High Performance Linear Algebra

Lecture 14: LAPACK and ScaLAPACK numerical algorithms

Ph.D. program in High Performance Scientific Computing

Fabio Durastante **Pasqua D'Ambra** **Salvatore Filippone**

Tuesday 03, 2026 — 14.00:16.00

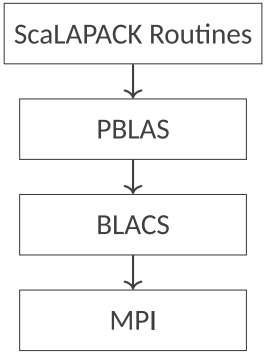
Dipartimento di Matematica
Università di Pisa

Table of Contents

Summary of ScaLAPACK Infrastructure

ScaLAPACK = Scalable Linear Algebra PACKage

- Distributed-memory extension of LAPACK for parallel computing
- Built on three key layers:
    1. **MPI** (Message Passing Interface) — low-level communication
    2. **BLACS** (Basic Linear Algebra Communication Subprograms) — higher-level communication primitives
    3. **PBLAS** (Parallel BLAS) — distributed matrix operations
- Matrix distributed across process grid using 2D block-cyclic layout



Each layer **abstracts communication details**, enabling scalable algorithms

Initialization: process grid and matrix distribution

Process grid setup:

```
    CALL BLACS_GRIDINIT( ICTXT, 'R', NPROW, NPCOL )
```

where ICTXT is the BLACS context, NPROW and NPCOL define the process grid dimensions.

Matrix descriptor creation:

```
    CALL DESCINIT( DESC, M, N, MB, NB, RSRC, CSRC, ICTXT, LLD, INFO )
```

where DESC is the descriptor array for the distributed matrix, M, N are global matrix dimensions, MB, NB are block sizes, and RSRC, CSRC specify the process owning the first block.

Information on the local part of the matrix can be obtained using:

```
    CALL NUMROC( N, NB, IPROC, ISRCPROC, NPROCS )
```

which computes the number of rows or columns of the distributed matrix owned by a specific process.

ScaLAPACK: Driver Routines for Linear Systems

Two types of driver routines are provided for solving systems of linear equations:

- **Simple driver** (name ending `-SV`):
  - Solves the system $AX = B$ by factorizing A and overwriting B with the solution X
- **Expert driver** (name ending `-SVX`):
  - Solve $A^\top X = B$ or $A^H X = B$ (unless A is symmetric or Hermitian)
  - Estimate the condition number of A, check for near-singularity and pivot growth
  - Refine the solution and compute forward and backward error bounds
  - Equilibrate the system if A is poorly scaled

Expert driver requires roughly twice as much storage to perform these extra functions.

Both types handle multiple right-hand sides (columns of B). Different drivers exploit special properties or storage schemes of matrix A.

Note: For band/tridiagonal matrices (`PxDBTRF`, `PxDTTRF`, `PxGBTRF`, `PxPBTRF`, `PxPTTRF`), the factorization differs from LAPACK due to additional permutations for parallelism.

ScaLAPACK: Linear System Solution Drivers

Type of matrix and storage scheme	Operation	Single precision		Double precision	
		Real	Complex	Real	Complex
general (partial pivoting)	simple driver	PSGESV	PCGESV	PDGESV	PZGESV
	expert driver	PSGESVX	PCGESVX	PDGESVX	PZGESVX
general band (partial pivoting)	simple driver	PSGBSV	PCGBSV	PDGBSV	PZGBSV
general band (no pivoting)	simple driver	PSDBSV	PCDBSV	PDDBSV	PZDBSV
general tridiagonal (no pivoting)	simple driver	PSDTSV	PCDTSV	PDDTSV	PZDTSV

ScaLAPACK: Linear System Solution Drivers

Type of matrix and storage scheme	Operation	Single precision		Double precision	
		Real	Complex	Real	Complex
symmetric/Hermitian positive definite	simple driver	PSPOSV	PCPOSV	PDPOSV	PZPOSV
	expert driver	PSPOSVX	PCPOSVX	PDPOSVX	PZPOSVX
symmetric/Hermitian positive definite band	simple driver	PSPBSV	PCPBSV	PDPBSV	PZPBSV
symmetric/Hermitian positive definite tridiagonal	simple driver	PSPTSV	PCPTSV	PDPTSV	PZPTSV

Divide and Conquer for Banded Linear Systems

The algorithm we discuss is based on the **divide and conquer** strategy introduced in

- J. Dongarra and L. Johnsson. Solving banded systems on a parallel processor. Parallel Computing, 5:219–246, 1987,
- A. Cleary and J. Dongarra. Implementation in ScaLAPACK of Divide-and-Conquer Algorithms for Banded and Tridiagonal Linear Systems. Computer Science Dept. Technical Report CS-97-358, University of Tennessee, Knoxville, TN, April 1997. (Also LAPACK Working Note 125).

The main idea is to **partition the banded matrix into smaller submatrices**, solve the smaller systems independently, and then combine the solutions to obtain the solution of the original system.

Divide and Conquer for Banded Linear Systems

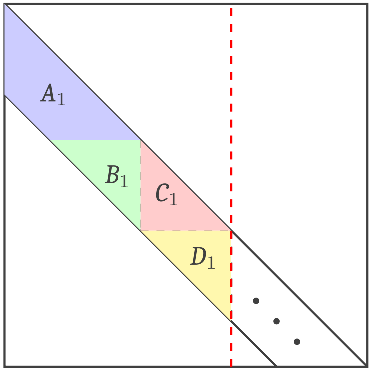
$$A\mathbf{x} = \mathbf{b},\ \text{lower bandwidth } \beta_l,\ \text{upper bandwidth } \beta_u$$

The algorithm follows these steps:

1. First produce a reordering: $PA(P^{-1}P)\mathbf{x} = P\mathbf{b}$, where P is a permutation matrix,
2. The reorderd matrix PAP^{-1} is factore as LU or $LL^\top$ via Gaussian Elimination/Chokesky,
3. Solve the system $LU\mathbf{x}' = \mathbf{b}'$ (or $LL^\top\mathbf{x}' = \mathbf{b}'$) where $\mathbf{x}' = P\mathbf{x}$ and $\mathbf{b}' = P\mathbf{b}$,
   3.1 Solve $L\mathbf{z} = \mathbf{b}'$ via forward substitution,
   3.2 Solve $U\mathbf{x}' = \mathbf{z}$ via back substitution,
4. Finally, recover the solution $\mathbf{x} = P^{-1}\mathbf{x}'$.

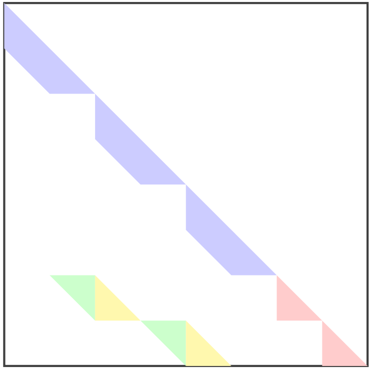
Find a good $n \times n$ permutation matrix P that reorders A to allow exploitation of parallelism,

The Symmetric and Positive Definite Case



- User inputs the matrix in **lower triangular form**,
- Each processor stores a contiguous set of columns of the matrix
- We partition each process matrix:
  - A_i: "trapezoidal" block along the diagonal of size O_i,
  - B_i, C_i, D_i: lower triangular blocks of size $\beta \times \beta$,
  - The **last processor** has only the A_i block.

The Symmetric and Positive Definite Case



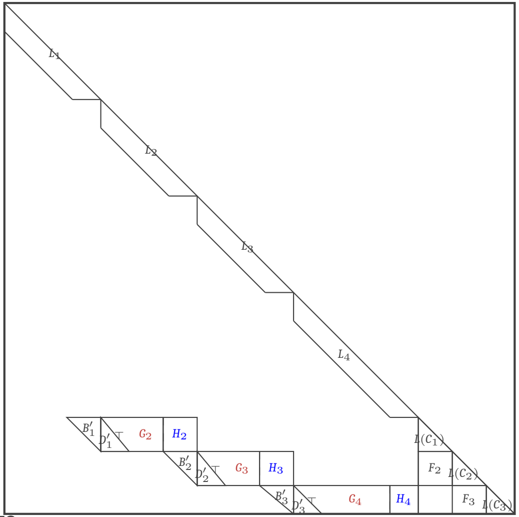
The **reordering goes** as follows:

- Number the equations in the A_i first, keeping the same relative order,
- Number the equations in the C_i next, keeping the same relative order,

The **Cholesky factorization** of the reordered matrix can be computed with sequential block operations.

⚠ We do not physically reorder the matrix but, rather, we base block operations on the reordering.

Cholesky Factor of the Permuted Matrix



- **Factorization**: largely computed with sequential block operations, minimal communication required.
- **Fill-in**: G_i and H_i represent fill-in, doubling nonzeros compared to sequential algorithms.
- **Operation Count**: $O(4N\beta^2)$
  - A_i factorization: $N\beta^2$.
  - Forming G_i: $2N\beta^2$.
  - Updating C_i with G_i: $N\beta^2$.
- **Total**: Approx. $4\times$ operations of the sequential algorithm.

The three phases of the Divide and Conquer Algorithm

Phase 1	Formation of the **reduced system**. Each processor does computations independently (for the most part) with local parts and then combines to form the Schur complement system corresponding to the parts already factored. The Schur complement is often called the reduced system.
Phase 2	The reduced system is solved, and the answers are communicated back to all of the processors.
Phase 3	The solutions from Phase 2 are applied in a backsolution process.

Phase 1: Formation of the Reduced System

We look at the i-th processor, first and last processors have special cases.

- **Communication Step:** D_i is sent to processor $i+1$. Since this is a small communication, it is completely overlapped with subsequent computations.

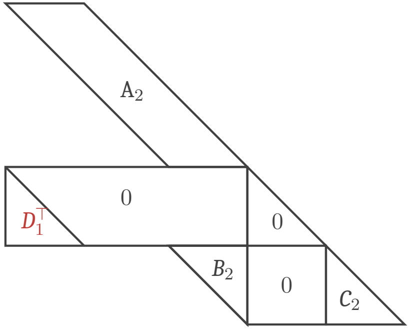
At this point, portions of the matrix are stored locally.

- We treat local computations as a **frontal computation**.
- We perform O_i factorization steps and apply them to the remaining submatrix of size 2β.
- This submatrix is subsequently used in **Phase 2** to form the reduced system.

The “divide” in the algorithm’s name stems from the reordering allowing each defined front to be independent.

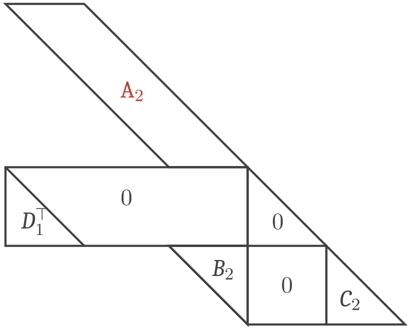
Only the 2β update equations at the end of each front need be coordinated with other processors.

Phase 1: Continued



- Start the exchange of D_i block with process $i+1$.

Phase 1: Continued

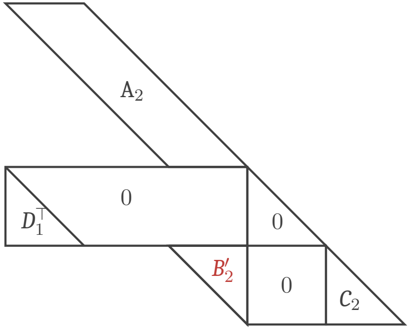


```
call dpbtrf( ... , A_i, ... )
```

- Start the exchange of D_i block with process $i+1$.
- Take O_i steps of **Cholesky factorization** on the local front:
  - Factor A_i to get $L(A_i)$,

Phase 1: Continued

1 ScaLAPACK Numerical Routines

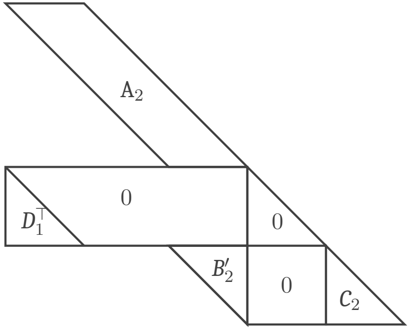


```
call dtrtrs( ... , B_i, ... )
```

- Start the exchange of D_i block with process $i+1$.
- Take O_i steps of **Cholesky factorization** on the local front:
  - Factor A_i to get $L(A_i)$,
  - Update B_i using $L_i = L(A_i)$: $L_i B_i'^\top = B_i^\top$,

Phase 1: Continued

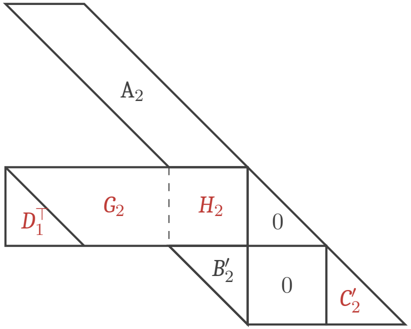
1 ScaLAPACK Numerical Routines



```
call dtrmm( ... , B_i, ... )
```

- Start the exchange of D_i block with process $i+1$.
- Take O_i steps of **Cholesky factorization** on the local front:
  - Factor A_i to get $L(A_i)$,
  - Update B_i using $L_i = L(A_i)$: $L_i B_i'^\top = B_i^\top$,
  - Update C_i using B_i' as $C_i' = C_i - B_i' B_i'^\top$.

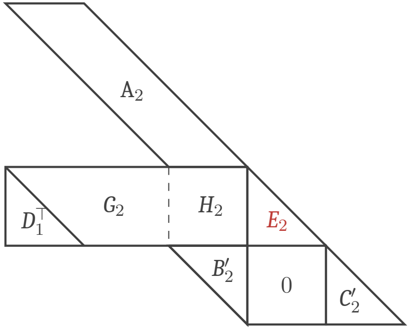
Phase 1: Continued



```
call dtbtrs( ... , B_i, ... )
```

- Start the exchange of D_i block with process $i+1$.
- Take O_i steps of **Cholesky factorization** on the local front:
  - Factor A_i to get $L(A_i)$,
  - Update B_i using $L_i = L(A_i)$: $L_i B_i'^\top = B_i^\top$,
  - Update C_i using B_i' as $C_i' = C_i - B_i' B_i'^\top$.
- The exchanged D_i is **now needed**:
  - Compute G_i from $L_i G_i^\top = D_i$,

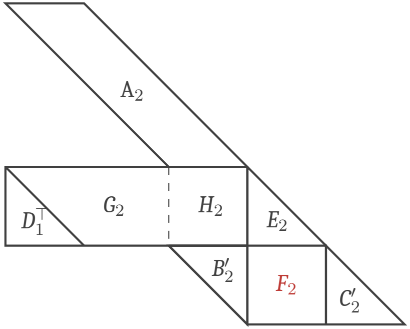
Phase 1: Continued



```
call dsyrk( ... , G_i, ... )
```

- Start the exchange of D_i block with process $i+1$.
- Take O_i steps of **Cholesky factorization** on the local front:
  - Factor A_i to get $L(A_i)$,
  - Update B_i using $L_i = L(A_i)$: $L_i B_i'^\top = B_i^\top$,
  - Update C_i using B_i' as $C_i' = C_i - B_i' B_i'^\top$.
- The exchanged D_i is **now needed**:
  - Compute G_i from $L_i G_i^\top = D_i$,
  - The matrix E_i represents the contribution from processor i to the diagonal block of the reduced system stored on process $i - i$, that is C_{i-1}': $E_i = G_i G_i^\top$

Phase 1: Continued

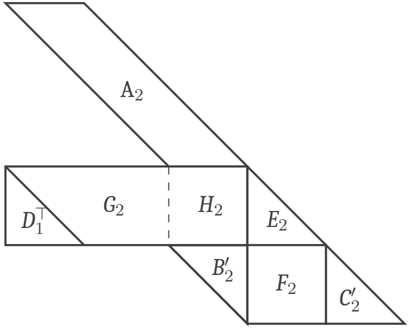


```
call dtrmm( ... , B_i, ... )
```

- The local computation is finished by computing F_i, using B_i' and the last β columns of G_i, which we have labelled H_i:

$$F_i^\top = H_i B_i'^\top$$

Phase 1: Continued



- The local computation is finished by computing F_i, using B'_i and the last β columns of G_i, which we have labelled H_i:

$$F_i^\top = H_i {B'_i}^\top$$

- The processor is now ready for **Phase 2**.

Phase 2: Solution of the Reduced System

Phase 2 consists of the forming and factorization of the Schur complement matrix.

- Each processor contributes three blocks of size $\beta \times \beta$ to this system: E_i, F_i, C_i'.
- Each C_i' is added to E_{i+1} to form the diagonal blocks of the matrix,.
- The F_i form the off-diagonal blocks.

The resultant system is **block tridiagonal**, with $P - 1$ blocks.

Several methods for factoring the reduced system have been proposed:

- **For small P or small β**: Perform an all-to-all broadcast of each processor's portion of the reduced system.
  - 👍 Disadvantage: Entire reduced system ends up on each processor.
  - 👍 Advantage: Only one (expensive) communication step.
  - 👎 Disadvantage: Redundant computation (serial algorithm), will not scale.
- **Parallel Solution**: For larger problems, ScaLAPACK uses a parallel block tridiagonal solver scaling as $O(\log P)$ or P.

Phase 2: Solution of the Reduced System

We use a block formulation of **odd-even (or cyclic) reduction**.

- This algorithm has $\log_2 P$ stages.
- At each stage, the **odd-numbered blocks** are used to “eliminate” the **even-numbered blocks**.
- The process decreases the number of blocks left by a factor of **two** at each stage.
- Symmetry is maintained throughout (Cholesky factorization of a symmetric permutation of the reduced system).

Reordering Strategy:

- Blocks are ordered so that even-numbered blocks in Step 1 are first, those in Step 2 correspond to second, and so on.
- Results in an elimination tree of minimal height.

⚠ Implementation requires additional space allocation for fill-in created by the reordering (though of much lower order than Phase 1 fill-in).

Phase 3: Backsolution

- Phase 3 is specific to the solution step (not factorization).
- Operations performed mirror the factorization steps (Phase 1 and 2) but operate on the **right-hand sides**.

Procedure At the end of Phase 2, processors hold portions of the solution to the reduced system.

Communication Each processor distributes 2β elements of this solution to neighboring processors.

Computation Partial solutions are back-substituted into locally stored factors. This is a completely local computation stage.

- Structure is similar to factorization but simpler.
- Uses block operations from **LAPACK** and **BLAS**.
- Multiple right-hand sides are handled efficiently in this context.

Running an example

We want to **run an example** of the parallel divide and conquer algorithm:

```
call pdpbsv(uplo, n, bw, nrhs, a, ja, desca, b, ib, descb, work, lwork, info)
```

Where:

- `uplo`: ‘L‘ for lower triangular storage,
- `n`: order of the matrix,
- `bw`: bandwidth of the matrix,
- `nrhs`: number of right-hand sides,
- `a`: local array containing the lower triangular part of the matrix,
- `ja`: global column index of the first local column of `a`,
- `desca`: array descriptor for matrix `a`,
- `b`: local array containing the right-hand side(s),
- `ib`: global row index of the first local row of `b`,
- `descb`: array descriptor for matrix `b`,
- `work`: workspace array,
- `lwork`: size of the workspace array,
- `info`: output status variable.

The construction of the input matrix

- Initialize matrix A in banded format,
- Create a symmetric positive definite banded matrix,
- In packed banded format for `UPLO='U'`:
  $A_{i,j}$ is stored in `A(BW+1+i-j, j)` for $\max(1, j - \text{BW}) \leq i \leq j$.

```fortran
a = 0.0_real64
do j = 1, loc_n_a
   ! Global column index
   jloc = (mycol * nb) + j
   if (jloc <= n) then
       ! Diagonal element (make it dominant for positive definiteness)
       a(bw + 1 + (j-1)*lld_a) = 4.0_real64 + 2.0_real64 * real(bw, real64)
```

The construction of the input matrix

```
      ! Off-diagonal elements
      do i = max(1, jloc - bw), jloc - 1
        if (i >= 1 .and. i < jloc) then
            iloc = bw + 1 + i - jloc
            if (iloc >= 1 .and. iloc <= bw + 1) then
              a(iloc + (j-1)*lld_a) = -1.0_real64
            end if
        end if
      end do
  end if
end do
```

The construction of the input matrix

The previous code build the following matrix for $n = 8$ and $bw = 2$:

$$A = \begin{bmatrix} 6 & -1 & -1 & 0 & 0 & 0 & 0 & 0 \\ -1 & 6 & -1 & -1 & 0 & 0 & 0 & 0 \\ -1 & -1 & 6 & -1 & -1 & 0 & 0 & 0 \\ 0 & -1 & -1 & 6 & -1 & -1 & 0 & 0 \\ 0 & 0 & -1 & -1 & 6 & -1 & -1 & 0 \\ 0 & 0 & 0 & -1 & -1 & 6 & -1 & -1 \\ 0 & 0 & 0 & 0 & -1 & -1 & 6 & -1 \\ 0 & 0 & 0 & 0 & 0 & -1 & -1 & 6 \end{bmatrix}$$

which is symmetric positive definite and banded with bandwidth 2.

The work spaces

- ScaLAPACK routines use a **workspace array** `WORK` of size `LWORK` to store temporary data.
- The size of `LWORK` depends on the routine and the problem size.
- If `LWORK` is too small, the routine returns an error.

For `PDPBSV`, the minimal size is:

$$\mathtt{LWORK} \geq (\mathtt{NB} + 2 \cdot \mathtt{BW}) \cdot \mathtt{BW} + \max((\mathtt{BW} \cdot \mathtt{NRHS}), \mathtt{BW}^2)$$

- `NB`: Block size,
- `BW`: Bandwidth,
- `NRHS`: Number of right-hand sides.

Query mechanism: If `LWORK = -1`, the routine calculates the optimal size and returns it in `WORK(1)`. This is the recommended way to allocate the workspace.

The work spaces

- ScaLAPACK routines use a **workspace array** WORK of size LWORK to store temporary data.
- The size of LWORK depends on the routine and the problem size.
- If LWORK is too small, the routine returns an error.

For PDPBSV, the minimal size is:

$$\mathtt{LWORK} \geq (\mathtt{NB} + 2 \cdot \mathtt{BW}) \cdot \mathtt{BW} + \max((\mathtt{BW} \cdot \mathtt{NRHS}), \mathtt{BW}^2)$$

- NB: Block size,
- BW: Bandwidth,
- NRHS: Number of right-hand sides.

```
lwork = (nb + 2*bw) * bw + max(bw*nrhs, bw*bw)
allocate(work(lwork), stat=info)
```

Calling the ScaLAPACK routine and measure time

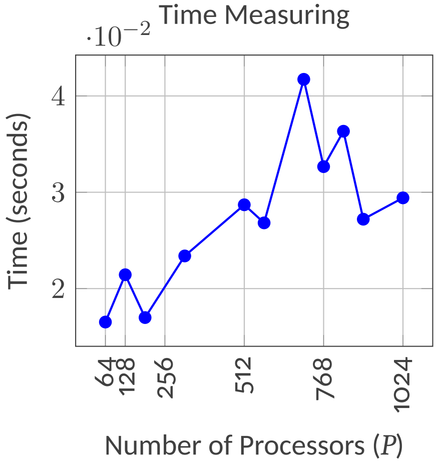
Finally, we can call the ScaLAPACK routine to solve the system $A\mathbf{x} = \mathbf{b}$:

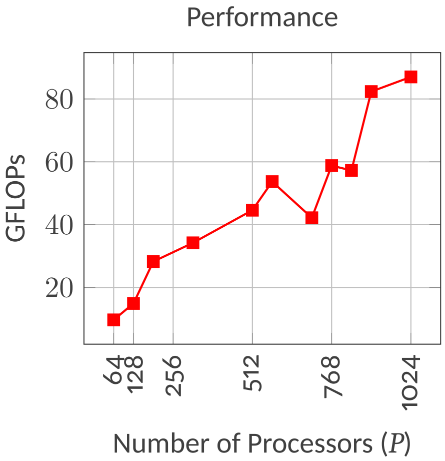
```fortran
! Start timing
t_start = mpi_wtime()
! Call PDPBSV to solve the system
call pdpbsv(uplo, n, bw, nrhs, a, ja, desca, b, ib, descb, &
            work, lwork, info)
! End timing
t_end = mpi_wtime()
t_elapsed = t_end - t_start
! All reduce to get maximum time across all processors
call mpi_allreduce(mpi_in_place, t_elapsed, 1, mpi_double_precision, &
                   mpi_max, mpi_comm_world, ierr)
```

Scaling Results

1 ScaLAPACK Numerical Routines

We run the **weak scaling** on the Amelia cluster at IAC-CNR, with Intel 2023.





Scaling Results

Observations:

- The time for the solution of the linear system remains constant as we increase the number of processors P while keeping the problem size N proportional to P.
- This is the expected behavior for a **weak scaling** experiment:
  - $N = P \times$ local_size.
  - Ideal weak scaling behavior: execution time is constant.
- The **GFLOPs** metric increases linearly with P, showing that we are able to effectively use the added computational power.

The divide and conquer algorithm for banded systems in ScaLAPACK exhibits good parallel scalability, effectively handling the inherent dependencies of the banded structure through the hierarchical decomposition.

Linear Least Squares Problems

The **linear least squares (LLS) problem** is defined as finding x that minimizes:

$$\min_x \|b - Ax\|_2$$

where A is an $m \times n$ matrix and b is an m-element vector.

Overdetermined ($m \geq n$)

- If full rank (n), the solution is unique.
- “Least squares solution”.

Underdetermined ($m < n$)

- If full rank (m), infinite solutions satisfy $b - Ax = 0$.
- We seek the **minimum norm solution** which minimizes $\|x\|_2$.

In the **rank-deficient** case ($rank(A) < \min(m, n)$), we seek the **minimum norm least squares solution** which minimizes both $\|b - Ax\|_2$ and $\|x\|_2$.

ScaLAPACK LLS Driver: PxGELS

The driver routine `PxGELS` solves the LLS problem assuming A has **full rank**.

- Finds the LLS solution for $m \geq n$ and the minimum norm solution for $m < n$.
- Uses **QR factorization** or **LQ factorization** of A.
- Can handle A or A^H (or $A^\top$ for real matrices).
- Handles multiple right-hand sides (columns of B) in a single call.

Driver Routines for Linear Least Squares:

Problem Type	Driver	Single precision		Double precision	
		Real	Complex	Real	Complex
Full rank LLS	`PxGELS`	`PSGELS`	`PCGELS`	`PDGELS`	`PZGELS`

Symmetric Eigenproblems (SEP)

The **symmetric/Hermitian eigenvalue problem** is to find the **eigenvalues** λ and corresponding **eigenvectors** $z \neq 0$ such that:

$$Az = \lambda z, \quad \text{where } A = A^\top \text{ (symmetric) or } A = A^H \text{ (Hermitian).}$$

The eigenvalues λ are always real.
When all eigenvalues and eigenvectors are computed, we can write the **spectral factorization** $A = Z\Lambda Z^H$, where Λ is diagonal containing eigenvalues, and Z is orthogonal (or unitary) containing eigenvectors.

Two types of driver routines are provided:

- **Simple driver** (name ending `-EV`):
  - Computes all eigenvalues and (optionally) eigenvectors.
- **Expert driver** (name ending `-EVX`):
  - Computes either all or a selected subset of eigenvalues.
  - Optionally computes corresponding eigenvectors.

Singular Value Decomposition (SVD)

The **Singular Value Decomposition (SVD)** of an $m \times n$ matrix A is given by:

$$A = U\Sigma V^\top \quad (\text{or } A = U\Sigma V^H \text{ for complex})$$

where U and V are orthogonal (unitary) and Σ is an m-by-n diagonal matrix with real diagonal elements σ_i, such that:

$$\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_{\min(m,n)} \geq 0$$

The σ_i are the **singular values**, and the first $\min(m,n)$ columns of U and V are the **left** and **right singular vectors**, satisfying:

$$Av_i = \sigma_i u_i \quad \text{and} \quad A^\top u_i = \sigma_i v_i$$

Driver Routine: `PxGESVD`

Singular Value Decomposition (SVD)

- Computes the “economy size” or “thin” SVD.
- If $m > n$: only the first n columns of U are computed.

Operation	Driver	Single precision		Double precision	
		Real	Complex	Real	Complex
Thin SVD	`PxGESVD`	`PSGESVD`	`PCGESVD`	`PDGESVD`	`PZGESVD`

Generalized Symmetric Definite Eigenproblems (GSEP)

An **expert driver** is provided to compute all (or selected) eigenvalues and (optionally) eigenvectors for the following problems:

1. $Az = \lambda Bz$
2. $ABz = \lambda z$
3. $BAz = \lambda z$

where A and B are symmetric (or Hermitian) and B is positive definite.

Properties:

- The eigenvalues λ are real.
- The matrix of eigenvectors Z satisfies orthogonality conditions relative to B (e.g., $Z^H BZ = I$ for type 1).

Generalized Symmetric Definite Eigenproblems (GSEP)

An **expert driver** is provided to compute all (or selected) eigenvalues and (optionally) eigenvectors for the following problems:

1. $Az = \lambda Bz$
2. $ABz = \lambda z$
3. $BAz = \lambda z$

where A and B are symmetric (or Hermitian) and B is positive definite.

Driver Routines (Expert):

Matrix Type	Single precision		Double precision	
	Real	Complex	Real	Complex
Symmetric	`PSSYGVX`	–	`PDSYGVX`	–
Hermitian	–	`PCHEGVX`	–	`PZHEGVX`

Computational Routines Overview

ScaLAPACK provides a suite of **computational routines** for solving linear algebra problems, including:

- Solving systems of linear equations,
- Computing matrix factorizations,
- Estimating condition numbers,
- Refining solutions and computing error bounds,
- Computing matrix inverses,
- Performing matrix equilibration.

These routines are designed to work with distributed matrices and leverage parallel computing architectures for efficiency and scalability.

Computational Routines for Linear Equations

Linear System Notation:

$$AX = B$$

where A is the coefficient matrix, B is the right-hand side, and X is the solution.

ScaLAPACK provides computational routines for factorizing A based on its properties:

- **General matrices**: LU factorization with partial pivoting ($A = PLU$).
- **Symmetric/Hermitian positive definite**: Cholesky factorization ($A = U^H U$ or $A = LL^H$).
- **Band matrices**: Generalized band factorizations.
- **Tridiagonal matrices**: Specialized LU or LDL^H factorizations.

Tasks Performed by Computational Routines

The last three characters of the routine name indicate the task:

`PxyyTRF`	**Factorize** the matrix (e.g., LU, Cholesky).
`PxyyTRS`	Use the factorization to **solve** $AX = B$ via forward/backward substitution.
`PxyyCON`	Estimate the reciprocal of the **condition number** $\kappa(A)$.
`PxyyRFS`	**Refine the solution** and compute error bounds (iterative refinement).
`PxyyTRI`	Compute the **matrix inverse** A^{-1} using the factorization.
`PxyyEQU`	Compute **equilibration** scaling factors to improve condition number.

Note: Not all routines are available for all matrix types (e.g., inversion is not provided for band matrices as the inverse is generally dense).

Summary of Computational Routines

Matrix type	Factorize	Solve	Condition Estimate	Error Bounds	Invert	Equilibrate
General	PxGETRF	PxGETRS	PxGECON	PxGERFS	PxGETRI	PxGEEQU
General Band	PxGBTRF	PxGBTRS	PxGBCON	PxGBRFS	–	PxGBEQU
General Tridiagonal	PxDTTRF	PxDTTRS	–	–	–	–
Sym./Herm. Pos. Def.	PxPOTRF	PxPOTRS	PxPOCON	PxPORFS	PxPOTRI	PxPOEQU
Sym./Herm. Pos. Def. Band	PxPBTRF	PxPBTRS	PxPBCON	PxPBRFS	–	PxPBEQU
Sym./Herm. Pos. Def. Tridiagonal	PxPTTRF	PxPTTRS	–	–	–	–
Triangular	–	PxTRTRS	PxTRCON	PxTRRFS	PxTRTRI	–

Computational Routines for Orthogonal Factorization and LLS

ScaLAPACK provides routines for **orthogonal factorizations** and solving **linear least squares (LLS)** problems.

Orthogonal Factorizations:

- QR factorization ($A = QR$) for general matrices.
- LQ factorization ($A = LQ$) for general matrices.

Linear Least Squares Problems:

- Solve overdetermined systems ($m \geq n$) to find the least squares solution.
- Solve underdetermined systems ($m < n$) to find the minimum norm solution.

QR Factorization

The most common of the factorizations is the **QR factorization** given by:

$$A = QR$$

where R is $n \times n$ upper triangular and Q is $m \times m$ orthogonal (or unitary). If A is of full rank n, then R is nonsingular.

It is often written as:

$$A = \begin{pmatrix} Q_1 & Q_2 \end{pmatrix} \begin{pmatrix} R_1 \\ 0 \end{pmatrix} = Q_1 R_1$$

where Q_1 consists of the first n columns of Q, and Q_2 the remaining $m - n$ columns.
If $m < n$:

$$A = Q \begin{pmatrix} R_1 & R_2 \end{pmatrix}$$

where R_1 is upper triangular and R_2 is rectangular.

ScaLAPACK support for QR

The routine `PxGEQRF` computes the QR factorization.

- Q is not formed explicitly but represented as a **product of elementary reflectors** ($H_i = I - \tau v v^H$).
- `PxORGQR` (or `PxUNGQR`): Generates all or part of Q.
- `PxORMQR` (or `PxUNMQR`): Pre- or post-multiplies a matrix by Q or Q^H.

Orthogonal or Unitary Matrices in ScaLAPACK

ScaLAPACK represents an orthogonal (or unitary) matrix Q as a product of **elementary reflectors** (Householder matrices):

$$Q = H_1 H_2 \cdots H_k$$

where each $H_i = I - \tau v v^H$.

- τ is a scalar and v is the **Householder vector**.
- $v_1 = 1$, so it does not need to be stored.

Working with Q: Most users do not operate on H_i directly but use provided routines:

- **Generate Q**: Routines ending in `-ORG` / `-UNG` (e.g., `PDORGQR`) form Q explicitly.
- **Apply Q**: Routines ending in `-ORM` / `-UNM` (e.g., `PDORMQR`) compute $Q^T C$ or QC without forming Q.

Note: In complex arithmetic, elementary reflectors are unitary but not Hermitian. This allows reducing complex Hermitian matrices to **real symmetric tridiagonal** form.

Solving Linear Least Squares with QR

When $m \geq n$ and A has full rank, the LLS problem minimizes $\|b - Ax\|_2$.

$$\|b - Ax\|_2 = \|Q^H b - Q^H Ax\|_2 = \left\| \begin{pmatrix} c_1 \\ c_2 \end{pmatrix} - \begin{pmatrix} R_1 \\ 0 \end{pmatrix} x \right\|_2$$

Steps:

1. Compute $c = Q^H b$ using `PxORMQR`,
2. Solve the upper triangular system $R_1 x = c_1$ using `PxTRTRS`,
3. The residual vector is $r = Q \begin{pmatrix} 0 \\ c_2 \end{pmatrix}$, its norm is $\|c_2\|_2$.

LQ Factorization

The **LQ factorization** is given by:

$$A = \begin{pmatrix} L & 0 \end{pmatrix} \begin{pmatrix} Q_1 \\ Q_2 \end{pmatrix} = LQ_1$$

where L is $m \times m$ lower triangular, Q is $n \times n$ orthogonal (or unitary), Q_1 consists of the first m rows of Q, and Q_2 the remaining $n - m$ rows.

ScaLAPACK Routines:

- `PxGELQF`: Computes the factorization. Q is represented as a product of elementary reflectors.
- `PxORGLQ` (or `PxUNGLQ`): Generates all or part of Q.
- `PxORMLQ` (or `PxUNMLQ`): Pre- or post-multiplies a matrix by Q or Q^H.

The LQ factorization of A corresponds to the QR factorization of $A^\top$ (or A^H):

$$A = LQ \iff A^\top = Q^\top L^\top$$

Solving Underdetermined Systems with LQ

The LQ factorization is used to find the **minimum norm solution** of an **underdetermined** system ($m < n$) with rank m.
The solution is given by:

$$x = Q^H \begin{pmatrix} L^{-1}b \\ 0 \end{pmatrix}$$

Computational Steps:

1. Solve the lower triangular system $Ly = b$ for y using `PxTRTRS`,
2. Form the vector $\tilde{y} = \begin{pmatrix} y \\ 0 \end{pmatrix}$,
3. Compute $x = Q^H\tilde{y}$ using `PxORMLQ` (or `PxUNMLQ`).

QR Factorization with Column Pivoting

If A is not of full rank, or the rank is in doubt, we can perform a **QR factorization with column pivoting**:

$$AP = QR$$

where P is a permutation matrix.

- P is chosen so that $|r_{11}| \geq |r_{22}| \geq \cdots \geq |r_{nn}|$.
- And for each k, the leading submatrix R_{11} of size $k \times k$ is well-conditioned, while the trailing submatrix R_{22} is negligible.

Rank Determination: If R_{22} is negligible, then k is the **effective rank** of A.

Basic Solution to LLS:

$$x = P \begin{pmatrix} R_{11}^{-1} c_1 \\ 0 \end{pmatrix}$$

where c_1 consists of the first k elements of $c = Q^H b$.

ScaLAPACK routine for QR with Pivoting

The routine `PxGEQPF` computes the QR factorization with column pivoting.

- It does not attempt to determine the rank of A automatically (user must inspect diagonal of R).
- Q is represented in the same way as in `PxGEQRF`.
- Routines `PxORMQR` (real) or `PxUNMQR` (complex) can be used to apply Q.

Complete Orthogonal Factorization

QR with column pivoting does not compute a **minimum norm** solution for rank-deficient LLS unless $R_{12} = 0$.

To solve this, we apply further orthogonal transformations from the right to the upper trapezoidal matrix R (using `PxTZRZF`) to eliminate R_{12}:

$$RP = \begin{pmatrix} T_{11} & 0 \\ 0 & 0 \end{pmatrix} Z$$

This yields the **complete orthogonal factorization**:

$$AP = Q \begin{pmatrix} T_{11} & 0 \\ 0 & 0 \end{pmatrix} Z$$

Complete Orthogonal Factorization

From this, the minimum norm solution is obtained as:

$$x = PZ^H \begin{pmatrix} T_{11}^{-1} c_1 \\ 0 \end{pmatrix}$$

Software Details:

- Z is represented as a product of elementary reflectors.
- `PxORMRZ` (or `PxUNMRZ`) is provided to multiple a matrix by Z or Z^H.

Summary of Orthogonal Factorization Routines (QR)

All the factorization routines discussed here (except `PxTZRZF`) allow arbitrary m and n, so that in some cases the matrices R or L are trapezoidal rather than triangular.

A routine that performs **pivoting** is provided **only for the QR factorization**.

Computational routines for QR factorization

Task	Single precision		Double precision	
	Real	Complex	Real	Complex
QR factorization and related operations				
Factorize generic	`PSGEQRF`	`PCGEQRF`	`PDGEQRF`	`PZGEQRF`
Factorize w/ pivoting	`PSGEQPF`	`PCGEQPF`	`PDGEQPF`	`PZGEQPF`
Apply Q	`PSORMQR`	`PCUNMQR`	`PDORMQR`	`PZUNMQR`
Generate Q	`PSORGQR`	`PCUNGQR`	`PDORGQR`	`PZUNGQR`

Summary of Orthogonal Factorization Routines (LQ)

Computational routines for LQ factorization

Task	Single precision		Double precision	
	Real	Complex	Real	Complex
LQ factorization and related operations				
Factorize generic	PSGELQF	PCGELQF	PDGELQF	PZGELQF
Apply Q	PSORMLQ	PCUNMLQ	PDORMLQ	PZUNMLQ
Generate Q	PSORGLQ	PCUNGLQ	PDORGLQ	PZUNGLQ

Generalized QR (GQR) Factorization

The **Generalized QR (GQR) factorization** of an $n \times m$ matrix A and an $n \times p$ matrix B is given by the pair of factorizations:

$$A = QR, \quad B = QTZ$$

where Q ($n \times n$) and Z ($p \times p$) are orthogonal (or unitary) matrices. R and T are generally upper triangular (or trapezoidal) matrices.

Implicit Factorization

If B is square and nonsingular, the GQR factorization implicitly gives the QR factorization of $B^{-1}A$:

$$B^{-1}A = Z^H(T^{-1}R)$$

without explicitly computing the inverse or the matrix product.

Generalized QR (GQR) Factorization

The **Generalized QR (GQR) factorization** of an $n \times m$ matrix A and an $n \times p$ matrix B is given by the pair of factorizations:

$$A = QR, \quad B = QTZ$$

where Q $(n \times n)$ and Z $(p \times p)$ are orthogonal (or unitary) matrices. R and T are generally upper triangular (or trapezoidal) matrices.

ScaLAPACK Routine: `PxGGQRF`

- Algorithms proceeds by computing the **QR factorization** of A and then the **RQ factorization** of $Q^H B$.
- Q and Z can be formed explicitly or used to multiply other matrices (like standard QR).

Generalized RQ (GRQ) Factorization

The **Generalized RQ (GRQ) factorization** of an $m \times n$ matrix A and a $p \times n$ matrix B is given by the pair of factorizations:

$$A = RQ, \quad B = ZTQ$$

where Q ($n \times n$) and Z ($p \times p$) are orthogonal (or unitary) matrices.

Structure:

- R is upper trapezoidal (triangular) with structure similar to the R in RQ.
- T is upper trapezoidal (triangular) with structure similar to the R in QR.

Implicit Factorization

If B is square and nonsingular, the GRQ factorization implicitly gives the RQ of AB^{-1}:

$$AB^{-1} = (RT^{-1})Z^H$$

without explicitly computing the inverse or the product.

Generalized RQ (GRQ) Factorization

The **Generalized RQ (GRQ) factorization** of an $m \times n$ matrix A and a $p \times n$ matrix B is given by the pair of factorizations:

$$A = RQ, \quad B = ZTQ$$

where Q $(n \times n)$ and Z $(p \times p)$ are orthogonal (or unitary) matrices.

Structure:

- R is upper trapezoidal (triangular) with structure similar to the R in RQ.
- T is upper trapezoidal (triangular) with structure similar to the R in QR.

ScaLAPACK Routine: `PxGGRQF`

- Computes the **RQ factorization** of A first, then the **QR factorization** of BQ^H.
- Q and Z can be formed explicitly or used to multiply other matrices.

Computational Routines for Symmetric Eigenproblems

Problem Definition: Let A be a real symmetric or complex Hermitian $N \times N$ matrix. Find eigenvalues λ and non-zero eigenvectors z such that:

$$Az = \lambda z$$

Computation Stages:

1. **Reduction to Tridiagonal Form**:
$$A = QTQ^H$$
where Q is orthogonal (unitary) and T is real symmetric tridiagonal.
2. **Solve Tridiagonal Problem**: Compute eigenvalues/vectors of T.
$$T = S\Lambda S^\top$$
The eigenvectors of A are recovered as $Z = QS$.

Reduction to Tridiagonal Form

The reduction $A = QTQ^H$ is performed by:

- `PxSYTRD`: Real symmetric matrices.
- `PxHETRD`: Complex Hermitian matrices.

Handling Q:

- The matrix Q is represented as a product of elementary reflectors.
- `PxORMTR` (Real) / `PxUNMTR` (Complex): Multiplies a matrix by Q without forming it explicitly.
- Used to transform eigenvectors of T back to eigenvectors of A.

Solving the Tridiagonal Problem

ScaLAPACK provides specific routines for the tridiagonal phase:

`xSTEQR2` Modified version of LAPACK's `xSTEQR`.

- Computes all eigenvalues and (optionally) eigenvectors using implicit QL or QR.
- Optimized look-ahead and partial updates for parallel execution.

`PxSTEBZ` Uses **bisection**.

- Computes some or all eigenvalues (e.g., in an interval $[a, b]$ or indices i to j).
- Tunable accuracy vs. speed.

`PxSTEIN` Uses **inverse iteration**.

- Computes eigenvectors given accurate eigenvalues.
- Performs reorthogonalization to ensure vector quality (limited by workspace).

Summary of Computational Routines for SEP

Task	Single precision		Double precision	
	Real	Complex	Real	Complex
Tridiagonal reduction				
Factorize $A = QTQ^T$	`PSSYTRD`	`PCHETRD`	`PDSYTRD`	`PZHETRD`
Multiply by Q	`PSORMTR`	`PCUNMTR`	`PDORMTR`	`PZUNMTR`
Tridiagonal solvers				
All eigs (QR/QL)	`SSTEQR2`	`CSTEQR2`	`DSTEQR2`	`ZSTEQR2`
Selected eigs (Bisection)	`PSSTEBZ`	`PCSTEBZ`	`PDSTEBZ`	`PZSTEBZ`
Selected vectors (Inv. It.)	`PSSTEIN`	`PCSTEIN`	`PDSTEIN`	`PZSTEIN`

Nonsymmetric Eigenproblems (NEP)

Problem Definition: Let A be a square $n \times n$ matrix.

- A scalar λ is an **eigenvalue** and a non-zero vector v is a **right eigenvector** if:

$$Av = \lambda v$$

- A non-zero vector u is a **left eigenvector** if:

$$u^H A = \lambda u^H$$

Goal: The basic task is to compute all n eigenvalues λ and, optionally, their associated right eigenvectors v and/or left eigenvectors u.

Schur Factorization

A fundamental step in solving NEP is the **Schur factorization**:

$$A = ZTZ^H \quad (\text{or } A = ZTZ^\top \text{ for real matrices})$$

- **Complex Case**: Z is unitary, and T is upper triangular. The eigenvalues appear on the diagonal of T.
- **Real Case**: Z is orthogonal, and T is **upper quasi-triangular**.
  - T has 1-by-1 and 2-by-2 blocks on the diagonal.
  - Complex conjugate eigenvalues correspond to the 2-by-2 blocks.
- The columns of Z are called the **Schur vectors** of A.

Computational Stages for NEP

The computation is typically performed in two stages:

1. **Reduction to Upper Hessenberg Form**:

$$A = QHQ^H$$

   where H is **upper Hessenberg** (zero below the first subdiagonal) and Q is orthogonal/unitary.

2. **Schur Factorization of H**:

$$H = STS^H$$

   where T is the Schur form. The Schur vectors of the original matrix A are recovered as $Z = QS$.

Stage 1: Reduction to Hessenberg Form

Factorization Routine: `PxGEHRD`

- Reduces a general matrix A to upper Hessenberg form H.
- Represents the orthogonal matrix Q in a factored form (product of elementary reflectors).

Orthogonal/Unitary Matrix Operations:

- `PxORMHR` (Real) / `PxUNMHR` (Complex).
- Used to multiply another matrix by Q (or Q^H) without explicitly forming Q.

Stage 2: Schur Factorization

Routine: `PxLAHQR`

- Computes the Schur factorization of the upper Hessenberg matrix H.
- Eigenvalues are obtained from the diagonal of T.

Parallel Algorithm Strategy:

- Unlike LAPACK's `xLAHQR` (single double shift) or `xHSEQR` (single large multi-shift), ScaLAPACK uses **multiple double shifts**.
- Shifts are spaced apart to allow parallelism across several processor rows/columns.
- Shifts are applied in a block fashion to maximize performance.

Heuristics for Eigenvalue Computations: Shifting

The convergence of the QR algorithm (or similar iterative methods like the Francis double-shift) depends on the ratio of eigenvalues $|\lambda_i|/|\lambda_j|$.

Goal: Accelerate convergence by subtracting a scalar shift σ:

$$A - \sigma I = QR \quad \longrightarrow \quad A_{\text{new}} = RQ + \sigma I$$

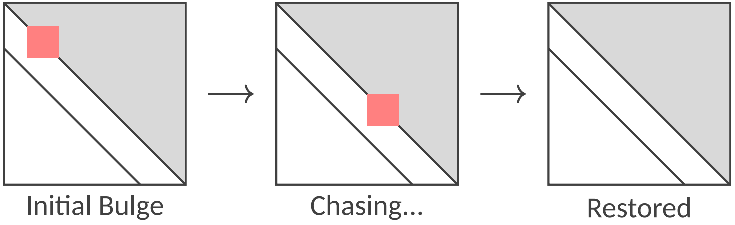
Ideally, if $\sigma \approx \lambda_n$, the deflation happens very quickly.

Techniques:

- **Francis Double Shift**: Standard for real non-symmetric matrices. Uses a 2×2 block from the bottom corner to perform implicitly shifted steps with complex conjugate shifts, keeping arithmetic real.
- **Aggressive Early Deflation**: Looks for convergence in a large window at the bottom of the matrix, deflating multiple eigenvalues at once.

Parallelism and Bulge Chasing

The implicit shift strategy creates a “bulge” (non-zero entries outside the Hessenberg form) that must be chased down the diagonal to restore the form.



Parallel Approach (ScaLAPACK):

- **Multi-shift:** Instead of chasing one bulge, introduce multiple bulges (chains of shifts) simultaneously.
- These bulges can be chased by different processors in a pipelined fashion, increasing the arithmetic intensity (BLAS 3) and parallelism.

Summary of Computational Routines for NEP

Task	Single precision		Double precision	
	Real	Complex	Real	Complex
Hessenberg reduction				
Factorize $A = QHQ^H$	PSGEHRD	PCGEHRD	PDGEHRD	PZGEHRD
Multiply by Q	PSORMHR	PCUNMHR	PDORMHR	PZUNMHR
Schur factorization				
Compute $H = STS^H$	PSLAHQR	PCLAHQR	PDLAHQR	PZLAHQR

Computational Stages for SVD

Let A be an $m \times n$ matrix. The computation of the SVD proceeds in two main stages:

1. **Reduction to Bidiagonal Form**:
$$A = QBP^H$$
where B is real bidiagonal, and Q ($m \times m$) and P ($n \times n$) are orthogonal (unitary).
   - If $m \geq n$, B is upper bidiagonal.
   - If $m < n$, B is lower bidiagonal.
2. **SVD of the Bidiagonal Matrix**:
$$B = U_1 \Sigma V_1^H$$
where U_1 and V_1 are orthogonal, and Σ contains the singular values.

The singular vectors of A are then $U = QU_1$ and $V = PV_1$.

Computational Routines for SVD

Reduction Routine: `PxGEBRD`

- Reduces A to bidiagonal form B.
- Represents Q and P as products of elementary reflectors.

Applying Q and P:

- `PxORMBR` (Real) / `PxUNMBR` (Complex).
- Routine to multiply a given matrix by Q or P (or their transposes).

Solving the Bidiagonal Problem:

- ScaLAPACK typically utilizes the **LAPACK** routine `xBDSQR` to compute the SVD of the bidiagonal matrix B.

Optimization for Non-Square Matrices

If $m \gg n$ or $n \gg m$, it is more efficient to perform a preliminary QR or LQ factorization.

Case $m \gg n$:

1. Compute QR factorization: $A = QR$ (using `PxGEQRF`).
2. Compute SVD of the $n \times n$ matrix R.

Case $n \gg m$:

1. Compute LQ factorization: $A = LQ$ (using `PxGELQF`).
2. Compute SVD of the $m \times m$ matrix L.

Note: The driver routine `PxGESVD` automatically handles these paths.

Rank-Deficient Linear Least Squares

The SVD is used to solve rank-deficient LLS problems ($\min \|b - Ax\|_2$) by finding the **minimum norm solution**.

Let k be the **effective rank** of A (number of singular values $\sigma_i >$ threshold).
The solution is given by:

$$x = V_k \Sigma_k^{-1} c_1$$

where:

- Σ_k is the leading $k \times k$ submatrix of Σ,
- V_k consists of the first k columns of V,
- c_1 consists of the first k elements of $c = U^H b$.

`PxORMBR` (or `PxUNMBR`) is used to compute $U^H b$.

Summary of SVD Computational Routines

Task	Single precision		Double precision	
	Real	Complex	Real	Complex
Bidiagonal reduction				
Factorize $A = QBP^H$	PSGEBRD	PCGEBRD	PDGEBRD	PZGEBRD
Multiply by Q or P	PSORMBR	PCUNMBR	PDORMBR	PZUNMBR
Bidiagonal SVD (LAPACK)				
SVD of B	SBDSQR	CBDSQR	DBDSQR	ZBDSQR

Computational Routines for GSEP

Reduction to Standard Form: The generalized problems are reduced to the standard symmetric eigenvalue problem $Cy = \lambda y$ using the Cholesky factorization of B ($B = U^H U$ or $B = LL^H$).

Reduction Strategy:

- Type 1 ($Az = \lambda Bz$): $C = U^{-H} A U^{-1}$ or $L^{-1} A L^{-H}$. $z = U^{-1} y$ or $L^{-H} y$.
- Type 2 ($ABz = \lambda z$): $C = UAU^H$ or $L^H AL$. $z = U^{-1} y$ or $L^{-H} y$.
- Type 3 ($BAz = \lambda z$): $C = UAU^H$ or $L^H AL$. $z = U^H y$ or Ly.

ScaLAPACK Routine: `PxyyGST`

- Overwrites A with the standard matrix C.
- After reduction, standard SEP routines (e.g., `PxSYTRD`) are used on C.

Summary of Computational Routines for GSEP

Task	Single precision		Double precision	
	Real	Complex	Real	Complex
Reduction to standard form				
Compute C from A, B	`PSSYGST`	`PCHEGST`	`PDSYGST`	`PZHEGST`

Note on Eigenvectors: No special routines are needed to recover eigenvectors z from y. These are simple triangular solves or matrix-vector multiplications handled by PBLAS (e.g., `PxTRSV` or `PxTRMM`).

Conclusions

ScaLAPACK provides a comprehensive suite of routines for distributed-memory parallel computation of a wide range of linear algebra problems, including:

- Orthogonal factorizations (QR, LQ, GQR, GRQ).
- Symmetric and nonsymmetric eigenproblems (SEP, NEP, GSEP).
- Singular Value Decomposition (SVD).

The library leverages efficient algorithms and parallelism strategies to ensure scalability and performance on large-scale systems.

These tools permits to implement **high-performance applications** in scientific computing and engineering that require robust linear algebra capabilities, e.g., *model reduction*, *computation of matrix functions*, and *solving large-scale optimization problems*.