SLATE WORKING NOTES

10

SLATE Users' Guide

Mark Gates
Ali Charara
Jakub Kurzak
Asim YarKhan
Mohammed Al Farhan
Dalal Sukkari
Treece Burgess
Neil Lindquist
Jack Dongarra

Innovative Computing Laboratory

November 5, 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**
07-2020	first publication
11-2020	copy editing; update install directions
11-2023	major revision

```
@techreport{gates2020slate,
  author={Gates, Mark and Charara, Ali and Kurzak, Jakub and YarKhan, Asim
          and Al Farhan, Mohammed and Sukkari, Dalal and Burgess, Treece
          and Lindquist, Neil and Dongarra, Jack},
  title={{SLATE} Users' Guide, {SWAN} No. 10},
  institution={Innovative Computing Laboratory, University of Tennessee},
  year={2020},
  month={July},
  number={ICL-UT-19-01},
  note={revision 2023-11},
  url={https://www.icl.utk.edu/publications/swan-010},
}
```

Contents

Contents ii

List of Figures v

List of Algorithms vii

1 Introduction 1

2 Essentials 2
- 2.1 SLATE . . . 2
- 2.2 Functionality and Goals of SLATE . . . 3
- 2.3 Software Components Required by SLATE . . . 3
- 2.4 Computers for Which SLATE Is Suitable . . . 5
- 2.5 Availability of SLATE . . . 5
- 2.6 User Support . . . 5
- 2.7 License and Commercial Use of SLATE . . . 6

3 Installation Instructions 7
- 3.1 Makefile-Based Build . . . 7
- 3.2 CMake . . . 8
- 3.3 Spack . . . 8
- 3.4 Verify Installation . . . 9

4 Getting Started with SLATE 10
- 4.1 Source Code for Example Program 1 . . . 10
- 4.2 Details of Example Program 1 . . . 10
- 4.3 Simplifying Assumptions Used in Example Program 1 . . . 11
- 4.4 Building and Running Example Program 1 . . . 14

5 Design and Fundamentals of SLATE 15
5.1 Design Principles . . . 15
5.1.1 Matrix Layout . . . 16
5.1.2 Parallelism Model . . . 18

6 SLATE API 21
6.1 C++ API . . . 21
6.1.1 BLAS and Auxiliary . . . 21
6.1.2 Linear Systems and Least Squares . . . 22
6.1.3 Unitary Factorizations . . . 23
6.1.4 Eigenvalue and Singular Value Decomposition . . . 24
6.2 C and Fortran API . . . 25
6.2.1 BLAS and Auxiliary . . . 25
6.2.2 Linear Systems and Least Squares . . . 26
6.2.3 Unitary Factorizations . . . 27
6.2.4 Eigenvalue and Singular Value Decomposition (SVD) . . . 27
6.3 Traditional LAPACK and ScaLAPACK API . . . 28

7 Using SLATE 31
7.1 Matrices in SLATE . . . 31
7.1.1 Matrix Hierarchy . . . 31
7.1.2 Creating and Accessing Matrices . . . 34
7.1.3 Matrices from ScaLAPACK . . . 38
7.1.4 Matrix Transpose . . . 38
7.1.5 Submatrices . . . 39
7.1.6 Matrix Slices . . . 39
7.1.7 Deep Matrix Copy . . . 40
7.2 Using SLATE Functions . . . 40
7.2.1 Execution Options . . . 41
7.2.2 Matrix Norms . . . 41
7.2.3 Matrix-Matrix Multiply . . . 42
7.2.4 Operations with Triangular Matrices . . . 43
7.2.5 Operations with Band Matrices . . . 44
7.2.6 Linear Systems: General Non-Symmetric Square Matrices (LU) . . . 44
7.2.7 Linear Systems: Hermitian/Symmetric Positive Definite (Cholesky) . . . 45
7.2.8 Linear Systems: Hermitian/Symmetric Indefinite (Aasen's) . . . 45
7.2.9 Least Squares: $AX \approx B$ Using QR or LQ . . . 46
7.2.10 Mixed-Precision Routines . . . 47
7.2.11 Matrix Inverse . . . 48
7.2.12 Singular Value Decomposition . . . 49
7.2.13 Hermitian/Symmetric Eigenvalues . . . 49
7.2.14 Generalized Hermitian/Symmetric Eigenvalues . . . 50

8 Testing Suite for SLATE 52
8.1 SLATE Tester . . . 53
8.2 Full Testing Suite . . . 55
8.3 Tuning SLATE . . . 56

8.3.1 Enabling Multi-threaded MPI Broadcast 56
8.4 Unit Tests 56

9 Compatibility APIs for ScaLAPACK and LAPACK Users **58**
9.1 LAPACK Compatibility API 58
9.2 ScaLAPACK Compatibility API 59

Bibliography **61**

List of Figures

2.1 Software layers in SLATE. 4

5.1 SLATE Software Stack. 15
5.2 General, symmetric, band, and symmetric band matrices. Only shaded tiles are stored; blank tiles are implicitly zero or known by symmetry, so are not stored. 16
5.3 View of symmetric matrix on process (0, 0) in 2×2 process grid. Darker blue tiles are local to process (0, 0); lighter yellow tiles are temporary workspace tiles copied from remote process (0, 1). 17
5.4 Broadcast of tile and its symmetric image to nodes owning a block row and block column in a symmetric matrix. 18
5.5 Block sizes can vary. Most algorithms require square diagonal tiles. 18
5.6 Tasks in Cholesky factorization. Arrows depict dependencies. 19

7.1 Matrix hierarchy in SLATE. Algorithms require the appropriate types for their operation. 33
7.2 Matrix layout of ScaLAPACK (left) and layout with contiguous tiles (right). SLATE matrix and tiles structures are flexible and accommodate multiple layouts. 38

8.1 Performance comparison with using listBcastMT. 57

List of Algorithms

4.1 LU solve: slate_lu.cc (1 of 3) 11
4.2 LU solve: slate_lu.cc (2 of 3) 12
4.3 LU solve: slate_lu.cc (3 of 3) 13
7.1 Conversions: ex02_conversion.cc 33
7.2 Creating matrices: ex01_matrix.cc 34
7.3 SLATE allocating CPU host memory for a matrix: ex01_matrix.cc 35
7.4 SLATE allocating GPU device memory for a matrix: ex01_matrix.cc 35
7.5 Inserting tiles using user-defined data: ex01_matrix.cc 35
7.6 Accessing tile elements: ex01_matrix.cc 37
7.7 Accessing tile elements, currently more efficient implementation: ex01_matrix.cc . . 37
7.8 Creating matrix from ScaLAPACK-style data: ex01_matrix.cc 38
7.9 Transposing matrices: ex01_matrix.cc 39
7.10 Sub-matrices: ex03_submatrix.cc 39
7.11 Matrix slice: ex03_submatrix.cc 40
7.12 Deep matrix copy:: ex01_matrix.cc 40
7.13 Options. 41
7.14 Passing options to multiply (gemm): ex05_blas.cc 41
7.15 Norms: ex04_norm.cc 42
7.16 Parallel matrix multiply: ex05_blas.cc 42
7.17 Parallel rank k and $2k$ updates: ex05_blas.cc 43
7.18 Parallel triangular multiply and solve: ex05_blas.cc 43
7.19 Band operations. 44
7.20 LU solve: ex06_linear_system_lu.cc 44
7.21 Cholesky solve: ex07_linear_system_cholesky.cc 45
7.22 Indefinite solve: ex08_linear_system_indefinite.cc 46
7.23 Least squares (overdetermined): ex09_least_squares.cc 47
7.24 Least squares (underdetermined): ex09_least_squares.cc 47
7.25 Mixed precision LU solve. ex06_linear_system_lu.cc 48
7.26 Mixed precision Cholesky solve. ex07_linear_system_cholesky.cc 48
7.27 LU inverse: ex06_linear_system_lu.cc 48

7.28 Cholesky inverse: ex07_linear_system_cholesky.cc 49
7.29 SVD: ex10_svd.cc 49
7.30 Hermitian/symmetric eigenvalues: ex11_hermitian_eig.cc 50
7.31 Generalized Hermitian/symmetric eigenvalues: ex12_generalized_hermitian_eig.cc . 51
9.1 LAPACK-compatible API. 59
9.2 ScaLAPACK-compatible API. 60

CHAPTER 1

Introduction

SLATE (Software for Linear Algebra Targeting Exascale)[1] has been 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). The objective of SLATE is to provide fundamental dense linear algebra capabilities to the U.S. Department of Energy and to the high-performance computing (HPC) community at large.

This *SLATE Users' Guide* is intended for application end users and focuses on the public SLATE application programming interface (API). The companion *SLATE Developers' Guide* [1] is intended to describe the internal workings of SLATE, to be of use for SLATE developers and contributors. These guides will be periodically revised as SLATE develops, with the revisions noted in the front matter notes and BibTeX.

[1] http://icl.utk.edu/slate/
[2] https://www.exascaleproject.org

CHAPTER 2

Essentials

2.1 SLATE

SLATE is a library providing dense linear algebra capabilities for high-performance systems supporting large-scale distributed-nodes with accelerators. SLATE provides coverage of existing ScaLAPACK functionality, including parallel Basic Linear Algebra Subprograms (BLAS), linear systems using LU and Cholesky, least squares problems using QR, eigenvalue problems, and the singular value decomposition (SVD). SLATE is designed to be a replacement for ScaLAPACK, which after two decades of operation cannot be adequately retrofitted for modern accelerated architectures. SLATE also seeks to deliver dense linear algebra capabilities beyond the capabilities of ScaLAPACK, including new features such as mixed-precision iterative refinement, threshold pivoting, the polar decomposition, communication-avoiding and randomized algorithms, as well as the potential to support variable size tiles and block low-rank compressed tiles. SLATE uses modern techniques such as communication-avoiding algorithms, lookahead panels to overlap communication and computation, task-based scheduling, and a modern C++ framework.

The SLATE project website is located at:
https://icl.utk.edu/slate/

The SLATE software can be downloaded from:
https://github.com/icl-utk-edu/slate/

The SLATE auto-generated function reference can be found at:
https://icl.bitbucket.io/slate/

2.2 Functionality and Goals of SLATE

SLATE operates on dense matrices, solving systems of linear equations, linear least squares problems, eigenvalue problems, and singular value problems. SLATE also handles many associated computations such as matrix factorizations and matrix norms. SLATE routines also support distributed parallel band factorization, band solve, and band BLAS.

SLATE is intended to fulfill the following design goals:

Target modern HPC hardware consisting of a large number of nodes with multi-core processors and several hardware accelerators per node.

Achieve portable high performance by relying on vendor optimized standard BLAS, batched BLAS, LAPACK, and standard parallel programming technologies such as MPI and OpenMP. Using the OpenMP runtime puts less of a burden on applications to integrate SLATE than adopting a proprietary runtime would.

Provide scalability by employing proven techniques in dense linear algebra, such as 2D block-cyclic data distribution and communication-avoiding algorithms, as well as modern parallel programming approaches, such as dynamic scheduling and communication overlapping.

Facilitate productivity by relying on the intuitive *single program, multiple data* (SPMD) programming model and a set of simple abstractions to represent dense matrices and dense matrix operations.

Assure maintainability by employing useful C++ facilities, such as templates and overloading of functions and operators, with a focus on minimizing code.

Ease transition to SLATE by natively supporting the ScaLAPACK 2D block-cyclic layout and providing a backwards-compatible ScaLAPACK API.

SLATE uses a modern testing framework, *TestSweeper* [1], which can exercise much of the functionality provided by the library. This framework sweeps over an input space of parameters to check valid combinations of parameters when calling SLATE routines.

2.3 Software Components Required by SLATE

SLATE builds on top of a small number of component packages, as depicted in Figure 2.1. The *BLAS++* library provides overloaded C++ wrappers around the Fortran BLAS routines, exposing a single function interface to a routine independent of the datatype of the operands. BLAS++ provides both column-major and row-major access to matrices with no-to-minimal performance overhead. The BLAS++ library provides Level 1, 2, and 3 BLAS on the CPU, and Level 1 and 3 BLAS on GPUs via CUDA's cuBLAS, AMD's rocBLAS, or Intel's oneMKL. (Level 2 BLAS can be added on GPUs as needed; please request by filing an issue on GitHub.) Where available, Level 3 Batched BLAS routines are provided on the CPU and GPU as well.

[1] https://github.com/icl-utk-edu/testsweeper

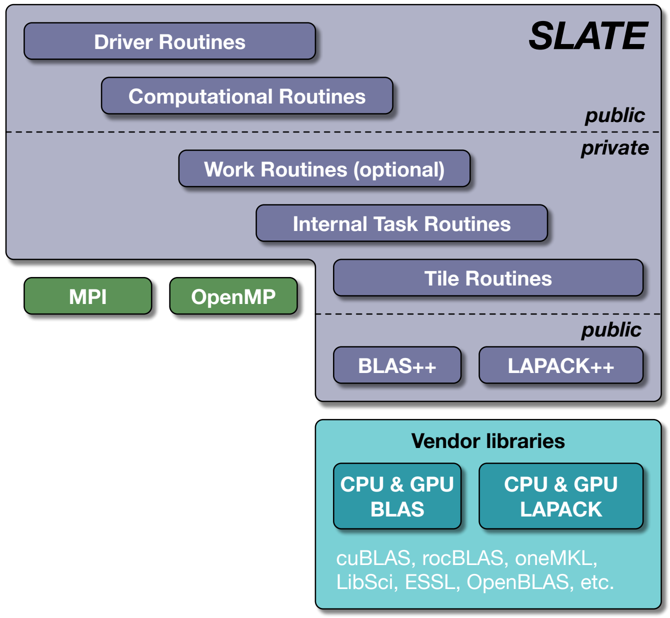


Figure 2.1: Software layers in SLATE.

The *LAPACK++* library provides similar datatype-independent overloaded C++ wrappers around the Fortran LAPACK routines. We are starting to add LAPACK-like GPU routines such as LU (`getrf`), Cholesky (`potrf`), QR (`geqrf`), and eigenvalues (`heevd`). Please file an issue on GitHub to request additional routines to be added. Note that LAPACK++ provides support only for column-major access to matrices. The LAPACK++ wrappers allocate optimal workspace sizes as needed by the routines so that the user is not required to allocate workspace in advance.

Within a process, multi-threading in SLATE is obtained using OpenMP constructs. More specifically, OpenMP task-depend clauses are used to specify high-level dependencies in SLATE algorithms and OpenMP task-based parallelism is used to distribute data parallel work to the processors.

Efficient use of GPUs and accelerators is obtained by using the Batched BLAS API. Batched BLAS is an emerging standard technique for aggregating many small, independent BLAS operations to efficiently use the hardware and obtain higher performance.

MPI is used for communication between processes in SLATE. If GPU-aware MPI is available, SLATE can take advantage of it to send data directly between GPUs. To enable GPU-aware MPI, set the environment variable:

```
export SLATE_GPU_AWARE_MPI=1
```

The job scheduler or MPI library may also need a flag set to enable GPU-aware MPI; see your HPC center's documentation. For Cray MPI as on Frontier and Perlmutter, set:

```
export MPICH_GPU_SUPPORT_ENABLED=1
```

2.4 Computers for Which SLATE Is Suitable

SLATE is primarily designed to solve dense linear algebra problems on large, distributed-memory machines where the primary compute power in each node may be in GPUs or accelerators. Nodes are expected to have close to identical hardware, with the same kind and number of CPUs and GPUs in each node for good load balancing. SLATE is also expected to run well on single-node, multi-processor machines, with or without accelerators. However, there are linear algebra libraries (e.g., from vendors) that are more closely focused on single-node machines. For single-node machines with accelerators, the MAGMA library[2] is also well suited.

2.5 Availability of SLATE

SLATE is available and distributed as C++ source code, and is intended to be readily compiled from source. Releases can be downloaded from the SLATE source repository:
https://github.com/icl-utk-edu/slate/releases
or cloned using git:

```
> git clone --recursive https://github.com/icl-utk-edu/slate.git
```

It has both Makefile (Section 3.1) and CMake (Section 3.2) build options. SLATE can also be downloaded and installed using the Spack scientific software package manager (Section 3.3).

Some HPC centers make SLATE available as an environment module to load using, e.g., `module load slate`. Check with your HPC support desk.

Papers and documentation for SLATE can be found on the SLATE website.
https://icl.utk.edu/slate/

2.6 User Support

General support for SLATE can be obtained by visiting the *SLATE User Forum* at https://groups.google.com/a/icl.utk.edu/g/slate-user. Join by signing in with your Google credentials, then clicking *Join group to post*. Messages can be posted online or by emailing slate-user@icl.utk.edu

Bug reports and issues should be filed on the SLATE, BLAS++, or LAPACK++ Issues trackers that can be found at their repositories:
https://github.com/icl-utk-edu/slate/issues/
https://github.com/icl-utk-edu/blaspp/issues/
https://github.com/icl-utk-edu/lapackpp/issues/

[2] http://icl.utk.edu/magma/

2.7 License and Commercial Use of SLATE

SLATE is licensed under the 3-clause BSD open-source software license. This means that SLATE can be included in commercial packages. The SLATE team asks only that proper credit be given to the authors.

Like all software, SLATE is copyrighted. It is not trademarked. However, if modifications are made that affect the interface, functionality, or accuracy of the resulting software, we request that the name or options of the routine be changed. Any modification to SLATE software should be noted in the modifier's documentation.

The SLATE team will gladly answer questions regarding this software. If modifications are made to the software, however, it is the responsibility of the individual or institution who modified the routine to provide support.

The SLATE software license is included here:

CHAPTER 3

Installation Instructions

SLATE requires BLAS and LAPACK library support for core mathematical routines. SLATE requires OpenMP 4.5 or better, and MPI with thread support, specifically `MPI_THREAD_MULTIPLE`. Currently, SLATE uses CUDA, ROCm, or SYCL for acceleration on NVIDIA, AMD, and Intel GPUs, respectively. SLATE can be built without GPU support, to run on only CPUs.

The SLATE source code has several methods that can be used to build and install the library: GNUmakefile, CMake, or Spack.

For Makefile or CMake builds, first download the source. Note that the source has several submodules and these need to be downloaded recursively.

```
# Use one of the following: https access
> git clone --recursive https://github.com/icl-utk-edu/slate.git
# or ssh access
> git clone --recursive git@github.com:icl-utk-edu/slate.git

> cd slate
```

3.1 Makefile-Based Build

The GNUmakefile build uses GNU-specific extensions and expects the user to provide some configuration options to the build process. These build configuration options may change as the build process is improved and modified. Please see INSTALL.md for the most current options.

The GNUmake build process expects to find the compilers, include files, and libraries for MPI, CUDA, ROCm, SYCL, BLAS, LAPACK, and ScaLAPACK via search path variables. The Makefile will auto-detect CUDA or ROCm if `nvcc` or `hipcc`, respectively, are in your `PATH`. The locations of header files to be included can be provided by extending the CPATH or CXXFLAGS environment variables. The location of libraries can be provided in LIBRARY_PATH, LDFLAGS, or LIBS.

With Makefile, creating a make.inc file with the necessary options is recommended, to ensure the same options are used by all make commands. It is recommended to use the MPI compiler wrappers such as mpicxx and mpif90. For instance, the following minimal make.inc file will produce a SLATE library that uses OpenBLAS; has CUDA, ROCm, or SYCL support if available; and is dynamically linked (default).

```
# make.inc
CXX  = mpicxx
FC   = mpif90
blas = openblas
```

SLATE is then compiled and installed using:

```
> make lib                        # build libraries
> make tester                     # build tester
> make check                      # run sanity check tests
> make install prefix=/usr/local  # install in /usr/local/{lib,include}
```

3.2 CMake

The CMake build has similar configuration options to the Makefile-based build, with some CMake-specific options. Please see INSTALL.md for the most current options.

With CMake, create a build directory and specify options to cmake using its -D variable=value syntax. It is recommended to set the CXX and FC environment variables to the desired C++ and Fortran compilers; CMake takes the plain C++ and Fortran compilers, not the MPI compiler wrappers. For instance, the following minimal configuration will produce a SLATE library that uses OpenBLAS; has CUDA, ROCm, or SYCL support if available; and is dynamically linked (default).

```
> mkdir build
> cd build
> export CXX=g++
> export FC=gfortran
> cmake -D blas=openblas -D CMAKE_INSTALL_PREFIX=/usr/local ..
> make lib        # build libraries
> make tester     # build tester
> make check      # run sanity check tests
> make install    # install in /usr/local/{lib,include}
```

3.3 Spack

Spack is a package manager for HPC software targeting supercomputers, Linux, and macOS. The following set of commands will install Spack in your directory and then install SLATE with all required dependencies. If Spack is already installed, use the local installation to install SLATE. Spack has many configuration options (which compiler to use, which libraries, etc.); make sure to use your desired setup. Here are some examples.

```
> git clone https://github.com/spack/spack.git
> source spack/share/spack/setup-env.sh
> spack compiler find
> spack info slate                              # see available options
> spack install slate                           # with defaults
> spack install slate %gcc@11.3.1               # compile with gcc 11.3.1
> spack install slate %gcc@11.3.1 ^openblas     # with OpenBLAS
> spack install slate %gcc@11.3.1 ^openblas ~cuda  # without CUDA
```

See the Spack Getting Started Guide for more information.

3.4 Verify Installation

Run a basic gemm tester on 4 distributed nodes using your local job launcher. This will produce output that indicates whether the test passed. Further explanation of SLATE's tester is in Chapter 8.

```
> export OMP_NUM_THREADS=8

# Using command line mpirun, with 4 MPI processes of 8 threads each.
> mpirun -n 4 ./test/tester gemm

# Using Slurm job manager, with 16 tasks (MPI processes) of 8 threads each on 4 nodes.
> srun --nodes=4 --ntasks=16 --cpus-per-task=${OMP_NUM_THREADS} ./test/tester gemm
```

CHAPTER 4

Getting Started with SLATE

4.1 Source Code for Example Program 1

The following example program will set up SLATE matrices and solve a linear system $AX = B$ using the SLATE LU solver by calling a distributed `lu_solve` implementation. This is also known as `gesv`, for *general matrix solve*, in the traditional LAPACK naming scheme.

4.2 Details of Example Program 1

The example program in Algorithms 4.1 to 4.3 shows how to set up matrices in SLATE and to call several SLATE routines to operate on those matrices. This example uses the SLATE LU solver `lu_solve` to solve a system of linear equations $AX = B$. In this example, the scalar data type for the coefficient matrix A and the right-hand side (RHS) matrix B are double-precision real numbers. However, this computation could have been instantiated for a number of different data types. The matrices are partitioned into $nb \times nb$ blocks, which are distributed over the $p \times q$ grid of processes. The default distribution is a 2D block-cyclic distribution of the blocks, similar to that of ScaLAPACK, although the distribution can be changed within SLATE.

After the A and B matrices are defined at line 60, the required local data space is allocated on each process by SLATE (line 64), and this local memory is then initialized with random values (line 70). Copies of the matrices are retained to do an error check after the solve (line 78).

After the call to SLATE's `lu_solve` at line 89, the distributed B matrix will contain the solution

Algorithm 4.1 LU solve: slate_lu.cc (1 of 3)

```
1  #include <slate/slate.hh>
2  #include <blas.hh>
3  #include <mpi.h>
4  #include <stdio.h>
5
6  // Forward function declarations
7  template <typename scalar_type>
8  void lu_example( int64_t n, int64_t nrhs, int64_t nb, int p, int q );
9
10 template <typename matrix_type>
11 void random_matrix( matrix_type& A );
12
13 int main( int argc, char** argv )
14 {
15     // Initialize MPI, requiring MPI_THREAD_MULTIPLE support.
16     int err=0, mpi_provided=0;
17     err = MPI_Init_thread( &argc, &argv, MPI_THREAD_MULTIPLE, &mpi_provided );
18     if (err != 0 || mpi_provided != MPI_THREAD_MULTIPLE) {
19         throw std::runtime_error( "MPI_Init failed" );
20     }
21
22     // Call the LU example.
23     int64_t n=5000, nrhs=1, nb=256, p=2, q=2;
24     lu_example<double>( n, nrhs, nb, p, q );
25
26     err = MPI_Finalize();
27     if (err != 0) {
28         throw std::runtime_error( "MPI_Finalize failed" );
29     }
30     return 0;
31 }
```

matrix X. A residual check is performed at line line 92 to verify the accuracy of the results:

$$\frac{\|AX - B\|_1}{\|X\|_1 \cdot \|A\|_1 \cdot n} < \epsilon.$$

The call to `lu_solve` uses an optional parameter (`opts`) at line 85 to set the execution target to running tasks on the host CPU `HostTask`. If SLATE is compiled for GPU devices, the execution target can be set to `Devices`. The `opts` can also be used to set a number of internal variables and is used here to give an example of how to pass options to SLATE.

4.3 Simplifying Assumptions Used in Example Program 1

Several assumptions and choices have been made in the Example Program 1:

- Choice of `nb=256`: The tile size `nb` should be tuned for the execution target.
- Choice of `p=2, q=2`: The $p \times q$ process grid was set to a square grid; however, other process grids may perform better depending on the number of processes and the problem size.
- Data distribution: The default data distribution in SLATE is 2D block-cyclic on CPUs.

Algorithm 4.2 LU solve: slate_lu.cc (2 of 3)

```
// Create matrices, call LU solver, and check result.
template <typename scalar_t>
void lu_example( int64_t n, int64_t nrhs, int64_t nb, int p, int q )
{
    // Get associated real type, e.g., double for complex<double>.
    using real_t = blas::real_type<scalar_t>;
    using llong = long long;  // guaranteed >= 64 bits
    const scalar_t one = 1;
    int err=0, mpi_size=0, mpi_rank=0;

    // Get MPI size. Must be >= p*q for this example.
    err = MPI_Comm_size( MPI_COMM_WORLD, &mpi_size );
    if (err != 0) {
        throw std::runtime_error( "MPI_Comm_size failed" );
    }
    if (mpi_size < p*q) {
        printf( "Usage: mpirun -np %d ... # %d ranks hard coded\n",
                p*q, p*q );
        return;
    }

    // Get MPI rank
    err = MPI_Comm_rank( MPI_COMM_WORLD, &mpi_rank );
    if (err != 0) {
        throw std::runtime_error( "MPI_Comm_rank failed" );
    }

    // Create SLATE matrices A and B.
    slate::Matrix<scalar_t> A( n, n,    nb, p, q, MPI_COMM_WORLD );
    slate::Matrix<scalar_t> B( n, nrhs, nb, p, q, MPI_COMM_WORLD );

    // Allocate local space for A, B on distributed nodes.
    A.insertLocalTiles();
    B.insertLocalTiles();

    // Set random seed so data is different on each MPI rank.
    srand( 100 * mpi_rank );
    // Initialize the data for A, B.
    random_matrix( A );
    random_matrix( B );

    // For residual error check,
    // create A0 as an empty matrix like A and copy A to A0.
    slate::Matrix<scalar_t> A0 = A.emptyLike();
    A0.insertLocalTiles();
    slate::copy( A, A0 );
    // Create B0 as an empty matrix like B and copy B to B0.
    slate::Matrix<scalar_t> B0 = B.emptyLike();
    B0.insertLocalTiles();
    slate::copy( B, B0 );
```

Algorithm 4.3 LU solve: slate_lu.cc (3 of 3)

```
    // Call the SLATE LU solver.
    slate::Options opts = {
        {slate::Option::Target, slate::Target::HostTask}
    };
    double time = omp_get_wtime();
    slate::lu_solve( A, B, opts );
    time = omp_get_wtime() - time;

    // Compute residual ||A0 * X  - B0|| / ( ||X|| * ||A0|| * n )
    real_t A_norm = slate::norm( slate::Norm::One, A0 );
    real_t X_norm = slate::norm( slate::Norm::One, B );
    slate::gemm( -one, A0, B, one, B0 );
    real_t R_norm = slate::norm( slate::Norm::One, B0 );
    real_t residual = R_norm / (X_norm * A_norm * n);
    real_t tol = std::numeric_limits<real_t>::epsilon();
    bool status_ok = (residual < tol);

    if (mpi_rank == 0) {
        printf( "lu_solve n %lld, nb %lld, p-by-q %lld-by-%lld, "
                "residual %.2e, tol %.2e, time %.2e sec, %s\n",
                llong( n ), llong( nb ), llong( p ), llong( q ),
                residual, tol, time,
                status_ok ? "pass" : "FAILED" );
    }
}

// Put random data in matrix A.
// todo: replace with:
//      auto rand_entry = []( int64_t i, int64_t j ) {
//          return 1.0 - rand() / double( RAND_MAX );
//      }
//      set( rand_entry, A );
template <typename matrix_type>
void random_matrix( matrix_type& A )
{
    // For each tile in the matrix
    for (int64_t j = 0; j < A.nt(); ++j) {
        for (int64_t i = 0; i < A.mt(); ++i) {
            if (A.tileIsLocal( i, j )) {
                // set data values in the local tile.
                auto tile = A( i, j );
                auto tiledata = tile.data();
                for (int64_t jj = 0; jj < tile.nb(); ++jj) {
                    for (int64_t ii = 0; ii < tile.mb(); ++ii) {
                        tiledata[ ii + jj*tile.stride() ]
                            = 1.0 - (rand() / double(RAND_MAX));
                    }
                }
            }
        }
    }
}
```

SLATE can specify other data distributions by using C++ lambda functions or functions from the `slate::func` namespace when defining the matrix.

- Execution target `slate::Target::HostTask`: The execution will happen on the host using OpenMP tasks. Other execution targets like `Device` for GPU accelerator devices may be preferable. Currently the default target is `HostTask`, but that may change to `Auto`, which would automatically select `Devices` if a GPU accelerator is available, otherwise `HostTask`.

4.4 Building and Running Example Program 1

The code in Example 1 requires SLATE, BLAS++, LAPACK++, and optionally CUDA or ROCm header files. The paths to SLATE, BLAS++, LAPACK++, and optionally CUDA or ROCm libraries are also needed. Using `-rpath` avoids the need to add these library paths to `LD_LIBRARY_PATH`. The shell commands below set up these paths. Note that SLATE typically requires `-fopenmp` to be used when compiling and linking applications.

```
# Locations of SLATE, BLAS++, LAPACK++ install or build directories.
export SLATE_ROOT=/path/to/slate
export BLASPP_ROOT=${SLATE_ROOT}/blaspp      # or ${SLATE_ROOT}, if installed
export LAPACKPP_ROOT=${SLATE_ROOT}/lapackpp  # or ${SLATE_ROOT}, if installed
# export CUDA_HOME=/usr/local/cuda           # wherever CUDA is installed
# export ROCM_PATH=/opt/rocm                 # wherever ROCm is installed

# Compile the example.
mpicxx -fopenmp -c slate_lu.cc
       -I${SLATE_ROOT}/include \
       -I${BLASPP_ROOT}/include \
       -I${LAPACKPP_ROOT}/include
       # -I${CUDA_HOME}/include         # For CUDA
       # -I${ROCM_PATH}/include         # For ROCm

mpicxx -fopenmp -o slate_lu slate_lu.o \
       -L${SLATE_ROOT}/lib    -Wl,-rpath,${SLATE_ROOT}/lib \
       -L${BLASPP_ROOT}/lib   -Wl,-rpath,${BLASPP_ROOT}/lib \
       -L${LAPACKPP_ROOT}/lib -Wl,-rpath,${LAPACKPP_ROOT}/lib \
       -lslate -llapackpp -lblaspp

       # For CUDA, may need to add:
       # -L${CUDA_HOME}/lib64 -Wl,-rpath,${CUDA_HOME}/lib64 \
       # -lcusolver -lcublas -lcudart

       # For ROCm, may need to add:
       # -L${ROCM_PATH}/lib -Wl,-rpath,${ROCM_PATH}/lib \
       # -lrocsolver -lrocblas -lamdhip64

# Run the slate_lu executable.
mpirun -n 4 ./slate_lu

# Output from the run will be something like the following:
# lu_solve n 5000, nb 256, p-by-q 2-by-2, residual 8.41e-20, tol 2.22e-16, time 7.65e-01 sec,
#   pass
#
```

CHAPTER 5

Design and Fundamentals of SLATE

5.1 Design Principles

Figure 5.1 shows the SLATE software stack, designed after a careful consideration of available implementation technologies [2]. The objective of SLATE is to provide dense linear algebra capabilities to the ECP applications (e.g., EXAALT, NWChemEx, QMCPACK, WarpX) as well as other software libraries and frameworks (e.g., STRUMPACK), and the HPC community at large. In that regard, SLATE is intended to be a replacement for ScaLAPACK, with superior performance and scalability in distributed-memory environments with multi-core processors and hardware accelerators.

Figure 5.1: SLATE Software Stack.

The SLATE project also encompasses the design and implementation of the BLAS++ and LAPACK++ C++ APIs [3], providing a portability layer for both CPU and GPU BLAS and LAPACK, including Batched BLAS. Underneath these APIs, highly optimized vendor libraries

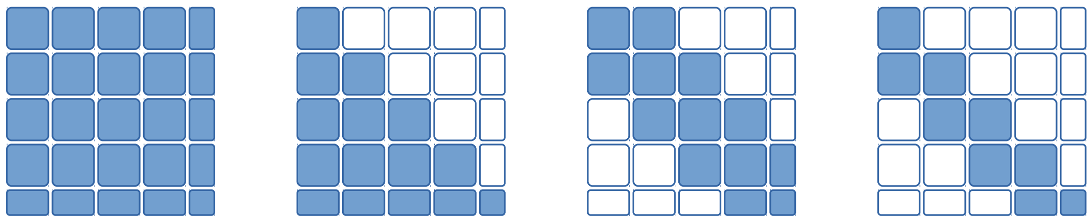


Figure 5.2: General, symmetric, band, and symmetric band matrices. Only shaded tiles are stored; blank tiles are implicitly zero or known by symmetry, so are not stored.

will be called for maximum performance (Intel oneMKL, IBM ESSL, Cray LibSci, OpenBLAS, NVIDIA cuBLAS, AMD rocBLAS, etc.).

To maximize portability, the design relies on the MPI standard for message passing and the OpenMP standard for multithreading and offload to hardware accelerators.

5.1.1 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 dense linear algebra software such as ScaLAPACK, Elemental, and PLASMA, where the local matrix occupies a contiguous memory region on each process. While PLASMA uses tiled algorithms, the tiles are stored in one contiguous memory block. In contrast, SLATE makes tiles first-class objects 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. Furthermore, SLATE supports tiles pointing to data in a traditional ScaLAPACK matrix layout, easing an application's transition from ScaLAPACK to SLATE. A similar strategy of allocating tiles individually has been successfully used in low-rank, data-sparse linear algebra libraries, such as hierarchical matrices [4, 5] in HLib [6] and with the block low-rank (BLR) format [7]. Compared to other distributed dense linear algebra formats, SLATE's matrix structure offers numerous advantages, outlined below.

First, the same structure can be used for holding many different matrix types: general, symmetric, triangular, band, symmetric band, etc., as shown in Figure 5.2. Little memory is wasted for storing parts of the matrix that hold no useful data, such as the upper triangle of a lower triangular matrix. Instead of wasting $\sim \frac{1}{2}n^2$ memory as ScaLAPACK does, only $\sim \frac{1}{2}n n_b$ memory is unused in the diagonal tiles for a block size n_b; all unused off-diagonal tiles are simply never allocated. There is no need for using complex matrix layouts—such as the *Recursive Packed Format* (RPF) [8] or *Rectangular Full Packed* (RFP) [9]—in order to save space.

Second, the matrix can be easily converted, in parallel, from one layout to another with $O(P)$ memory overhead for P processors (cores/threads). Possible conversions include changing tile layout from column-major to row-major, "packing" of tiles for efficient BLAS execution [10], and low-rank compression of tiles. Notably, transposition of the matrix can be accomplished by transposition of each tile and remapping of the indices. There is no need for complex in-place

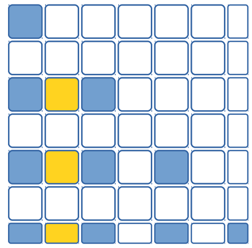


Figure 5.3: View of symmetric matrix on process (0, 0) in 2 × 2 process grid. Darker blue tiles are local to process (0, 0); lighter yellow tiles are temporary workspace tiles copied from remote process (0, 1).

layout translation and transposition algorithms [11].

Moreover, tiles can be easily allocated and copied among different memory spaces. Both inter-node and intra-node communication are vastly simplified. Tiles can be easily and efficiently transferred between nodes using MPI. Tiles can be easily moved in and out of fast memory, such as the MCDRAM in Xeon Phi processors. Tiles can also be copied to one or more device memories in the case of GPU acceleration.

In practical terms, a SLATE matrix is implemented using the `std::map` container from the C++ standard library as:

```
std::map< std::tuple< int64_t, int64_t >,
          TileNode<scalar_t>* >
```

The map's key is a tuple consisting of the tile's (i, j) block row and column indices in the matrix. SLATE relies on global indexing of tiles, meaning that each tile is identified by the same unique tuple across all processes. The map's value is a TileNode object that stores tiles on each device (host or accelerator), and is indexed by the device number where the tile is located. The tile itself is a lightweight object that stores a tile's data and properties (dimensions, uplo, etc.).

In addition to facilitating the storage of different types of matrices, this structure also readily accommodates partitioning of the matrix to the nodes of a distributed-memory system. Each node stores only its local subset of tiles, as shown in Figure 5.3. Mapping of tiles to nodes is defined by a C++ lambda function, and set to 2D block cyclic mapping by default, but the user can supply an arbitrary mapping function. Similarly, distribution to accelerators within each node is 1D block cyclic by default, but the user can substitute an arbitrary function.

Remote access is realized by replicating remote tiles in the local matrix for the duration of the operation. This is shown in Figure 5.3 for the trailing matrix update in Cholesky, where portions of the remote panel (yellow) have been copied locally.

Communication in SLATE relies on explicit dataflow information. When tiles are needed for computation, they are broadcast to all the processes where they are required. Figure 5.4 shows a single tile being broadcast from the Cholesky panel to a block row and block column for the trailing matrix update. The broadcast is expressed in terms of the tiles to be updated, which are internally mapped by SLATE to explicit MPI ranks as destinations.

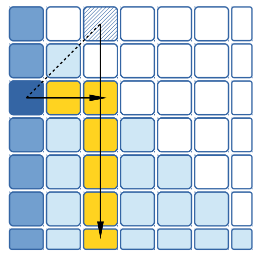


Figure 5.4: Broadcast of tile and its symmetric image to nodes owning a block row and block column in a symmetric matrix.

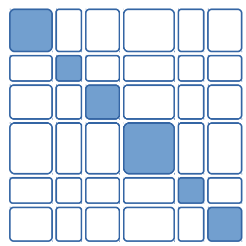


Figure 5.5: Block sizes can vary. Most algorithms require square diagonal tiles.

Finally, SLATE can support non-uniform tile sizes (Figure 5.5). Most factorizations require that the diagonal tiles be square, but the block row heights and block column widths can, in principle, be arbitrary. Non-uniform tile sizes does, however, complicate using the Batched BLAS, which work on tiles that are a fixed size. Due to this, most SLATE algorithms currently require fixed tile size for GPU Device execution, while CPU HostTask execution supports non-uniform tile sizes in most cases. We are currently working to resolve this issue for GPUs; see PRs with regions. Non-uniform tile sizes is useful in applications where the block structure is significant, for instance in *Adaptive Cross Approximation* (ACA) linear solvers [12].

5.1.2 Parallelism Model

SLATE utilizes up to four levels of parallelism: distributed parallelism between nodes using MPI, explicit thread parallelism using OpenMP, implicit thread parallelism within the vendor's node-level BLAS, and, at the lowest level, vector parallelism for the processor's single instruction, multiple data (SIMD) vector instructions. For multi-core CPUs, SLATE typically uses all the threads explicitly, and uses the vendor's BLAS in sequential mode. For GPU accelerators, SLATE uses a batched BLAS call, utilizing the thread block parallelism built into the accelerator's BLAS.

The cornerstones of SLATE are (1) the single program, multiple data (SPMD) programming model for productivity and maintainability, (2) dynamic task scheduling using OpenMP for maximum node-level parallelism and portability, (3) the *lookahead* technique for prioritizing the *critical path*, (4) primary reliance on the 2D block cyclic distribution for scalability, and (5) reliance on the `gemm` operation, specifically its batched rendition, for maximum hardware utilization.

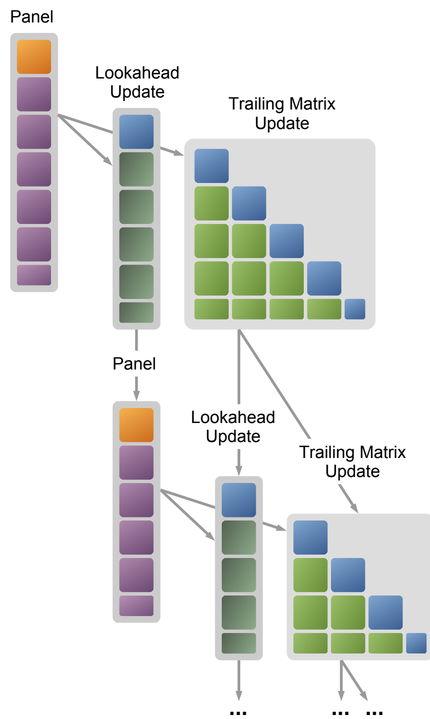


Figure 5.6: Tasks in Cholesky factorization. Arrows depict dependencies.

Cholesky factorization demonstrates the basic framework, with its task graph shown in Figure 5.6. Dataflow tasking (using `omp task depend`), is used for scheduling operations with dependencies on large blocks of the matrix. Within each large block, either nested tasking (forking multiple `omp task`), or batched operations of independent tile operations are used for scheduling individual tile operations to individual cores, without dependencies. For accelerators, batched BLAS calls are used for fast processing of large blocks of the matrix.

Compared to pure tile-by-tile dataflow scheduling—as is used by DPLASMA and Chameleon—this approach minimizes the size of the task graph and number of dependencies to track. For a matrix of $N \times N$ tiles, tile-by-tile scheduling creates $O(N^3)$ tasks and dependencies, which can lead to significant scheduling overheads. This is one of the main performance handicaps of the OpenMP version of the PLASMA library [13] for many-core processors such as the Xeon Phi family. In contrast, SLATE's approach creates $O(N)$ dependencies, eliminating the issue of scheduling overheads. At the same time, this approach is a necessity for scheduling a large set of independent tasks to accelerators to fully occupy their massive compute resources. It also eliminates the need to use a hierarchical task graph to satisfy the vastly different levels of parallelism on CPUs versus on accelerators [14].

At each step of Cholesky, one or more columns of the trailing submatrix are prioritized for processing, using the OpenMP `priority` clause, to facilitate faster advance along the critical path, implementing a lookahead. At the same time, the lookahead depth needs to be limited, as the amount of extra memory required for storing temporary tiles is proportional to the lookahead. Deep lookahead translates to depth-first processing of the task graph, synonymous with left-looking algorithms, but can also lead to catastrophic memory overheads in distributed-memory environments [15].

Distributed-memory computing is implemented by filtering operations based on the matrix distribution function; in most cases, the owner of the output tile performs the computation to update the tile. Appropriate communication calls are issued to send tiles to where the computation will occur. Management of multiple accelerators is handled by a node-level memory consistency protocol.

The user can choose among various target implementations. In the case of accelerated execution, the updates are executed as calls to `batched gemm` (`Target::Devices`). In the case of multi-core execution, the updates can be executed as:

- a set of OpenMP tasks (`Target::HostTask`),
- a nested parallel for loop (`Target::HostNest`), or
- a call to batched `gemm` (`Target::HostBatch`).

For CPUs, `HostTask` is the most thoroughly implemented target; `HostNest` and `HostBatch` are not implemented for all algorithms and are less tested.

SLATE intentionally relies on standards in MPI, OpenMP, and BLAS to maintain easy portability. Any CPU platform with good implementations of these standards should work well for SLATE. For accelerators, SLATE's reliance on batched gemm means any platform that implements batched gemm is a good target. Differences between vendors' BLAS implementations are abstracted at a low level in the BLAS++ library to ease porting. There are few accelerator (e.g., CUDA, HIP, SYCL) kernels in SLATE—currently just matrix norms, add, copy, set, scale, and transpose—and they are relatively simple, Level 2 BLAS-type operations, so porting to a new architecture has proven to be a lightweight task.

CHAPTER 6

SLATE API

SLATE provides two naming schemes. The first is based on the traditional BLAS and LAPACK naming scheme, which is explained in Section 6.3, with routines like `trsm` and `gesv`. Given the cryptic nature of the traditional BLAS and LAPACK names, and that SLATE can identify the matrix type from the matrix classes in arguments, we also provide a second, simplified C++ API, with routine names that are spelled out, such as `triangular_solve` and `lu_solve`. Detailed information about each routine is provided in the online routine reference at https://icl.bitbucket.io/slate/.

To make SLATE accessible from C and Fortran, we also provide C and Fortran 2003 APIs, given in Section 6.2.

To aid transition for existing codes, we provide a compatibility layer using ScaLAPACK routines that requires no application source code changes, and an LAPACK-like API that requires minimal source code changes, adding a `slate_` prefix, as described in Chapter 9.

6.1 C++ API

All routines here are in the `slate::` namespace.

6.1.1 BLAS and Auxiliary

The BLAS perform basic operations such as matrix-multiply and norms. In most cases, A and B can be transposed or conjugate-transposed.

Simplified API	Traditional API	Operation	Matrix type
`multiply`	`gemm`	$C = \alpha AB + \beta C$	A, B, C all general
`multiply`	`hemm`	$C = \alpha AB + \beta C$	A xor B Hermitian
`multiply`	`symm`	$C = \alpha AB + \beta C$	A xor B symmetric
`rank_k_update`	`herk`	$C = \alpha AA^H + \beta C$	C Hermitian
`rank_k_update`	`syrk`	$C = \alpha AA^T + \beta C$	C symmetric
`rank_2k_update`	`her2k`	$C = \alpha AB^H + \bar{\alpha} BA^H + \beta C$	C Hermitian
`rank_2k_update`	`syr2k`	$C = \alpha AB^T + \alpha BA^T + \beta C$	C symmetric
`triangular_multiply`	`trmm`	$B = \alpha AB$ or $B = \alpha BA$	A triangular
`triangular_solve`	`trsm`	Solve $AX = \alpha B$ or $XA = \alpha B$	A triangular
`add`	`geadd` [1]	$B = \alpha A + \beta B$	any
`copy`	`lacpy` [1]	$B = A$, local	any
`copy`	`zlag2c` [1]	$B = A$, local, precision conversion	any
`redistribute`	`gemr2d` [1]	$B = A$	any
`norm`	`lange, ...` [1]	$\|A\|_1, \|A\|_\infty, \|A\|_{\text{fro}}, \|A\|_{\max}$	any
`scale`	`lascl` [1]	$B = \alpha A$	any
`scale_row_col`	`laqge` [1]	$A = \text{diag}(R) A \,\text{diag}(C)$, equilibration	general
`set`	`laset` [1]	$A_{ij} = \begin{cases} \alpha & \text{if } i \neq j, \\ \beta & \text{if } i = j \end{cases}$	any
`print`	n/a	print full or subset of matrix	any

6.1.2 Linear Systems and Least Squares

Linear system and least squares solvers factor a matrix into simpler matrices, typically triangular or unitary, that are easily solved. The factored matrix is then used to solve $AX = B$ or $AX \approx B$. LU factors a general non-symmetric matrix into a lower-triangular matrix L, upper-triangular matrix U, and permutation matrix P, yielding $PA = LU$. Cholesky factors a Hermitian or symmetric positive-definite matrix (HPD/SPD) into a lower-triangular matrix L and its conjugate-transpose, L^H, with $A = LL^H$. Aasen factors a Hermitian or symmetric indefinite matrix into a lower-triangular matrix L, permutation matrix P, and a band matrix T, with $A = LTL^T$. This is different than the familiar Bunch-Kaufman algorithm for indefinite matrices, which generates a block diagonal matrix D, with $A = LDL^T$, instead of a band matrix T. The standard approach for solving least squares problems uses a unitary factorization such as $A = QR$ or $A = LQ$.

To solve a single system $AX = B$, potentially with multiple right-hand sides, the `*_solve` drivers are recommended. To factor a matrix once, then solve different right-hand side matrices, such as in a time-stepping loop, call `*_factor` once, then repeatedly call `*_solve_using_factor`.

[1] SLATE does not provide these traditional (Sca)LAPACK names, only the simplified names.

Simplified API	Traditional API	Operation
General non-symmetric (LU)		
lu_solve	gesv	Solve $AX = B$ using LU
lu_solve_mixed [3]	gesv_mixed	Solve $AX = B$ using LU, mixed precision with iterative refinement
lu_solve_mixed_gmres [3]	gesv_mixed_gmres	Solve $AX = B$ using LU, mixed precision with GMRES
lu_factor	getrf	Factor $A = PLU$
lu_solve_using_factor	getrs	Solve $AX = (PLU)X = B$
lu_inverse_using_factor	getri	Form $A^{-1} = (PLU)^{-1}$
lu_condest_using_factor	gecondest [2]	$\kappa_p(A) = \|A\|_p \cdot \left\|A^{-1}\right\|_p$ for $p \in \{1, \infty\}$
Hermitian/symmetric positive definite (Cholesky)		
chol_solve	posv	Solve $AX = B$ using Cholesky
chol_solve_mixed [3]	posv_mixed	Solve $AX = B$ using Cholesky, mixed precision with iterative refinement
chol_solve_mixed_gmres [3]	posv_mixed_gmres	Solve $AX = B$ using Cholesky, mixed precision with GMRES
chol_factor	potrf	Factor $A = LL^H$
chol_solve_using_factor	potrs	Solve $AX = (LL^H)X = B$
chol_inverse_using_factor	potri	Form $A^{-1} = (LL^H)^{-1}$
chol_condest_using_factor	pocondest	$\kappa_p(A) = \|A\|_p \cdot \left\|A^{-1}\right\|_p$ for $p \in \{1, \infty\}$
Hermitian/symmetric indefinite (block Aasen, permutation not shown)		
indefinite_solve	hesv, sysv	Solve $AX = B$ using Aasen
indefinite_factor	hetrf, sytrf	Factor $A = LTL^H$
indefinite_solve_using_factor	hetrs, sytrs	Solve $AX = (LTL^H)X = B$
indefinite_inverse_using_factor [3]	*hetri, sytri* [3]	Form $A^{-1} = (LTL^H)^{-1}$
indefinite_condest [3]	*sycondest* [2] [3]	$\kappa_p(A) = \|A\|_p \cdot \left\|A^{-1}\right\|_p$ for $p \in \{1, \infty\}$
Triangular		
triangular_inverse [3]	trtri	Form L^{-1}
triangular_condest_using_factor	trcondest [2]	$\kappa_p(L) = \|L\|_p \cdot \left\|L^{-1}\right\|_p$ for $p \in \{1, \infty\}$
Least squares		
least_squares_solve	gels	Solve $AX \approx B$ for rectangular A

6.1.3 Unitary Factorizations

Unitary factorizations factor a matrix A into a unitary matrix Q and an upper-triangular matrix R or lower-triangular matrix L. The Q is represented implicitly by a sequence of vectors representing Householder reflectors. SLATE uses CAQR (communication avoiding QR), so its representation does not match LAPACK's or ScaLAPACK's.

[2] renamed from gecon, pocon, sycon, trcon in LAPACK
[3] not yet implemented

Simplified API	Traditional API	Operation
qr_factor	geqrf	Factor $A = QR$
qr_multiply_by_q	gemqr	Multiply $C = \text{op}(Q)C$ or $C = C\,\text{op}(Q)$
qr_generate_q [3]	*gegqr* [3]	Form Q
lq_factor	gelqf	Factor $A = LQ$
lq_multiply_by_q	gemlq	Multiply $C = \text{op}(Q)C$ or $C = C\,\text{op}(Q)$
lq_generate_q [3]	*geglq* [3]	Form Q
rq_factor [3]	*gerqf* [3]	Factor $A = RQ$
rq_multiply_by_q [3]	*gemrq* [3]	Multiply $C = \text{op}(Q)C$ or $C = C\,\text{op}(Q)$
rq_generate_q [3]	*gegrq* [3]	Form Q
ql_factor [3]	*geqlf* [3]	Factor $A = QL$
ql_multiply_by_q [3]	*gemql* [3]	Multiply $C = \text{op}(Q)C$ or $C = C\,\text{op}(Q)$
ql_generate_q [3]	*gegql* [3]	Form Q

$\text{op}(Q)$ is Q or Q^H.

6.1.4 Eigenvalue and Singular Value Decomposition

SLATE currently has Hermitian/symmetric eigenvalue solvers, generalized Hermitian/symmetric eigenvalue solvers, and the Singular Value Decomposition (SVD). The non-symmetric eigensolver is planned for future work.

The `_values` routines compute only eigen/singular values, while the regular routines also compute eigen/singular vectors.

Variants of methods can be provided by specifying an option in the input arguments, rather than a different routine name as in LAPACK (`gesvd`, `gesdd`, `gesvdx`, `gesvj`, etc.). Currently, the Hermitian eigensolver supports two methods: QR iteration (`MethodEig::QR`) and divide & conquer (`MethodEig::DC`). The SVD supports only QR iteration; divide & conquer is planned for future work. Other methods such as bisection and Jacobi may also be added as needed.

Simplified API	Traditional API	Operation
Hermitian eigenvalues		
eig, eig_values	heev, syev	Factor $A = X \Lambda X^H$
eig, eig_values	hegv, sygv	For positive-definite B: Type 1: $AX = BX\Lambda$ Type 2: $ABX = X\Lambda$ Type 3: $BAX = X\Lambda$
SVD		
svd, svd_values	*gesvd* [1]	Factor $A = U \Sigma V^H$
General non-symmetric eigenvalues		
eig, eig_values [3]	*geev* [3]	Factor $A = X \Lambda X^{-1}$
eig, eig_values [3]	*ggev* [3]	Factor $A = BX \Lambda X^{-1}$

6.2 C and Fortran API

To make SLATE accessible from C and Fortran, we also provide C and Fortran 2003 APIs. Generally, these APIs replaces the `::` in the C++ API with `_` underscore. Because C does not provide overloading, some routine names include an extra term to differentiate. Following the BLAS G2 convention, a suffix is added indicating the type: `_r32` (single), `_r64` (double), `_c32` (complex-single), `_c64` (complex-double). This notation is easily expanded to other data types such as `_r16`, `_c16` for 16-bit half precision, and `_r128` and `_c128` for 128-bit quad precision, as well as mixed precisions. The `_c64` version is shown below.

6.2.1 BLAS and Auxiliary

Simplified API	Traditional API	C/Fortran API
multiply	gemm	slate_multiply_c64
multiply	hemm	slate_hermitian_multiply_c64
multiply	symm	slate_symmetric_multiply_c64
rank_k_update	herk	slate_hermitian_rank_k_update_c64
rank_k_update	syrk	slate_symmetric_rank_k_update_c64
rank_2k_update	her2k	slate_hermitian_rank_2k_update_c64
rank_2k_update	syr2k	slate_symmetric_rank_2k_update_c64
triangular_multiply	trmm	slate_triangular_multiply_c64
triangular_solve	trsm	slate_triangular_solve_c64
add	*geadd* [1]	slate_add_c64
copy	*lacpy* [1]	slate_copy_c64
copy	*zlag2c* [1]	slate_copy_c64c32
norm	*lange* [1]	slate_norm_c64
norm	*lanhe* [1]	slate_hermitian_norm_c64
norm	*lansy* [1]	slate_symmetric_norm_c64
norm	*lantr* [1]	slate_triangular_norm_c64
scale	*lascl* [1]	slate_scale_c64
scale_row_col	*laqge* [1]	slate_scale_row_col_c64
set	*laset* [1]	slate_set_c64

LAPACK does not have a matrix add routine, only the vector add routine `axpy`. ScaLAPACK has `geadd` and `tradd`.

6.2.2 Linear Systems and Least Squares

Simplified API	Traditional API	C/Fortran API
General non-symmetric (LU)		
lu_solve	gesv	slate_lu_solve_c64
lu_factor	getrf	slate_lu_factor_c64
lu_solve_using_factor	getrs	slate_lu_solve_using_factor_
lu_inverse_using_factor	getri	slate_lu_inverse_using_facto
lu_condest_using_factor	gecondest [2]	slate_lu_condest_c64
Hermitian/symmetric positive definite (Cholesky)		
chol_solve	posv	slate_chol_solve_c64
chol_factor	potrf	slate_chol_factor_c64
chol_solve_using_factor	potrs	slate_chol_solve_using_facto
chol_inverse_using_factor	potri	slate_chol_inverse_using_fac
chol_condest_using_factor	pocondest} \cref{cond_renamed	slate_chol_condest_using_fac
Hermitian/symmetric indefinite (block Aasen, permutation not shown)		
indefinite_solve	hesv	slate_indefinite_solve_c64
indefinite_factor	hetrf	slate_indefinite_factor_c64
indefinite_solve_using_factor	hetrs	slate_indefinite_solve_using
indefinite_inverse_using_factor [3]	*hetri* [3]	*slate_indefinite_inverse_usi*
indefinite_condest [3]	*hecondest* [2] [3]	*slate_indefinite_condest_c64*
Triangular		
triangular_inverse [3]	trtri	*slate_triangular_inverse_c64*
triangular_condest_using_factor	trcondest [2]	slate_triangular_condest_usi
Least squares		
least_squares_solve	gels	slate_least_squares_solve_c6

6.2.3 Unitary Factorizations

Simplified API	Traditional API	C/Fortran API
qr_factor	geqrf	slate_qr_factor_c64
qr_multiply_by_q	gemqr	slate_qr_multiply_by_q_c64
qr_generate_q [3]	*gegqr* [3]	*slate_qr_generate_q_c64* [3]
lq_factor	gelqf	slate_lq_factor_c64
lq_multiply_by_q	gemlq	slate_lq_multiply_by_q_c64
lq_generate_q [3]	*geglq* [3]	*slate_lq_generate_q_c64* [3]
rq_factor [3]	*gerqf* [3]	*slate_rq_factor_c64* [3]
rq_multiply_by_q [3]	*gemrq* [3]	*slate_rq_multiply_by_q_c64* [3]
rq_generate_q [3]	*gegrq* [3]	*slate_rq_generate_q_c64* [3]
ql_factor [3]	*geqlf* [3]	*slate_ql_factor_c64* [3]
ql_multiply_by_q [3]	*gemql* [3]	*slate_ql_multiply_by_q_c64* [3]
ql_generate_q [3]	*gegql* [3]	*slate_ql_generate_q_c64* [3]

6.2.4 Eigenvalue and Singular Value Decomposition (SVD)

Simplified API	Traditional API	C/Fortran API
Hermitian		
eig, eig_values	heev	slate_hermitian_eig_c64
		slate_hermitian_eig_values_c64
eig, eig_values	hegv	slate_generalized_hermitian_eig_c64
		slate_generalized_hermitian_eig_values_c64
SVD		
svd, svd_values	*gesvd* [1]	slate_svd_c64
		slate_svd_values_c64
General non-symmetric		
eig, eig_values [3]	*geev* [3]	*slate_eig_c64* [3]
		slate_eig_values_c64 [3]
eig, eig_values [3]	*ggev* [3]	*slate_generalized_eig_c64* [3]
		slate_generalized_eig_values_c64 [3]

6.3 Traditional LAPACK and ScaLAPACK API

SLATE implements many routines from BLAS, LAPACK, and ScaLAPACK. The traditional BLAS, LAPACK, and ScaLAPACK APIs rely on a 5–6 character naming scheme. This systematic scheme was designed to fit into the 6 character limit of Fortran 77. Compared to LAPACK, in SLATE the precision character has been dropped in favor of overloading (`slate::gemm` instead of `sgemm`, `dgemm`, `cgemm`, `zgemm`), and the arguments are greatly simplified by packing information into the matrix classes.

- One or two characters for precision (dropped in SLATE)
  - `s`: single
  - `d`: double
  - `c`: complex-single
  - `z`: complex-double
  - `zc`: mixed complex-double/single (e.g., zcgesv)
  - `ds`: mixed double/single (e.g., dsgesv)
  - `sc`: real-single output, complex-single input (e.g., scnrm2)
  - `dz`: real-double output, complex-double input (e.g., dznrm2)
- Two character matrix type
  - `ge`: general non-symmetric matrix
  - `he`: Hermitian matrix
  - `sy`: symmetric matrix
  - `po`: positive definite, Hermitian or symmetric matrix
  - `tr`: triangular or trapezoidal matrix
  - `tz`: trapezoidal matrix
  - `hs`: Hessenberg matrix
  - `or`: orthogonal matrix
  - `un`: unitary matrix
  - Band matrices
    - `gb`: general band non-symmetric matrix
    - `hb`: Hermitian band matrix
    - `sb`: symmetric band matrix
    - `pb`: positive definite, Hermitian or symmetric band matrix
    - `tb`: triangular band matrix
  - Bi- or tridiagonal matrices
    - `bd`: bidiagonal matrix
    - `st`: symmetric tridiagonal matrix
    - `ht`: Hermitian tridiagonal matrix

  - `pt`: positive definite, Hermitian or symmetric tridiagonal matrix

- Several characters for function
  - Level 1 BLAS: $O(n)$ data, $O(n)$ operations (vectors; no matrix type)
    - `axpy` : $y = \alpha x + y$
    - `scal` : $x = \alpha x$
    - `copy` : copy vector
    - `swap` : swap vectors
    - `dot, dotu, dotc`: dot products (u = unconjugated, c = conjugated)
    - `nrm2` : vector 2-norm
    - `asum` : vector 1-norm (absolute value sum)
    - `iamax`: vector ∞-norm
    - `rot` : apply plane (Givens) rotation
    - `rotg` : generate plane rotation
    - `rotm` : apply modified (fast) plane rotation
    - `rotmg`: generate modified plane rotation
  - Level 2 BLAS: $O(n^2)$ data, $O(n^2)$ operations
    - `mv` : matrix-vector multiply
    - `sv` : solve, one vector RHS
    - `r` : rank-1 update
    - `r2` : rank-2 update
    - `lan`: matrix norm (1, infinity, frobenius, max)
  - Level 3 BLAS: $O(n^2)$ data, $O(n^3)$ operations
    - `mm` : matrix multiply
    - `sm` : solve, multiple RHS
    - `rk` : rank-k update
    - `r2k`: rank-2k update
  - Linear systems and least squares
    - `sv` : solve
    - `ls` : least squares solve (several variants)
    - `trf`: triangular factorization
    - `trs`: solve, using triangular factorization
    - `tri`: inverse, using triangular factorization
    - `con`: condition number, using triangular factorization
  - Unitary (orthogonal) factorizations
    - `qrf, qlf, rqf, lqf`: QR, QL, RQ, LQ unitary factorization
    - `mqr, mlq, mrq, mlq`: multiply by Q from factorization
    - `gqr, glq, grq, glq`: generate Q from factorization
  - Eigenvalue and singular value
    - `ev` : eigenvalue decomposition (variants: `ev, evd, evx, evr`)

- `gv` : generalized eigenvalue decomposition (variants: `gv, gvd, gvx`)
- `svd`: singular value decomposition (variants: `svd, sdd, svdq, svdx, svj, jsv`)

There are many more lower level or specialized routines, but the above routines are the main routines users may encounter. Traditionally, there are also packed (hp, sp, pp, tp, op, up) and rectangular full-packed (RFP: hf, sf, pf, tf, op, up) matrix formats, but these don't apply in SLATE.

CHAPTER 7

Using SLATE

Many of the code snippets in this section reference the SLATE tutorial, available at
https://github.com/icl-utk-edu/slate/tree/master/examples.
Links to individual files are given where applicable.

7.1 Matrices in SLATE

A SLATE matrix consists of a collection of individual tiles, with no correlation between their positions in the matrix and their memory locations. In SLATE the tiles of a matrix are first-class objects that can be individually allocated and passed to low-level tile routines.

7.1.1 Matrix Hierarchy

The usage of SLATE revolves around a Tile class and a Matrix class hierarchy (Figure 7.1). The Tile class is intended as a simple class for maintaining the properties of individual tiles and used in 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.

Grayed out classes are abstract base classes that cannot be directly instantiated.

- BaseMatrix Abstract base class for all matrices.
  - **Matrix** General, $m \times n$ matrix.
  - BaseTrapezoidMatrix Abstract base class for all upper or lower trapezoid storage, $m \times n$ matrices. For upper, tiles $A(i, j)$ for $i \leq j$ are stored; for lower, tiles $A(i, j)$ for $i \geq j$ are stored.
    - **TrapezoidMatrix** Upper or lower trapezoid, $m \times n$ matrix with unit or non-unit diagonal; the opposite triangle is implicitly zero.
      - **TriangularMatrix** Upper or lower triangular, $n \times n$ matrix.
    - **SymmetricMatrix** Symmetric, $n \times n$ matrix, stored by its upper or lower triangle; the opposite triangle is known implicitly by symmetry ($a_{j,i} = a_{i,j}$).
    - **HermitianMatrix** Hermitian, $n \times n$ matrix, stored by its upper or lower triangle; the opposite triangle is known implicitly by symmetry ($a_{j,i} = \overline{a_{i,j}}$).
  - BaseBandMatrix Abstract base class for band matrices, with a lower bandwidth k_l (number of sub-diagonals) and upper bandwidth k_u (number of super-diagonals).
    - **BandMatrix** General, $m \times n$ band matrix. All tiles intersecting the band exist, e.g., $A(i, j)$ for $j = i - \left\lceil \frac{k_l}{n_b} \right\rceil, \ldots, i + \left\lceil \frac{k_u}{n_b} \right\rceil$.
    - BaseTriangularBandMatrix Abstract base class for all upper or lower triangular storage, $n \times n$ band matrices. For upper, tiles within the band in the upper triangle exist; for lower, tiles within the band in the lower triangle exist.
      - **TriangularBandMatrix** Upper or lower triangular, $n \times n$ band matrix; the opposite triangle is implicitly zero.
      - **SymmetricBandMatrix** Symmetric, $n \times n$ band matrix, stored by its upper or lower triangle; the opposite triangle is known implicitly by symmetry ($a_{j,i} = a_{i,j}$).
      - **HermitianBandMatrix** Hermitian, $n \times n$ band matrix, stored by its upper or lower triangle; the opposite triangle is known implicitly by symmetry ($a_{j,i} = \overline{a_{i,j}}$).

The `BaseMatrix` class stores the matrix dimensions; whether the matrix is upper, lower, or general; whether it is non-transposed, transposed, or conjugate transposed; how the matrix is distributed; and the set of tiles—both local tiles and temporary workspace tiles, as needed, during the computation. It also stores the distribution parameters and MPI communicator that would traditionally be stored in a ScaLAPACK context. As such, there is no separate structure to maintain state, nor any need to initialize or finalize the SLATE library.

Currently, in the band matrix hierarchy there is no TrapezoidBandMatrix. This is simply because we haven't found a need for it; if a need arises, it can be added.

SLATE routines require the correct matrix types for their arguments, which helps to ensure correctness, while inexpensive shallow copy conversions exist between the various matrix types. For instance, a general `Matrix` can be converted to a `TriangularMatrix` for doing a triangular solve (`trsm`), without copying. The two matrices have a reference-counted C++ shared pointer

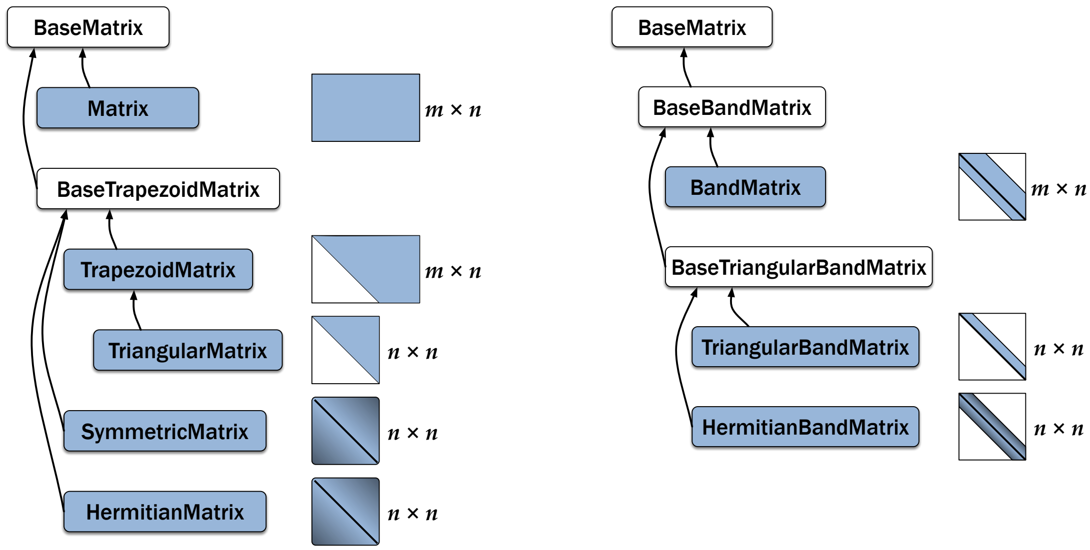


Figure 7.1: Matrix hierarchy in SLATE. Algorithms require the appropriate types for their operation.

to the same underlying data (`std::map` of tiles). Algorithm 7.1 shows some conversions between various matrix types.

Algorithm 7.1 Conversions: ex02_conversion.cc

```
    // A is defined to be a general m x n matrix of type scalar_type
    // (float, std::complex<float>, double, std::complex<double>, etc.).
    slate::Matrix<scalar_type>
        A( m, n, nb, grid_p, grid_q, MPI_COMM_WORLD );

    // Lz is a trapezoid matrix view of the lower trapezoid of A,
    // assuming Unit diagonal.
    slate::TrapezoidMatrix<scalar_type>
        Lz( slate::Uplo::Lower, slate::Diag::Unit, A );

    // Triangular, symmetric, and Hermitian matrices must be square --
    // take square slice if needed.
    int64_t min_mn = std::min( m, n );
    auto A_square = A.slice( 0, min_mn-1, 0, min_mn-1 );

    // L is a triangular matrix view of the lower triangle of A,
    // assuming Unit diagonal.
    slate::TriangularMatrix<scalar_type>
        L( slate::Uplo::Lower, slate::Diag::Unit, A_square );

    // U is a triangular matrix view of the upper triangle of A.
    slate::TriangularMatrix<scalar_type>
        U( slate::Uplo::Upper, slate::Diag::NonUnit, A_square );

    // S is a symmetric matrix view of the upper triangle of A.
    slate::SymmetricMatrix<scalar_type>
        S( slate::Uplo::Upper, A_square );

    // H is a Hermitian matrix view of the upper triangle of A.
    slate::HermitianMatrix<scalar_type>
        H( slate::Uplo::Upper, A_square );
```

Likewise, copying a matrix object is an inexpensive shallow copy, using a C++ shared pointer. Submatrices are also implemented by creating an inexpensive shallow copy, with the matrix object storing the offset from the top left of the original matrix and the transposition operation with respect to the original matrix.

Transpose and conjugate transpose are supported by creating an inexpensive shallow copy and changing the transposition operation flag stored in the new matrix object. For a matrix `A` that is a possibly transposed copy of an original matrix `A0`, the function `A.op()` returns `Op::NoTrans`, `Op::Trans`, or `Op::ConjTrans`, indicating whether `A` is non-transposed, transposed, or conjugate transposed, respectively. The functions `A = transpose(A0)` and `A = conj_transpose(A0)` return new matrices with the operation flag set appropriately. Querying properties of a matrix object takes the transposition and submatrix offsets into account. For instance, `A.mt()` is the number of block rows of $\mathrm{op}(A_0)$, where $A = \mathrm{op}(A_0) = A_0$, A_0^T, or A_0^H. The function `A(i, j)` returns the i, j-th tile of $\mathrm{op}(A_0)$, with the tile's operation flag set to match the `A` matrix.

SLATE supports upper and lower storage with `A.uplo()` returning `Uplo::Upper` or `Uplo::Lower`. Tiles likewise have a flag indicating upper or lower storage, accessed by `A(i, j).uplo()`. For tiles on the matrix diagonal, the `uplo` flag is set to match the matrix, while for off-diagonal tiles it is set to `Uplo::General`.

7.1.2 Creating and Accessing Matrices

A SLATE matrix can be defined and created empty with no data tiles attached.

Algorithm 7.2 Creating matrices: ex01_matrix.cc

```
    // Create an empty matrix (2D block cyclic layout, p x q grid,
    // no tiles allocated, square nb x nb tiles)
    slate::Matrix<scalar_type>
        A( m, n, nb, grid_p, grid_q, MPI_COMM_WORLD );

    // Create an empty matrix (2D block cyclic layout, p x q grid,
    // no tiles allocated, rectangular mb x nb tiles)
    slate::Matrix<scalar_type>
        B( m, n, mb, nb, grid_p, grid_q, MPI_COMM_WORLD );

    // Create an empty TriangularMatrix (2D block cyclic layout, no tiles)
    slate::TriangularMatrix<scalar_type>
        T( slate::Uplo::Lower, slate::Diag::NonUnit, n, nb,
           grid_p, grid_q, MPI_COMM_WORLD );

    // Create an empty matrix based on another matrix structure.
    slate::Matrix<scalar_type> A2 = A.emptyLike();
```

At this point, data tiles can be inserted into the matrix. The tile data can be allocated by SLATE in CPU memory (Algorithm 7.3) or GPU memory (Algorithm 7.4), in which case SLATE is responsible for deallocating the data. The tile data can also be provided by the user (Algorithm 7.5), so that the user retains ownership of the data, and the user is responsible for deallocating the data. Below are examples of different modes of allocating data.

Algorithm 7.3 SLATE allocating CPU host memory for a matrix: ex01_matrix.cc

```
    // Create two empty matrices.
    slate::Matrix<scalar_type> A( m, n, nb, grid_p, grid_q, MPI_COMM_WORLD );
    auto A2 = A.emptyLike();

    // Insert tiles on the CPU host.
    A.insertLocalTiles( slate::Target::Host );

    // A2.insertLocalTiles( slate::Target::Host ) is equivalent to:
    for (int64_t j = 0; j < A2.nt(); ++j)
        for (int64_t i = 0; i < A2.mt(); ++i)
            if (A2.tileIsLocal( i, j ))
                A2.tileInsert( i, j, slate::HostNum );
```

Algorithm 7.4 SLATE allocating GPU device memory for a matrix: ex01_matrix.cc

```
    // Create two empty matrices.
    slate::Matrix<scalar_type> A( m, n, nb, grid_p, grid_q, MPI_COMM_WORLD );
    auto A2 = A.emptyLike();

    // Insert tiles on the GPU devices.
    A.insertLocalTiles( slate::Target::Devices );

    // A2.insertLocalTiles( slate::Target::Devices ) is equivalent to:
    for (int64_t j = 0; j < A2.nt(); ++j)
        for (int64_t i = 0; i < A2.mt(); ++i)
            if (A2.tileIsLocal( i, j ))
                A2.tileInsert( i, j, A2.tileDevice( i, j ) );
```

SLATE can take memory pointers directly from the user to initialize the tiles in a Matrix. The user's tile size must match the tile size `mb` × `nb` for the Matrix.

Algorithm 7.5 Inserting tiles using user-defined data: ex01_matrix.cc

```
    // Create an empty matrix (2D block cyclic layout, no tiles).
    slate::Matrix<scalar_type> A( m, n, nb, grid_p, grid_q, MPI_COMM_WORLD );

    // Attach user allocated tiles, from pointers in data( i, j )
    // with local stride lld between columns.
    for (int64_t j = 0; j < A.nt(); ++j) {
        for (int64_t i = 0; i < A.mt(); ++i) {
            if (A.tileIsLocal( i, j ))
                A.tileInsert( i, j, data( i, j ), lld );
        }
    }
```

Now that the matrix is created and tiles are attached to the matrix, the elements of data in the tiles can be accessed locally on different processes.

For a matrix `A`, calling `A( i, j )` returns its (i, j)-th block, in block row i and block column j. If a matrix is transposed, the indices get transposed and the transposition operation is set on the tile, that is, if `AT = transpose( A )`, then `AT( i, j )` is `transpose( A( j, i ) )`. Similarly, with conjugate transposed, if `AH = conj_transpose( A )`, then `AH( i, j )` is `conj_transpose( A( j, i ) )`. The `A.at( i, j )` operator is equivalent to `A( i, j )`.

For a tile `T`, calling `T( i, j )` returns its (i, j)-th element. If a tile is transposed, the transposition operation is included, that is, if `TT = transpose( T )`, then `TT( i, j )` is `T( j, i )`. If a tile is conjugate transposed, the conjugation is also included, that is, if `TH = conj_transpose(T)`, then `TH( i, j )` is `conj( T( j, i ) )`. This makes `TH( i, j )` read-only. The `T.at( i, j )` operator includes transposition *but not conjugation* in order to return a reference that can be updated. As this is a rather subtle distinction for which we may devise a better solution in the future; feedback and suggestions are welcome.

Also, at the moment, the `mb()`, `nb()`, `T( i, j )`, and `T.at( i, j )` operators have an **`if`** condition inside to check the transposition; thus, they are not efficient for use inside inner loops. It is better to get the data pointer and index it directly. Compare Algorithm 7.6 and Algorithm 7.7.

Algorithm 7.6 Accessing tile elements: ex01_matrix.cc

```
    slate::Matrix<scalar_type> A( m, n, nb, grid_p, grid_q, MPI_COMM_WORLD );
    A.insertLocalTiles( slate::Target::Host );

    // Loop over tiles in A.
    int64_t jj_global = 0;
    for (int64_t j = 0; j < A.nt(); ++j) {
        int64_t ii_global = 0;
        for (int64_t i = 0; i < A.mt(); ++i) {
            if (A.tileIsLocal( i, j )) {
                // For local tiles, loop over entries in tile.
                // Make sure CPU tile exists for writing.
                A.tileGetForWriting( i, j, slate::HostNum, LayoutConvert::ColMajor );
                slate::Tile<scalar_type> T = A( i, j, slate::HostNum );
                for (int64_t jj = 0; jj < T.nb(); ++jj) {
                    for (int64_t ii = 0; ii < T.mb(); ++ii) {
                        // Note: currently using T.at() is inefficient
                        // in inner loops; see below.
                        T.at( ii, jj )
                            = std::abs( (ii_global + ii) - (jj_global + jj) );
                    }
                }
            }
            ii_global += A.tileMb( i );
        }
        jj_global += A.tileMb( j );
    }
```

Algorithm 7.7 Accessing tile elements, currently more efficient implementation: ex01_matrix.cc

```
    // Loop over tiles in A, more efficient implementation.
    jj_global = 0;
    for (int64_t j = 0; j < A.nt(); ++j) {
        int64_t ii_global = 0;
        for (int64_t i = 0; i < A.mt(); ++i) {
            if (A.tileIsLocal( i, j )) {
                // For local tiles, loop over entries in tile.
                // Make sure CPU tile exists for writing.
                A.tileGetForWriting( i, j, slate::HostNum, LayoutConvert::ColMajor );
                slate::Tile<scalar_type> T = A( i, j, slate::HostNum );
                scalar_type* data = T.data();
                int64_t   mb      = T.mb();
                int64_t   nb      = T.nb();
                int64_t   stride  = T.stride();
                for (int64_t jj = 0; jj < T.nb(); ++jj) {
                    for (int64_t ii = 0; ii < T.mb(); ++ii) {
                        // Currently more efficient than using T.at().
                        data[ ii + jj*stride ]
                            = std::abs( (ii_global + ii) - (jj_global + jj) );
                    }
                }
            }
            ii_global += A.tileMb( i );
        }
        jj_global += A.tileMb( j );
    }
```

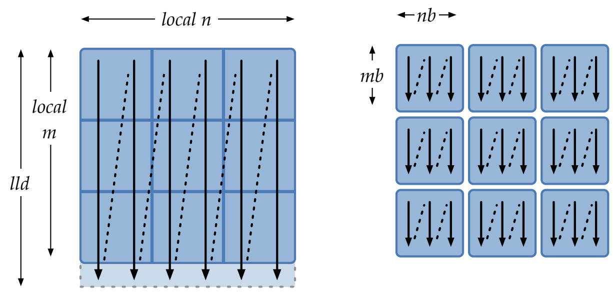


Figure 7.2: Matrix layout of ScaLAPACK (left) and layout with contiguous tiles (right). SLATE matrix and tiles structures are flexible and accommodate multiple layouts.

7.1.3 Matrices from ScaLAPACK

SLATE also supports tiles laid out in memory using the traditional ScaLAPACK matrix storage allowing a leading dimension stride when accessing the matrix (Figure 7.2). This eases an application's transition from ScaLAPACK to SLATE.

SLATE can map its Matrix datatype over matrices that are laid out in ScaLAPACK format.

Algorithm 7.8 Creating matrix from ScaLAPACK-style data: ex01_matrix.cc

```
    // User-allocated data, in ScaLAPACK format (assuming column-major grid).
    int myrow = mpi_rank % grid_p;
    int mycol = mpi_rank / grid_p;
    int64_t mlocal = slate::num_local_rows_cols( m, nb, myrow, 0, grid_p );
    int64_t nlocal = slate::num_local_rows_cols( n, nb, myrow, 0, grid_p );
    int64_t lld = mlocal; // local leading dimension
    scalar_type* A_data = new scalar_type[ lld*nlocal ];

    // Create matrix from ScaLAPACK data.
    auto A = slate::Matrix<scalar_type>::fromScaLAPACK(
        m, n,                   // global matrix dimensions
        A_data,                 // local ScaLAPACK array data
        lld,                    // local leading dimension (column stride) for data
        nb, nb,                 // block size
        slate::GridOrder::Col,  // col- or row-major MPI process grid
        grid_p, grid_q,         // MPI process grid
        MPI_COMM_WORLD          // MPI communicator
    );
```

7.1.4 Matrix Transpose

In SLATE the transpose is a structural property and is associated with the Matrix or Tile object. Using the transpose operation is a lightweight operations that sets a flag in a shallow copy of the matrix or tile.

Algorithm 7.9 Transposing matrices: ex01_matrix.cc

```
    slate::Matrix<scalar_type> A( m, n, nb, grid_p, grid_q, MPI_COMM_WORLD );

    // Transpose
    // AT is a transposed view of A, with flag AT.op() == Op::Trans.
    // The Tile AT( i, j ) == transpose( A( j, i ) ).
    auto AT = transpose( A );

    // Conjugate transpose
    // AH is a conjugate-transposed view of A, with flag AH.op() == Op::ConjTrans.
    // The Tile AH( i, j ) == conj_transpose( A( j, i ) ).
    auto AH = conj_transpose( A );
```

7.1.5 Submatrices

SLATE submatrices are views of SLATE matrices based on tile indices. The submatrix that is created uses shallow copy semantics.

Algorithm 7.10 Sub-matrices: ex03_submatrix.cc

```
    // view of A( i1 : i2, j1 : j2 ) as tile indices, inclusive
    auto B = A.sub( i1, i2, j1, j2 );

    // view of all of A
    B = A;

    // same, view of all of A
    B = A.sub( 0, A.mt()-1, 0, A.nt()-1 );

    // view of first block-column, A[ 0:mt-1, 0:0 ] as tile indices
    B = A.sub( 0, A.mt()-1, 0, 0 );

    // view of first block-row, A[ 0:0, 0:nt-1 ] as tile indices
    B = A.sub( 0, 0, 0, A.nt()-1 );
```

7.1.6 Matrix Slices

Matrix slices use column and row indices instead of tile indices. Note that the slice operations are less efficient than the submatrix operations, and the matrices produced have less algorithm support, especially on GPUs, which uses batch operations where all tiles must be the same size. We are in the process of fixing this so GPUs can handle arbitrary mixtures of tile sizes (Oct 2023).

Algorithm 7.11 Matrix slice: ex03_submatrix.cc

```
    // view of A( row1 : row2, col1 : col2 ), inclusive
    B = A.slice( row1, row2, col1, col2 );

    // view of all of A
    B = A.slice( 0, A.m()-1, 0, A.n()-1 );

    // view of first column, A[ 0:m-1, 0:0 ]
    B = A.slice( 0, A.m()-1, 0, 0 );

    // view of first row, A[ 0:0, 0:n-1 ]
    B = A.slice( 0, 0, 0, A.n()-1 );
```

7.1.7 Deep Matrix Copy

SLATE can make a deep copy of a matrix and do precision conversion as needed. This is a heavy-weight operation and makes a full copy of the matrix. Currently it copies host-to-host or device-to-device, depending on the Target in options (Section 7.2.1). Copying from a matrix on the host to a matrix on devices works, but currently incurs extra overhead.

Algorithm 7.12 Deep matrix copy:: ex01_matrix.cc

```
    // scalar_type is double or complex<double>;
    // low_type    is float  or complex<float>.
    slate::Matrix<scalar_type> A_hi( m, n, nb, grid_p, grid_q, MPI_COMM_WORLD );
    slate::Matrix<low_type>    A_lo( m, n, nb, grid_p, grid_q, MPI_COMM_WORLD );
    A_hi.insertLocalTiles();
    A_lo.insertLocalTiles();

    auto A_hi_2 = A_hi.emptyLike();
    A_hi_2.insertLocalTiles();

    // Copy with precision conversion from double to float.
    copy( A_hi, A_lo );

    // Copy with precision conversion from float to double.
    copy( A_lo, A_hi );

    // Copy without conversion.
    copy( A_hi, A_hi_2 );
```

7.2 Using SLATE Functions

This user's guide describes some of the high-level, commonly used functionality available in SLATE. For details on the current implementation, please access the online SLATE Function Reference, generated from the source code documentation and are available at https://icl.bitbucket.io/slate/

7.2.1 Execution Options

SLATE routines take an optional map of options as the last argument. These options can help tune the execution or specify the execution target.

Algorithm 7.13 Options.

```
// Commonly used options in SLATE (slate::Option::name)
Target              // computation method:
                    //     HostTask (default), Devices, HostNest, HostBatch
Lookahead,          // lookahead depth for algorithms (default 1)
InnerBlocking       // inner blocking size for panel operations  (default 16)
MaxPanelThreads     // max number of threads for panel operation (default omp_get_max_threads() / 2)
PivotThreshold      // pivoting threshold in LU (default 1.0)

MethodCholQR        // method for Cholesky QR: Auto (default), HerkC, GemmA, GemmC
MethodEig           // method to solve tridiagonal eig: QR or DC
MethodGels          // method for least squares: Auto, Cholqr, Geqrf (default)
MethodGemm          // method for gemm: Auto (default), GemmA, GemmC
MethodHemm          // method for hemm: Auto (default), HemmA, HemmC
MethodLU            // method for pivoting in LU: PartialPiv (default), CALU, NoPiv
MethodTrsm          // method for trsm: Auto (default), TrsmA, TrsmB

PrintPrecision      // floating point precision to print (default 4)
PrintWidth          // floating point width to print (default 10)
PrintVerbose        // which matrix entries to print, level 0-4. See `./test/tester -h gemm`.

MaxIterations       // max number of iterations in iterative refinement (default 30)
Tolerance           // tolerance for iterative methods (default epsilon)
UseFallbackSolver   // whether to fall back to full double-precision solve (default true)
```

These options are passed via an optional map of name–value pairs. In the following example, the `gemm` execution options are set to execute on GPU devices with a lookahead of 2. For more details, see the SLATE function reference.

Algorithm 7.14 Passing options to multiply (gemm): ex05_blas.cc

```
        // Execute on GPU devices with lookahead of 2.
        slate::Options opts = {
            { slate::Option::Lookahead, 2 },
            { slate::Option::Target, slate::Target::Devices },
        };
        slate::multiply( alpha, A, B, beta, C, opts );
```

7.2.2 Matrix Norms

The following distributed parallel general matrix norms are available in SLATE and are defined for any SLATE matrix type: Matrix, SymmetricMatrix, HermitianMatrix, TriangularMatrix, etc.

Algorithm 7.15 Norms: ex04_norm.cc

```
    slate::Matrix<scalar_type> A( m, n, nb, grid_p, grid_q, MPI_COMM_WORLD );
    // ...
    real_type A_norm_one = slate::norm( slate::Norm::One, A );
    real_type A_norm_inf = slate::norm( slate::Norm::Inf, A );
    real_type A_norm_max = slate::norm( slate::Norm::Max, A );
    real_type A_norm_fro = slate::norm( slate::Norm::Fro, A );

    // norm() is overloaded for all matrix types: Symmetric, Triangular, etc.
    slate::SymmetricMatrix<scalar_type>
        S( slate::Uplo::Lower, n, nb, grid_p, grid_q, MPI_COMM_WORLD );
    // ...
    real_type S_norm_one = slate::norm( slate::Norm::One, S );
    real_type S_norm_inf = slate::norm( slate::Norm::Inf, S );
    real_type S_norm_max = slate::norm( slate::Norm::Max, S );
    real_type S_norm_fro = slate::norm( slate::Norm::Fro, S );
```

7.2.3 Matrix-Matrix Multiply

SLATE implements matrix multiply for the matrices in the matrix hierarchy (e.g., `gemm, gbmm, hemm, symm, trmm, trsm`). A matrix can set several flags that get recorded within its structure and define the view of the matrix. For example, the transpose flag can be set (e.g., `AT = transpose( A )`, or `AC = conj_transpose( A )`), so that the user can access the matrix data as needed.

Algorithm 7.16 Parallel matrix multiply: ex05_blas.cc

```
    // C = alpha A B + beta C, where A, B, C are all general matrices.
    slate::multiply( alpha, A, B, beta, C );  // simplified API
    slate::gemm( alpha, A, B, beta, C );      // traditional API

    // Matrices can be transposed or conjugate-transposed beforehand.
    // C = alpha A^T B^H + beta C
    auto AT = transpose( A );
    auto BH = conj_transpose( B );
    slate::multiply( alpha, AT, BH, beta, C );  // simplified API
    slate::gemm( alpha, AT, BH, beta, C );      // traditional API

    // C = alpha A B + beta C, where A is symmetric, on left side
    slate::multiply( alpha, A, B, beta, C );                  // simplified API
    slate::symm( slate::Side::Left, alpha, A, B, beta, C );   // traditional API

    // C = alpha B A + beta C, where A is symmetric, on right side
    // Note B, A order reversed in multiply compared to symm.
    slate::multiply( alpha, B, A, beta, C );                  // simplified API
    slate::symm( slate::Side::Right, alpha, A, B, beta, C );  // traditional API

    // C = alpha A B + beta C, where A is Hermitian, on left side
    slate::multiply( alpha, A, B, beta, C );                  // simplified API
    slate::hemm( slate::Side::Left, alpha, A, B, beta, C );   // traditional API

    // C = alpha B A + beta C, where A is Hermitian, on right side
    // Note B, A order reversed in multiply compared to hemm.
    slate::multiply( alpha, B, A, beta, C );                  // simplified API
    slate::hemm( slate::Side::Right, alpha, A, B, beta, C );  // traditional API
```

Rank k and $2k$ matrix multiply have different semantics, namely that the A and B matrices are each used twice—once un-transposed, once (conjugate) transposed.

Algorithm 7.17 Parallel rank k and $2k$ updates: ex05_blas.cc

```
    // C = alpha A A^T + beta C, where C is symmetric
    slate::rank_k_update( alpha, A, beta, C );      // simplified API
    slate::syrk( alpha, A, beta, C );               // traditional API

    // C = alpha A B^T + alpha B A^T + beta C, where C is symmetric
    slate::rank_2k_update( alpha, A, B, beta, C );  // simplified API
    slate::syr2k( alpha, A, B, beta, C );           // traditional API

    // C = alpha A A^H + beta C, where C is Hermitian
    slate::rank_k_update( alpha, A, beta, C );      // simplified API
    slate::herk( alpha, A, beta, C );               // traditional API

    // C = alpha A B^H + conj(alpha) B A^H + beta C, where C is Hermitian
    slate::rank_2k_update( alpha, A, B, beta, C );  // simplified API
    slate::her2k( alpha, A, B, beta, C );           // traditional API
```

7.2.4 Operations with Triangular Matrices

For triangular matrices, the `uplo (Lower, Upper)`, `diag (Unit, NonUnit)` and `transpose op (NoTrans, Trans, ConjTrans)` flags set matrix-specific information about whether the matrix is upper or lower triangular, the status of the diagonal, and whether the matrix is transposed.

Algorithm 7.18 Parallel triangular multiply and solve: ex05_blas.cc

```
    //----- left
    // B = alpha A B, where A is triangular, on left side
    slate::triangular_multiply( alpha, A, B );       // simplified API
    slate::trmm( slate::Side::Left, alpha, A, B );   // traditional API

    // Solve AX = B, where A is triangular, on left side; X overwrites B.
    // That is, B = alpha A^{-1} B.
    slate::triangular_solve( alpha, A, B );          // simplified API
    slate::trsm( slate::Side::Left, alpha, A, B );   // traditional API

    //----- right
    // B = alpha B A, where A is triangular, on right side
    // Note B, A order reversed in multiply compared to trmm.
    slate::triangular_multiply( alpha, B, A );       // simplified API
    slate::trmm( slate::Side::Right, alpha, A, B );  // traditional API

    // Solve XA = B, where A is triangular, on right side; X overwrites B.
    // That is, B = alpha B A^{-1}.
    // Note B, A order reversed in solve compared to trsm.
    slate::triangular_solve( alpha, B, A );          // simplified API
    slate::trsm( slate::Side::Right, alpha, A, B );  // traditional API
```

7.2.5 Operations with Band Matrices

Band matrices include the BandMatrix, TriangularBandMatrix, SymmetricBandMatrix, and HermitianBandMatrix classes. For an upper-block bandwidth k_u and lower-block bandwidth k_l, only the tiles $A(i, j)$ for $j - k_u \leq i \leq j + k_l$ are stored. Band matrices have multiply, factorize, solve and norm operations defined for them.

Algorithm 7.19 Band operations.

```
// band matrix with block bandwidth (kl, ku)
auto A = slate::BandMatrix<scalar_t>(
    m, n, kl, ku, nb, p, q, mpi_comm);
// A needs memory to be allocated and initialized ...
// general matrix B
slate::Matrix<double> B( n, nrhs, ... );
// B needs memory to be allocated and initialized ...

// Solve AX = B where A is band; X overwrites B
slate::lu_solve( A, B );        // simplified

slate::Pivots pivots;
slate::gbsv( A, pivots, B );    // traditional
```

7.2.6 Linear Systems: General Non-Symmetric Square Matrices (LU)

Distributed parallel LU factorization and solve computes the solution to a system of linear equations

$$AX = B,$$

where A is an $n \times n$ matrix and X and B are $n \times nrhs$ matrices. LU decomposition with partial pivoting and row interchanges is used to factor A as

$$A = PLU,$$

where P is a permutation matrix, L is unit lower triangular, and U is upper triangular. The factored form of A is then used to solve the system of equations $AX = B$.

Algorithm 7.20 LU solve: ex06_linear_system_lu.cc

```
    slate::Matrix<scalar_type> A( n, n,    nb, grid_p, grid_q, MPI_COMM_WORLD );
    slate::Matrix<scalar_type> B( n, nrhs, nb, grid_p, grid_q, MPI_COMM_WORLD );
    // ...

    slate::lu_solve( A, B );        // simplified API

    slate::Pivots pivots;
    slate::gesv( A, pivots, B );    // traditional API
```

Because pivoting can be expensive, SLATE provides several pivoting variants for LU. These variants are controlled with the options argument. First, Option::MethodLU can be set to MethodLU::PartialPiv for partial pivoting, MethodLU::CALU [16] for tournament pivoting, or MethodLU::NoPiv for no pivoting. Furthermore, partial pivoting can be relaxed by setting the

`Option::PivotThreshold` option between 0 and 1 [17]. A threshold of 1 gives regular partial pivoting, and reducing the threshold reduces the number of row exchanges.

Not pivoting is the fastest variant but is only numerically stable for select classes of matrices, such as diagonal-dominant ones. Partial pivoting with a threshold of 1 is the slowest, but most stable, variant. Reducing the pivoting threshold reduces the number of rows that are exchanged; experimental results suggest that a threshold of 0.5 or 0.1 usually gives a nice speedup with little loss of accuracy. Finally, tournament pivoting reduces the number of MPI reductions in the pivot search, so tournament pivoting should provide better scaling than partial pivoting.

Currently, LU for banded matrices ignores `Option::MethodLU` and always uses partial pivoting, but it does support the `Option::PivotThreshold` option. The mixed-precision LU routines support both options.

7.2.7 Linear Systems: Hermitian/Symmetric Positive Definite (Cholesky)

Distributed parallel Cholesky factorization and solve computes the solution to a system of linear equations

$$AX = B,$$

where A is an $n \times n$ Hermitian or symmetric positive definite matrix and X and B are $n \times nrhs$ matrices. The Cholesky decomposition is used to factor A as

$$A = LL^H,$$

if A is stored lower, where L is a lower-triangular matrix, or

$$A = U^H U,$$

if A is stored upper, where U is an upper-triangular matrix. The factored form of A is then used to solve the system of equations $AX = B$.

Algorithm 7.21 Cholesky solve: ex07_linear_system_cholesky.cc

```
    slate::HermitianMatrix<scalar_type>
        A( slate::Uplo::Lower, n, nb, grid_p, grid_q, MPI_COMM_WORLD );
    slate::Matrix<scalar_type> B( n, nrhs, nb, grid_p, grid_q, MPI_COMM_WORLD );
    // ...
    slate::chol_solve( A, B );  // simplified API

    slate::posv( A, B );        // traditional API
```

7.2.8 Linear Systems: Hermitian/Symmetric Indefinite (Aasen's)

Distributed parallel Hermitian or symmetric indefinite LTL^T factorization and solve computes the solution to a system of linear equations

$$AX = B,$$

where A is an $n \times n$ Hermitian or symmetric matrix and X and B are $n \times nrhs$ matrices. Aasen's 2-stage algorithm is used to factor A as

$$A = LTL^H,$$

if A is stored lower, or

$$A = U^H TU,$$

if A is stored upper. U (or L) is a product of permutation and unit upper (lower) triangular matrices, and T is Hermitian and banded. The matrix T is then factored using LU with partial pivoting. The factored form of A is then used to solve the system of equations $AX = B$.

Algorithm 7.22 Indefinite solve: ex08_linear_system_indefinite.cc

```
    slate::HermitianMatrix<scalar_type>
        A( slate::Uplo::Lower, n, nb, grid_p, grid_q, MPI_COMM_WORLD );
    slate::Matrix<scalar_type> B( n, nrhs, nb, grid_p, grid_q, MPI_COMM_WORLD );
    // ...

    // simplified API
    slate::indefinite_solve( A, B );

    // traditional API
    // workspaces
    // todo: drop H (internal workspace)
    slate::Matrix<scalar_type>     H( n, n, nb, grid_p, grid_q, MPI_COMM_WORLD );
    slate::BandMatrix<scalar_type> T( n, n, nb, nb, nb, grid_p, grid_q, MPI_COMM_WORLD );
    slate::Pivots pivots, pivots2;

    slate::hesv( A, pivots, T, pivots2, H, B );
```

7.2.9 Least Squares: AX ≈ B Using QR or LQ

Distributed parallel least squares solve via QR or LQ factorization solves overdetermined or underdetermined complex linear systems involving an $m \times n$ matrix A, using a QR or LQ factorization of A. It is assumed that A has full rank. X is $n \times nrhs$, B is $m \times nrhs$. The routine takes a single matrix B_X, which is $\max(m, n) \times nrhs$, to represent both the input right-hand side B and the output solution X.

If $m \geq n$, it solves the overdetermined system $AX \approx B$ with least squares solution X that minimizes $\|AX - B\|_2$. The matrix B_X is $m \times nrhs$. On input, B is all m rows of B_X. On output, X is the first n rows of B_X. Currently, in this case A must be not transposed.

If $m < n$, it solves the underdetermined system $AX = B$ with minimum norm solution X that minimizes $\|X\|_2$. The matrix B_X is $n \times nrhs$. On input, B is first the m rows of B_X. On output, X is all n rows of B_X. Currently, in this case A must be transposed (only if real) or conjugate-transposed.

Several right-hand side vectors b and solution vectors x can be handled in a single call; they are stored as the columns of the $m \times nrhs$ right-hand side matrix B and the $n \times nrhs$ solution matrix X.

Note that these (m, n) differ from (M, N) in ScaLAPACK, where the original A is $M \times N$ before applying any transpose, while here A is $m \times n$ after applying any transpose.

The solution vector X is contained in the same storage as B, so the space provided for the right-hand side B should accommodate the solution vector X. The example in Algorithm 7.23 shows how to handle overdetermined systems ($m \geq n$).

Algorithm 7.23 Least squares (overdetermined): ex09_least_squares.cc

```
    int64_t max_mn = std::max( m, n );
    slate::Matrix<scalar_type> A( m, n, nb, grid_p, grid_q, MPI_COMM_WORLD );
    slate::Matrix<scalar_type> BX( max_mn, nrhs, nb, grid_p, grid_q, MPI_COMM_WORLD );
    // ...
    auto B = BX;  // == BX.slice( 0, m-1, 0, nrhs-1 );
    auto X = BX.slice( 0, n-1, 0, nrhs-1 );

    // solve AX = B, solution in X
    slate::least_squares_solve( A, BX );  // simplified API

    slate::gels( A, BX );                 // traditional API
```

The example in Algorithm 7.24 shows how to handle underdetermined systems ($m < n$).

Algorithm 7.24 Least squares (underdetermined): ex09_least_squares.cc

```
    int64_t max_mn = std::max( m, n );
    slate::Matrix<scalar_type> A( m, n, nb, grid_p, grid_q, MPI_COMM_WORLD );
    slate::Matrix<scalar_type> BX( max_mn, nrhs, nb, grid_p, grid_q, MPI_COMM_WORLD );
    // ...
    auto B = BX.slice( 0, n-1, 0, nrhs-1 );
    auto X = BX;  // == BX.slice( 0, m-1, 0, nrhs-1 );

    // solve A^H X = B, solution in X
    auto AH = conj_transpose( A );
    slate::least_squares_solve( AH, BX );  // simplified API

    slate::gels( AH, BX );                 // traditional API
```

7.2.10 Mixed-Precision Routines

Mixed-precision routines do their heavy computation in lower precision (e.g., single precision), taking advantage of the higher number of operations per second that are available at lower precision. Then, the answers obtained in the lower precision are improved using iterative refinement or GMRES in higher precision (e.g., double precision) to achieve the accuracy desired. If iterative refinement fails to reach desired accuracy, the computation falls back and runs the high-precision algorithm. Mixed-precision algorithms are implemented for LU and Cholesky solvers.

Algorithm 7.25 Mixed precision LU solve. ex06_linear_system_lu.cc

```
    // mixed precision: factor in single, iterative refinement to double
    slate::Matrix<scalar_type> A( n, n,    nb, grid_p, grid_q, MPI_COMM_WORLD );
    slate::Matrix<scalar_type> B( n, nrhs, nb, grid_p, grid_q, MPI_COMM_WORLD );
    slate::Matrix<scalar_type> X( n, nrhs, nb, grid_p, grid_q, MPI_COMM_WORLD );
    slate::Matrix<scalar_type> B1( n, 1,   nb, grid_p, grid_q, MPI_COMM_WORLD );
    slate::Matrix<scalar_type> X1( n, 1,   nb, grid_p, grid_q, MPI_COMM_WORLD );
    int iters = 0;
    // ...

    // todo: simplified API

    // traditional API
    slate::gesv_mixed( A, pivots, B, X, iters );
    slate::gesv_mixed_gmres( A, pivots, B1, X1, iters );  // only one RHS
```

Algorithm 7.26 Mixed precision Cholesky solve. ex07_linear_system_cholesky.cc

```
    // mixed precision: factor in single, iterative refinement to double
    slate::HermitianMatrix<scalar_type>
        A( slate::Uplo::Lower, n, nb, grid_p, grid_q, MPI_COMM_WORLD );
    slate::Matrix<scalar_type> B( n, nrhs, nb, grid_p, grid_q, MPI_COMM_WORLD );
    slate::Matrix<scalar_type> X( n, nrhs, nb, grid_p, grid_q, MPI_COMM_WORLD );
    slate::Matrix<scalar_type> B1( n, 1,   nb, grid_p, grid_q, MPI_COMM_WORLD );
    slate::Matrix<scalar_type> X1( n, 1,   nb, grid_p, grid_q, MPI_COMM_WORLD );
    int iters = 0;

    // todo: simplified API

    // traditional API
    slate::posv_mixed( A, B, X, iters );
    slate::posv_mixed_gmres( A, B1, X1, iters );  // only one RHS
```

7.2.11 Matrix Inverse

Matrix inversion requires that the matrix first be factored, and then the inverse is computed from the factors. Note: it is generally recommended that you solve $AX = B$ using the solve routines (e.g., `lu_solve`, `chol_solve`) rather than computing the inverse and multiplying $X = A^{-1}B$. Solves are both faster and more accurate. Matrix inversion is implemented for LU and Cholesky factorization.

Algorithm 7.27 LU inverse: ex06_linear_system_lu.cc

```
    slate::HermitianMatrix<scalar_type>
        A( slate::Uplo::Lower, n, nb, grid_p, grid_q, MPI_COMM_WORLD );
    // ...

    // simplified API
    slate::chol_factor( A );
    slate::chol_inverse_using_factor( A );

    // traditional API
    slate::potrf( A );  // factor
    slate::potri( A );  // inverse
```

Algorithm 7.28 Cholesky inverse: ex07_linear_system_cholesky.cc

```
    slate::HermitianMatrix<scalar_type>
        A( slate::Uplo::Lower, n, nb, grid_p, grid_q, MPI_COMM_WORLD );
    // ...

    // simplified API
    slate::chol_factor( A );
    slate::chol_inverse_using_factor( A );

    // traditional API
    slate::potrf( A );  // factor
    slate::potri( A );  // inverse
```

7.2.12 Singular Value Decomposition

The SLATE singular value decomposition (SVD) algorithm uses a 2-stage reduction that involves reduction first to a triangular band matrix, and then to bidiagonal, which is used to compute the singular values.

Algorithm 7.29 SVD: ex10_svd.cc

```
    int64_t min_mn = std::min( m, n );
    slate::Matrix<scalar_type> A( m, n, nb, grid_p, grid_q, MPI_COMM_WORLD );
    std::vector<real_t> Sigma( min_mn );
    // ...

    // A = U Sigma V^H, singular values only
    slate::svd_vals( A, Sigma );
    slate::svd( A, Sigma );

    // Singular vectors
    // U is m x min_mn (reduced SVD) or m x m (full SVD)
    // V is min_mn x n (reduced SVD) or n x n (full SVD)
    slate::Matrix<scalar_type>  U( m, min_mn, nb, grid_p, grid_q, MPI_COMM_WORLD );
    slate::Matrix<scalar_type> VH( min_mn, n, nb, grid_p, grid_q, MPI_COMM_WORLD );
    // empty, 0-by-0 matrices as placeholders for U and VH.
    slate::Matrix<scalar_type> Uempty, Vempty;
    // ...

    slate::svd( A, Sigma, U, VH );      // both U and V^H
    slate::svd( A, Sigma, U, Vempty );  // only U
    slate::svd( A, Sigma, Uempty, VH ); // only V^H
```

7.2.13 Hermitian/Symmetric Eigenvalues

The SLATE eigenvalue algorithm uses a 2-stage reduction that involves reduction first to a Hermitian band matrix, then to real symmetric tridiagonal, which is used to compute the eigenvalues. Even in the complex-valued case, the tridiagonal matrix is real.

It currently has two methods: `MethodEig::DC` for divide-and-conquer (default for vectors), and `MethodEig::QR` for QR iteration. Eigenvalues are always found using a variant of QR iteration.

Algorithm 7.30 Hermitian/symmetric eigenvalues: ex11_hermitian_eig.cc

```
    slate::HermitianMatrix<scalar_type>
        A( slate::Uplo::Lower, n, nb, grid_p, grid_q, MPI_COMM_WORLD );
    slate::Matrix<scalar_type>
        Z( n, n, nb, grid_p, grid_q, MPI_COMM_WORLD );
    std::vector<real_t> Lambda( n );
    // ...
    // A = Z Lambda Z^H, eigenvalues only
    slate::eig_vals( A, Lambda );  // simplified API, or
    slate::eig( A, Lambda );       // simplified API
    slate::heev( A, Lambda );      // traditional API
    // A = Z Lambda Z^H, eigenvalues and eigenvectors
    slate::eig( A, Lambda, Z );    // simplified API
    slate::heev( A, Lambda, Z );   // traditional API
```

7.2.14 Generalized Hermitian/Symmetric Eigenvalues

The generalized eigenvalue problem adds a Hermitian positive definite matrix B in one of three places, set by `itype`:

(1) $Az = \lambda Bz$

(2) $ABz = \lambda z$

(3) $BAz = \lambda z$

It uses a Cholesky factorization to reduce the problem to a standard eigenvalue problem. All of the options for a standard eig apply.

Algorithm 7.31 Generalized Hermitian/symmetric eigenvalues: ex12_generalized_hermitian_eig.cc

```
    slate::HermitianMatrix<scalar_type>
        A( slate::Uplo::Lower, n, nb, grid_p, grid_q, MPI_COMM_WORLD ),
        B( slate::Uplo::Lower, n, nb, grid_p, grid_q, MPI_COMM_WORLD );
    slate::Matrix<scalar_type>
        Z( n, n, nb, grid_p, grid_q, MPI_COMM_WORLD );
    std::vector<real_t> Lambda( n );
    // ...
    // Type 1: A = B Z Lambda Z^H, eigenvalues only
    slate::eig_vals( 1, A, B, Lambda );  // simplified API, or
    slate::eig( 1, A, B, Lambda );       // simplified API
    slate::hegv( 1, A, B, Lambda );      // traditional API

    // Type 2: A B = Z Lambda Z^H, eigenvalues only
    slate::eig_vals( 2, A, B, Lambda );  // simplified API

    // Type 3: A = B Z Lambda Z^H, eigenvalues only
    slate::eig_vals( 3, A, B, Lambda );  // simplified API

    // Types 1, 2, and 3, with eigenvectors
    slate::eig( 1, A, B, Lambda, Z );    // simplified API
    slate::eig( 2, A, B, Lambda, Z );    // simplified API
    slate::eig( 3, A, B, Lambda, Z );    // simplified API
    slate::hegv( 1, A, B, Lambda, Z );   // traditional API
    slate::hegv( 2, A, B, Lambda, Z );   // traditional API
    slate::hegv( 3, A, B, Lambda, Z );   // traditional API
```

CHAPTER 8

Testing Suite for SLATE

SLATE comes with a testing suite to check the correctness and accuracy of the functionality provided by the library. The testing suite can also be used to obtain timing results for the routines. For many of the routines, the SLATE testers can be used to run a reference ScaLAPACK execution of the same routine (with some caveats with respect to threading).

Most routines using backwards error checks to check the accuracy, similar to the checks done in LAPACK. See LAPACK working note 41 for descriptions of the error formulas.

For the parallel BLAS routines, with `--ref=n`, accuracy checks are done by multiplying the expression by a random matrix X with two different parenthesizations, e.g., for `gemm`:

$$
\begin{aligned}
C_{\text{out}} &= \alpha AB + \beta C_{\text{in}}, \\
Y_1 &= \alpha A(BX) + (\beta C_{\text{in}} X), \\
Y_2 &= C_{\text{out}} X, \\
\text{error} &= \frac{\|Y_1 - Y_2\|}{\|Y_1\|}.
\end{aligned}
$$

This is fast but cannot detect all errors. With `--ref=y`, accuracy checks are done compared with the ScaLAPACK reference result. This is slower but more robust, since it relies on a different implementation for the reference solution.

For the parallel norm routines, accuracy checks are done by comparing the answer with a reference ScaLAPACK execution. Older versions of ScaLAPACK have accuracy errors in the norms that can cause apparent failures.

The SLATE test suite should be built by default in the `test` directory. A number of the tests require ScaLAPACK to run reference versions, so the build process will try to link the `tester` binary with a ScaLAPACK library.

The SLATE tests are all driven by the `TestSweeper` testing framework, which enables the tests to sweep over a combination of input choices.

8.1 SLATE Tester

Some basic examples of using the SLATE `tester` are shown here.

```
cd test
# list all the available tests
./tester --help

# do a quick test of gemm using small default settings
./tester gemm

# list the options for testing gemm
./tester --help gemm

# do a larger single-process sweep of gemm
./tester --nb 256 --dim 1000:5000:1000 gemm

# do a multi-process sweep of gemm using MPI
mpirun -n 4 ./tester --nb 256 --dim 1000:5000:1000 --grid 2x2 gemm

# do a multi-process sweep of gemm using MPI and target devices (CUDA / ROCm / oneMKL)
mpirun -n 4 ./tester --nb 256 --dim 1000:5000:1000 --target d gemm
```

The `./tester --help gemm` command will generate a list of available parameters for `gemm`. Other routines can be checked similarly.

```
> ./tester --help gemm
% SLATE version 2023.08.25, id 965f1d63
% input: ./tester --help gemm
% 2023-11-05 04:16:47, 1 MPI ranks, CPU-only MPI, 4 OpenMP threads per MPI rank
Usage: test [-h|--help]
       test [-h|--help] routine
       test [parameters] routine

Parameters for gemm:
    --check           check the results; default y; valid: [ny]
    --error-exit      check error exits; default n; valid: [ny]
    --ref             run reference; sometimes check implies ref; default n; valid: [nyo]
    --trace           enable/disable traces; default n; valid: [ny]
    --trace-scale     horizontal scale for traces, in pixels per sec; default 1000
    --tol             tolerance (e.g., error < tol*epsilon to pass); default 50
    --repeat          number of times to repeat each test; default 1
    --verbose         verbose level:
                      0: no printing (default)
                      1: print metadata only (dimensions, uplo, etc.)
                      2: print first & last edgeitems rows & cols from the four corner tiles
                      3: print 4 corner elements of every tile
                      4: print full matrix; default 0
    --print-edgeitems for verbose=2, number of first & last rows & cols to print
                      from the four corner tiles; default 16
    --print-width     minimum number of characters to print per value; default 10
    --print-precision number of digits to print after the decimal point; default 4
    --cache           total cache size, in MiB; default 20
    --debug           given rank waits for debugger (gdb/lldb) to attach; default -1

Parameters that take comma-separated list of values and may be repeated:
    --type            s=single (float), d=double, c=complex-single, z=complex-double; default d
    --origin          origin: h=Host, s=ScaLAPACK, d=Devices; default host
    --target          target: t=HostTask, n=HostNest, b=HostBatch, d=Devices; default task
    --method-gemm     auto=auto, A=gemmA, C=gemmC; default auto
    --grid-order      (go) MPI grid order: c=Col, r=Row; default col
    --matrix          test matrix kind; see 'test --help-matrix'; default 'rand'
    --cond            requested matrix condition number; default NA
    --condD           matrix D condition number; default NA
    --seed            Randomization seed (-1 randomizes the seed for each matrix); default -1
    --matrixB         test matrix kind; see 'test --help-matrix'; default 'rand'
    --condB           requested matrix condition number; default NA
    --condD_B         matrix D condition number; default NA
    --seedB           Randomization seed (-1 randomizes the seed for each matrix); default -1
    --matrixC         test matrix kind; see 'test --help-matrix'; default 'rand'
    --condC           requested matrix condition number; default NA
    --condD_C         matrix D condition number; default NA
    --seedC           Randomization seed (-1 randomizes the seed for each matrix); default -1
    --norm            norm: o=one, 2=two, i=inf, f=fro, m=max; default 1
    --transA          transpose of A: n=no-trans, t=trans, c=conj-trans; default notrans
    --transB          transpose of B: n=no-trans, t=trans, c=conj-trans; default notrans
    --dim             m x n x k dimensions
    --nrhs            number of right hand sides; default 10
    --alpha           alpha value
    --beta            beta value
    --nb              block size; default 384
    --grid            MPI grid p x q dimensions
    --lookahead       (la) number of lookahead panels; default 1
```

The SLATE tester can be used to check the accuracy and tune the performance of specific routines (e.g., `gemm`).

```
# Run gemm, single precision, targeting cpu tasks, matrix dimensions
# 500 to 2000 with step 500 and tile size 256.
./tester --type s --target t --dim 500:2000:500 --nb 256 gemm

# The following command could be used to tune tile sizes.  Run gemm,
# single precision, target devices, matrix dimensions 5000, 10000 and
# use tile sizes 192 to 512 with step 64.
./tester --type s --target d --dim 5000,10000 --nb 192:256:64 gemm

# Run distributed gemm, double precision, target devices, matrix
# dimensions 10000, use tile sizes 192 to 512 with step 64,
# and use 2x2 MPI process grid.
mpirun -n 4 ./tester --type d --target d --dim 10000 --nb 192:256:64 \
                     --grid 2x2 gemm

# Run distributed gemm, double precision, target devices, matrix
# dimensions 10000, use tile size 256, and a 1x4 MPI process grid.
mpirun -n 4 ./tester --type d --target d --dim 10000 --nb 256 \
                     --grid 1x4 gemm
```

8.2 Full Testing Suite

The SLATE tester contains a Python test driver script `run_tests.py` that can run the available routines, sweeping over combinations of parameters and running the SLATE `tester` to ensure that SLATE is functioning correctly. By default, the test driver will run the tester for all the known functions; however, it can be restricted to run only specific functions.

The `run_tests.py` script has a large number of parameters that can be passed to the `tester`.

```
cd test

# Get a list of available parameters
python3 ./run_tests.py --help

# The default full test suite used by SLATE
python3 ./run_tests.py --xml ../report_unit.xml

# Run a small run using the full testing suite
python3 ./run_tests.py --xsmall

# You can also send jobs to a job manager or use mpirun by changing
# the test command.  Run gesv tests using SLURM plus mpirun, running
# on 4 process Note, if the number of processes is a square number,
# the tester will set p and q to the root of that number.
python3 ./run_tests.py --test "salloc -N 4 -n 4 -t 10 mpirun -n 4 ./tester" \
                       --xsmall gesv

# Run on execution target devices, assuming all the nodes have NVIDIA GPUs
python3 ./run_tests.py --test "mpirun -n 4 ./tester " --xsmall --target d gesv
```

8.3 Tuning SLATE

There are several parameters that can affect the performance of SLATE routines on different architectures. The most basic parameter is the tile size nb. For execution on the CPU using OpenMP tasks (HostTask), SLATE tile sizes tend to be in the order of hundreds. A sweep over tile sizes can be used to determine the appropriate tile size for an algorithm. Note that the appropriate tile sizes are likely to vary for different execution targets and process grids.

```
cd test
# Trying tile sizes for gesv, double precision data, target HostTask
./tester --type d --target t --dim 3000 --nb 128:512:32 gesv

# Trying tile sizes for gesv, double precision data, target Devices
# For NVIDIA GPUs, nb tends to be larger and a multiple of 64
./tester --type d --target d --dim 10000 --nb 192:1024:64 gesv
```

Similarly, for a distributed execution a number of process grids may need to be tested to determine the appropriate choice. For many of SLATE algorithms, a $p \times q$ grid where $p \leq q$, but not too far from square, will work well. A 1D grid ($p = 1$ or $q = 1$) is usually bad for performance, as it leads to higher communication.

```
cd test
# Trying grid sizes for gemm, double precision data, target HostTask
mpirun -n 4 ./tester --target t --nb 256 --dim 10000 --grid 1x4,2x2 gemm
```

There are several other parameters that can be tested—for example, the algorithmic lookahead (`--lookahead 1` default is usually sufficient) and the number of threads to be used in panel operations.

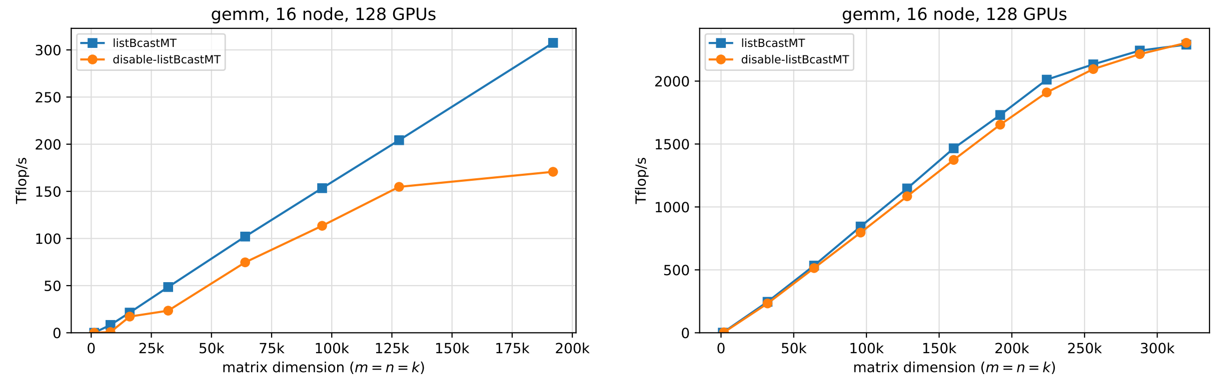
8.3.1 Enabling Multi-threaded MPI Broadcast

Sending tiles to MPI ranks in the list of submatrices during computations can be accomplished using OpenMP tasksloop and multi-threaded MPI. To enable the multithreaded MPI broadcast, the flag `CXXFLAGS += -DSLATE_HAVE_MT_BCAST` have to be added to the make.inc file. Figure 8.1 illustrates a performance comparison of `gemm` with and without enabling the multithreaded MPI broadcast. On Summit (Figure 8.1a), enabling this option results in approximately 2X performance improvements. However, on Frontier (Figure 8.1b), it exhibits similar performance to disabling it but can cause SLATE to hang when using GPU-aware MPI.

8.4 Unit Tests

SLATE also contains a set of unit tests that are used to check the functionality of smaller parts of the source code. For example, the unit tests can ensure that the SLATE memory manager, the various Matrix objects, and the Tile objects are functioning as expected. These unit tests are of more use to the SLATE developer and are not discussed in more detail here.

The unit tests also have a Python script that will run a sweep over these tests.



(a) 16 nodes of Summit (672 Power9 CPUs, 96 V100 GPUs). **(b)** 16 nodes of Frontier (896 EPYC CPUs, 128 MI250X GPUs).

Figure 8.1: Performance comparison with using listBcastMT.

```
cd unit_test
# The default unit_test run used by SLATE
python3 ./run_tests.py --xml ../report_unit.xml
```

CHAPTER 9

Compatibility APIs for ScaLAPACK and LAPACK Users

In order to facilitate easy and quick adoption of SLATE, a set of compatibility APIs is provided for routines that will allow ScaLAPACK and LAPACK functions to execute using their matching SLATE routines. SLATE can support such compatibility because the flexible tile layout adopted by SLATE was purposely designed to match LAPACK and ScaLAPACK matrix layouts.

9.1 LAPACK Compatibility API

The SLATE-LAPACK compatibility API is parameter matched to standard LAPACK calls with the function names prefaced by `slate_`. The prefacing was necessary because SLATE uses standard LAPACK routines internally, and the function names would clash if the SLATE-LAPACK compatibility API used the standard names.

Each supported LAPACK routine (e.g., `gemm`) added to the compatibility library provides interfaces for all data types (single, double, single complex, double complex, mixed) that may be required. These interfaces (e.g., `slate_sgemm`, `slate_dgemm`) call a type-generic routine that sets up other SLATE requirements.

The LAPACK data is then mapped to a SLATE matrix type using a support routine `fromLAPACK`. SLATE requires a block/tile size (`nb`) because SLATE algorithms view matrices as composed of tiles of data. This tiling does not require the LAPACK data to be moved; it is a view on top of the pre-existing LAPACK data layout.

SLATE will attempt to manage the number of available threads such that threads are used to generate and manage tasks and the internal lower-level BLAS calls all run single threaded. These settings may need to be altered to support different BLAS libraries since each library may have its own methods for controlling the threads used for BLAS computations.

The SLATE execution target (e.g., HostTask, Devices, ...) is not something available from the LAPACK function parameters (e.g. dgemm). The execution target information defaults to HostTask (running on the CPUs), but the user can specify the execution target to the compatibility routine using environment variables, allowing the LAPACK call (e.g., slate_dgemm) to execute on Device/GPU targets.

The compatibility library will then call the SLATE version of the routine (slate::gemm) and execute it on the selected target.

Algorithm 9.1 LAPACK-compatible API.

C example

```
// Compile with, e.g.,
//     mpicc -o example example.c -lslate_lapack_api

// Original call to LAPACK
dgetrf_( &m, &n, A, &lda, ipiv, &info );

// New call to SLATE
slate_dgetrf_( &m, &n, A, &lda, ipiv, &info );
```

Fortran example

```
!! Compile with, e.g.,
!!     mpif90 -o example example.f90 -lslate_lapack_api

!! Original call to LAPACK
call dgetrf( m, n, A, lda, ipiv, info )

!! New call to SLATE
call slate_dgetrf( m, n, A, lda, ipiv, info )
```

9.2 ScaLAPACK Compatibility API

The SLATE-ScaLAPACK compatibility API is intended to be link-time compatible with standard ScaLAPACK, matching both function names and parameters to the degree possible.

Each supported ScaLAPACK routine (e.g., gemm) has interfaces for all the supported data types (e.g., pdgemm, psgemm) and all the standard Fortran name manglings (i.e., uppercase, lowercase, added underscore). So, a call to a ScaLAPACK function will be intercepted using a function name expected by the end user.

All the defined Fortran interface routines (e.g., pdgemm, PDGEMM, pdgemm_) call a single type-generic SLATE function that sets up the translation between the ScaLAPACK and SLATE parameters. The ScaLAPACK matrix data can be mapped to SLATE matrix types using a support function fromScaLAPACK provided by SLATE. This mapping does not move the ScaLAPACK data from its original locations. A SLATE matrix structure is defined, referencing the ScaLAPACK data using the ScaLAPACK blocking factor to define SLATE tiles. Note: SLATE algorithms tend to perform better at larger block sizes, especially on GPU devices, so it is preferable if ScaLAPACK uses a larger blocking factor.

The SLATE execution target (e.g., HostTask, Devices, ...) defaults to HostTask (running on the CPUs) but the user can specify the execution target to the compatibility routine using environment variables. This allows an end user to use ScaLAPACK and SLATE within the same executable. ScaLAPACK functions that have an analog in SLATE will benefit from any algorithmic or GPU speedup, and any functions that are not yet in SLATE will transparently fall through to the pre-existing ScaLAPACK implementations.

Algorithm 9.2 ScaLAPACK-compatible API.

C example

```c
// Compile with, e.g.,
//     mpicc -o example example.c -lslate_scalapack_api -lscalapack

// Call to ScaLAPACK will be intercepted by SLATE
pdgetrf_( &m, &n, A, &ia, &ja, descA, ipiv, &info );
```

Fortran example

```fortran
!! Compile with, e.g.,
!!     mpif90 -o example example.f90 -lslate_scalapack_api -lscalapack

!! Call to ScaLAPACK will be intercepted by SLATE
call pdgetrf( m, n, A, ia, ja, descA, ipiv, info )
```

Bibliography

[1] Ali Charara, Mark Gates, Jakub Kurzak, Asim YarKhan, and Jack Dongarra. SLATE Developers' Guide, SWAN no. 11. Technical Report ICL-UT-19-02, Innovative Computing Laboratory, University of Tennessee, December 2019. revision 12-2019.

[2] Ahmad Abdelfattah, Hartwig Anzt, Aurelien Bouteiller, Anthony Danalis, Jack Dongarra, Mark Gates, Azzam Haidar, Jakub Kurzak, Piotr Luszczek, Stanimire Tomov, Stephen Wood, Panruo Wu, Ichitaro Yamazaki, and Asim YarKhan. Roadmap for the Development of a Linear Algebra Library for Exascale Computing: SLATE: Software for Linear Algebra Targeting Exascale. SLATE Working Notes 1, ICL-UT-17-02, 2017. URL http://www.icl.utk.edu/publications/swan-001.

[3] Mark Gates, Piotr Luszczek, Ahmad Abdelfattah, Jakub Kurzak, Jack Dongarra, Konstantin Arturov, Cris Cecka, and Chip Freitag. C++ API for BLAS and LAPACK. Technical Report ICL-UT-17-03, SLATE Working Note 2, Innovative Computing Laboratory, University of Tennessee, 2017. URL https://www.icl.utk.edu/publications/swan-002.

[4] Wolfgang Hackbusch. A sparse matrix arithmetic based on H-Matrices. part i: Introduction to H-Matrices. *Computing*, 62:89–108, 1999. doi:https://doi.org/10.1007/s006070050015.

[5] Steffen Börm, Lars Grasedyck, and Wolfgang Hackbusch. Introduction to hierarchical matrices with applications. *Engineering analysis with boundary elements*, 27(5):405–422, 2003. doi:https://doi.org/10.1016/S0955-7997(02)00152-2.

[6] Wolfgang Hackbusch, Steffen Börm, and Lars Grasedyck. *HLib 1.4*. Max-Planck-Institut, Leipzig, 2012. URL http://www.hlib.org.

[7] Clément Weisbecker. *Improving multifrontal solvers by means of algebraic block low-rank representations*. PhD thesis, Institut National Polytechnique de Toulouse-INPT, 2013. URL https://tel.archives-ouvertes.fr/tel-00934939/.

[8] Fred Gustavson, André Henriksson, Isak Jonsson, Bo Kågström, and Per Ling. Recursive blocked data formats and BLAS's for dense linear algebra algorithms. *Applied Parallel Computing Large Scale Scientific and Industrial Problems*, 1541:195–206, 1998. doi:https://doi.org/10.1007/BFb0095337.

[9] Fred G Gustavson, Jerzy Waśniewski, Jack J Dongarra, and Julien Langou. Rectangular full packed format for cholesky's algorithm: factorization, solution, and inversion. *ACM Transactions on Mathematical Software (TOMS)*, 37(2):18, 2010. doi:https://doi.org/10.1145/1731022.1731028.

[10] *Introducing the new Packed APIs for GEMM*. Intel Corp., 2016. URL https://software.intel.com/en-us/articles/introducing-the-new-packed-apis-for-gemm.

[11] Fred Gustavson, Lars Karlsson, and Bo Kågström. Parallel and cache-efficient in-place matrix storage format conversion. *ACM Transactions on Mathematical Software (TOMS)*, 38(3):17, 2012. doi:https://doi.org/10.1145/2168773.2168775.

[12] Stefan Kurz, Oliver Rain, and Sergej Rjasanow. The adaptive cross-approximation technique for the 3d boundary-element method. *IEEE Transactions on Magnetics*, 38(2):421–424, 2002. doi:https://doi.org/10.1109/20.996112.

[13] Jack Dongarra, Mark Gates, Azzam Haidar, Jakub Kurzak, Piotr Luszczek, Panruo Wu, Ichitaro Yamazaki, Asim YarKhan, Maksims Abalenkovs, Negin Bagherpour, Sven Hammarling, Jakub Šíšístek, David Stevens, Mawussi Zounon, and Samuel Relton. PLASMA: Parallel Linear Algebra Software for Multicore Using OpenMP. *ACM Transactions on Mathematical Software (TOMS)*, 45:16:1–16:35, 2019. doi:https://doi.org/10.1145/3264491.

[14] Wei Wu, Aurelien Bouteiller, George Bosilca, Mathieu Faverge, and Jack Dongarra. Hierarchical DAG scheduling for hybrid distributed systems. In *2015 IEEE International Parallel and Distributed Processing Symposium*, pages 156–165. IEEE, 2015. doi:https://doi.org/10.1109/IPDPS.2015.56.

[15] Jakub Kurzak, Piotr Luszczek, Ichitaro Yamazaki, Yves Robert, and Jack Dongarra. Design and implementation of the PULSAR programming system for large scale computing. *Supercomputing Frontiers and Innovations*, 4(1):4–26, 2017. doi:http://dx.doi.org/10.14529/jsfi170101.

[16] Laura Grigori, James W Demmel, and Hua Xiang. CALU: a communication optimal LU factorization algorithm. *SIAM Journal on Matrix Analysis and Applications*, 32(4):1317–1350, 2011. doi:10.1137/100788926.

[17] Neil Lindquist, Mark Gates, Piotr Luszczek, and Jack Dongarra. Threshold pivoting for dense LU factorization. In *2022 IEEE/ACM Workshop on Latest Advances in Scalable Algorithms for Large-Scale Heterogeneous Systems (ScalAH)*, pages 34–42. IEEE, 2022. doi:10.1109/ScalAH56622.2022.00010.