SLATE Users' Guide

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

Innovative Computing Laboratory

July 7, 2020

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

```
@techreport{gates2020slate,
  author={Gates, Mark and Charara, Ali and Kurzak, Jakub and YarKhan, Asim
          and Al Farhan, Mohammed and Sukkari, Dalal 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 07-2020}
}
```

Contents

Contents ii

List of Figures v

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 License and Commercial Use of SLATE . . . 5
- 2.7 Support for SLATE . . . 6

3 Getting Started with SLATE 7
- 3.1 Quick Installation of SLATE . . . 7
- 3.2 Source Code for Example Program 1 . . . 8
- 3.3 Details of Example Program 1 . . . 11
- 3.4 Simplifying Assumptions Used in Example Program 1 . . . 11
- 3.5 Building and Running Example Program 1 . . . 12

4 Design and Fundamentals of SLATE 13
- 4.1 Design Principles . . . 13
  - 4.1.1 Matrix Layout . . . 14
  - 4.1.2 Parallelism Model . . . 16

5 SLATE API 19
- 5.1 Simplified C++ API . . . 19
  - 5.1.1 BLAS and auxiliary . . . 19

5.1.2 Linear systems . . . 20
5.1.3 Unitary factorizations . . . 21
5.1.4 Eigenvalue and singular value decomposition . . . 22
5.2 C and Fortran API . . . 23
5.2.1 BLAS and auxiliary . . . 23
5.2.2 Linear systems . . . 24
5.2.3 Unitary factorizations . . . 24
5.2.4 Eigenvalue and singular value decomposition . . . 25
5.3 Traditional (Sca)LAPACK API . . . 26

6 Using SLATE **29**
6.1 Matrices in SLATE . . . 29
6.1.1 Matrix Hierarchy . . . 29
6.1.2 Creating and Accessing Matrices . . . 32
6.1.3 Matrices from ScaLAPACK and LAPACK . . . 34
6.1.4 Matrix Transpose . . . 35
6.1.5 Tile Submatrices . . . 36
6.1.6 Matrix Slices . . . 36
6.1.7 Deep Matrix Copy . . . 36
6.2 Using SLATE Functions . . . 36
6.2.1 Execution Options . . . 37
6.2.2 Matrix Norms . . . 37
6.2.3 Matrix-Matrix Multiply . . . 38
6.2.4 Operations with Triangular Matrices . . . 39
6.2.5 Operations with Band Matrices . . . 39
6.2.6 Linear Systems: General non-symmetric square matrices (LU) . . . 40
6.2.7 Linear Systems: Symmetric positive definite (Cholesky) . . . 40
6.2.8 Linear Systems: Hermitian indefinite (Aasen's) . . . 41
6.2.9 Least squares: AX ≈ B using QR or LQ . . . 42
6.2.10 Mixed Precision Routines . . . 43
6.2.11 Matrix Inverse . . . 43
6.2.12 Singular Value Decomposition . . . 44
6.2.13 Hermitian (symmetric) eigenvalues . . . 44

7 Installation Instructions **46**
7.1 Makefile based build . . . 46
7.2 CMake . . . 48
7.3 Spack . . . 48

8 Testing Suite for SLATE **49**
8.1 SLATE Tester . . . 49
8.2 Full Testing Suite . . . 51
8.3 Tuning SLATE . . . 52
8.4 Unit Tests . . . 52

9 Compatibility APIs for ScaLAPACK and LAPACK Users **53**
9.1 LAPACK Compatibility API 53
9.2 ScaLAPACK Compatibility API 54

Bibliography **56**

List of Figures

2.1 Software layers in SLATE. 4
2.2 Broadcast of tile and its symmetric image to nodes owning a block row and block column in a symmetric matrix. 4

4.1 SLATE Software Stack. 13
4.2 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. 14
4.3 View of symmetric matrix on process (0, 0) in 2 $\times$ 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). 15
4.4 Block sizes can vary. Most algorithms require square diagonal tiles. 16
4.5 Tasks in Cholesky factorization. Arrows depict dependencies. 17

6.1 Matrix hierarchy in SLATE. Algorithms require the appropriate types for their operation. 31
6.2 Matrix layout of ScaLAPACK (left) and layout with contiguous tiles (right). SLATE matrix and tiles structures are flexible and accommodate multiple layouts. 34

List of Algorithms

3.1 LU solve: slate_lu.cc (1 of 3) . . . 8
3.2 LU solve: slate_lu.cc (2 of 3) . . . 9
3.3 LU solve: slate_lu.cc (3 of 3) . . . 10
6.1 Conversions: slate01_conversion.cc . . . 31
6.2 Creating matrices . . . 32
6.3 SLATE allocating CPU memory for a matrix. . . . 32
6.4 SLATE allocating GPU memory for a matrix. . . . 33
6.5 Inserting tiles using user-defined data. . . . 33
6.6 Accessing tile elements . . . 34
6.7 Creating matrix from ScaLAPACK-style data. . . . 35
6.8 Creating matrix from LAPACK-style data. . . . 35
6.9 Conjugating matrices. . . . 35
6.10 Sub-matrices: slate02_submatrix.cc . . . 36
6.11 Matrix slice: slate02_submatrix.cc . . . 36
6.12 Deep matrix copy . . . 36
6.13 Options . . . 37
6.14 multiply (gemm) options. . . . 37
6.15 Norms: slate03_norm.cc . . . 38
6.16 Parallel BLAS: slate04_blas.cc . . . 38
6.17 Parallel BLAS, triangular: slate04_blas.cc . . . 39
6.18 Band operations . . . 40
6.19 LU solve: slate05a_linear_system_lu.cc . . . 40
6.20 Cholesky solve: slate05b_linear_system_cholesky.cc . . . 41
6.21 Indefinite solve: slate05c_linear_system_hesv.cc . . . 41
6.22 Least squares (overdetermined): slate06_least_squares.cc . . . 42
6.23 Least squares (underdetermined): slate06_least_squares.cc . . . 43
6.24 Mixed precision . . . 43
6.25 LU inverse: slate05a_linear_system_lu.cc . . . 44
6.26 Cholesky inverse: slate05b_linear_system_cholesky.cc . . . 44
6.27 SVD: slate07_svd.cc . . . 44

6.28 Hermitian/symmetric eigenvalues: slate08_hermitian_eig.cc 45
9.1 LAPACK compatible API 54
9.2 ScaLAPACK compatible API 55

CHAPTER 1

Introduction

SLATE (Software for Linear Algebra Targeting Exascale) [1] is being developed as part of the Exascale Computing Project (ECP) [2], which is a joint project of the U.S. Department of Energy's Office of Science and National Nuclear Security Administration (NNSA). The objective of SLATE is to provide fundamental dense linear algebra capabilities to the US 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 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 revision 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 (Software for Linear Algebra Targeting Exascale) 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 the parallel BLAS; linear systems using LU and Cholesky; least squares problems using QR; and eigenvalue and singular value problems. SLATE is designed as a replacement for ScaLAPACK, which after two decades of operation, cannot adequately be retrofitted for modern accelerated architectures. SLATE uses modern techniques such as communication-avoiding algorithms, lookahead panels to overlap communication and computation, and task-based scheduling, along with a modern C++ framework.

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

The SLATE software can be downloaded from:
https://bitbucket.org/icl/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 and solve and the associated band routines.

SLATE is intended to fulfill the following design goals:

Targets modern HPC hardware consisting of a large number of nodes with multicore processors and several hardware accelerators per node.

Achieves portable high performance by relying on vendor optimized standard BLAS, batched BLAS, and 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.

Provides 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.

Facilitates 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.

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

Eases 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. The *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-3 BLAS on GPUs (via CUDA's cuBLAS at present). If Level-3 Batched BLAS routines exist, they are provided on the CPU and GPU as well.

[1] https://bitbucket.org/icl/testsweeper

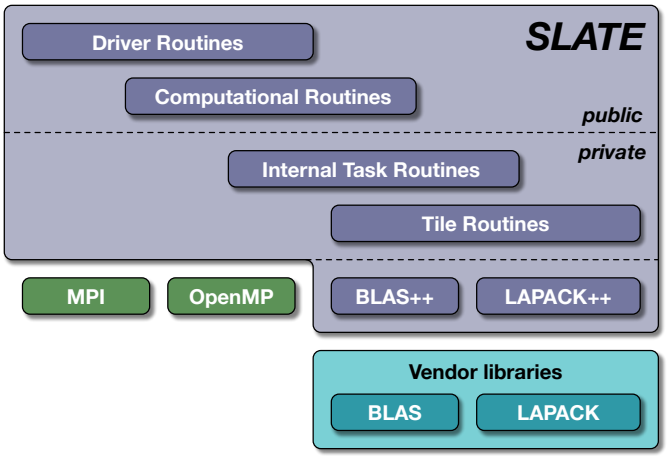

Figure 2.1: Software layers in SLATE.

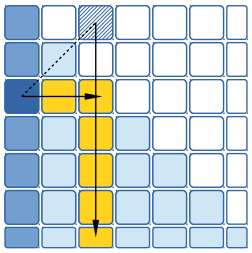

Figure 2.2: Broadcast of tile and its symmetric image to nodes owning a block row and block column in a symmetric matrix.

The *LAPACK*++ library provides similar datatype independent overloaded C++ wrappers around the Fortran *LAPACK* routines. 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 is obtained using OpenMP constructs. More specifically, OpenMP task-depend clauses are used to specify high level dependencies in SLATE algorithms and OpenMP task-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. The Batched BLAS is an emerging standard technique for aggregating smaller BLAS operations to efficiently use the hardware and obtaining higher performance.

MPI is used for communication between processes in SLATE. 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 2.2 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 destination tiles, which are internally mapped by SLATE to explicit MPI ranks.

2.4 Computers for which SLATE is Suitable

SLATE is primarily designed to solve the problem of running dense linear algebra problems on large, distributed-memory machines where the compute power in each node may be in accelerators. The nodes are currently expected to be homogeneous in their compute capabilities. SLATE is also expected to run on well on single-node multi-processor machines (with or without accelerators), however there may be 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. Source code, papers and documentation for SLATE can be found via the SLATE website.
https://icl.utk.edu/slate/

You may also download directly from the SLATE source repository.
https://bitbucket.org/icl/slate/

SLATE can also be installed be downloaded and installed using the Spack scientific software package manager using `spack install slate` or some variant of that command.

2.6 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 ask that proper credit be given to the authors.

Like all software, SLATE software 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 should 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:

Copyright ©2017–2020, University of Tennessee. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

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

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the name of the University of Tennessee nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

This software is provided by the copyright holders and contributors "as is" and any express or implied warranties, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose are disclaimed. In no event shall the copyright holders or contributors be liable for any direct, indirect, incidental, special, exemplary, or consequential damages (including, but not limited to, procurement of substitute goods or services; loss of use, data, or profits; or business interruption) however caused and on any theory of liability, whether in contract, strict liability, or tort (including negligence or otherwise) arising in any way out of the use of this software, even if advised of the possibility of such damage.

2.7 Support for SLATE

General support for SLATE can be obtained by visiting the *SLATE User Forum* at https://groups.google.com/a/icl.utk.edu/forum/#!forum/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 Issues tracker which can be found at the SLATE repository.
https://bitbucket.org/icl/slate/

CHAPTER 3

Getting Started with SLATE

3.1 Quick Installation of SLATE

This section presents the instructions for installing SLATE and running one simple test from the SLATE test suite, using a distributed memory machine with CUDA GPUs. In this example, Make is used to build SLATE rather than using CMake. Detailed installation options can be found later in this document.

- Fetch SLATE from the repository. The recursive flag will fetch the BLAS++, LAPACK++ and TestSweeper submodules.

```
> git clone --recursive https://bitbucket.org/icl/slate.git
> cd slate
```

- SLATE builds with BLAS++, LAPACK++, TestSweeper, MPI and CUDA (optional). The testers use ScaLAPACK for reference runs. All these libraries and their headers should be available in the search paths. One way is to set the `LIBRARY_PATH` and `LD_LIBRARY_PATH` for the library locations and `CPATH` for the include locations.

- Create a 'make.inc' file to set a few variables (see top of `slate/GNUmakefile` for all options). For instance, the following will build an MPI and CUDA enabled library:

```
> cat make.inc
CXX     = mpicxx    # C++ compiler, implies that MPI is enabled
FC      = mpif90    # Fortran compiler (currently required)
blas    = mkl       # One of: mkl essl openblas
cuda_arch = volta   # One or more of: kepler maxwell pascal volta turing
```

Algorithm 3.1 LU solve: slate_lu.cc (1 of 3)

```
#include <slate/slate.hh>
#include <blas.hh>
#include <mpi.h>
#include <stdio.h>

// Forward function declarations
template <typename scalar_type>
void lu_example( int64_t n, int64_t nrhs, int64_t nb, int64_t p, int64_t q );

template <typename matrix_type>
void random_matrix( matrix_type& A );


int main( int argc, char** argv )
{
    // Initialize MPI, require MPI_THREAD_MULTIPLE support
    int err=0, mpi_provided=0;
    err = MPI_Init_thread( &argc, &argv, MPI_THREAD_MULTIPLE, &mpi_provided );
    assert( err == 0 && mpi_provided == MPI_THREAD_MULTIPLE );

    // Call the LU example
    int64_t n=5000, nrhs=1, nb=256, p=2, q=2;
    lu_example< double >( n, nrhs, nb, p, q );

    err = MPI_Finalize();
    assert( err == 0 );
}
```

- Build SLATE using GNU Make. The BLAS++ and LAPACK++ libraries will also be configured and built by the SLATE build process as needed.

```
> make
```

- Run a basic gemm tester on 4 distributed nodes using your local job launcher. This will produce output that tells if the test passed.

```
# Using command line mpirun
> mpirun -n 4 ./test/tester gemm

# Using Slurm job manger
> salloc -N 4 -n 4 mpirun -n 4 ./test/tester gemm
```

3.2 Source Code for Example Program 1

The following example program will setup SLATE matrices and solve a linear system `AX = B` using the SLATE LU solver by calling a distributed gesv implementation.

Algorithm 3.2 LU solve: slate_lu.cc (2 of 3)

```
template <typename scalar_t>                                                    29
void lu_example( int64_t n, int64_t nrhs, int64_t nb, int64_t p, int64_t q)    30
{                                                                               31
    // Get associated real type, e.g., double for complex<double>.             32
    using real_t = blas::real_type<scalar_t>;                                   33
    using llong = long long;  // guaranteed >= 64 bits                         34
    const scalar_t one = 1;                                                     35
    int err=0, mpi_size=0, mpi_rank=0;                                          36
                                                                                37
    // Get MPI size, must be p*q for this example                               38
    err = MPI_Comm_size( MPI_COMM_WORLD, &mpi_size );                           39
    assert( err == 0 );                                                         40
    if (mpi_size != p*q) {                                                      41
        printf( "Usage: mpirun -np %lld ... # %lld ranks hard coded\n",         42
                llong( p*q ), llong( p*q ) );                                   43
        assert( mpi_size == p*q );                                              44
    }                                                                           45
                                                                                46
    // Get MPI rank                                                             47
    err = MPI_Comm_rank( MPI_COMM_WORLD, &mpi_rank );                           48
    assert( err == 0 );                                                         49
                                                                                50
    // Create SLATE matrices A and B                                            51
    slate::Matrix<scalar_t> A( n, n,    nb, p, q, MPI_COMM_WORLD );             52
    slate::Matrix<scalar_t> B( n, nrhs, nb, p, q, MPI_COMM_WORLD );             53
                                                                                54
    // Allocate local space for A, B on distributed nodes                       55
    A.insertLocalTiles();                                                       56
    B.insertLocalTiles();                                                       57
                                                                                58
    // Set random seed so data is different on each rank                        59
    srand( 100 * mpi_rank );                                                    60
    // Initialize the data for A, B                                             61
    random_matrix( A );                                                         62
    random_matrix( B );                                                         63
                                                                                64
    // For residual error check,                                                65
    // create A0 as an empty matrix like A and copy A to A0                     66
    slate::Matrix<scalar_t> A0 = A.emptyLike();                                 67
    A0.insertLocalTiles();                                                      68
    slate::copy( A, A0 );                                                       69
    // Create B0 as an empty matrix like B and copy B to B0                     70
    slate::Matrix<scalar_t> B0 = B.emptyLike();                                 71
    B0.insertLocalTiles();                                                      72
    slate::copy( B, B0 );                                                       73
```

Algorithm 3.3 LU solve: slate_lu.cc (3 of 3)

```
    slate::Pivots pivots;
    slate::Options opts = {{slate::Option::Target, slate::Target::HostTask}};
    double time = omp_get_wtime();
    slate::gesv( A, pivots, B, opts);
    time = omp_get_wtime() - time;

    // Compute residual ||A0 * X  - B0|| / ( ||X|| * ||A|| * N )
    real_t A_norm = slate::norm( slate::Norm::One, A );
    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();
    int status_ok = (residual < tol);

    if (mpi_rank == 0) {
        printf("gesv n %lld nb %lld pxq %lldx%lld "
               "residual %.2e tol %.2e sec %.2e ",
               llong( n ), llong( nb ), llong( p ), llong( q ),
               residual, tol, time);
        if (status_ok) printf("PASS\n");
        else printf("FAILED\n");
    }
}

// put random data in matrix 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))) * 100;
                    }
                }
            }
        }
    }
}
```

3.3 Details of Example Program 1

The example program in Algorithms 3.1 to 3.3 shows how to setup matrices in SLATE and to call several SLATE routines to operate on those matrices. This example uses the SLATE LU solver `gesv` 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 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 in ScaLAPACK, however the distribution can be changed within SLATE.

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

After the call to the SLATE `gesv` solver the distributed B matrix will contain the solution vector X. A residual check is performed to verify the accuracy of the results:

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

The call to the `gesv` solver uses an optional parameter (`opts`) that is used 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 be used to set a number of internal variables and is used here to give an example of how to pass options to SLATE.

3.4 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 x 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. SLATE can specify other data distributions by using C++ lambda functions when defining the matrix.
- Execution target `slate::Target::HostTask`: The execution will happen on the host using OpenMP tasks (the default if no target is specified). Other execution targets like `Device` for GPU accelerator devices may be preferable.

3.5 Building and Running Example Program 1

The code in Example 1 requires SLATE header files, and from there it needs the BLAS++ and LAPACK++ header files to build. The shell commands below set up the locations of these headers. Note, if SLATE was compiled with -fopenmp, -DSLATE_NO_MPI, or -DSLATE_NO_CUDA, those flags must be used during compile commands. If SLATE was built as a shared library with all the paths to its dependencies embedded, than the link command below will pull in all the dependencies. Otherwise additional library paths and linked libraries may be required to build the example.

```
# locations of slate, blas++, lapack++
export SLATE_ROOT=/path/to/slate/directory
export BLASPP_ROOT=$SLATE_ROOT/blaspp      # or $SLATE_ROOT, if installed
export LAPACKPP_ROOT=$SLATE_ROOT/lapackpp  # or $SLATE_ROOT, if installed

# compile the example
# NB: if SLATE was compiled with -fopenmp, -DSLATE_NO_MPI, or
# -DSLATE_NO_CUDA, those must be used during compile and link commands
mpicxx -fopenmp -I${SLATE_ROOT}/include -I${BLASPP_ROOT}/include \
    -I${LAPACKPP_ROOT}/include -c slate_lu.cc

# if some libraries cannot be found using dynamic rpath magic,
# BLASPP and LAPACKPP paths may need to be added
mpicxx -fopenmp -o slate_lu slate_lu.o \
    -L${SLATE_ROOT}/lib -Wl,-rpath,${SLATE_ROOT}/lib \
    -lslate -lcublas -lcudart

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

# output from the run will be something like the following
gesv n 5000 nb 256 pxq 2x2 residual 2.50e-20 tol 2.22e-16 sec 3.37e+00 PASS
```

CHAPTER 4

Design and Fundamentals of SLATE

4.1 Design Principles

Figure 4.1 shows the SLATE software stack, designed after a careful consideration of all available implementation technologies [2]. The objective of SLATE is to provide dense linear algebra capabilities to the ECP applications, e.g., EXAALT, NWChemEx, QMCPACK, GAMESS, as well as other software libraries and frameworks, e.g., FBSS. In that regard, SLATE is intended as a replacement for ScaLAPACK, with superior performance and scalability in distributed memory environments with multicore processors and hardware accelerators.

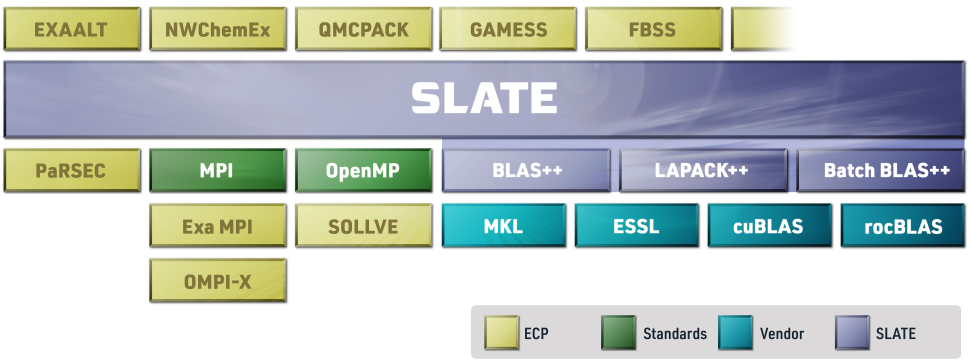

Figure 4.1: SLATE Software Stack.

The SLATE project also encompasses the design and implementation of C++ APIs for BLAS and LAPACK [3], and for batched BLAS. Underneath these APIs, highly optimized vendor libraries will be called for maximum performance (Intel MKL, IBM ESSL, NVIDIA cuBLAS,

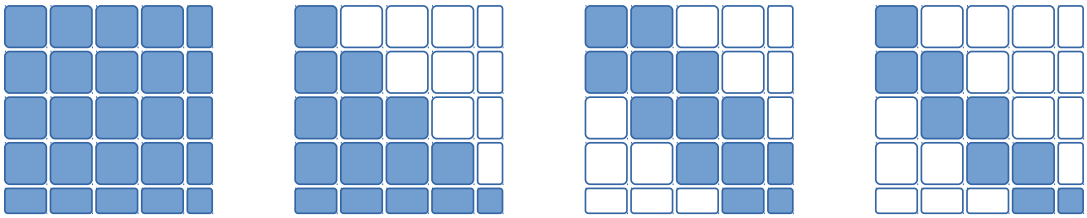

Figure 4.2: General, symmetric, band, and symmetric band matrices. Only shaded tiles are stored; blank tiles are implicitly zero or known by symmetry, so are not stored.

AMD rocBLAS, etc.).

To maximize portability, the current design relies on the MPI standard for message passing, and the OpenMP standard for multithreading and offload to hardware accelerators. The collaborations with the ECP Exa-MPI and OMPI-X projects are intended to improve message passing capabilities, while the collaboration with the ECP SOLLVE project is intended to improve multithreading capabilities.

4.1.1 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 DPLASMA, where the local matrix occupies a contiguous memory region on each process. While DPLASMA uses tiled algorithms, the tiles are stored in one contiguous memory block on each process. 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. At the same time, SLATE also 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:

First, the same structure can be used for holding many different matrix types: general, symmetric, triangular, band, symmetric band, etc., as shown in Figure 4.2. Little memory is wasted for storing parts of the matrix that hold no useful data, e.g., the upper triangle of a lower triangular matrix. Instead of wasting $O(n^2)$ memory as ScaLAPACK does, only $O(nn_b)$ memory is wasted 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],

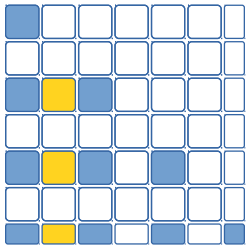

Figure 4.3: View of symmetric matrix on process (0, 0) in 2 × 2 process grid. Darker blue tiles are local to process (0, 0); lighter yellow tiles are temporary workspace tiles copied from remote process (0, 1).

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 layout translation and transposition algorithms [11].

Also, tiles can be easily allocated and copied among different memory spaces. Both inter-node communication and intra-node communication is 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 >,                    1
          TileNode<scalar_t>* >                              2
```

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 4.3. 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 4.3 for the trailing matrix update in Cholesky, where portions of the remote panel (yellow) have been copied locally.

Finally, SLATE can support non-uniform tile sizes (Figure 4.4). Most factorizations require that the diagonal tiles are square, but the block row heights and block column widths can, in principle, be arbitrary. The current implementation has a fixed tile size, but the structure does not require it. This will facilitate applications where the block structure is significant, for

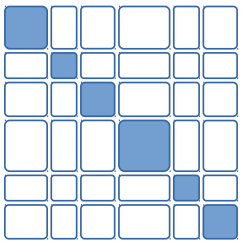

Figure 4.4: Block sizes can vary. Most algorithms require square diagonal tiles.

instance in *Adaptive Cross Approximation* (ACA) linear solvers [12].

4.1.2 Parallelism Model

SLATE utilizes three or 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 SIMD vector instructions. For multicore, 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 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) primarily reliance on the 2D block cyclic distribution for scalability, 5) reliance on the `gemm` operation, specifically its batched rendition, for maximum hardware utilization.

The Cholesky factorization demonstrates the basic framework, with its task graph shown in Figure 4.5. Dataflow tasking (`omp task depend`, is used for scheduling operations with dependencies on large blocks of the matrix. Within each large block, either nested tasking (`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 using accelerators.

Compared to pure tile-by-tile dataflow scheduling, as 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] in the case of manycore processors such as the Xeon Phi family. In contrast, the SLATE 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 vs. on accelerators [14].

At each step of Cholesky, one or more columns of the trailing submatrix are prioritized for

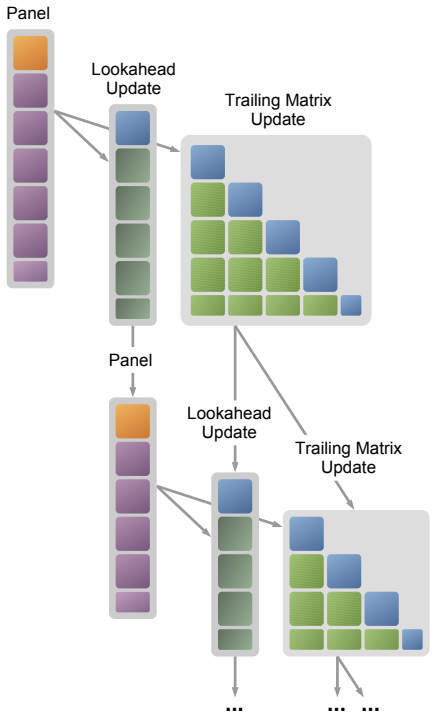

Figure 4.5: Tasks in Cholesky factorization. Arrows depict dependencies.

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 it is proportional to the amount of extra memory required for storing temporary tiles. 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 multicore 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`).

To motivate our choices of CPU tasks on individual tiles and GPU tasks using batches of tiles, we examine the performance of `dgemm`. Libraries such as DPLASMA and Chameleon have

demonstrated that on CPUs doing operations on a tile-by-tile basis can achieve excellent CPU performance.

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 will be abstracted at a low level in the BLAS++ library to ease porting. There are very few accelerator (e.g., CUDA) kernels in SLATE – currently just matrix norms and transposition – so porting should be a lightweight task.

CHAPTER 5

SLATE API

Initially, SLATE used the traditional (Sca)LAPACK naming scheme for routines, with minor variations. In 2020, we introduced a new simplified scheme that we recommend applications use.

5.1 Simplified C++ API

Given the rather cryptic nature of the traditional BLAS/LAPACK naming scheme, and that SLATE can identify the matrix type from the matrix classes in arguments, we are proposing a new simplified C++ API.

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

5.1.1 BLAS and auxiliary

These routines perform basic operations such as matrix-multiply and norms. For simplicity, transpose and conjugate-transpose operations are not shown here, but are handled in the matrix classes.

C++ API	**traditional**	**operation**	**matrix type**
`multiply`	`gemm`	$C = \alpha AB + \beta C$	A, B, C all general
`multiply`	`hemm`	$C = \alpha AB + \beta C$	A or B Hermitian
`multiply`	`symm`	$C = \alpha AB + \beta C$	A or B symmetric
`rank_k_update`	`herk`	$C = \alpha AA^H + \beta C$	C Hermitian
`rank_k_update`	`syrk`	$C = \alpha AA^H + \beta C$	C symmetric
`rank_2k_update`	`her2k`	$C = \alpha AB^H + \bar{\alpha} BA^H + \beta C$	C Hermitian
`rank_2k_update`	`syr2k`	$C = \alpha AB^H + \alpha BA^H + \beta C$	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
`norm`	`lan`	$\|A\|_1$, $\|A\|_{\text{inf}}$, $\|A\|_{\text{fro}}$, $\|A\|_{\max}$	any
`copy`	`lacpy`	$B = A$	any
`copy`	`lag2`	$B = A$, precision conversion	any
`set`	`laset`	$A_{ij} = \begin{cases} \alpha & \text{if } i \neq j, \\ \beta & \text{if } i = j \end{cases}$	any

5.1.2 Linear systems

Depending on the method, these routines factor a matrix into a lower triangular matrix L, upper triangular matrix U, permutation matrix P, and band matrix T. The factored matrix is then used to solve $AX = B$. For least squares, a unitary factorization is used.

To solve one linear system per matrix A, potentially with multiple right-hand sides (RHS), the `*_solve` drivers are recommended. To factor a matrix once, then solve different systems, such as in a time-stepping loop, call `*_factor` once, then repeatedly call `*_solve_using_factor`.

C++ API	traditional	operation
General non-symmetric (LU)		
`lu_solve`	`gesv`	Solve $AX = B$ using LU
`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_cond_using_factor`	`gecon`	Est. $\|A\|_p \cdot \|A^{-1}\|_p$ for $p \in \{1, \text{inf}\}$
Hermitian/symmetric positive definite (Cholesky)		
`chol_solve`	`posv`	Solve $AX = B$ using Cholesky
`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_cond_using_factor`	`pocon`	Est. $\|A\|_p \cdot \|A^{-1}\|_p$ for $p \in \{1, \text{inf}\}$
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`	`hetri, sytri`	Form $A^{-1} = (LTL^H)^{-1}$
`indefinite_cond_using_factor`	`sycon`	Est. $\|A\|_p \cdot \|A^{-1}\|_p$ for $p \in \{1, \text{inf}\}$
Least squares		
`least_squares_solve`	`gels*`	Solve $AX \approx B$ for rectangular A

5.1.3 Unitary factorizations

These 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 (Sca)LAPACK's.

For simplicity, the conjugate-transpose operation for multiplying by Q is not shown here, but is implemented.

C++ API	traditional	operation
`qr_factor`	`geqrf`	Factor $A = QR$
`qr_multiply_by_q`	`gemqr`	Multiply $C = QC$ or $C = CQ$
`qr_generate_q` [1]	`gegqr`	Form Q
`lq_factor`	`gelqf`	Factor $A = LQ$
`lq_multiply_by_q`	`gemlq`	Multiply $C = QC$ or $C = CQ$
`lq_generate_q` [1]	`geglq`	Form Q
`rq_factor` [1]	`gerqf`	Factor $A = RQ$
`rq_multiply_by_q` [1]	`gemrq`	Multiply $C = QC$ or $C = CQ$
`rq_generate_q` [1]	`gegrq`	Form Q
`ql_factor` [1]	`geqlf`	Factor $A = QL$
`ql_multiply_by_q` [1]	`gemql`	Multiply $C = QC$ or $C = CQ$
`ql_generate_q` [1]	`gegql`	Form Q

5.1.4 Eigenvalue and singular value decomposition

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

Variants of methods—such as QR iteration, divide-and-conquer, or Jacobi—will be provided by specifying an option in the input arguments, rather than a different routine name.

C++ API	traditional	operation
`eig, eig_values` [1]	`geev`	Factor $A = X\Lambda X^{-1}$
`eig, eig_values` [1]	`ggev`	Factor $A = X\Lambda BX^{-1}$
`eig, eig_values`	`heev, syev`	Factor $A = X\Lambda X^H$
`eig, eig_values`	`hegv, sygv`	Factor $AB = X\Lambda X^H$ (and variants)
`svd, svd_values`	`gesvd, etc.`	Factor $A = U\Sigma V^H$

[1] not yet implemented

5.2 C and Fortran API

To make SLATE accessible from C and Fortran, we are also providing a C and Fortran 2003 API. Generally, this API 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`, `_c128` for 128-bit quad precision, as well as mixed precisions. The `_c64` version is shown below.

5.2.1 BLAS and auxiliary

C++ API	**C/Fortran API**
`multiply`	`slate_multiply_c64`
`multiply`	`slate_hermitian_multiply_c64`
`multiply`	`slate_symmetric_multiply_c64`
`rank_k_update`	`slate_hermitian_rank_k_update_c64`
`rank_k_update`	`slate_symmetric_rank_k_update_c64`
`rank_2k_update`	`slate_hermitian_rank_2k_update_c64`
`rank_2k_update`	`slate_symmetric_rank_2k_update_c64`
`triangular_multiply`	`slate_triangular_multiply_c64`
`triangular_solve`	`slate_triangular_solve_c64`
`norm`	`slate_norm_c64`
`norm`	`slate_hermitian_norm_c64`
`norm`	`slate_symmetric_norm_c64`
`norm`	`slate_triangular_norm_c64`
`copy`	`slate_copy_c64`
`copy`	`slate_copy_c64c32`
`set`	`slate_set_c64`

5.2.2 Linear systems

C++ API	C/Fortran API
General non-symmetric (LU)	
`lu_solve`	`slate_lu_solve_c64`
`lu_factor`	`slate_lu_factor_c64`
`lu_solve_using_factor`	`slate_lu_solve_using_factor_c64`
`lu_inverse_using_factor`	`slate_lu_inverse_using_factor_c64`
`lu_cond_using_factor`	`slate_lu_cond_using_factor_c64`
Hermitian/symmetric positive definite (Cholesky)	
`chol_solve`	`slate_chol_solve_c64`
`chol_factor`	`slate_chol_factor_c64`
`chol_solve_using_factor`	`slate_chol_solve_using_factor_c64`
`chol_inverse_using_factor`	`slate_chol_inverse_using_factor_c64`
`chol_cond_using_factor`	`slate_chol_cond_using_factor_c64`
Hermitian/symmetric indefinite (block Aasen, permutation not shown)	
`indefinite_solve`	`slate_indefinite_solve_c64`
`indefinite_factor`	`slate_indefinite_factor_c64`
`indefinite_solve_using_factor`	`slate_indefinite_solve_using_factor_c64`
`indefinite_inverse_using_factor`	`slate_indefinite_inverse_using_factor_c64`
`indefinite_cond_using_factor`	`slate_indefinite_cond_using_factor_c64`
Least squares	
`least_squares_solve`	`slate_least_squares_solve_c64`

5.2.3 Unitary factorizations

C++ API	C/Fortran API
`qr_factor`	`slate_qr_factor_c64`
`qr_multiply_by_q`	`slate_qr_multiply_by_q_c64`
`qr_generate_q`	`slate_qr_generate_q_c64` [1]
`lq_factor`	`slate_lq_factor_c64`
`lq_multiply_by_q`	`slate_lq_multiply_by_q_c64`
`lq_generate_q`	`slate_lq_generate_q_c64` [1]
`rq_factor`	`slate_rq_factor_c64` [1]
`rq_multiply_by_q`	`slate_rq_multiply_by_q_c64` [1]
`rq_generate_q`	`slate_rq_generate_q_c64` [1]
`ql_factor`	`slate_ql_factor_c64` [1]
`ql_multiply_by_q`	`slate_ql_multiply_by_q_c64` [1]
`ql_generate_q`	`slate_ql_generate_q_c64` [1]

5.2.4 Eigenvalue and singular value decomposition

C++ API	traditional
`eig, eig_values`	`slate_eig_c64, slate_eig_values_c64` [1]
`eig, eig_values`	`slate_eig_c64, slate_eig_values_c64` [1]
`eig, eig_values`	`slate_eig_c64, slate_eig_values_c64`
`eig, eig_values`	`slate_eig_c64, slate_eig_values_c64`
`svd, svd_values`	`slate_svd_c64, slate_svd_values_c64`

5.3 Traditional (Sca)LAPACK API

SLATE implements many routines from BLAS and (Sca)LAPACK. 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
  - `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` : alpha x plus y
    - `scal` : alpha x
    - `copy` : copy vector
    - `swap` : swap vectors
    - `dot*` : dot products
    - `nrm2` : vector 2-norm
    - `asum` : vector 1-norm (absolute value sum)
    - `iamax`: vector inf-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, inf, fro, 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
    - `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 (several variants)
    - `gv*` : generalized eigenvalue decomposition (several variants)

- `svd*`: singular value decomposition (several variants, some spelled differently)

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 6

Using SLATE

Many of the code snippets in this section reference the SLATE tutorial, available at https://bitbucket.org/icl/slate-tutorial. Links to individual files are given where applicable.

6.1 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.

6.1.1 Matrix Hierarchy

The usage of SLATE revolves around a Tile class and a Matrix class hierarchy (Figure 6.1). 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 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; 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 within the band exist, e.g., $A(i, j)$ for $j = i - k_l, \ldots, i + k_u$.

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 to the same underlying data (`std::map` of tiles).

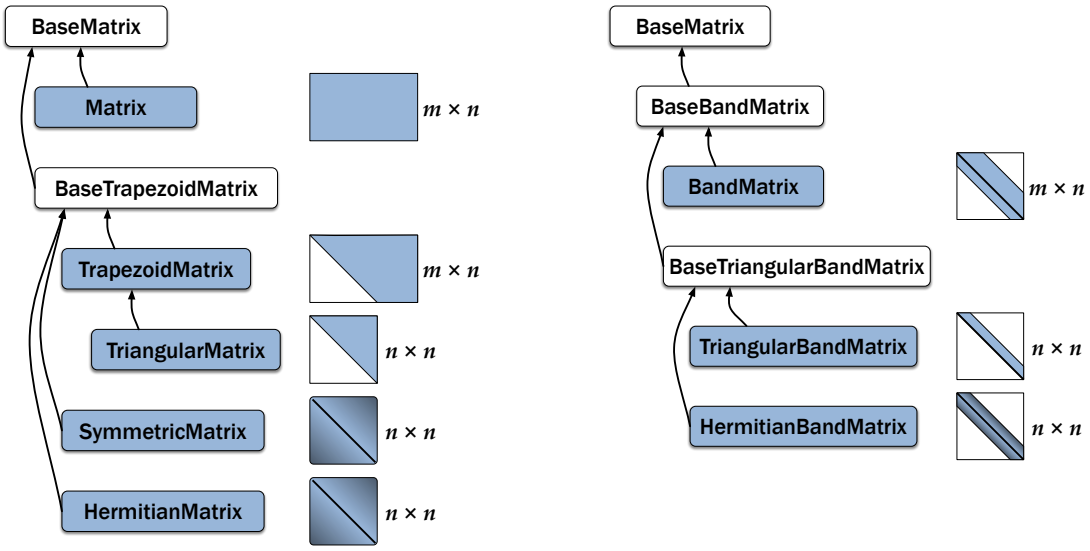

Figure 6.1: Matrix hierarchy in SLATE. Algorithms require the appropriate types for their operation.

Algorithm 6.1 Conversions: slate01_conversion.cc

```
// A is defined to be a general m x n double precision matrix
slate::Matrix<double>
    A( m, n, nb, p, q, MPI_COMM_WORLD );

// U is the triangular matrix obtained from the upper triangle of A
slate::TriangularMatrix<double>
    U( slate::Uplo::Upper, slate::Diag::NonUnit, A );

// L is the triangular matrix obtained from the lower triangle of A,
// assuming Unit diagonal.
slate::TriangularMatrix<double>
    L( slate::Uplo::Lower, slate::Diag::Unit, A );

// Sy is the symmetric matrix obtained from the upper triangle of A
slate::SymmetricMatrix<double>
    Sy( slate::Uplo::Upper, A );
```

Likewise, copying a matrix object is an inexpensive shallow copy, using a C++ shared pointer. Sub-matrices 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 sub-matrix 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`.

6.1.2 Creating and Accessing Matrices

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

Algorithm 6.2 Creating matrices

```
// create an empty matrix (2D block cyclic layout, p x q grid,
// no tiles allocated, square nb x nb tiles)
slate::Matrix<double> A( m, n, nb, p, q, mpi_comm );

// create an empty matrix (2D block cyclic layout, p x q grid,
// no tiles allocated, rectangular mb x nb tiles)
slate::Matrix<double> A( m, n, mb, nb, p, q, mpi_comm );

// create an empty TriangularMatrix (2D block cyclic layout, no tiles)
slate::TriangularMatrix<double> T( uplo, diag, n, nb, p, q, mpi_comm );

// create an empty matrix based on another matrix structure
slate::Matrix<double> 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 6.3) or GPU memory (Algorithm 6.4), in which case SLATE is responsible for deallocating the data. Or the tile data can be provided by the user (Algorithm 6.5), 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 6.3 SLATE allocating CPU memory for a matrix.

```
// create an empty matrix (no tiles allocated)
slate::Matrix<double> A( m, n, nb, p, q, mpi_comm );

// allocate and insert matrix tiles in CPU memory
A.insertLocalTiles();

// A.insertLocalTiles() is equivalent to:
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 );
```

Algorithm 6.4 SLATE allocating GPU memory for a matrix.

```
// create an empty matrix (2D block cyclic layout, no tiles)
slate::Matrix<double> A( m, n, nb, p, q, mpi_comm );

// allocate and insert matrix tiles in GPU memory
A.insertLocalTiles( Target::Devices );

// A.insertLocalTiles( Target::Devices ) is equivalent to:
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, A.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` $\times$ `nb` for the Matrix.

Algorithm 6.5 Inserting tiles using user-defined data.

```
// create an empty matrix (2D block cyclic layout, no tiles)
slate::Matrix<double> A( m, n, nb, p, q, mpi_comm );

// 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`, `A( i, j )` returns its (i,j)-th block, in block row i and block column j. If a matrix is transposed, the transposition operation is included, and set on the tile: 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`, `T( i, j )` returns its (i,j)-th element. If a tile is transposed, the transposition operation is included: if `TT = transpose(T)`, then `TT( i, j )` is `T( j, i )`. If a tile as conjugate transposed, the conjugation is also included: 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, 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; hence they are not efficient for use inside inner loops. It is better to get the data pointer and index it directly.

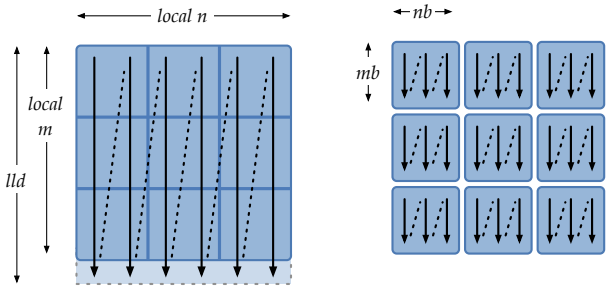

Figure 6.2: Matrix layout of ScaLAPACK (left) and layout with contiguous tiles (right). SLATE matrix and tiles structures are flexible and accommodate multiple layouts.

Algorithm 6.6 Accessing tile elements

```
// loop over tiles in A
for (int64_t j = 0; j < A.nt(); ++j) {
    for (int64_t i = 0; i < A.mt(); ++i) {
        if (A.tileIsLocal( i, j )) {
            // For local tiles, loop over entries in tile
            // (assuming it exists on CPU)
            auto T = A( i, j );
            for (int64_t jj = 0; jj < T.nb(); ++jj) {
                for (int64_t ii = 0; ii < T.mb(); ++ii) {
                    // note: using T.at() is inefficient in inner loops
                    T.at( ii, jj ) = ...;
                }
            }
        }
    }
}
```

6.1.3 Matrices from ScaLAPACK and LAPACK

SLATE also supports tiles laid out in memory using the traditional ScaLAPACK matrix storage allowing a leading dimension stride when accessing the matrix (Figure 6.2). 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. Note, the ScaLAPACK process grid is expected to be in column-major natural ordering.

Algorithm 6.7 Creating matrix from ScaLAPACK-style data.

```
// User has a matrix in ScaLAPACK format with some data
// SLATE can then create a Matrix from that
auto A = slate::Matrix<double>::fromScaLAPACK(
    m, n, data, lld, nb, p, q, mpi_comm );
    // m, n: global matrix dimensions
    // data: local ScaLAPACK array data
    // lld:  local leading dimension (stride) for data
    // nb:   block size
    // p, q: MPI process grid
    // mpi_comm: MPI communicator
```

Similarly, SLATE can map its Matrix datatype over matrices that are in LAPACK layout, and replicated across all nodes.

Algorithm 6.8 Creating matrix from LAPACK-style data.

```
// User has a matrix in LAPACK format with some data
// SLATE can then create a Matrix from that
auto A = slate::Matrix<scalar_t>::fromLAPACK(
    m, n, data, lda, nb, p, q, mpi_comm);
    // m, n: global matrix dimensions
    // data: local LAPACK data
    // lda:  leading dimension (stride) for data
    // nb:   block size
    // p, q: normally both are 1
    // mpi_comm: MPI communicator
```

6.1.4 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 6.9 Conjugating matrices.

```
// Transpose
AT = transpose( A );
// AT is shallow copy of A, with flag AT.op() == Op::Trans
// The Tile AT( i, j ) == transpose( A( j, i ) )

// Conjugate transpose
AH = conj_transpose( A );
// AH is shallow copy of A, with flag AH.op() == Op::ConjTrans
// The Tile AH( i, j ) == conj_transpose( A( j, i ) )
```

6.1.5 Tile Submatrices

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

Algorithm 6.10 Sub-matrices: slate02_submatrix.cc

```
A.sub( i1, i2, j1, j2 )            // A( i1 : i2, j1 : j2 ) inclusive
A.sub( 0, A.mt()-1, 0, 0 )         // first block-column
A.sub( 0, 0, 0, A.nt()-1 )         // first block-row
A.sub( 1, A.mt()-1, 1, A.nt()-1 ) // trailing matrix
```

6.1.6 Matrix Slices

Matrix slices use column and row indices instead of tile indices. Note, the slice operations are less efficient than the submatrix operations and the matrices produced have less algorithm support, especially on GPUs.

Algorithm 6.11 Matrix slice: slate02_submatrix.cc

```
A.slice( row1, row2, col1, col2 ) // A( row1 : row2, col1 : col2 ) inclusive
A.slice( 0, A.m()-1, 0, 0 )       // first column
A.slice( 0, 0, 0, A.n()-1 )       // first row
A.slice( i1, i2, j1, j2 )         // arbitrary region of A
```

6.1.7 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.

Algorithm 6.12 Deep matrix copy

```
slate::Matrix<double> A( m, n, nb, p, q, mpi_comm );
A.insertLocalTiles();
// fill data into A here ...
slate::Matrix<float> A_lo( m, n, nb, p, q, mpi_comm );
A_lo.insertLocalTiles();
// This copy will do precision conversion from double to float
slate::copy( A, A_lo );
```

6.2 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. The SLATE Function Reference is generated from the source code documentation and is available online.
https://icl.bitbucket.io/slate/

6.2.1 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 6.13 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 (usually 1)
Tolerance        // tolerance for iterative methods (default epsilon)
```

These options are passed in via an optionally provided 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.

Algorithm 6.14 multiply (gemm) options.

```
// multiply (gemm) call without options
slate::multiply( alpha, A, B, beta, C );

// multiply (gemm) call with options map provided
slate::multiply( alpha, A, B, beta, C, {
  // Set execution target to GPU Devices
  { slate::Option::Target, slate::Target::Devices }
  // Use an algorithmic lookahead of 2
  { slate::Option::Lookahead, 2 } } );
```

6.2.2 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 6.15 Norms: slate03_norm.cc

```
enum class Norm { Max, One, Inf, Fro }

slate::Matrix<double> A(...);
double Anorm = norm( Norm::One, A );

slate::SymmetricMatrix<double> S(...);
double Snorm = norm( Norm::One, S );

slate::TrapezoidMatrix<double> T(...);
double Tnorm = norm( Norm::One, T );
```

6.2.3 Matrix-Matrix Multiply

SLATE implements matrix multiply for the matrices in the matrix hierarchy (i.e. `gemm`, `gbmm`, `hemm`, `symm`, `trmm`. Note: a matrix can set several flags that get recorded within its structure that 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 6.16 Parallel BLAS: slate04_blas.cc

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

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

// C = alpha BA + beta C, where A is symmetric
slate::multiply( alpha, B, A, beta, C );            // simplified (right)
slate::symm( Side::Right, alpha, A, B, beta, C );   // traditional

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

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

// matrices can be transposed or conjugate-transposed beforehand
auto AT = slate::transpose( A );
auto BT = slate::conj_transpose( B );
slate::multiply( alpha, AT, BT, beta, C );          // simplified
slate::gemm( alpha, AT, BT, beta, C );              // traditional
```

6.2.4 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 6.17 Parallel BLAS, triangular: slate04_blas.cc

```
// B = AB, where A is triangular
// Note: diag is a property of matrix A
slate::triangular_multiply( alpha, A, B );      // simplified (left)
slate::trmm( Side::Left, alpha, A, B );         // traditional

// B = BA, where A is triangular
slate::triangular_multiply( alpha, B, A );      // simplified (right)
slate::trmm( Side::Right, alpha, A, B );        // traditional

// B = A^T B, where A is triangular
slate::TriangularMatrix<double> A
    (uplo, diag, n, nb, p, q, mpi_comm);
auto AT = transpose(A);
slate::triangular_multiply( alpha, AT, B );     // simplified (left)
slate::trmm( Side::Left, alpha, AT, B );        // traditional

// Solve AX = B where A is triangular
// X overwrites B
slate::triangular_solve( alpha, A, B );         // simplified (right)
slate::trsm( Side::Left, alpha, A, B );         // traditional

// Solve XA = B where A is triangular
slate::triangular_solve( alpha, B, A );         // simplified (right)
slate::trsm( Side::Right, alpha, A, B );        // traditional
```

6.2.5 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 and solve operations defined for them.

Algorithm 6.18 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
```

6.2.6 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. The 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 6.19 LU solve: slate05a_linear_system_lu.cc

```
slate::Matrix<double> A( n, n,    ... );
slate::Matrix<double> B( n, nrhs, ... );

slate::lu_solve( A, B );                    // simplified
slate::lu_solve( A, B, {options} );         // simplified, with options

slate::Pivots pivots;
slate::gesv( A, pivots, B );                // traditional
slate::gesv( A, pivots, B, {options} );     // traditional, with options
```

6.2.7 Linear Systems: 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 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 6.20 Cholesky solve: slate05b_linear_system_cholesky.cc

```
slate::SymmetricMatrix<double>                                    1
    A( slate::Uplo::Lower, n, ... );                              2
slate::Matrix<double> B( n, nrhs, ... );                          3
                                                                  4
slate::chol_solve( A, B );       // simplified                    5
slate::posv( A, B );             // traditional                   6
```

6.2.8 Linear Systems: Hermitian indefinite (Aasen's)

Distributed parallel Hermitian 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 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 LU-factored with partial pivoting. The factored form of A is then used to solve the system of equations $AX = B$.

Algorithm 6.21 Indefinite solve: slate05c_linear_system_hesv.cc

```
slate::HermitianMatrix<double>  A( slate::Uplo::Lower, n, ... );  1
slate::Matrix<double> B( n, nrhs, ... );                          2
                                                                  3
slate::indefinite_solve( A, B );                 // simplified    4
                                                                  5
// workspaces                                                     6
slate::Matrix<double> H( n, n, ... );                             7
slate::BandMatrix<double> T( n, n, nb, nb, ... );                 8
slate::Pivots pivots, pivots2;                                    9
// call solver                                                    10
slate::hesv( A, pivots, T, pivots2, H, B );      // traditional   11
```

6.2.9 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 $m \times nrhs$, BX is $\max(m,n) \times nrhs$.

If $m \geq n$, solves over-determined $AX = B$ with least squares solution X that minimizes $\|AX - B\|_2$. BX is $m \times nrhs$. On input, B is all m rows of BX. On output, X is first n rows of BX. Currently in this case A must be not transposed.

If $m < n$, solves under-determined $AX = B$ with minimum norm solution X that minimizes $\|X\|_2$. BX is $n \times nrhs$. On input, B is first m rows of BX. On output, X is all n rows of BX. 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 these (m,n) differ from (M,N) in (Sca)LAPACK, 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 6.22 shows how to handle over-determined systems ($m > n$).

Algorithm 6.22 Least squares (overdetermined): slate06_least_squares.cc

```
// Solve AX = B                                                          1
int64_t m=2000, n=1000, nrhs=100, nb=100, p=2, q=2;                      2
int64_t max_mn = std::max( m, n );                                       3
slate::Matrix<double> A( m, n, nb, p, q, mpi_comm );                     4
slate::Matrix<double> BX( max_mn, nrhs, nb, p, q, mpi_comm );            5
// user should fill in A, B here                                         6
auto B = BX;                                 // B is all m rows of BX    7
auto X = BX.slice( 0, n-1, 0, nrhs-1 ); // X is first n rows of BX       8
                                                                         9
// solve AX = B, solution in X                                           10
slate::least_squares_solve( A, BX );     // simplified                   11
                                                                         12
slate::TriangularFactors<double> T;                                      13
slate::gels( A, T, BX );                 // traditional                  14
```

The example in Algorithm 6.23 shows how to handle under-determined systems ($m < n$).

Algorithm 6.23 Least squares (underdetermined): slate06_least_squares.cc

```
// Solve A^T X = B
int64_t m=2000, n=1000, nrhs=100, nb=100, p=2, q=2;
int64_t max_mn = std::max( m, n );
slate::Matrix<double> A( m, n, nb, p, q, mpi_comm );
slate::Matrix<double> BX( max_mn, nrhs, nb, p, q, mpi_comm );
// user should fill in A, B here
auto B = BX.slice( 0, n-1, 0, nrhs-1 ); // B is first n rows of BX
auto X = BX;                            // X is all m rows of BX

// solve A^T X = B, solution in X
auto AT = transpose(A);
slate::least_squares_solve( A, BX );    // simplified

slate::TriangularFactors<double> T;
slate::gels( AT, T, BX );               // traditional
```

6.2.10 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 in higher precision (e.g., double precision) to get to the accuracy desired. If the iterative refinement fails to get to the desired accuracy, the computation falls back and runs the high precision algorithm. Mixed precision algorithms are implemented for LU and Cholesky solvers.

Algorithm 6.24 Mixed precision

```
// Mixed precision LU factorization
slate::Matrix<double> A( n, n,    ... );
slate::Matrix<double> B( n, nrhs, ... );
slate::Matrix<double> X( n, nrhs, ... );
int iters;

slate::lu_solve( A, B, X, &iters );           // simplified not yet implemented

slate::Pivots pivots;
slate::gesvMixed( A, pivots, B, X, &iters );  // traditional
```

6.2.11 Matrix Inverse

Matrix inversion requires that the matrix first be factored, then the inverse is computed from the factors. Note: it is generally recommended that you solve $AX = B$ using the solve routines (e.g., gesv, posv) 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 6.25 LU inverse: slate05a_linear_system_lu.cc

```
slate::Matrix<double> A( n, n, ... );

slate::lu_factor( A, pivots );                 // simplified, factor
slate::lu_inverse_using_factor( A, pivots );  // simplified, inverse

slate::Pivots pivots;
slate::getrf( A, pivots );  // traditional, factor
slate::getri( A, pivots );  // traditional, inverse
```

Algorithm 6.26 Cholesky inverse: slate05b_linear_system_cholesky.cc

```
slate::SymmetricMatrix<double> A( slate::Uplo::Lower, n, ... );

slate::chol_factor( A );                 // simplified, factor
slate::chol_inverse_using_factor( A );  // simplified, inverse

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

6.2.12 Singular Value Decomposition

The SLATE SVD algorithm uses a 2-stage reduction that first reduces to a triangular band matrix, then to bidiagonal, which is used to compute the singular values.

Algorithm 6.27 SVD: slate07_svd.cc

```
slate::Matrix<double> A( m, n, ... );
std::vector<double> Sigma( min(m, n) );

// singular values only
slate::svd_values( A, Sigma );          // simplified
slate::gesvd( A, Sigma );               // traditional

// singular vectors forthcoming
slate::Matrix<double> U( m, n, ... );
slate::Matrix<double> VH( n, n, ... );
slate::svd( A, U, Sigma, VH );          // simplified
slate::gesvd( A, U, Sigma, VH );        // traditional
```

6.2.13 Hermitian (symmetric) eigenvalues

The SLATE eigenvalue algorithm uses a 2-stage reduction that first reduces to a Hermitian band matrix, then to symmetric tridiagonal, which is used to compute the eigenvalues.

Algorithm 6.28 Hermitian/symmetric eigenvalues: slate08_hermitian_eig.cc

```
slate::HermitianMatrix<double> A( Uplo::Lower, n, ... );
std::vector<double> Lambda( n );

// eigenvalues only
slate::eig_values( A, Lambda );          // simplified
slate::heev( A, Lambda );                // traditional

// eigenvectors forthcoming
slate::Matrix<double> X( n, n, ... );
slate::eig( A, Lambda, X );              // simplified
slate::heev( A, Lambda, X );             // traditional
```

CHAPTER 7

Installation Instructions

SLATE requires BLAS and LAPACK library support for core mathematical routines. SLATE can be built without multi-threaded support, but for any multicore architecture a compiler with support for OpenMP 4.5 or better should be used. SLATE can be built without MPI support, but to support distributed computing MPI is required. Currently, SLATE uses CUDA and cuBLAS for acceleration on NVIDIA GPUs. We are working to support AMD and Intel GPUs. SLATE can be built without CUDA 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.

First download the source. Note, the source has several submodules and these need to be downloaded recursively.

```
# Use one of the following: git or https access
> git clone --recursive git@bitbucket.org:icl/slate.git
> git clone --recursive https://bitbucket.org/icl/slate.git

> cd slate
```

SLATE contains Python configuration scripts that will probe the available environment (compilers, libraries) to determine configuration options for OpenMP, BLAS and LAPACK.

7.1 Makefile based build

The GNUmakefile build uses GNU specific extensions and expects the user provide some configuration options to the build process. These build configuration options may change as

the build process is improved and modified. Please look at the top of the GNUmakefile for the most current information. The current instructions in the GNUmakefile are shown here.

```
# Relies on settings in environment. These can be set in ~/.cshrc,
# via modules, or in make.inc.
#
# Set C++ compiler by $CXX; usually want CXX=mpicxx.
# Set Fortran compiler by $FC; usually want FC=mpifort or mpif90.
# Add include directories to $CPATH or $CXXFLAGS for CUDA, MKL, etc.
# Add lib directories to $LIBRARY_PATH or $LDFLAGS for CUDA, MKL, etc.
# At runtime, these lib directories need to be in $LD_LIBRARY_PATH,
# or on MacOS, $DYLD_LIBRARY_PATH, or set as rpaths in $LDFLAGS.
# MPI is usually found using the mpicxx compiler wrapper, instead of
# adding to these paths.
#
# Set SLATE options on the command line or in a make.inc file.
# An example make.inc file:
#     CXX  = mpicxx
#     FC   = mpifort
#     blas = openblas
#     cuda_arch = volta
#
# CXX=mpicxx or mpic++ for MPI using compiler wrapper.
# Alternatively:
#     mpi=1       for MPI (-lmpi).
#     spectrum=1  for IBM Spectrum MPI (-lmpi_ibm).
#
# blas=mkl        for Intel MKL. Additional sub-options:
#     mkl_intel=1           for Intel MKL with Intel Fortran conventions;
#                           otherwise uses GNU gfortran conventions.
#                           Automatically set if CXX=icpc or on macOS.
#     mkl_threaded=1        for multi-threaded Intel MKL.
#     mkl_blacs=openmpi     for OpenMPI BLACS in SLATE's testers.
#     mkl_blacs=intelmpi    for Intel MPI BLACS in SLATE's testers (default).
#     ilp64=1               for ILP64. Currently only with Intel MKL.
# blas=essl       for IBM ESSL.
# blas=openblas   for OpenBLAS.
#
# openmp=1        for OpenMP (default).
# static=1        for static library (libslate.a);
#                 otherwise default is shared library (libslate.so).
#
# If $(NVCC) compiler is found, sets cuda=1 by default. NVCC=nvcc by default.
# cuda_arch="ARCH" for CUDA architectures, where ARCH is one or more of:
#                     kepler maxwell pascal volta turing sm_XX
#                  and sm_XX is a CUDA architecture (see nvcc -h).
# cuda_arch="kepler pascal" by default.
```

The GNUmake build process expects to find the include files or libraries for MPI, CUDA, BLAS, LAPACK and ScaLAPACK via search path variables. The locations of 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`.

The following minimal build command will produce a SLATE library that is configured with OpenMP (default), enabled for distributed computing, has CUDA support (if the `nvcc` command can be found) and is dynamically linked (default).

```
# make echo will print out all the options for the build
make CXX=mpicxx FC=mpifort blas=mkl echo

# do the actual build
make CXX=mpicxx FC=mpifort blas=mkl

# install files in /usr/local/{lib,include}
make install prefix=/usr/local
```

7.2 CMake

CMake based builds are currently in development in SLATE. These builds are included in the source code and can be used. Note: for the CMake build process, the TestSweeper, BLAS++ and LAPACK++ submodules will need to be built before SLATE can be built.

7.3 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 and missing dependencies. If Spack is already installed, use the local installation to install SLATE. Spack has many configuration option (which compiler to use, which libraries, etc), make sure you use your desired setup.

Note: the current SLATE Spack package is limited. We are actively developing a more flexible one based on CMake.

```
> git clone https://github.com/spack/spack.git
> cd spack/bin
> ./spack info slate
> ./spack install slate
```

CHAPTER 8

Testing Suite for SLATE

SLATE comes with a testing suite that is intended to check the correctness and accuracy of a majority 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 about threading).

For the parallel BLAS routines (e.g. `gemm, symm`), the testing is done by comparing the answer with a reference ScaLAPACK execution.

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 gemm --help

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

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

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

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

```
> ./tester gemm --help
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; ...
    --trace           enable/disable traces; default n; valid: [ny]
    --trace-scale     horizontal scale for traces, in pixels per sec; ...
    --tol             tolerance (e.g., error < tol*epsilon to pass); ...
    --repeat          number of times to repeat each test; default 1
    --verbose         verbose level; default 0
    --cache           total cache size, in MiB; default 20

Parameters that take comma-separated list of values and may be repeated:
    --type            s=single (float), d=double, c=complex-single, ...
    --origin          origin: h=Host, s=ScaLAPACK, d=Devices; default host
    --target          target: t=HostTask, n=HostNest, b=HostBatch, ...
    --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; ...
    --transB          transpose of B: n=no-trans, t=trans, c=conj-trans; ...
    --dim             m x n x k dimensions
    --alpha           scalar alpha; default 3.142
    --beta            scalar beta; default 2.718
    --nb              nb; default 50
    --p               p; default 1
    --q               q; default 1
    --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 gemm --type s --target t --dim 500:2000:500 --nb 256

# 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 gemm --type s --target d --dim 5000,10000 --nb 192:256:64

# 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 gemm --type d --target d --dim 10000 --nb 192:256:64 \
                          --p 2 --q 2

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

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
python ./run_tests.py --help

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

# Run a small run using the full testing suite
python ./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.
python ./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
python ./run_tests.py --test "mpirun -n 4 ./tester " --xsmall --target ges
```

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 tiles sizes tend to be on the order of hundreds. A sweep over tiles sizes can be used to determine the appropriate tile size for an algorithm. Note, 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 gesv --type d --target t --dim 3000 --nb 128:512:32

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

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 $p * q$ grid where $p \leq q$ is going to work well.

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

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 (`--panel-threads 1` is usually correct).

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 assure 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.

```
cd unit_test
# The default unit_test run used by SLATE
python ./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 the 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 set up other SLATE requirements.

The LAPACK data is then mapped 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 so that SLATE uses the threads 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 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 that references 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 analogue 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

```
// 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

```
!! 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, and Jack Dongarra. SLATE Developers' Guide, SWAN no. 11. Technical Report ICL-UT-19-02, Innovative Computing Laboratory, University of Tennessee, January 2019. Revision 01-2019.

[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.