SLATE WORKING NOTES 8

Linear Systems Performance Report

Jakub Kurzak
Mark Gates
Ichitaro Yamazaki
Ali Charara
Asim YarKhan
Jamie Finney
Gerald Ragghianti
Piotr Luszczek
Jack Dongarra

Innovative Computing Laboratory

October 1, 2018

ICL INNOVATIVE COMPUTING LABORATORY

THE UNIVERSITY OF TENNESSEE KNOXVILLE

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of two U.S. Department of Energy organizations (Office of Science and the National Nuclear Security Administration) responsible for the planning and preparation of a capable exascale ecosystem, including software, applications, hardware, advanced system engineering and early testbed platforms, in support of 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**
09-2018	first publication

```
@techreport{kurzak2018linear,
  author={Kurzak, Jakub and Gates, Mark and Yamazaki, Ichitaro and
          Charara, Ali and YarKhan, Asim and Finney, Jamie and
          Ragghianti, Gerald and Luszczek, Piotr and Dongarra, Jack},
  title={{SLATE} Working Note 8: Linear Systems Performance Report},
  institution={Innovative Computing Laboratory, University of Tennessee},
  year={2018},
  month={September},
  number={ICL-UT-XX-XX},
  note={revision 09-2018}
}
```

Contents

Contents **ii**

List of Figures **iv**

List of Tables **v**

1 Introduction **1**

2 Implementation **2**

2.1 Parallelization . . . 3

2.1.1 LLT . . . 3

2.1.2 LU . . . 4

2.1.3 LDLT . . . 5

3 Experiments **6**

3.1 Environment . . . 6

3.2 Performance . . . 7

3.2.1 LLT . . . 7

3.2.2 LU . . . 7

3.2.3 LDLT . . . 8

4 Summary **9**

Bibliography **10**

Appendices **12**

A Function Signatures **13**

B Implementation Snippets **15**

C MPI Thread Safety Considerations **17**
C.1 MPI Implementations 17
C.2 Multithreading Tests 18
C.3 Testing Methodology 18

List of Figures

2.1 Cholesky factorization with lookahead of one. 3
2.2 LU factorization with lookahead of one. 4
2.3 Left-looking tiled Aasen's algorithm. 5

3.1 Performance of dpotrf. 7
3.2 Performance of dgetrf. 8
3.3 Performance of dsytrf. 8

List of Tables

C.1 MPI thread safety testing results. 19

CHAPTER 1

Introduction

Software for Linear Algebra Targeting Exascale (SLATE) [1] [1] is being developed as part of the Exascale Computing Project (ECP) [2], which is a collaborative effort between two US Department of Energy (DOE) organizations, the Office of Science and the National Nuclear Security Administration (NNSA). The purpose of SLATE is to serve as a replacement for ScaLAPACK for the upcoming pre-exascale and exascale DOE machines. SLATE will accomplish this objective by leveraging recent progress in parallel programming models and by strongly focusing on supporting hardware accelerators.

This report focuses on the set of SLATE routines that solve linear systems of equations. Specifically, initial performance numbers are reported, alongside ScaLAPACK performance numbers, on the SummitDev machine at the Oak Ridge Leadership Computing Facility (OLCF). More details about the design of the SLATE software infrastructure can be found in the report by Kurzak et al. [1].

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

CHAPTER 2

Implementation

The principles of the SLATE software framework were laid out in SLATE Working Note 3 [1] [1]. SLATE's design relies on the following principles:

- The matrix is represented as a set of individual tiles with no constraints on their locations in memory with respect to one another. Any tile can reside anywhere in memory and have any stride. Notably, a SLATE matrix can be created from a LAPACK matrix or a ScaLAPACK matrix without making a copy of the data.
- Node-level scheduling relies on nested Open Multi Processing (OpenMP) tasking, with the top level responsible for resolving data dependencies and the bottom level responsible for deploying large numbers of independent tasks to multi-core processors and accelerator devices.
- Batch BLAS is used extensively for maximum node-level performance. Most routines spend the majority of their execution in the call to batch `gemm`.
- The Message Passing Interface (MPI) is used for message passing with emphasis on collective communication, with the majority of communication being cast as broadcasts.

Also, the use of a runtime scheduling system, such as the Parallel Runtime Scheduling and Execution Controller (PaRSEC) [2] [2] or Legion [3,4] [3], is currently under investigation.

[1] http://www.icl.utk.edu/publications/swan-003
[2] http://icl.utk.edu/parsec/
[3] http://legion.stanford.edu
[4] http://www.lanl.gov/projects/programming-models/legion.php

2.1 Parallelization

SLATE linear solvers are marked by much higher complexity than (Sca)LAPACK due to a totally different representation of the matrix. Consider the following factors:

- SLATE matrix is a "loose" collection of tiles, i.e., there are no constraints on the memory location of any tile with respect to the other tiles. Notably, however, a ScaLAPACK matrix can still be mapped to a SLATE matrix without making a copy of the data.
- SLATE matrix can be partitioned to distributed memory nodes in any possible way, i.e., no assumptions are made about the placement of any tiles with respect to the other tiles. The same applies to the partitioning of tiles within each node to multiple accelerators.
- In principle, SLATE can support non-uniform tile sizes within the same matrix, although this mode of operation has not been well tested, as currently supporting the standard 2D block cyclic partitioning, for compatilibity with ScaLAPACK, is the top priority.

The following subsections describe the current set of linear solvers in SLATE, in the order of increasing implementation complexity: LLT, LU, and LDLT.

2.1.1 LLT

SLATE provides routines for solving linear systems of equations, where the coefficient matrix is symmetric (Hermitian) positive definite. These routines compute the factorization $A = LL^T$ ($A = LL^H$) using the Cholesky decomposition, and follow with the steps of forward and backward substitution. The routines are mathematically equivalent to their ScaLAPACK counterparts [4].

Figure 2.1 shows the basic mechanics of the Cholesky factorization in SLATE. Like most routines in SLATE, the implementation relies on nested tasking using the OpenMP standard, with the top level responsible for scheduling a small number of coarse grained, inter-dependent tasks, and the nested level responsible for dispatching large numbers of fine grained, independent tasks. In the case of GPU acceleration, the nested level is implemented using calls to batch BLAS routines, to exploit the efficiency of processing large numbers of tiles in one call to a GPU kernel.

The Cholesky factorization in SLATE applies the technique of *lookahead*, where one or more columns, immediately following the panel, are prioritized for faster processing, to allow for speedier advancement along the critical path. Lookahead provides large performance improvements, as it allows for overlapping the panel factorization, which is usually inefficient, with updating of the trailing submatrix, which is usually very efficient and can be GPU accelerated. Usually, the lookahead of one results in a large performance gain, while bigger values deliver diminishing returns.

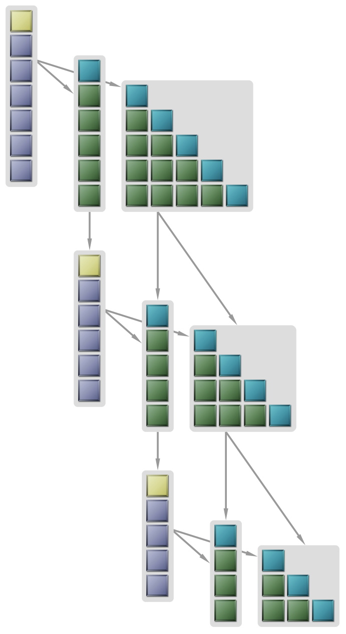


Figure 2.1: Cholesky factorization with lookahead of one.

2.1.2 LU

SLATE provides routines for solving linear systems of equations, where the coefficient matrix is a general (nonsymmetric) matrix. These routines compute the factorization $PA = LU$ using the process of Gaussian elimination with partial (row) pivoting, and follow with the steps of forward and backward substitution. The routines are mathematically equivalent to their ScaLAPACK counterparts [4].

Figure 2.2 shows the basic mechanics of the LU factorization in SLATE. While the parallelization is based on the same principles as the Cholesky factorization, the implementation is significantly more challenging, due to the application of row pivoting. The primary consequence of row pivoting is a fairly complex, and heavily synchronous, panel factorization procedure. The secondary effect is the communication overhead of swapping rows to the left and to the right of the panel. Further complication is introduced by GPU acceleration, which requires layout translation, as the row swapping operation is extremely inefficient in column major.

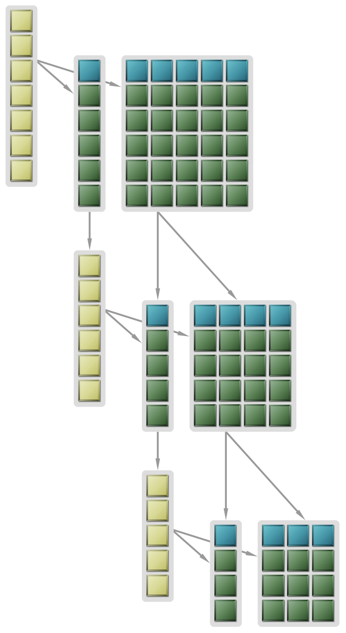


Figure 2.2: LU factorization with lookahead of one.

The critical component of the LU factorization is the step of factoring the panel, which in SLATE is an arbitrary selection of tiles from one column of the matrix. This operation is on the critical path of the algorithms and has to be optimized to the maximum. Resorting to a simple, memory-bound implementation, could have profound consequences for performance. The current implementation of the LU panel factorization in SLATE derives from the technique of *Parallel Cache Assignment* (PCA) by Castaldo et al. [5], and the work on parallel panel factorization by Dongarra et al. [6].

The LU panel factorization in SLATE relies on internal blocking and persistent assignment of tiles to threads within each MPI process. Unlike past implementations, it is not recursive, as plain recursion proved inferior to blocking. Memory residency provides some level of cache reuse, while blocking provides some level of compute intensity. The resulting implementation is no longer memory bound, and scales well with the number of processes and the number of threads in each process. The procedure is heavily synchronous and relies on MPI collective communication, to exchange pivot information, and on thread barriers for intra-node synchronization. An MPI sub-communicator is created for each set of processes participating in each panel factorization.

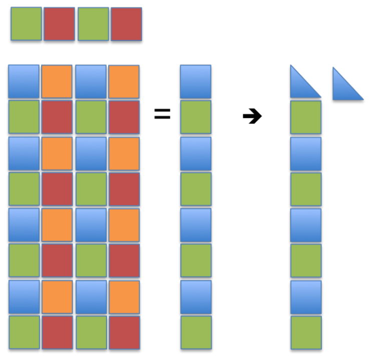
Finally, the LU factorization in SLATE introduces the complication of multiple OpenMP tasks issuing, possibly concurrently, independent communications. Specifically, the collective communication of the panel factorization may coincide with sends and receives of multiple simultaneous row swaps. This requires that the underlaying MPI implementation is thread safe, and supports the `MPI_THREAD_MULTIPLE` mode, i.e., multiple threads simultaneously issuing MPI communications (further discussed in Appendix C). It also requires that the different communications be distinguished by different MPI tags.

2.1.3 LDLT

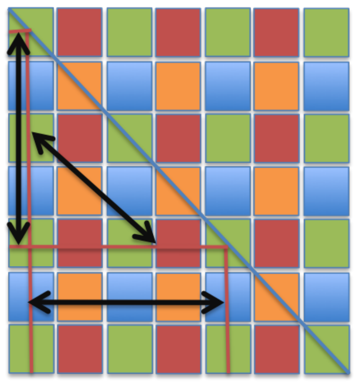
SLATE provides routines for solving linear systems of equations, where the coefficient matrix is symmetric (Hermitian) indefinite, based on a two-stage decomposition of the matrix. The first stage uses the tiled Aasen's algorithm [7] to compute the factorization $PAP^T = LDL^T$ ($PAP^H = LDL^H$) where D is a symmetric band matrix (with the bandwidth equal to the tile size). The second stage then computes the LU factorization of the band matrix with partial (row) pivoting. The solution is then computed by the forward and backward substitutions. ScaLAPACK currently does not provide a symmetric indefinite solver, but our solver is mathematically equivalent to the LAPACK and PLASMA counterparts for shared memory CPUs [8, 9]. Our current SLATE distributed memory implementation extends the OpenMP implementation of PLASMA.

The three main phases of the Aasen's algorithm are: 1) the left-looking block update of the panel, 2) the LU factorization of the panel, and 3) the symmetric pivoting of the trailing submatrix, each of which presents challenges in term of parallel scalability. In addition, each of these phases cannot be overlapped with other main computational tasks. Hence, the parallelism needs to be exploited within each phase. For instance, at the first phase, the right-looking algorithm can update multiple tiles of the panel in parallel, but the `gemm` operations in the same row update the same tile of the panel, and they have the write conflict. In our implementation, each of the processes locally computes the partial updates, which are then accumulated into the panel using a single all-reduce operation. Though workspace may be used to further increase the scalability of the left-looking algorithm, compared with the right-looking algorithm, each process has a limitted thread parallelism to exploit (limited by the number of local block rows).

At the second phase, the panel is factorized using the LU panel factorization routine from Section 2.1.2. Unfortunately, the task dependencies of the Aasen's algorithm prevents us from overlapping the panel factorization with other computational tasks. Hence, during the panel factorization, the parallelism is limited by the number of tiles in the panel. Finally, compared with the partial pivoting, the symmetric pivoting introduces significantly more irregular data communication patterns (e.g., when the lower triangular part of the matrix is stored, the columns in the upper triangular part are stored as the transpose of the corresponding rows in the lower triangular part).



(a) Left-looking update & LU panel



(b) Symmetric pivoting.

Figure 2.3: Left-looking tiled Aasen's algorithm.

For computing the LU factorization of the band matrix at the second stage of the factorization, we extended the LU factorization of Section 2.1.2 such that only the operations with non-empty blocks are computed.

CHAPTER 3

Experiments

3.1 Environment

Performance numbers were collected using the SummitDev system [1] at the OLCF, which is intended to mimic the OLCF's next supercomputer, Summit. SummitDev is based on IBM POWER8 processors and NVIDIA P100 (Pascal) accelerators, and is one generation behind Summit, which will be based on IBM POWER9 processors and NVIDIA V100 (Volta) accelerators.

The SummitDev system contains three racks, each with eighteen IBM POWER8 S822LC nodes, for a total of fifty-four nodes. Each node contains two POWER8 CPUs, ten cores each, and four P100 GPUs. Each node has 256 GB of DDR4 memory. Each GPU has 16 GB of HBM2 memory. The GPUs are connected by NVLink 1.0 at 80 GB/s. The nodes are connected with a fat-tree enhanced data rate (EDR) InfiniBand.

The software environment used for the experiments included GNU Compiler Collection (GCC) 7.1.0, CUDA 9.0.69, Engineering Scientific Subroutine Library (ESSL) 5.5.0, Spectrum MPI 10.1.0.4, Netlib LAPACK 3.6.1, and Netlib ScaLAPACK 2.0.2—i.e., the output of `module list` included:

```
gcc/7.1.0
cuda/9.0.69
essl/5.5.0-20161110
spectrum-mpi/10.1.0.4-20170915
netlib-lapack/3.6.1
netlib-scalapack/2.0.2
```

[1]https://www.olcf.ornl.gov/kb_articles/summitdev-quickstart/

3.2 Performance

All runs were performed using sixteen nodes of the SummitDev system, which provides $16\ nodes \times 2\ sockets \times 10\ cores = 320$ IBM POWER8 cores and $16\ nodes \times 4\ devices = 64$ NVIDIA P100 accelerators. SLATE was run with one process per node, while ScaLAPACK was run with one process per core, which is still the prevailing method of getting the best performance from ScaLAPACK. Only rudimentary performance tuning was done in both cases.

3.2.1 LLT

Figure 3.2 shows the performance of the Cholesky factorization. The Cholesky factorization in SLATE delivers very good performance. SLATE implementation delivers better CPU performance than ScaLAPACK and the GPU acceleration provides further improvement by an order of magnitude. This is thanks to the factorization being a fairly simple workload with no complex operations and with straightforward communication patterns.

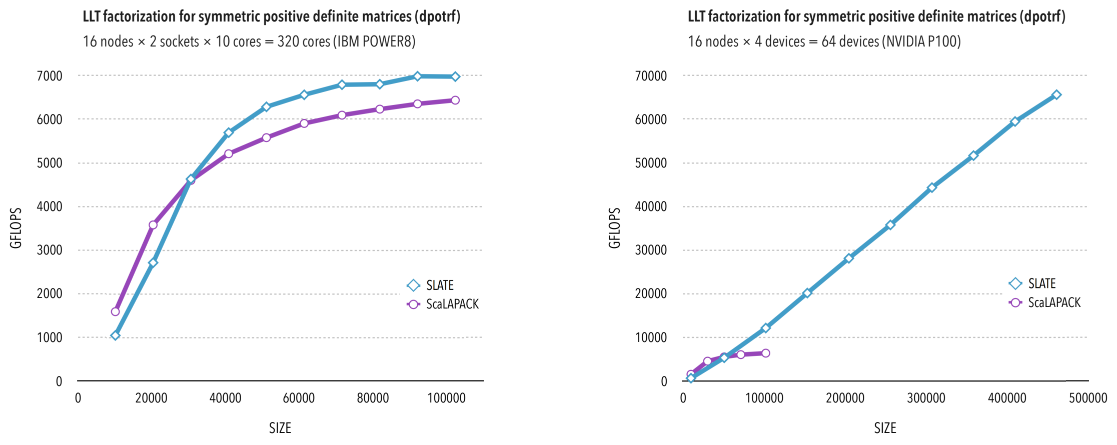
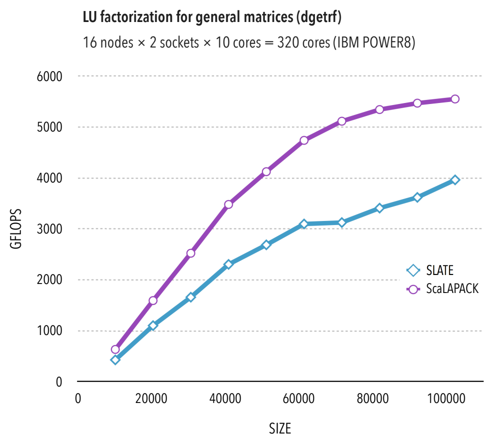


Figure 3.1: Performance of `dpotrf` without acceleration (left) and with acceleration (right).

3.2.2 LU

Figure 3.2 shows the performance of the LU factorization. At this point, SLATE implementation delivers lower CPU performance than ScaLAPACK, and acceleration provides only moderate advantage, This is despite employing state-of-the-art solutions, such as the multithreaded and cache-efficient panel factorization and the technique of lookahead. Judging from the traces, the main reason seems to be the cost of swapping rows in the process of pivoting. This will require further investigation, both into the inner workings of SLATE and the mechanics of the underlaying MPI implementations.



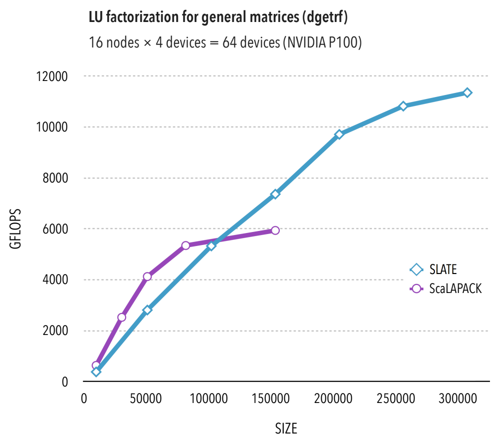


Figure 3.2: Performance of `dgetrf` without acceleration (left) and with acceleration (right).

3.2.3 LDLT

Figure 3.3 shows the performance of the Aasen's LDLT factorization. Since ScaLAPACK does not provide a symmetric indefinite factorization, there is no ScaLAPACK performance cuver here. Also, the GPU performance line is missing, as the algorithm turned out to be unfit for acceleration.

Despite its communication avoiding properties [7], the Aasen's algorithm turned out to be ill suited for a distributed memory implementation and basically impossible to GPU accelerate. Because of symmetric pivoting, the Aasen's algorithm is inherently left-looking. This makes it harder to implement the main `gemm` operation, and makes it impossible to overlap the panel factorization with other parts of the algorithm. Finally, swapping both rows and columns, in the course of symmetric pivoting, is fairly complex and inefficient, and ultimately prevents GPU acceleration, as rows can only be swapped efficiently in row-major and columns in column-major. Swapping row in column-major or columns in row-major creates the worst possible memory access pattern, ultimately nullifying any potential gains from GPU acceleration.

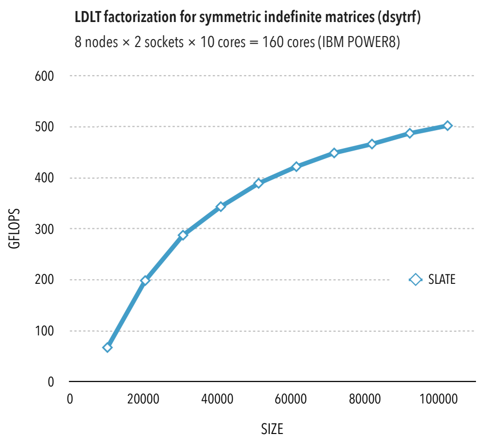


Figure 3.3: Performance of `dsytrf` without acceleration.

An alternative to the Aasen's LDLT algorithms is clearly needed, to solve symmetric indefinite problems at large scale and with GPU acceleration. One viable option may be the solver proposed by Becker and Baboulin [10, 11], which avoids pivoting by applying random butterfly transformations (RBT) and iterative refinement.

CHAPTER 4

Summary

Bibliography

[1] Jakub Kurzak, Panruo Wu, Mark Gates, Ichitaro Yamazaki, Piotr Luszczek, Gerald Ragghianti, and Jack Dongarra. SLATE working note 3: Designing SLATE: Software for linear algebra targeting exascale. Technical Report ICL-UT-17-06, Innovative Computing Laboratory, University of Tennessee, September 2017. revision 09-2017.

[2] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Mathieu Faverge, Thomas Hérault, and Jack J Dongarra. PaRSEC: Exploiting heterogeneity to enhance scalability. *Computing in Science & Engineering*, 15(6):36–45, 2013.

[3] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. Legion: Expressing locality and independence with logical regions. In *Proceedings of the international conference on high performance computing, networking, storage and analysis*, page 66. IEEE Computer Society Press, 2012.

[4] Jaeyoung Choi, Jack Dongarra, Susan Ostrouchov, Antoine Petitet, David Walker, and Clint Whaley. Design and implementation of the ScaLAPACK LU, QR, and Cholesky factorization routines. *Scientific Programming*, 5(3):173–184, 1996.

[5] Anthony Castaldo and Clint Whaley. Scaling LAPACK panel operations using parallel cache assignment. In *ACM Sigplan Notices*, volume 45, pages 223–232. ACM, 2010.

[6] Jack Dongarra, Mathieu Faverge, Hatem Ltaief, and Piotr Luszczek. Achieving numerical accuracy and high performance using recursive tile LU factorization with partial pivoting. *Concurrency and Computation: Practice and Experience*, 26(7):1408–1431, 2014.

[7] Grey Ballard, Dulceneia Becker, James Demmel, Jack Dongarra, Alex Druinsky, Inon Peled, Oded Schwartz, Sivan Toledo, and Ichitaro Yamazaki. Communication-avoiding symmetric-indefinite factorization. *SIAM Journal on Matrix Analysis and Applications*, 35(4): 1364–1406, 2014.

[8] Ichitaro Yamazaki and Jack Dongarra. LAPACK working note 294: Aasen's symmetric indefinite linear solvers in LAPACK. Technical Report ICL-UT-17-13, Innovative Computing Laboratory, University of Tennessee, Decembter 2017.

[9] Ichitaro Yamazaki, Jakub Kurzak, Panruo Wu, Mawussi Zounon, and Jack Dongarra. Symmetric indefinite linear solver using OpenMP task on multicore architecture. *IEEE Transactions on Parallel and Distributed Systems*, 2018.

[10] Dulceneia Becker, Marc Baboulin, and Jack Dongarra. Reducing the amount of pivoting in symmetric indefinite systems. In *International Conference on Parallel Processing and Applied Mathematics*, pages 133–142. Springer, 2011.

[11] Marc Baboulin, Dulceneia Becker, and Jack Dongarra. A parallel tiled solver for dense symmetric indefinite systems on multicore architectures. In *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 14–24. IEEE, 2012.

[12] Rajeev Thakur and William Gropp. Test suite for evaluating performance of multithreaded MPI communication. *Parallel Computing*, 35(12):608–617, 2009.

Appendices

APPENDIX A

Function Signatures

SLATE follows the (Sca)LAPACK convention of allowing to factor the coefficient matrix once, and applying the factorization multiple times to different sets of right-hand side vectors. Same as (Sca)LAPACK, SLATE provides three routines for each linear solver:

- a `[po|ge|he]sv` routine that factors the coefficient matrix and applies the forward and backward substitution to solve the system of linear equations,
- a `[po|ge|he]trf` routine that only factors the coefficient matrix, and requires a follow up call to the `[po|ge|he]trs` routine,
- a `[po|ge|he]trs` routine that only applies the forward and backward substitution, using a previously factored matrix.

The signatures of the functions for solving symmetric positive definite systems of equations, using the Cholesky factorization, are as follows:

```
template <typename scalar_t>
void posv(HermitianMatrix<scalar_t>& A,
          Matrix<scalar_t>& B,
          const std::map<Option, Value>& opts = std::map<Option, Value>());

template <typename scalar_t>
void potrf(HermitianMatrix<scalar_t>& A,
           const std::map<Option, Value>& opts = std::map<Option, Value>());

template <typename scalar_t>
void potrs(HermitianMatrix<scalar_t>& A,
           Matrix<scalar_t>& B,
           const std::map<Option, Value>& opts = std::map<Option, Value>());
```

The signatures of the functions for solving general (nonsymmetric) systems of equations, using Gaussian elimination with partial pivoting, are as follows:

```
template <typename scalar_t>
void gesv(Matrix<scalar_t>& A, Pivots& pivots,
          Matrix<scalar_t>& B,
          const std::map<Option, Value>& opts = std::map<Option, Value>());

template <typename scalar_t>
void getrf(Matrix<scalar_t>& A, Pivots& pivots,
           const std::map<Option, Value>& opts = std::map<Option, Value>());

template <typename scalar_t>
void getrs(Matrix<scalar_t>& A, Pivots& pivots,
           Matrix<scalar_t>& B,
           const std::map<Option, Value>& opts = std::map<Option, Value>());
```

The signatures of the functions for solving symmetric indefinite systems of equations, using Aasen's LDLT factorizatin, are as follows:

```
template <typename scalar_t>
void hesv(HermitianMatrix<scalar_t>& A, Pivots& pivots,
          BandMatrix<scalar_t>& T, Pivots& pivots2,
          Matrix<scalar_t>& B,
          const std::map<Option, Value>& opts = std::map<Option, Value>());

template <typename scalar_t>
void hetrf(HermitianMatrix<scalar_t>& A, Pivots& pivots,
           BandMatrix<scalar_t>& T, Pivots& pivots2,
           const std::map<Option, Value>& opts = std::map<Option, Value>());

template <typename scalar_t>
void hetrs(HermitianMatrix<scalar_t>& A, Pivots& pivots,
           BandMatrix<scalar_t>& T, Pivots& pivots2,
           Matrix<scalar_t>& B,
           const std::map<Option, Value>& opts = std::map<Option, Value>());
```

The Aasen's LDLT algorithm requires the solution of a general band system of linear equations, using the band rendition of the Gaussian elimination with partial pivoting. Since a standalone band solver is a useful tool, the following routines were also made available to the users:

```
template <typename scalar_t>
void gbsv(BandMatrix<scalar_t>& A, Pivots& pivots,
          Matrix<scalar_t>& B,
          const std::map<Option, Value>& opts = std::map<Option, Value>());

template <typename scalar_t>
void gbtrf(BandMatrix<scalar_t>& A, Pivots& pivots,
           const std::map<Option, Value>& opts = std::map<Option, Value>());

template <typename scalar_t>
void gbtrs(BandMatrix<scalar_t>& A, Pivots& pivots,
           Matrix<scalar_t>& B,
           const std::map<Option, Value>& opts = std::map<Option, Value>());
```

APPENDIX B

Implementation Snippets

The following code snippet shows the body of the top-level routine implementing the Cholesky factorization, which is the simplest of the three factorizations. The Gaussian elimination routine is significantly more complex, and the Aasen's LDLT routine is the most complex of all three. The main loop of the Cholesky routine contains three blocks of code:

Panel factorization contains factorization of the diagonal block, broadcast of the diagonal block down the column below, triangular solves below the diagonal block, and broadcast of the panel across the trailing submatrix.

Lookahead update contains Hermitian rank-k updates to the diagonal blocks of the lookahead columns of the trailing submatrix, and `gemm` updates to the rest of the blocks.

Trailing update contains Hermitian rank-k updates to the diagonal blocks of the trailing submatrix, and `gemm` updates to the rest of the of the blocks.

```
#pragma omp parallel
#pragma omp master
for (int64_t k = 0; k < A_nt; ++k) {
    // panel, high priority
    #pragma omp task depend(inout:column[k]) priority(1)
    {
        // factor A(k, k)
        internal::potrf<Target::HostTask>(A.sub(k, k), 1);

        // send A(k, k) down col A(k+1:nt-1, k)
        if (k+1 <= A_nt-1)
            A.tileBcast(k, k, A.sub(k+1, A_nt-1, k, k));

        // A(k+1:nt-1, k) * A(k, k)^{-H}
        if (k+1 <= A_nt-1) {
            auto Akk = A.sub(k, k);
            auto Tkk = TriangularMatrix<scalar_t>(Diag::NonUnit, Akk);
            internal::trsm<Target::HostTask>(
                Side::Right,
                scalar_t(1.0), conj_transpose(Tkk),
                A.sub(k+1, A_nt-1, k, k), 1);
        }

        BcastList bcast_list_A;
        for (int64_t i = k+1; i < A_nt; ++i) {
            // send A(i, k) across row A(i, k+1:i) and down col A(i:nt-1, i)
            bcast_list_A.push_back({i, k, {A.sub(i, i, k+1, i),
                                           A.sub(i, A_nt-1, i, i)}});
        }
        A.template listBcast(bcast_list_A);
    }
    // update lookahead column(s), high priority
    for (int64_t j = k+1; j < k+1+lookahead && j < A_nt; ++j) {
        #pragma omp task depend(in:column[k]) \
                         depend(inout:column[j]) priority(1)
        {
            // A(j, j) -= A(j, k) * A(j, k)^H
            internal::herk<Target::HostTask>(
                real_t(-1.0), A.sub(j, j, k, k),
                real_t( 1.0), A.sub(j, j), 1);

            // A(j+1:nt-1, j) -= A(j+1:nt-1, k) * A(j, k)^H
            if (j+1 <= A_nt-1) {
                auto Ajk = A.sub(j, j, k, k);
                internal::gemm<Target::HostTask>(
                    scalar_t(-1.0), A.sub(j+1, A_nt-1, k, k),
                                    conj_transpose(Ajk),
                    scalar_t(1.0), A.sub(j+1, A_nt-1, j, j), 1);
            }
        }
    }
    // update trailing submatrix, normal priority
    if (k+1+lookahead < A_nt) {
        #pragma omp task depend(in:column[k]) \
                         depend(inout:column[k+1+lookahead]) \
                         depend(inout:column[A_nt-1])
        {
            // A(kl+1:nt-1, kl+1:nt-1) -=
            //     A(kl+1:nt-1, k) * A(kl+1:nt-1, k)^H
            // where kl = k + lookahead
            internal::herk<target>(
                real_t(-1.0), A.sub(k+1+lookahead, A_nt-1, k, k),
                real_t( 1.0), A.sub(k+1+lookahead, A_nt-1));
        }
    }
}
```

APPENDIX C

MPI Thread Safety Considerations

Distributed memory processing in SLATE is based on MPI, while node-level parallelism relies on OpenMP. It was not until now, tough, that we faced the problem of thread safety of MPI. The LU and LDLT factorizations, presented int his report, actually require support for the strongest thread safety mode of MPI, where multiple threads can issue MPI calls simultaneously, as explained in Section 2.1.2.

The MPI specification outlines four modes of thread safety in version 2 of the MPI standard [1]: single, funneled, serialized, and multiple. Of these, we are specifically interested in the strong concurrency protections and flexibility provided by `MPI_THREAD_MULTIPLE` option (MTM), and we will evaluate this thread safety level in four popular MPI implementations: Intel MPI [2], OpenMPI [3], MVAPICH [4], and IBM Spectrum MPI [5].

C.1 MPI Implementations

Intel MPI has provided MTM support since version 3.0 [6]. We will evaluate version 2018. The user should link the Intel MPI library with the `-lmt_mpi` linking option and can optionally provide the environment variables `I_MPI_PIN_DOMAIN` and `KMP_AFFINITY` to control process and thread affinity [7].

[1] http://micro.ustc.edu.cn/Linux/MPI/mpi-20.pdf
[2] https://software.intel.com/en-us/mpi-library
[3] https://www.open-mpi.org/
[4] http://mvapich.cse.ohio-state.edu/
[5] https://www.ibm.com/us-en/marketplace/spectrum-mpi
[6] https://software.intel.com/en-us/articles/intel-mpi-library-for-linux-main-features-faq
[7] https://software.intel.com/en-us/articles/hybrid-applications-intelmpi-openmp

OpenMPI provides MTM support since release version 2.0 [8], and this support is enabled by default since version 3.0. We test both versions 2.1.1 and 3.0.0 in this investigation.

MVAPICH (based on MPICH) has provided MTM support since release version 2.0 [9], and we will test version 2.3. Use of MTM support in MVAPICH requires setting of the environment variable `MV2_ENABLE_AFFINITY=0` at runtime.

IBM Spectrum MPI provides MTM support since version 10.1 [10], and we evalutaed version 10.2.0.0. As IBM Spectrum MPI is based on the OpenMPI 3.x release, no additional configuration is required for MTM support.

C.2 Multithreading Tests

We are interested in verification of the correctness and stability of the MTM support in each MPI implementation by use of direct tests of MTM functionality as described by Thakur and Gropp [12] using the provided source code [11]. We evaluate each MPI implementation using these tests:

(1) Concurrent latency

(2) Concurrent bandwidth

(3) Message rate

(4) Concurrent short-long messages

(5) Computation and communication overlap

(6) Concurrent collectives

(7) Concurrent collectives and computation

C.3 Testing Methodology and Results

We compile the threading test code against each MPI implementation and run each test on either two or three compute nodes (as required by the specific test) using the Slurm job scheduler [12] `srun` process launcher. We define a test as “passed” if the test code builds and runs against the given MPI implementation with correct output (independent of code performance).

Table C.1 summaries the testing results. The five MPI versions passed all test with the exception of test #5 (Computation and communication overlap) which caused an apparent deadlock in Intel MPI and MVAPICH. This will require further investigation to determine the cause of this failure.

[8] https://www.open-mpi.org/doc/v2.0/man3/MPI_Init_thread.3.php
[9] http://mvapich.cse.ohio-state.edu/features/
[10] https://www.ibm.com/support/knowledgecenter/en/SSZTET_10.1/smpi02/smpi02_features_threadsafety.html
[11] http://www.mcs.anl.gov/~thakur/thread-tests/
[12] https://slurm.schedmd.com/

Table C.1: MPI thread safety testing results.

MPI implementation	Version tested	Test notes
Intel MPI	2018	Failed test #5
OpenMPI	2.1.1	All tests passed
OpenMPI	3.0.0	All tests passed
MVAPICH	2.3	Failed test #5
IBM Spectrum MPI	10.2.0.0	All tests passed