SLATE WORKING NOTES 6

# Parallel Norms Performance Report

Jakub Kurzak
Mark Gates
Asim YarKhan
Ichitaro Yamazaki
Piotr Luszczek
Jamie Finney
Jack Dongarra

Innovative Computing Laboratory

July 1, 2018

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** |
|---|---|
| 06-2018 | first publication |

```
@techreport{kurzak2018parallel,
  author={Kurzak, Jakub and Gates, Mark and YarKhan, Asim and
          Yamazaki, Ichitaro and Luszczek, Piotr and Finney, Jamie and
          Dongarra, Jack},
  title={{SLATE} Working Note 6: Parallel Norms Performance Report},
  institution={Innovative Computing Laboratory, University of Tennessee},
  year={2018},
  month={June},
  number={ICL-UT-XX-XX},
  note={revision 06-2018}
}
```

# Contents

# List of Figures

# 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 compute matrix norms. 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 norms routines, described in this report, are a notable exception to this rule, as the BLAS does not implement the underlying components.
- 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

All norm routines are embarrassingly parallel and basically boil down to a sequence of reductions. At the same time, SLATE implementations 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.
- 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.

Figure 2.1 illustrates the most complicated case, where the matrix is spread across multiple distributed memory nodes and across multiple accelerators in each node. The figure shows the stages of computing the one norm, i.e., finding the maximum column sum. This means computing the sum of all elements in each column and then finding the maximum sum. Figure 2.1 illustrates the case with four nodes in a 2D block cyclic arrangement and four devices per node, also in a 2D block cyclic arrangement. The process consists of four steps:

(1) Within each node, each device computes column sums for each of its tiles, using a specialized device kernel.
(2) Within each node, contributions from all devices are summed up to a local vector of partial sums using `blas::axpy`.
(3) Partial sums from all the nodes are summed up using `MPI_Allreduce`.
(4) Within each node, the maximum sum is found using `lapack::lange`.

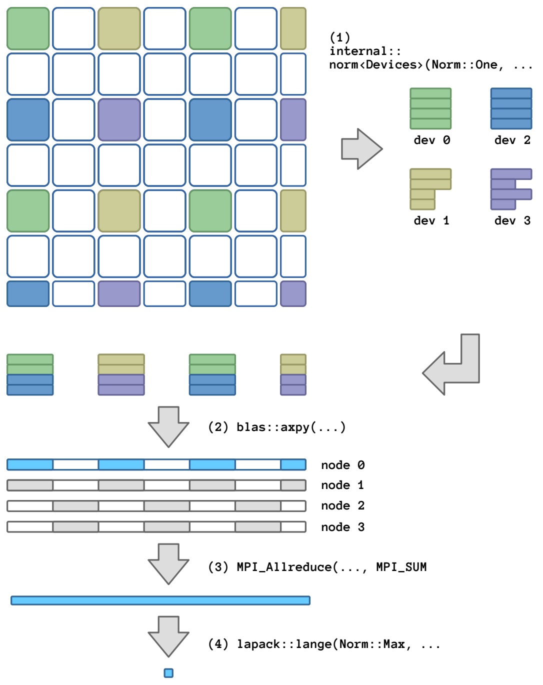

(1)
internal::
norm<Devices>(Norm::One, ...
dev 0
dev 2
dev 1
dev 3
(2) blas::axpy(...)
node 0
node 1
node 2
node 3
(3) MPI_Allreduce(..., MPI_SUM
(4) lapack::lange(Norm::Max, ...

Figure 2.1: Stages of the one norm using four nodes in a 2D block cyclic arrangement and four devices per node in a 2D block cyclic arrangement.

The same basic approach is used to implement the other types of norms (max, infinity, Frobenius) and to support the other types or matrices (triangular, symmetric, Hermitian) with slightly higher complexity in the one norm and the infinity norm, due to the need to accumulate partial sums both along rows and columns.

The device kernels needed for step (1) are are currently implemented in CUDA. Each type of norm requires a slightly different implementation, and specialized kernels are needed for triangular and symmetric/Hermitian matrices.

**Infinity Norm:** Each thread block computes the partial row sums for one tile. The thread block is one dimensional and the number of threads is equal to to the number of rows. Each thread sums up the elements of one row. For column major tiles, the access is naturally coalesced.

**Max Norm:** Each thread block finds the maximum absolute value in one tile. The thread block is one dimensional and the number of threads is equal to to the number of rows. First, each thread finds the maximum absolute value in one row. For column major tiles, the access is naturally coalesced. Then all threads write their maximums to the shared memory, and then a binary tree reduction follows to find the tile maximum. The kernel properly propagates NaNs by using the expression `isnan(y) || x < y ? y : x` for element-wise comparison.

**Frobenius Norm:** This kernel is very similar to the max norm kernel. Each thread block computes the sum of squares for one tile. The thread block is one dimensional and the number of threads is equal to to the number of rows. First, each thread computes the sum of squares for one row. For column major tiles, the access is naturally coalesced. Then all threads write their sums to the shared memory, and then the partial sums are reduced to compute the sum of squares for the tile. To avoid unnecessary under or overflow, the sum of squares is computed using a scaled representation, in a similar fashion to the `lapack::lassq` function.

**One Norm:** This kernel is somewhat similar to the infinity norm kernel. Each thread block computes the partial column sums for one tile. Here, however, the simple implementation, with one thread per column, produces non coalesced memory access pattern. Therefore, the operation is blocked with a fixed blocking factor $B$ (currently simply one warp of 32 threads). The thread block consists of $B$ threads, which go through the tile in $B \times B$ blocks. Each block is first loaded to the shared memory, using a coalesced access pattern, and then read from the shared memory to compute the partial sums.

In the case of multithreaded execution, without acceleration, operations on individual tiles are dispatched as OpenMP tasks. For a single tile, the max norm is implemented as a single call to `lapack::lange`, the Frobenius norm is implemented as a single loop over `lapack::lassq`, and the one and infinity norms are implemented as double nested loops over the tile's elements. Otherwise, the same procedure applies for reducing local contributions within each node and for finding the global value across all nodes.

## 2.2 Handling Different Types of Matrices

SLATE has a single `slate::norm()` function that is templated for the matrix type. Internally it dispatches to different implementations for general, trapezoid (triangular), and symmetric/Hermitian matrices. The implementations are similar, the difference being what tiles are accessed and whether diagonal tiles are handled specially.

For trapezoid and symmetric matrices, only half of the matrix is stored, either the lower half or the upper half. The other side is assumed to be zero for trapezoid matrices, and known by symmetry for symmetric matrices. SLATE loops over tiles only in the given half, so it reads half the data as for the same sized general matrix.

For trapezoid and triangular matrices, off-diagonal tiles use the same kernels as for a general matrix, but diagonal tiles require specialized kernels that access only either the lower or upper triangle, assuming the opposite triangle is zero. The specialized kernels also deal with the diag option that specifies whether the matrix is assumed to have unit diagonal or not.

For symmetric matrices, the max norm is essentially identical to the triangular matrix case (less the diag option). For the symmetric matrix Frobenius norm, off-diagonal tiles use the same kernel as for general matrices, the result of which is doubled to account for symmetry. A specialized kernel is used for diagonal tiles to sum up squares of off-diagonal entries, double the result, then add squares of diagonal entries. The symmetric matrix one norm, which is the same as the infinity norm, has specialized kernels for both diagonal and off-diagonal tiles. The diagonal kernel computes column sums, taking symmetry of the tile into account. The off-diagonal kernel computes both column and row sums of the tile, the row sums being column sums for the symmetric tile. This allows each tile to be read only once. The tile column and row sums are then reduced into the appropriate matrix column sums. With respect to these four norms, Hermitian matrices can be treated as symmetric matrices, since the absolute value of each element is taken.

## 2.3 Handling of Multiple Precisions

SLATE handles multiple precisions by C++ templating, so there is only one precision-independent version of the code, which is then instantiated for the desired precisions. SLATE's LAPACK++ component [4] provides overloaded, precision-independent wrappers for all the underlying LAPACK routines, which SLATE's parallel norms are built on top of. For instance, `lapack::lange` in LAPACK++ maps to the classical `slange`, `dlange`, `clange`, or `zlange` LAPACK routines, depending on the precision of its arguments.

Where a data type is always real, `blas::real_type<scalar_t>` is a C++ type trait to provide the real type associated with the type `scalar_t`, so `blas::real_type< std::complex<double> >` is `double`. Since norms of complex matrices are real values, this is used across the norms routines.

Currently, the SLATE library has explicit instantiations of the four main data types: `float`, `double`, `std::complex<float>`, and `std::complex<double>`. The SLATE norms code should be able to accommodate other data types, such as quad precision, given appropriate implementations of the elemental operations.

# 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/

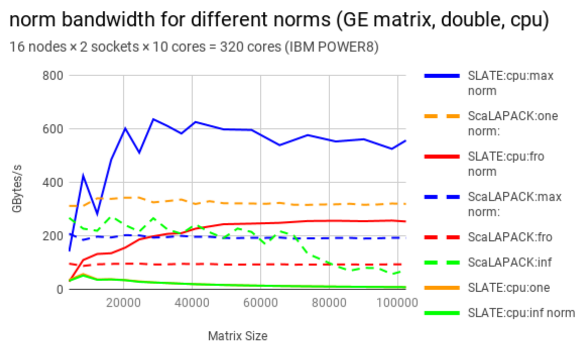
## 3.2 Performance

In order to avoid excessive numbers of run, while still getting the complete picture, we asses the performance of SLATE norms in the following way. First, in Section 3.2.1, we look at the performance of different kinds of norms for general matrices in double precision. Then, in Section 3.2.2, we present the performance of the max norm for general matrices in different precisions. Finally, in Section 3.2.3 we show the performance of the max norm for different types of matrices in double precision.

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 Different Kinds of Norms

Figure 3.1 show the performance of different kinds of norms for general matrices in double precision. SLATE runs were made with and without acceleration, while no acceleration was used for ScaLAPACK runs. We are not aware of a viable solution for accelerating ScaLAPACK. All runs used 16 nodes of the SummiDev system. For the accelerated runs, this translates to 64 accelerators.



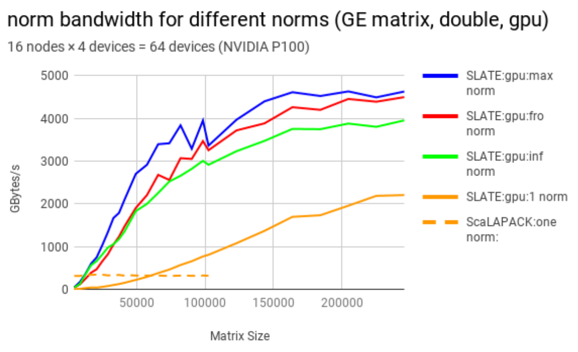


Figure 3.1: Performance for different types of norms (double precision, general matrix) without acceleration (left) and with acceleration (right).

In the cases of multithreaded runs, SLATE delivers superior performance for the max norm and the Frobenius norm, while the SLATE's implementations of the one norm and the infinity norm clearly suffer from a performance deficiency. In the case of accelerated runs, SLATE delivers superior performance across all norms, with the performance of the one norm significantly below the performance of other norms.

### 3.2.2 Different Precisions

Figure 3.2 shows the performance of the max norm for general matrices in different precisions. SLATE delivers superior performance for all cases, with huge performance benefit for the accelerated runs. At the same time, it is surprising that different precisions deliver different performance. In principle, norm calculations are memory bound, regardless of the precision, and should be saturating the bandwidth equally well. This outcome calls for further investigation.

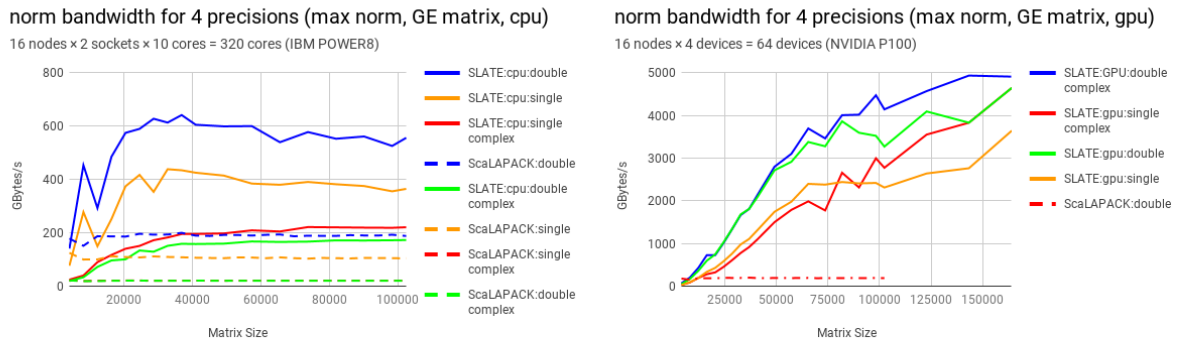


Figure 3.2: Performance for different precisions (max norm, general matrix) without acceleration (left) and with acceleration (right).

### 3.2.3 Different Types of Matrices

Figure 3.3 shows the performance of the max norm for different types of matrices in double precision. SLATE delivers superior performance for all cases, with huge performance benefit for the accelerated runs. At this point, we do not have an explanation for the spikes in the multithreaded performance of SLATE. They are reproducible. This calls for further investigation.

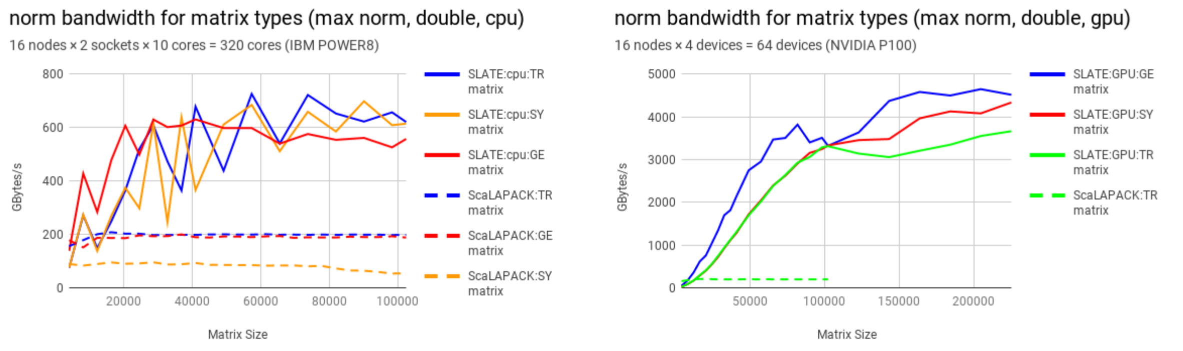


Figure 3.3: Performance for different types of matrices (double precision, max norm) without acceleration (left) and with acceleration (right).

# CHAPTER 4

# Summary

Due to the complexity of the target hardware, the implementations of norms are non trivial, despite the simplicity of their mathematical definitions. More complexity is caused by removing ScaLAPACK constraints on the arrangement of the matrix in the memory and the mapping of the matrix to the distributed memory nodes. This results in multiple reduction stages across multiple levels of the memory system.

Despite the complexity of the implementation, SLATE delivers superior performance across all accelerated runs. The remaining area of concern is SLATE's multithreaded performance for the one norm and infinity norm. The follow-up performance engineering effort by the SLATE team should result in a swift resolution of these issues.

# 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] Mark Gates, Piotr Luszczek, Ahmad Abdelfattah, Jakub Kurzak, Jack Dongarra, Konstantin Arturov, Cris Cecka, and Chip Freitag. SLATE working note 2: C++ API for BLAS and LAPACK. Technical Report ICL-UT-17-03, Innovative Computing Laboratory, University of Tennessee, June 2017. revision 03-2018.

[5] Anthony M Castaldo, R Clint Whaley, and Anthony T Chronopoulos. Reducing floating point error in dot product using the superblock family of algorithms. *SIAM Journal on Scientific Computing*, 31(2):1156–1174, 2008.

## Appendices

# APPENDIX A

## Function Signature

SLATE has a single `slate::norm()` function that is templated for the matrix type. Internally it dispatches to different implementations for general, trapezoid (triangular), and symmetric/Hermitian matrices. The `norm_type` option specifies whether to compute the max, infinity, one, or Frobenius norm. The optional `opts` argument specifies additional options, such as where the computation should be done as `Target::Host` or `Target::Devices`.

```
template <typename matrix_type>
blas::real_type<typename matrix_type::value_type>
norm(Norm norm_type, matrix_type& A,
     const std::map<Option, Value>& opts = std::map<Option, Value>());
```

# APPENDIX B

# Discovered ScaLAPACK Problems

## B.1 Indexing Bugs

In testing SLATE, it was discovered that the trapezoid norm, `p*lantr`, in ScaLAPACK had several indexing bugs in the one, infinity, and Frobenius norms. This lead to ScaLAPACK including either one extra row or one less row than ought to be included in the norm calculation. For testing purposes, SLATE has local copies of these routines with corrections, and a bug report has been sent to the ScaLAPACK maintainers.

## B.2 Accuracy Problems

The Frobenius norm uses a sum-of-squares. Ignoring scaling, a simplified implementation is (in Matlab notation):

```
sum = 0;
for j = 1:n
    for i = 1:m
        sum = sum + abs(A(i, j))^2;
     end
end
result = sqrt(sum);
```

As is well known [5], summing a large number of elements in the above naïve fashion can lead to significant error in floating point, since `sum` $\gg$ `abs(A(i, j))^2`. Summation using a blocked algorithm is more accurate, since the numbers that are added are of similar magnitude. SLATE naturally sums numbers in a blocked fashion: each tile is summed, and then the tile sums are reduced to the final result. While testing SLATE's Frobenius norms, the single precision results showed large differences compared to ScaLAPACK. Within each process, ScaLAPACK does a simple summation, leading to an error that grows with the matrix size, becoming noticeable for $n \geq 1000$. This is easily fixed with a modification in ScaLAPACK to sum each column individually, and accumulate the column sums. This adds trivial overhead, so does not change the performance of ScaLAPACK.

```
sum = 0;
for j = 1:n
    colsum = 0;
    for i = 1:m
        colsum = colsum + abs(A(i, j))^2;
     end
     sum = sum + colsum;
end
result = sqrt(sum);
```

For testing purposes, SLATE has local copies of ScaLAPACK's norm routines (`p*lange, p*lansy, p*lantr`) with this modification. The issue likewise affects LAPACK. An issue report has been sent to the (Sca)LAPACK maintainers [1].

[1]https://github.com/Reference-LAPACK/lapack/issues/261