SLATE WORKING NOTES 18

Communication Avoiding LU with Tournament Pivoting in SLATE

Rabab Alomairy
Mark Gates
Sebastien Cayrols
Dalal Sukkari
Kadir Akbudak
Asim YarKhan
Paul Bagwell
Jack Dongarra

Innovative Computing Laboratory

January 19, 2022

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 Shaheen-II from King Abdullah University of Science and Technology (KAUST). It used as well 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**
01-2022	first publication

```
@techreport{rabab2022slate-calu,
  author={Rabab Alomairy and Mark Gates and Sebastien Cayrols and Dalal Sukkari
  and Kadir Akbudak  and Asim YarKhan and Paul Bagwell and Jack Dongarra},
  title={Communication Avoiding {LU} with Tournament Pivoting in {SLATE}, {SWAN} No. 18},
  institution={Innovative Computing Laboratory, University of Tennessee},
  year={2022},
  month={11},
  number={ICL-UT-22-01},
  note={revision 01-2022},
  url={https://www.icl.utk.edu/publications/swan-018},
}
```

Contents

List of Figures i

List of Tables i

1 Introduction 1

2 Communication Avoiding LU Factorization Algorithm 1

2.1 Overview . . . 1

2.2 Implementation Details . . . 1

2.3 CALU Algorithm in SLATE . . . 2

3 Performance Results 7

References 9

List of Figures

1 Panel Factorization of CALU. . . . 2

2 CALU numerical kernels and the involved inter-communication. The highlighted tiles presents the working set of each row, where red color denotes input tiles and black color denotes input/output tiles. . . . 3

3 Performance of LU variant in SLATE on 16 nodes of Shaheen-II. . . . 7

4 Strong scaling results on Shaheen-II. . . . 8

5 Overhead of CALU panel factorization of 80k matrix size and 320 tile size on 16 nodes of Shaheen-II. . . . 8

6 Performance of LU tntpiv and LU RBT on Summit. . . . 9

7 CALU based on local and global tree reduction. . . . 10

List of Algorithms

2.1 Panel factorization, `getrf_tntpiv` and lower `trsm` computational routines . . . 4

2.2 Trailing Lookahead update, upper `trsm` and `gemm` and computational routines . . . 5

2.3 Trailing update of the remaining matrix, upper `trsm` and `gemm` and computational routines . . . 6

List of Tables

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 distributed, GPU-accelerated dense linear algebra capabilities to the US Department of Energy and to the high-performance computing (HPC) community at large.

This report discusses the implementation of communication avoiding LU algorithm in SLATE. It is based on tournament pivoting strategy, that is shown to be stable in practice.

2 Communication Avoiding LU Factorization Algorithm

2.1 Overview

Since the cost of data communication has significantly outpaced the costs of calculations on current and future architectures, we are motivated to use algorithms that communicate as little as possible at the expense of doing more computation, or storing redundant data. Therefore, we extend the current implementation of the LU factorization in SLATE using the tournament pivoting strategy designed by Grigori et al [1]. It is called Communication-Avoiding LU factorization (CALU) and it has been proven to be stable in practice [2]. The main idea of CALU is to reduce the number of messages exchanged during the panel factorization by performing redundant arithmetic operations. The main difference between CALU and classical LU (i.e partial pivoting LU) lies in the panel factorization. In LU, the processors need to synchronize for each column of the panel, while in CALU processors need to synchronize only for each column block of the panel.

2.2 Implementation Details

The panel factorization using CALU is illustrated in Fig. 1. The algorithm presented in this report bears some similarities to Communication-Avoiding QR (CAQR) described in [3]. CALU divides the panel into tiles of size b and distributes those tiles following block cyclic distribution. First, each processor performs standard LU on its local tiles (equivalent of the LAPACK `getrf` routine) to identify in parallel pivot rows, applies the resulting permutation vector on the original local tiles, then keeps the first b rows of its permuted local blocks as the pivot candidates for the next level in the computation. This step is represented by the dash arrows in 1. The next step applies a binary tree reduction of the pairwise reductions of the permuted rows, where the pivot candidates are placed one on top another at each node of the tree. Then, LU with partial pivoting is applied again as in the first step. The obtained pivot vector is applied to the original merged tiles and the pivot candidates are chosen for the next level of the reduction. At the end of the reduction step, the good pivot candidate are moved on to the top of the panel with L and

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

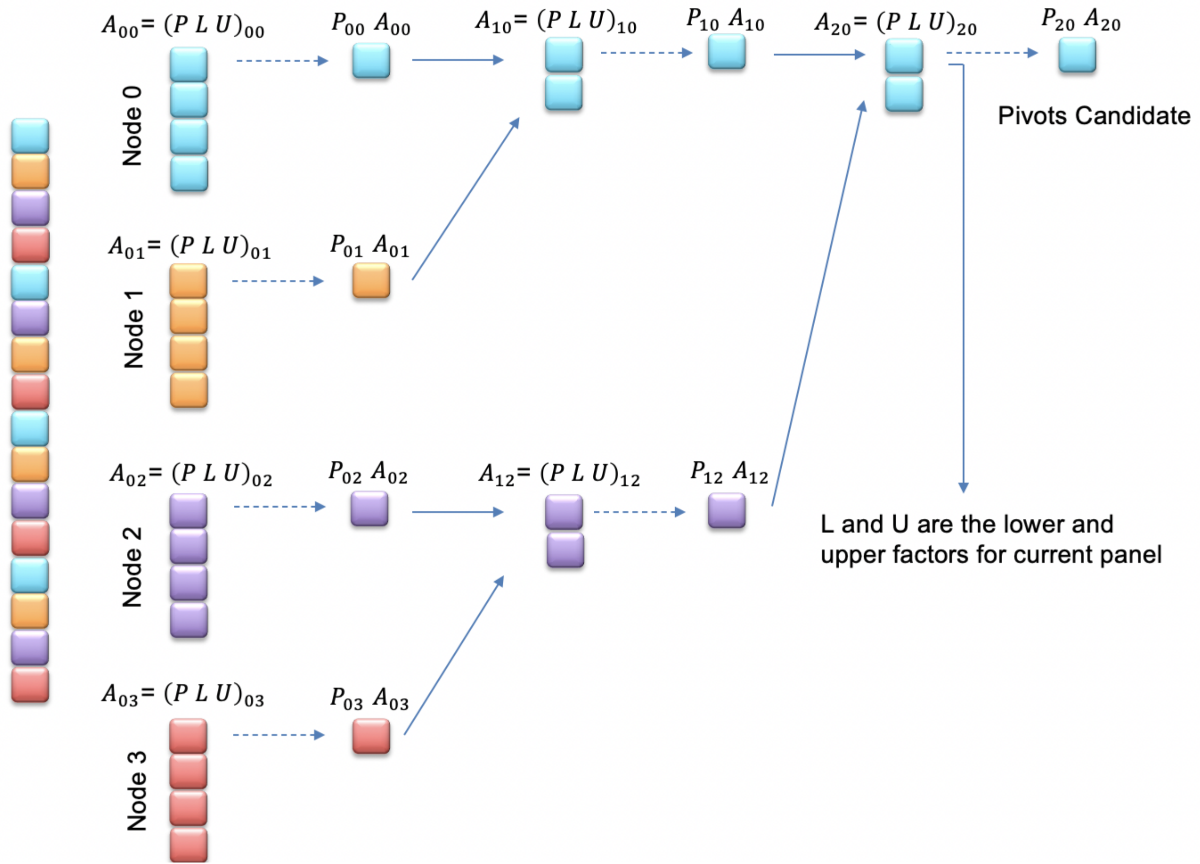

Figure 1: Panel Factorization of CALU.

U factors of the current panel. The resulting pivot rows are different than the one produced by the standard LU factorization of LAPACK and ScaLAPACK.

The steps of the algorithm are illustrated on Fig. 2. After obtaining a good pivot of the current panel, the algorithm proceeds to update the trailing submatrix, which which involves: (1) applying row swaps (LASWAP), (2) upper/lower triangular solve (TRSM), (3) matrix-matrix multiplication (GEMM). This require the following communication: (1) "horizontal" broadcasting of the panel to the right, (2) "vertical" exchanges of the rows being swapped, and (3) "vertical" broadcasting of the top row or tiles down the matrix.

2.3 CALU Algorithm in SLATE

CALU algorithm is very similar to classical LU algorithm in SLATE. Algorithm in 2.1, 2.2, 2.3 shows the computational routines within `#omp pragma task`. Algorithm starts iterating over those tasks until it reaches the number of tiles. It consists of four computational kernels i.e., GETRF_TNTPIV (CALU factorization), TRSM (triangular solve), GEMM (general matrix multiply) and LASWAP (row swaps). Moreover, CALU applies same optimization as LU such us lookahead, internal blocking, and cache reuse,

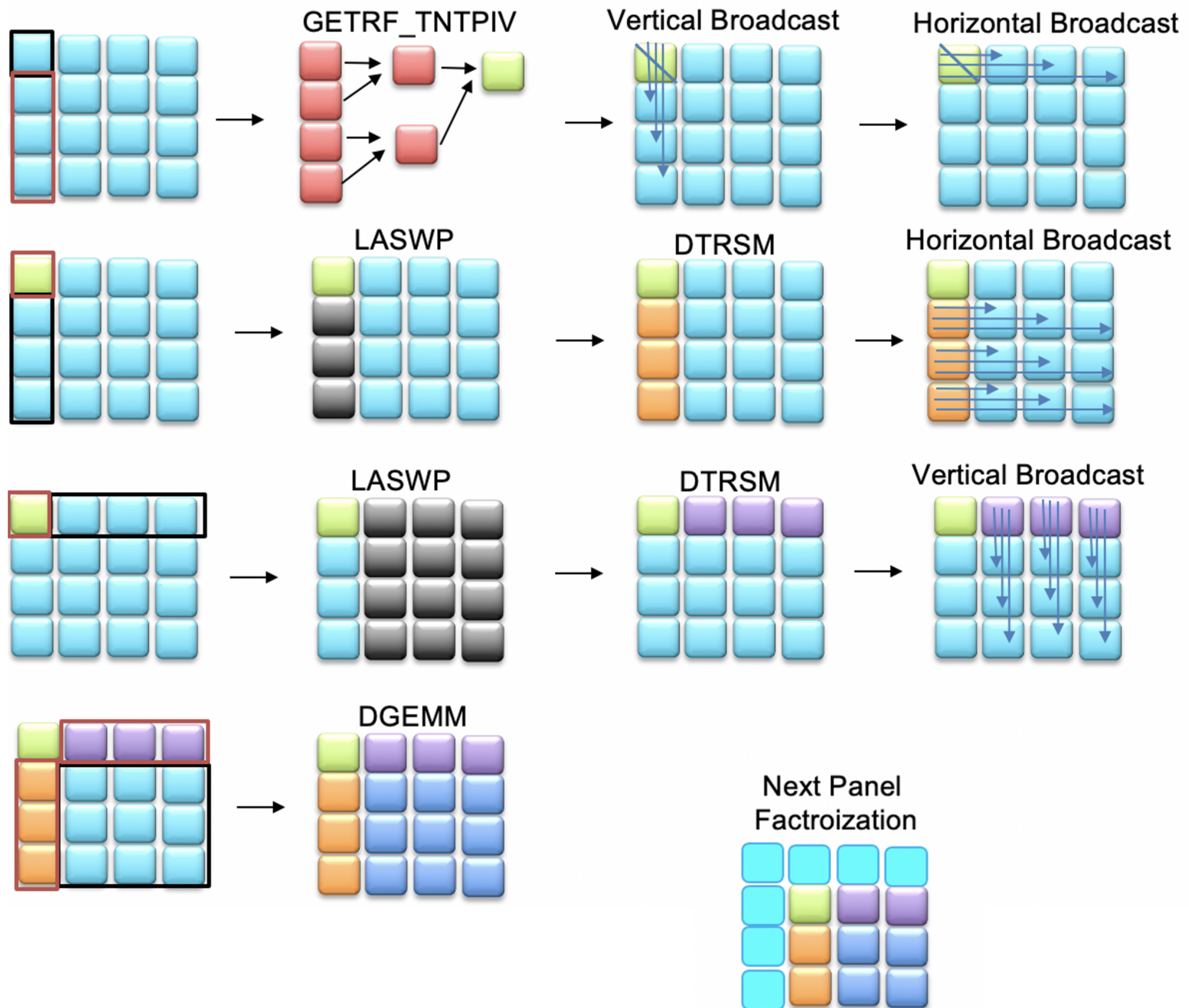

Figure 2: CALU numerical kernels and the involved inter-communication. The highlighted tiles presents the working set of each row, where red color denotes input tiles and black color denotes input/output tiles.

Algorithm 2.1 Panel factorization, getrf_tntpiv and lower trsm computational routines

```
// panel factorization                                                    1
#pragma omp task depend(inout:column[k])                                  2
{                                                                         3
    auto Apanel = Awork.sub( k, A_mt-1, k, k );                           4
    Apanel.insertLocalTiles();                                            5
    // factor A(k:mt-1, k)                                                6
    internal::getrf_tntpiv<Target::HostTask>(                             7
        A.sub(k, A_mt-1, k, k), std::move(Apanel), diag_len,              8
        ib, pivots.at(k), max_panel_threads, priority_one);               9
    // root broadcasts the pivot to all ranks.                            10
    MPI_Bcast(pivots.at(k).data(),                                        11
        sizeof(Pivot)*pivots.at(k).size(),                                12
        MPI_BYTE, A.tileRank(k, k), A.mpiComm());                         13
    // swap rows in A(k+1:A_mt-1, k)                                      14
    int tag_k = k;                                                        15
    internal::permuteRows<target>(                                        16
        Direction::Forward, A.sub(k, A_mt-1, k, k),                       17
        pivots.at(k),target_layout, priority_one, tag_k, queue_0);        18
                                                                          19
    internal::copy<Target::HostTask>(Apanel.sub( 0, 0, 0, 0 ),            20
        A.sub( k, k, k, k ));                                             21
    //broadcast panel                                                     22
     BcastList bcast_list_A;                                              23
     bcast_list_A.push_back({k, k, {A.sub(k+1, A_mt-1, k, k),             24
        A.sub(k, k, k+1, A_nt-1)}});                                      25
        A.template listBcast<target>(bcast_list_A,                        26
        host_layout, tag_k, life_factor_one, is_shared);                  27
                                                                          28
      Apanel.clear();                                                     29
}                                                                         30
//lower trsm                                                              31
#pragma omp task depend(inout:column[k]) depend(inout:listBcastMT_token)  32
{                                                                         33
    auto Akk = A.sub(k, k, k, k);                                         34
    auto Tkk = TriangularMatrix<scalar_t>(Uplo::Upper,                    35
         Diag::NonUnit, Akk);                                             36
                                                                          37
    internal::trsm<target>(                                               38
        Side::Right,                                                      39
        scalar_t(1.0), std::move(Tkk),                                    40
        A.sub(k+1, A_mt-1, k, k),                                         41
        priority_one, Layout::ColMajor, queue_0);                         42
    // bcast the tiles of the panel to the right hand side                43
    BcastListTag bcast_list;                                              44
    for (int64_t i = k+1; i < A_mt; ++i) {                                45
        const int64_t tag = i;                                            46
        bcast_list.push_back({i, k, {A.sub(i, i, k+1, A_nt-1)}, tag});    47
          }                                                               48
    A.template listBcastMT<target>(                                       49
      bcast_list, Layout::ColMajor, life_factor_one, is_shared);          50
}                                                                         51
```

Algorithm 2.2 Trailing Lookahead update, upper `trsm` and `gemm` and computational routines

```
                                                                          52
      // update lookahead column(s), high priority                        53
      for (int64_t j = k+1; j < k+1+lookahead && j < A_nt; ++j) {         54
          #pragma omp task depend(in:column[k]) \                         55
                           depend(inout:column[j]) \                      56
          {                                                               57
              int tag_j = j;                                              58
              internal::permuteRows<target>(                              59
                      Direction::Forward, A.sub(k, A_mt-1, j, j),         60
                      pivots.at(k), target_layout, priority_one,          61
                      tag_j, j-k+1);                                      62
                                                                          63
              auto Akk = A.sub(k, k, k, k);                               64
              auto Tkk = TriangularMatrix<scalar_t>(Uplo::Lower,          65
                       Diag::Unit, Akk);                                  66
                                                                          67
              // solve A(k, k) A(k, j) = A(k, j)                          68
              internal::trsm<target>(                                     69
                  Side::Left,                                             70
                  scalar_t(1.0), std::move(Tkk),                          71
                  A.sub(k, k, j, j), priority_one,                        72
                  Layout::ColMajor, j-k+1);                               73
              // send A(k, j) across column A(k+1:mt-1, j)                74
              A.tileBcast(k, j, A.sub(k+1, A_mt-1, j, j),                 75
              Layout::ColMajor, tag_j);                                   76
              // A(k+1:mt-1, j) -= A(k+1:mt-1, k) * A(k, j)               77
              internal::gemm<target>(                                     78
                      scalar_t(-1.0), A.sub(k+1, A_mt-1, k, k),           79
                                      A.sub(k, k, j, j),                  80
                      scalar_t(1.0),  A.sub(k+1, A_mt-1, j, j),           81
                      host_layout, priority_one, j-k+1);                  82
             }                                                            83
          }                                                               84
                                                                          85
     // pivot to the left                                                 86
     if (k > 0) {                                                         87
        #pragma omp task depend(in:column[k]) \                           88
                         depend(inout:column[0]) \                        89
                         depend(inout:column[k-1])                        90
        {                                                                 91
             // swap rows in A(k:mt-1, 0:k-1)                             92
             int tag = k;                                                 93
             internal::permuteRows<Target::HostTask>(                     94
                 Direction::Forward, A.sub(k, A_mt-1, 0, k-1),            95
                 pivots.at(k), host_layout, priority_zero,                96
                 tag, queue_0);                                           97
        }                                                                 98
     }                                                                    99
```

Algorithm 2.3 Trailing update of the remaining matrix, upper `trsm` and `gemm` and computational routines

```

// 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:listBcastMT_token) \
                     depend(inout:column[A_nt-1])
    {
        // swap rows in A(k:mt-1, kl+1:nt-1)
        int tag_kl1 = k+1+lookahead;
        internal::permuteRows<target>(
            Direction::Forward, A.sub(k, A_mt-1, k+1+lookahead, A_nt-1),
            pivots.at(k), target_layout, priority_zero, tag_kl1, queue_1);

        auto Akk = A.sub(k, k, k, k);
        auto Tkk = TriangularMatrix<scalar_t>(Uplo::Lower,
               Diag::Unit, Akk);

        // solve A(k, k) A(k, kl+1:nt-1) = A(k, kl+1:nt-1)
        internal::trsm<target>(
               Side::Left,
               scalar_t(1.0), std::move(Tkk),
                              A.sub(k, k, k+1+lookahead, A_nt-1),
               priority_zero, Layout::ColMajor, queue_1);

        // send A(k, kl+1:A_nt-1) across A(k+1:mt-1, kl+1:nt-1)
        BcastListTag bcast_list;
            for (int64_t j = k+1+lookahead; j < A_nt; ++j) {
                // send A(k, j) across column A(k+1:mt-1, j)
                // tag must be distinct from sending left panel
                const int64_t tag = j + A_mt;
                bcast_list.push_back({k, j,
                   {A.sub(k+1, A_mt-1, j, j)}, tag});
            }

        A.template listBcastMT<target>(
                bcast_list, Layout::ColMajor);

        // A(k+1:mt-1, kl+1:nt-1) -= A(k+1:mt-1, k) * A(k, kl+1:nt-1)
        internal::gemm<target>(
                scalar_t(-1.0), A.sub(k+1, A_mt-1, k, k),
                                A.sub(k, k, k+1+lookahead, A_nt-1),
                scalar_t(1.0),  A.sub(k+1, A_mt-1, k+1+lookahead, A_nt-1),
                host_layout, priority_zero, queue_1);
    }
}
```

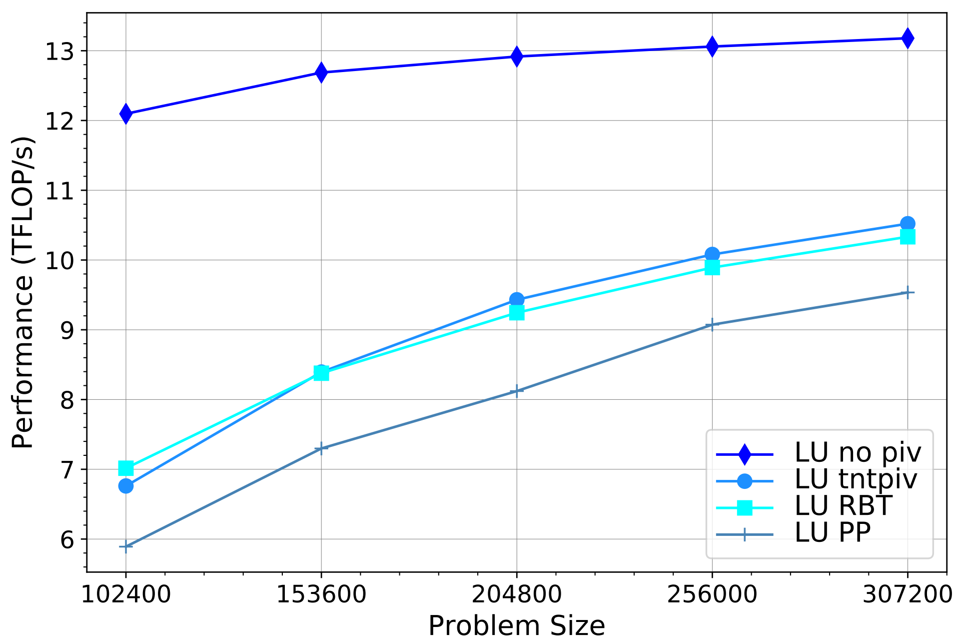

Figure 3: Performance of LU variant in SLATE on 16 nodes of Shaheen-II.

3 Performance Results

Figs. 3 shows performance of four different variant of LU factorization in SLATE: (1) LU no pivoting (`LU nopiv`) which work well for specific classes of matrices, such as diagonally dominant matrices and totally nonnegative matrice, (2) CALU (`LU tntpiv`) which is based on tournament pivoting, (3) LU based on Recursive Butterfly Transformation RBF (`LU RBT`) [4], and (4) LU with partial pivoting (`LU PP`) [5]. The performance is reported in double precision using 16 nodes of Shaheen-II, Cray XC40 with Dual-Socket 16-core Intel Haswell System. We can notice that CALU and RBT behave similarly and they outperform LU with partial pivoting. CALU achieves 1.16x speedup compared to partial pivoting.

To farther asses performance of the implementation, Figs. 4 shows the results when scaling number of node. We can see that for small number of nodes CALU and LU partial pivoting deliver similar performance, However, as we add more node we can notice that communication avoiding LU outperform partial pivoting LU.

Figs. 6 shows the performance of LU tntpiv and LU PP on 8 nodes of Summit each with 6 GPUs. LU PP outperforms LU tntpiv for dominant random matrices because the reduction step overhead in the panel factorization, which is not nessasray in the type of matrices. However, for completely random matrices, CALU achieves 1.15x speedup compared to LU partial pivoting.

The critical component of the CALU factorization is the step of factorizing the panel. We assess performance of this step by showing the overhead of the first phase where each node is doing the first LU for its local tiles. And the tree reduction phase. Figs. 5 shows that 42% of time is spent in this first phase and 39% on tree reduction and the remaining 19% of the panel factorization is spent for different auxiliary SLATE functions such as `copy`, and `tileGetForWriting` routines.

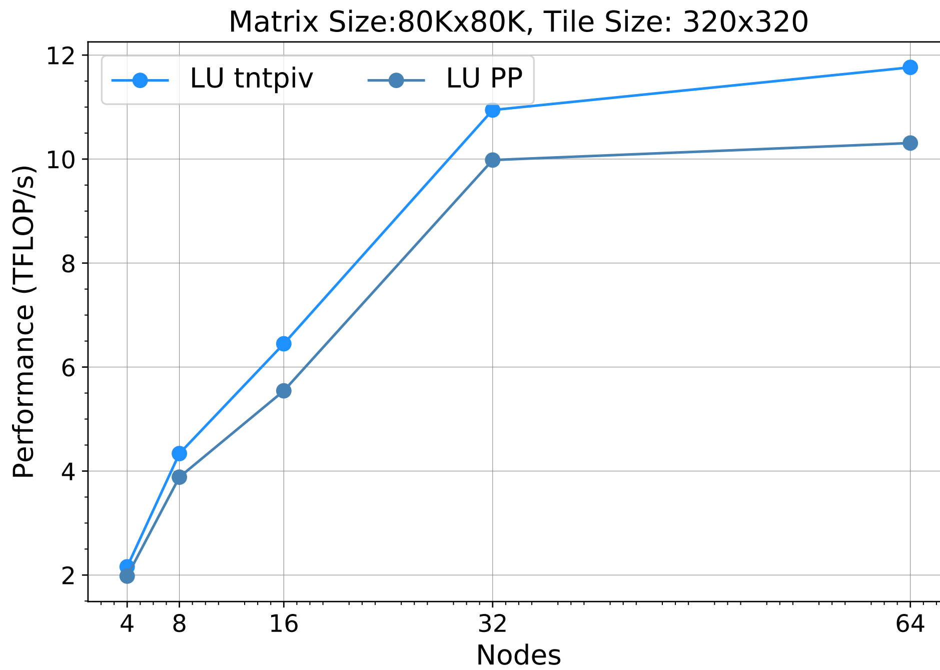

Figure 4: Strong scaling results on Shaheen-II.

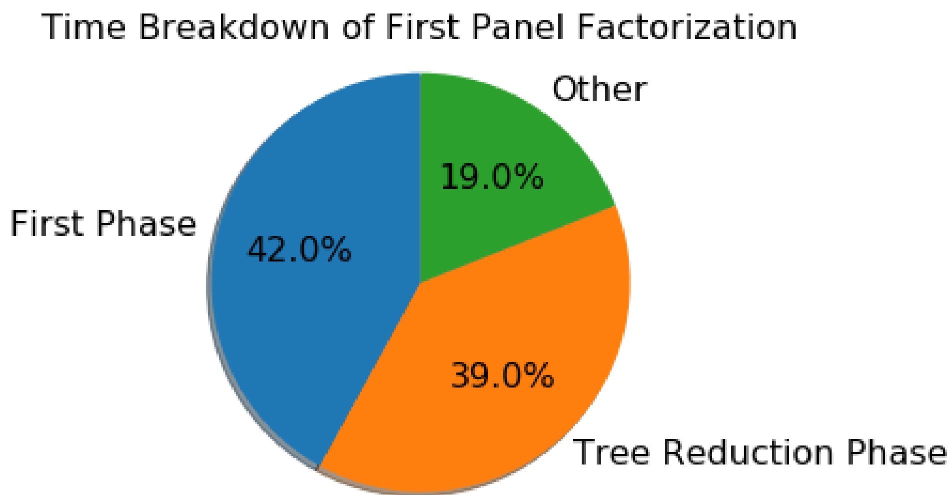

Figure 5: Overhead of CALU panel factorization of 80k matrix size and 320 tile size on 16 nodes of Shaheen-II.

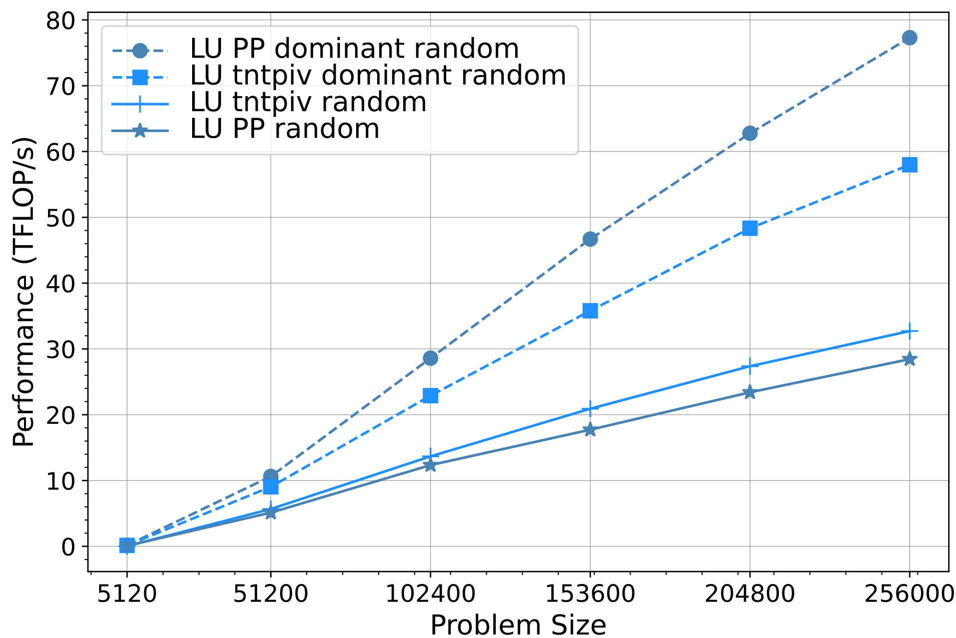

Figure 6: Performance of LU tntpiv and LU RBT on Summit.

This pie chart shows that there is a room for improvements. As future work, we propose to do local tree reduction within each node using OpenMP thread, followed be the global tree reduction, as shown in 7.

References

[1] Laura Grigori, James W Demmel, and Hua Xiang. Communication Avoiding Gaussian Elimination. In *SC'08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, pages 1–12. IEEE, 2008.

[2] Laura Grigori, James W Demmel, and Hua Xiang. CALU: a Communication Optimal LU Factorization Algorithm. *SIAM Journal on Matrix Analysis and Applications*, 32(4):1317–1350, 2011.

[3] Jakub Kurzak, Mark Gates, Ali Charara, Asim YarKhan, and Jack Dongarra. Least Squares Solvers for Distributed-Memory Machines with GPU Accelerators. In *Proceedings of the ACM International Conference on Supercomputing*, pages 117–126, 2019.

[4] Neil Lindquist, Piotr Luszczek, and Jack Dongarra. Replacing pivoting in distributed gaussian elimination with randomized techniques. In *2020 IEEE/ACM 11th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (ScalA)*, pages 35–43. IEEE, 2020.

[5] Jakub Kurzak, Mark Gates, Ali Charara, Asim YarKhan, Ichitaro Yamazaki, and Jack Dongarra. Linear Systems Solvers for Distributed-Memory Machines with GPU Accelerators. In *European Conference on Parallel Processing*, pages 495–506. Springer, 2019.

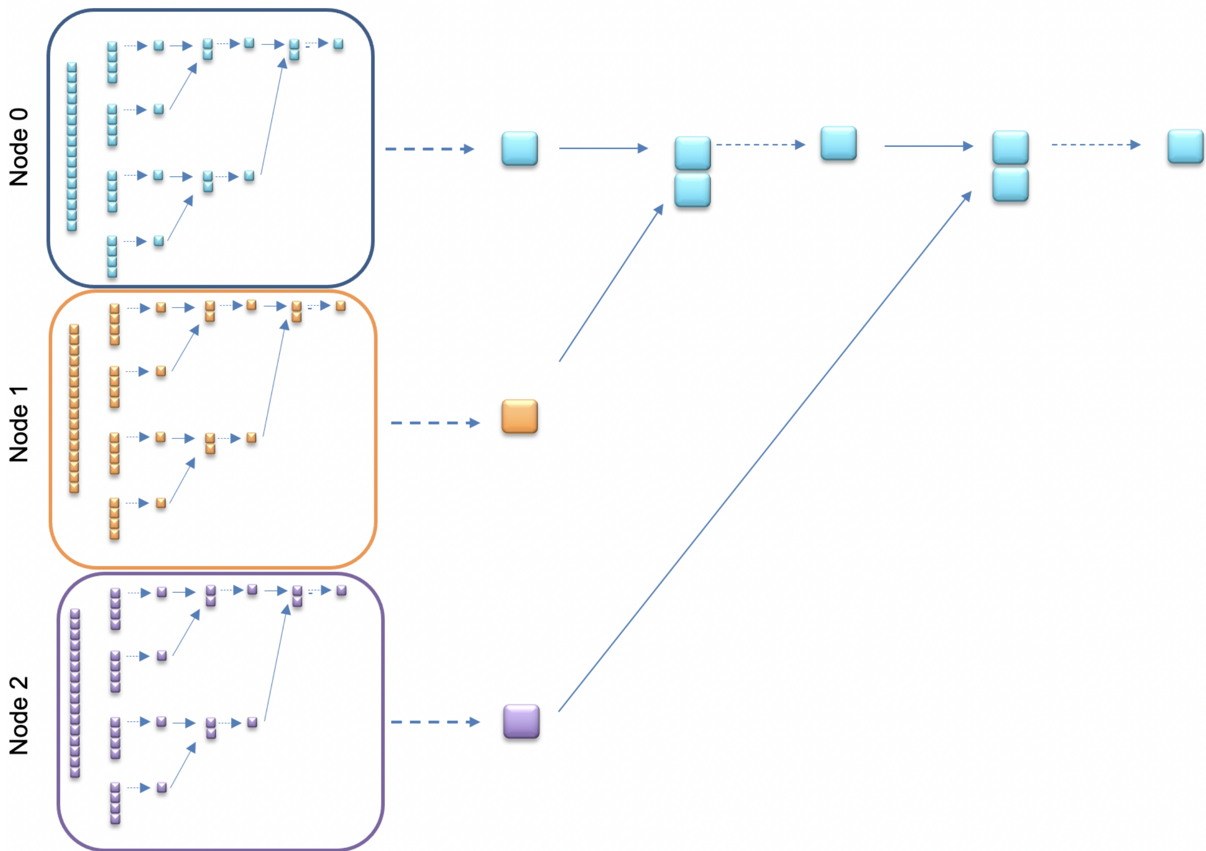

Figure 7: CALU based on local and global tree reduction.