Unified Communication Optimization Strategies for Sparse Triangular Solver on CPU and GPU Clusters

Yang Liu
liuyangzhuan@lbl.gov
Lawrence Berkeley National Laboratory
Berkeley, CA, USA

Nan Ding
nanding@lbl.gov
Lawrence Berkeley National Laboratory
Berkeley, CA, USA

Piyush Sao
saopk@ornl.gov
Oak Ridge National Laboratory
Oak Ridge, TN, USA

Samuel Williams
swwilliams@lbl.gov
Lawrence Berkeley National Laboratory
Berkeley, CA, USA

Xiaoye Sherry Li
xsli@lbl.gov
Lawrence Berkeley National Laboratory
Berkeley, CA, USA

ABSTRACT

This paper presents a unified communication optimization framework for sparse triangular solve (SpTRSV) algorithms on CPU and GPU clusters. The framework builds upon a 3D communication-avoiding (CA) layout of $P_x \times P_y \times P_z$ processes that divides a sparse matrix into P_z submatrices, each handled by a $P_x \times P_y$ 2D grid with block-cyclic distribution. We propose three communication optimization strategies: First, a new 3D SpTRSV algorithm is developed, which trades the inter-grid communication and synchronization with replicated computation. This design requires only one inter-grid synchronization, and the inter-grid communication is efficiently implemented with sparse allreduce operations. Second, broadcast and reduction communication trees are used to reduce message latency of the intra-grid 2D communication on CPU clusters. Finally, we leverage GPU-initiated one-sided communication to implement the communication trees on GPU clusters. With these nested inter- and intra-grid communication optimization strategies, the proposed 3D SpTRSV algorithm can attain up to 3.45x speedups compared to the baseline 3D SpTRSV algorithm using up to 2048 Cori Haswell CPU cores. In addition, the proposed GPU 3D SpTRSV algorithm can achieve up to 6.5x speedups compared to the proposed CPU 3D SpTRSV algorithm with P_z up to 64. Finally it is remarkable that the proposed GPU 3D SpTRSV can scale to 256 GPUs using the Perlmutter system while the existing 2D SpTRSV algorithm can only scale up to 4 GPUs.

CCS CONCEPTS

• **Mathematics of computing → Mathematical software**; • **Computing methodologies → Parallel computing methodologies**; **Distributed computing methodologies**.

SC '23, November 12–17, 2023, Denver, CO, USA

ACM ISBN 979-8-4007-0109-2/23/11.
https://doi.org/10.1145/3581784.3607092

KEYWORDS

communication-avoiding algorithm, communication optimization, SpTRSV, triangular solve, supernodal method, sparse matrix, NVSHMEM

ACM Reference Format:
Yang Liu, Nan Ding, Piyush Sao, Samuel Williams, and Xiaoye Sherry Li. 2023. Unified Communication Optimization Strategies for Sparse Triangular Solver on CPU and GPU Clusters. In *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC '23), November 12–17, 2023, Denver, CO, USA*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3581784.3607092

1 INTRODUCTION

Sparse triangular solves (SpTRSV) are important computational kernels in many direct sparse linear solvers and preconditioners for a wide range of scientific and engineering applications. Taking LU factorization of a sparse matrix $A = LU$ as an example, the solution vector x from $Ax = b$ given a right-hand side (RHS) vector b can be computed following one lower-triangular and upper-triangular SpTRSV operations. Although SpTRSV typically has many fewer arithmetic operations compared to LU factorization, it can become a computational bottleneck for linear systems with many RHSs or preconditioned iterative solvers requiring repeated application of SpTRSV. The low arithmetic intensity and sequential nature of SpTRSV post significant challenges for their efficient implementation on modern shared-memory and distributed-memory computing architectures. Shared-memory implementations [2, 14, 28, 30, 31, 42, 45, 46] such as multi-core CPU and GPU implementations rely on level-set, color-set or blocking methods to exploit available parallelism from the directed acyclic graph (DAG) representation of the sparse LU factors.

For realistic and large-scale multi-physics and multi-scale simulations, shared-memory SpTRSV implementation quickly becomes incapable of handling large linear systems and one needs to turn to distributed-memory SpTRSV and LU factorization algorithms [3, 7–10, 16, 18–25, 25–27, 35, 36]. However, distributed-memory parallel SpTRSV algorithms are even more challenging as communication quickly becomes dominant as the number of processors increases. Existing works include supernodal representation with a 2D process layout [12, 13, 22, 29], non-blocked representation with a 1D

process layout [41], multifrontal representation with sparse RHSs [34], and selective inversion-based algorithms for dense systems [32, 43]. Among them, communication optimization techniques such as customized communication trees [29], one-sided MPI communication [13] and GPU-initiated communication [12] have been exploited and performance prediction studies such as critical path analysis [12, 13], roofline modeling [44] and machine learning-based performance tuning [1, 15] have been considered. Despite these advances in distributed-memory SpTRSV algorithms, their strong scalability remains very limited for many classes of triangular matrices. More specifically, parallel 2D CPU SpTRSV can exhibit flattened strong scaling for large numbers of MPIs [13, 29], and parallel 2D GPU SpTRSV can only scale up to the number of GPUs in one node (typically less than 10) [12].

In addition to the above-mentioned communication optimization strategies, communication-avoiding (CA) algorithms have drawn many interests in numerical linear algebra research, including sparse LU factorization [37, 38], sparse GEMM [6], QR factorization [5], and Krylov methods [33], etc. Many of these algorithms rely on a 3D process layout of $P_x \times P_z \times P_z$ processes arranged as layers of 2D process grids that replicate the memory and computation loads across the grids. These algorithms oftentimes yield asymptotic reduction in communication at the cost of manageable memory overheads. Recently, a 3D CA SpTRSV algorithm [39] has also been developed by following the layout of the 3D CA sparse LU factorization algorithm [37]. The 3D SpTRSV algorithm divides the triangular matrix into multiple levels with a binary elimination tree, where each node corresponds to several submatrices that reside on one 2D grid. The algorithm proceeds with a bottom-up tree traversal where SpTRSV of each node requires only communication inside one 2D grid (intra-grid communication), and the solution subvectors need to be reduced across the 2D grids (inter-grid communication) before moving to the next level. This algorithm can effectively reduce the communication volume by a factor of $\sqrt{P_z}$ for matrices arising from many 2D and 3D PDEs [39].

Despite its preliminary success, neither the intra-grid communication pattern nor inter-grid communication pattern of the 3D SpTRSV (henceforth refereed to as the baseline algorithm) [39] is optimal: the algorithm requires $O(\log P_z)$ synchronizations between the 2D grids which significantly increase runtime when the intra-grid computation and computation are load-imbalanced. Moreover, these synchronizations make it seemingly impossible to efficiently integrate other communication reduction techniques such as customized communication trees [29] and GPU-initiated one-sided communication [12]. As a result, the baseline algorithm can yield even worse performance than the 2D SpTRSV algorithms [29]. In this paper, we propose a unified 3D SpTRSV algorithm that can efficiently leverage multiple communication optimization strategies on both CPU and GPU clusters. The contributions of this paper include: (1) Develop a new 3D SpTRSV algorithm that requires only one inter-grid synchronization in between the L and U solves. (2) Develop an efficient sparse allreduce communication scheme that dramatically reduces the inter-grid communication cost. (3) Integrate the communication tree-based optimization [29] to reduce message latency for the intra-grid communication on CPU clusters, which yields significantly faster SpTRSV time compared to the baseline algorithm. (4) Integrate NVSHMEM-based one-sided communication for intra-grid communication and GPU acceleration for the computation workloads using GPU clusters, which significantly improves the strong scalability of multi-GPU SpTRSV algorithms.

The rest of this paper is organized as follows: Section 2 briefly overviews the SpTRSV with a supernodal representation and the baseline 3D SpTRSV algorithm. Section 3 describes the proposed synchronization-reduced 3D SpTRSV algorithm (Subsection 3.1) and sparse inter-grid communication (Subsection 3.2), as well as the integration of communication tree-based intra-grid communication optimization for both CPUs (Subsection 3.3) and GPUs (Subsection 3.4). Section 4 presents several numerical experiments to demonstrate the improved scalability of the proposed 3D SpTRSV algorithm on Cori Haswell CPU nodes, Perlmutter GPU nodes, and Crusher GPU nodes.

2 BACKGROUND

2.1 Overview of SpTRSV With Supernodal Representation

Consider a LU factorized sparse $n \times n$ matrix $A = LU$, the solution vector (or matrix) x subject to $Ax = b$ with a dense right-hand side (RHS) vector (or matrix) b can be computed by a lower-triangular SpTRSV (L-solve) $Ly = b$ followed by a upper-triangular SpTRSV (U-solve) $Ux = y$. Furthermore, we assume a supernodal representation of L and U with N supernodes from supernodal sparse direct solvers [7]. A supernode consists of a set of contiguous columns and rows whose nonzero patterns are similar across the rows and columns, respectively. Let $b(K)$, $y(K)$ and $x(K)$ denote the subvectors corresponding to supernode K, and $L(I,K)$ and $U(I,K)$ denote the nonzero submatrix corresponding to supdernodes I and K. Each $L(I,K)$, $I < K$ consists of a set of full rows, each $U(I,K)$, $I > K$ consists of a set of columns, and $L(K,K)$ and $U(K,K)$ are dense. $U(I,K)$ typically follows the "skyline" format assuming each nonzero column has different length, but in this work we assume all nonzero columns in each $U(I,K)$ have the same length. Compared with column/row-based representation, these supernodal column/row-based representations yield higher floating point (FP) operation performance.

The subvectors $y(K)$ and $x(K)$ can be computed as

$$y(K) = L(K,K)^{-1}\Big(b(K) - \sum_{I=1}^{K-1} L(K,I) \cdot y(I)\Big) \tag{1}$$

$$x(K) = U(K,K)^{-1}\Big(y(K) - \sum_{I=N}^{K+1} U(K,I) \cdot x(I)\Big) \tag{2}$$

Throughout this paper, we assume that diagonal blocks $L(K,K)^{-1}$ and $U(K,K)^{-1}$ have been precomputed and the significant FP operations are the GEMV (single RHS) or GEMM (multiple RHSs) associated with diagonal and nonzero off-diagonal computation, which yields intrinsic low arithmetic intensity of SpTRSV. Furthermore, note that computation of $y(K)$ depends on $y(I)$ if $L(K,I)$ is nonzero, and computation of $x(K)$ depends on $x(I)$ if $U(K,I)$ is nonzero. This computation dependency can be modeled by directed acyclic graphs (DAGs) consisting of vertices K. The DAGs can post significant challenges for their efficient parallelization on

distributed-memory machines. One can expect that the communication will dominate the overall SpTRSV time as the number of (distributed-memory) processes increases due to the low arithmetic intensity and sequential nature of DAG. Communication reduction techniques become crucial for scaling up the SpTRSV to large numbers of CPU and GPU processors [12, 13, 29, 39]. Next, we briefly review the CA 3D SpTRSV algorithm [39] as the baseline algorithm for the proposed unified communication optimization strategies.

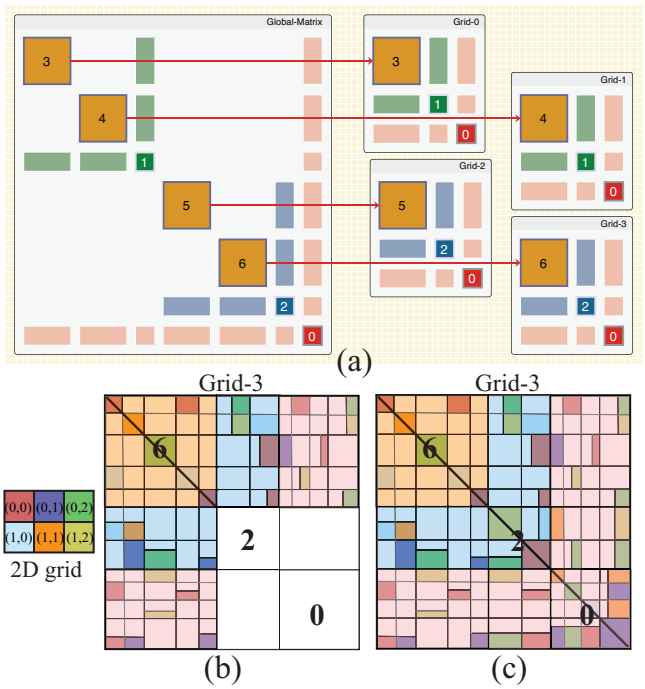

Figure 1: Parallel data layout of 3D SpTRSV with $P_x = 2$, $P_y = 3$ and $P_z = 4$, assuming a LU-factorized symmetric sparse matrix. (a) Mapping of the matrix onto the 4 2D grids. (b) 2D process layout for computation tasks in Grid-3 of the baseline algorithm [39]. (c) 2D process layout for computation tasks in Grid-3 of the proposed algorithm.

2.2 Baseline CA 3D SpTRSV Algorithm

The baseline algorithm [39] trades off some communication with memory replication similar to several other CA algorithms[5, 6, 37]. As opposed to traditional distributed-memory SpTRSV algorithms [22, 24] that use a 2D block-cyclic process layout with $P = P_x \times P_y$ processes, the CA 3D SpTRSV algorithm [39] uses a 3D process layout of $P = P_x \times P_y \times P_z$ processes consisting of P_z 2D process grids. The sparse matrices L and U are distributed onto the P_z grids as follows.

First we assume that during the numerical factorization of A, an ordering of the matrix has been applied to reduce the number of fill-ins in L and U, such as minimum degree ordering or nested-dissection (ND) ordering. The ordering generates a multilevel dependency tree, known as the elimination tree. Each node of the tree represents a group of supernodes and the SpTRSVs of the nodes at the same level are independent of each other. Therefore the elimination tree can be used to exploit parallelism in SpTRSV. In this paper, we uses ND ordering computed by METIS [17], assuming that the number of 2D grids P_z is power-of-two and the top $\log(P_z)$ levels of the elimination tree are a binary subtree. Following the elimination tree, the 3D SpTRSV algorithm finds the leaf level where the number of nodes is equal to P_z. Each leaf node k and all its ancestors a are assigned to one 2D grid. In other words, the ancestor nodes are replicated across multiple 2D grids. Fig. 1(a) shows the distribution of the LU factors with $P_z = 4$ 2D grids according to the first three levels of the elimination tree. The root node 0 is replicated across all 2D grids, node 1 is replicated across Grid-0 and Grid-1, node 2 is replicated across Grid-2 and Grid-3, and each of the nodes 3,4,5,6 belongs to one 2D grid. Here the submatrix of a node consists of the diagonal block corresponding to all supernodes of the node, as well as the off-diagonal blocks below and to the right of the diagonal block. This submatrix of each grid is distributed with a 2D block-cyclic layout using $P_x \times P_y$ processes. Fig. 1(b)(c) shows the 2D layout with $P_x = 2$ and $P_y = 3$ for the submatrix in Grid-3 of Fig. 1(a). Here we have assumed that the matrix A has symmetric nonzero patterns for simplicity.

Following this 3D layout, the CA SpTRSV algorithm [39] proceeds as follow.[1] First, each leaf node k performs 2D SpTRSV independently using the diagonal block. Once the solution subvectors corresponding to the node has been obtained, they are used to perform GEMV/GEMM with the off-diagonal blocks to generate partial summation results corresponding to RHS of Eq (1). This step only involves intra-grid communication. Next, an inter-grid communication between a pair of grids is performed to reduce the partial summation results to the grid whose grid number z is the smallest among all grids replicating the parent p of k. All other grids among those grids remain idle ever since. Next, the active grid z performs 2D SpTRSV for the node p, and GEMV/GEMM operations with its off-diagonal blocks, which then requires inter-grid communication before moving to the next level. Using the example of Fig. 1(a), Grid-1 and Grid-3 are active at level 2 (i.e., the leaf level), Grid-2 are active at levels 2 and 1, and only Grid-0 is active at all levels. Fig. 1(b) shows the computation and communication workloads at Grid-3.

It has been estimated and validated that when compared with 2D SpTRSV using the same number of processes P, the 3D SpTRSV algorithm can effectively reduce the communication volume from $O(\frac{n}{\sqrt{P}})$ to $O(\frac{n}{\sqrt{PP_z}})$ for many 2D and 3D PDEs [39]. That said, this algorithm has two main drawbacks due to the alternating intra-grid and inter-grid communication required at each level.

- Synchronization across the 2D grids is required at each level of the elimination tree for a total of $\log P_z$ times, which can significantly increase the overall runtime due to possible load imbalance among diagonal and off-diagonal blocks of different grids.
- The separation of diagonal block solve and off-diagonal block GEMV/GEMM for each node of the tree makes it cumbersome for further communication optimization in each grid

[1] Here we only describe the L-solve as the U-solve follows a similar but reversed computation order.

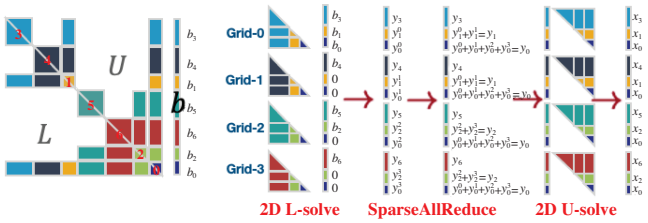

Figure 2: Workflow of the proposed 3D SpTRSV algorithm. 2D L-solve and U-solve only involve intra-grid communication, and SparseAllReduce only involves inter-grid communication.

such as latency reduction with binary communication trees [29] and one-sided CPU/GPU communication [11, 12].

These difficulties have to be addressed to further improve the parallel scalability of SpTRSV with large numbers of CPU and GPU nodes.

3 UNIFIED COMMUNICATION OPTIMIZATION STRATEGIES FOR 3D SPTRSV ALGORITHM

In this paper, we propose a unified communication optimization framework based on a novel 3D SpTRSV algorithm. The proposed 3D algorithm requires only one inter-grid synchronization at the cost of replicated computation as opposed to the baseline 3D algorithm requiring $O(\log P_z)$ inter-grid synchronizations. As a result, the new 3D SpTRSV algorithm can integrate multiple communication optimization strategies such as sparse inter-grid communication, latency reduction [29] and one-sided GPU communication [12].

3.1 New 3D SpTRSV With Synchronization Reduction

The proposed 3D SpTRSV algorithm can be described using the example in Fig. 1(a). Unlike the baseline algorithm where each grid is active for a few bottom levels, the proposed algorithm ensures that all grids are active at all levels for allowing them to perform replicated computation (see Fig. 1(c) for Grid-3). Specifically, the baseline algorithm performs operations for each shaded block (see Fig. 1(b) for Grid-3) separately and proceeds level by level with inter-grid communication in between. In contrast, the proposed algorithm treats the blocks of one grid (including the replicated blocks) as *one* 2D distributed matrix, denoted as L^z and U^z for Grid-z. These 2D solves require only one inter-grid communication/synchronization in between at the cost of replicated computation. To ensure correct solutions, judiciously selected subvectors of the RHS b^z for each 2D L-solve need to be set to 0, and a sparse allreduce operation is needed for the solution vectors y^z before the 2D U-solve.

This procedure is illustrated in Fig. 2 using the same matrix as Fig. 1(a) except that the blocks associated with one node of the elimination tree is marked with the same color. The subvectors of the RHS corresponding to one node in Grid-z are set to 0 if z is not the smallest grid number sharing that node (see the RHS for each grid in Fig. 2). The 2D L-solve does not involve any inter-grid communication, which leads to partial solution vectors y_k^z for non-leaf nodes k. Therefore an allreduce operation is performed across the grids to obtain the complete solution vectors, which then become the RHSs for the 2D U-solve. Again, the 2D U-solve does not involve any inter-grid communication. This algorithm is summarized as Algorithm 1. Line 4 forms the proper RHS b^z for L^z, Lines 11 and 21 perform the 2D L-solve and U-solve whose communication costs can be further optimized based on CPU (see Section 3.3) or GPU (see Section 3.4) implementations, and Line 20 performs the inter-grid communication (see Section 3.2).

Remark. Although the proposed 3D SpTRSV algorithm introduces extra FP operations and communication due to replicated computation, the FP operations are performed in parallel among all the 2D grids, whereas many 2D grids stay idle at higher tree levels in the baseline 3D SpTRSV algorithm. Therefore, the FP operations do not require extra overall runtime for the proposed algorithm. That said, the extra communication introduced by the proposed 3D SpTRSV algorithm can have an impact on the total communication time in each 2D grid. However, they can be highly optimized using the communication schemes in Subsection 3.2.

3.2 Inter-Grid Communication Optimization

The inter-grid communication and synchronization are needed in between the 2D L-solve and U-solve of Algorithm 1 to reduce the partial solution vectors y_k^z for all grids z sharing the non-leaf node k. Straightforward implementations using MPI_allreduce for each node k can become costly both in terms of latency and synchronization. Instead, we propose a sparse allreduce algorithm which requires only $O(\log P_z)$ pair-wise sends and receives. The algorithm is illustrated with Fig. 3 using the example of Fig. 2. The algorithm is composed of a sparse reduce phase and a sparse broadcast step. The reduce phase (Fig. 3(a)) proceeds from leaf to root: Step ① performs pairwise send/receive between two grids for $[y_0, y_1]$ and $[y_0, y_2]$, Step ② performs pairwise send/receive between Grid-0 and Grid-2 for y_0. Similarly, the broadcast phase (Fig. 3(b)) proceeds from root to leaf: Step ③ performs pairwise send/receive between Grid-0 and Grid-2 for y_0, Step ④ performs pairwise send/receive between two grids for $[y_0, y_1]$ and $[y_0, y_2]$. Note that each y_k consists of many supernodes which are distributed according to the 2D block-cyclic layout of L^z. That said, each process only communicates for $O(\log P_z)$ times when the communication buffer packs data from all the required supernodes. This sparse allreduce algorithm is summarized as Algorithm 2.

3.3 Intra-Grid Communication Optimization on CPU

For each 2D L-solve and U-solve, the intra-grid communication can be further improved by reducing message latency using customized communication trees [29]. In Subsection 3.3 and 3.4, we only describe the communication optimization strategies for the L-solve as the U-solve algorithm is very similar. Consider the L^z matrix in Fig. 1(c) of Grid-3 with a 2×3 process layout. For each supernode column I, the process computing $y(I)$ needs to send it to the process in charge of $L(K, I)$ to perform the partial sum (see Eq (1)), for all $K > I$ residing on a different process, which

Algorithm 1 Proposed 3D SpTRSV on a 3D grid of size $P_x \times P_y \times P_z$.

1: **procedure** SOLVE_3D(L,U,b,x)
2: Suppose I am on grid-z with leaf node k, form L^z and U^z with a $P_x \times P_y$ layout (see Fig. 1(c))
3: $b^z = b_k$
4: **for** each ancester a of k **do** ▷ form RHS b^z for L^z
5: **if** z is the smallest grid ID that replicates a **then**
6: $b^z = [b^z; b_a]$
7: **else**
8: $b^z = [b^z; 0]$
9: **end if**
10: **end for**
11: **if** CPU **then**
12: SOLVE_L_CPU(L^z,b^z,y^z)▷ 2D CPU L-solve with binary communication tree (Algorithm 3)
13: **else if** GPU **then**
14: **if** $P_x \times P_y$=1 **then**
15: SOLVE_L_SINGLE_GPU(L^z,b^z,y^z)▷ single-GPU 2D L-solve (see Algorithm 4)
16: **else**
17: SOLVE_L_MULTI_GPU(L^z,b^z,y^z) ▷ NVSHMEM-based multi-GPU 2D L-solve (see Algorithm 5)
18: **end if**
19: **end if**
20: SPARSE_ALLREDUCE(y^z) ▷ Inter-grid communication to form the correct solution vector (see Algorithm 2)
21: **if** CPU **then**
22: SOLVE_U_CPU(U^z,y^z,x^z) ▷ 2D CPU U-solve with binary communication tree
23: **else if** GPU **then**
24: **if** $P_x \times P_y$=1 **then**
25: SOLVE_U_SINGLE_GPU(U^z,y^z,x^z) ▷ single-GPU 2D U-solve
26: **else**
27: SOLVE_U_MULTI_GPU(U^z,y^z,x^z) ▷ NVSHMEM-based multi-GPU 2D U-solve
28: **end if**
29: **end if**
30: **end procedure**

Algorithm 2 SparseAllReduce of the partial solution vectors after 3D L-solve across P_z grids

1: **procedure** SPARSE_ALLREDUCE(y)
2: Suppose I am on grid-z with nodes k_l, $l = 0, \ldots, l_{max}$, complete solution vector y_{k_0}, and partial solution vectors $y_{k_l} = y^z_{k_l}$, $l > 0$.
3: **for** $l = 0 : l_{max} - 1$ **do** ▷ sparse reduce from leaf to root
4: $y_{buf} := \{y_a | a \text{ is an ancester of } k_l\}$ ▷ each y_a follows the same 2D layout as L
5: **if** $z\%2^{l+1} = 0$ **then** ▷ pairwise inter-grid communication
6: Send y_{buf}
7: **else if** $z\%2^{l+1} = 2^l$ **then**
8: Receive and reduce y_{buf}
9: **end if**
10: **end for**
11: **for** $l = l_{max} - 1 : 0$ **do** ▷ sparse broadcast from root to leaf
12: $y_{buf} := \{y_a | a \text{ is an ancester of } k_l\}$ ▷ each y_a follows the same 2D layout as L
13: **if** $z\%2^{l+1} = 2^l$ **then** ▷ pairwise inter-grid communication
14: Send y_{buf}
15: **else if** $z\%2^{l+1} = 0$ **then**
16: Receive y_{buf}
17: **end if**
18: **end for**
19: **end procedure**

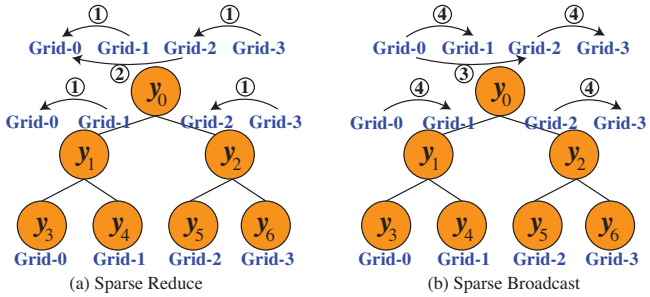

Figure 3: The SparseAllReduce phase of Fig. 2 with 4 2D grids, consisting of (a) a sparse reduce step and (b) a sparse broadcast step.

is a broadcast operation for each column I. Similarly, for each supernode row K, the process computing $lsum(K) = \sum L(K,I) \cdot y(I)$ for all $L(K,I)$ it owns needs to send $lsum$ to the process storing $L(K,K)^{-1}$ (see Eq (1)). This induces a reduction operation for each row K. These broadcast and reduction operations can be improved using customized binary communication trees (one per column and row) to significantly reduce the total message counts [29]. For ease of illustration, Algorithm 3 summarizes the communication tree-enhanced 2D L-solve with a $P_x \times 1$ process layout, where only broadcast operation is needed. Our numerical results in Section 4.1 will demonstrate the functionality and efficiency of using $P_x \times P_y$ process layouts with both broadcast and reduction operations. In Algorithm 3, we use an indicator array $fmod$ to keep track of the number of $y(I)$ each row K needs to receive. The algorithm is MPI message driven with a blocking MPI_Recv at Line 7 for any incoming message until the total number of expected messages have been received by each process.

Remark. The proposed 3D SpTRSV algorithm with synchronization reduction (see Fig. 1(c)) makes it very convenient to integrate the binary communication tree-based optimization as each grid treats L^z as a single-level 2D block-cyclic distributed matrix. For example, each row and column in Fig. 1(b) require one broadcast and communication tree. In contrast, the baseline 3D SpTRSV algorithm needs to compute the broadcast trees and reduction trees for each row and column, for all diagonal and off-diagonal blocks. For example, each row and column in Fig. 1(b) require three broadcast and

Algorithm 3 CPU SpTRSV for L on a 2D grid of size $P_x \times 1$

1: **procedure** SOLVE_L_CPU(L,b,y)
2: **for** each leaf node K my MPI rank owns **do**
3: $y(K) = L(K,K)^{-1} \cdot y(K)$
4: MPI_Isend $y(K)$ to my children in the binary broadcast tree of column K
5: **end for**
6: **while** I expect more messages **do** ▷ message count is pre-computed in $fmod$
7: $y(K)$=MPI_Recv(MPI_ANY_SOURCE) ▷ blocking receive any message
8: MPI_Isend $y(K)$ to my children in the binary broadcast tree of column K
9: **for** each $L(I,K)! = 0, I > K$ **do**
10: $lsum(I) = lsum(I) + L(I,K) \cdot y(K)$
11: $fmod(I) = fmod(I) - 1$
12: **end for**
13: **end while**
14: **end procedure**

Algorithm 4 Single-GPU SpTRSV for L on a 2D grid of size 1×1.

1: **procedure** SOLVE_L_SINGLE_GPU(L,b,y)
2: $K = bid$ ▷ one thread block handles one block column
3: while ($fmod(K)! = 0$); ▷ spin wait, called by thread 0
4: $y(K) += lsum(K)$
5: $y(K) = L(K,K)^{-1} \cdot y(K)$ ▷ parallelize GEMV/GEMM over threads
6: **for** each $L(I,K)! = 0, I > K$ **do**
7: $lsum(I) = lsum(I) + L(I,K) \cdot y(K)$ ▷ parallelize I and GEMV/GEMM over threads
8: $fmod(I) = fmod(I) - 1$
9: **end for**
10: **end procedure**

reduction trees (i.e., for the orange, blue and pink colored blocks), which unfavorably increases the message counts and greatly complicates the implementation.

3.4 Intra-Grid Communication Optimization on GPU

We also implement the proposed 3D SpTRSV algorithm on GPU clusters using synchronization reduction with replicated computation and latency reduction with communication trees. In all our implementations, we assume that one GPU is assigned to one MPI for simplicity. Just like Subsection 3.3, we only describe the L-solve implementation for simplicity. In our implementation, it is assumed that L^z, b^z and y^z reside on GPU and the computation (i.e., GEMV/GEMM involving $L(K,I)$ and $L(K,K)^{-1}$) is performed on GPU. While the SparseAllReduce operation in Line 20 of Algorithm 1 is implemented with MPIs, the communication operations in 2D L-solve and U-solve cannot use MPI efficiently. This is due to the fact that MPI communications are CPU-initiated (even for GPU-aware MPI) and CPU control flow will significantly slow down performance due to the sequential nature of SpTRSV. In other words, both the CPU and GPU need to keep track of the DAG of SpTRSV with frequent CPU-GPU data communication. Instead of MPI, we rely on the GPU-initiated one-sided communication libraries such as NVSHMEM to implement the binary communication trees. The GPU-initiated communication feature allows the code to execute both computation and communication in GPU kernels without CPU interference, hence traverse the DAG much more efficiently.

Before describing the NVSHMEM-based 2D L-solve, we first describe the single-GPU accelerated L-solve algorithm. The reason is two-fold: 1) When $P_x = P_y = 1$ and $P_z > 1$, the GPU 2D L-solve of L^z requires no intra-grid communication, hence the L-solve kernel can be greatly simplified. 2) Currently, the GPU-initiated communication-based 2D solves only work on NVIDIA GPUs with NVSHMEM. The AMD GPU's counterpart ROC-SHMEM currently does not support MPI subcommunicators, which are used throughout the 3D SpTRSV algorithm. Adding support for MPI subbcommunicators in ROC-SHMEM will enable significantly improved scalability of SpTRSV for large nubmers of GPU nodes. In this paper our numerical results for AMD GPUs will only consider $P_x = P_y = 1$. The single-GPU 2D L-solve algorithm is described in Algorithm 4. The kernel assigns one thread block (with ID bid) per supernode column K, whose thread 0 spin waits for the dependency indicator $fmod(K)$ until $y(K)$ can be computed. The GEMV/GEMM operations for $L(I,K)y(K)$, $I > K$ are parallelized over the threads in each thread block. When the number of RHSs nrhs is 1, the GEMV is parallelized over the row dimension; when nrhs>1, each GEMM $L(I,K)y(K)$ is implemented as dense blocked operations involving only the nonzero rows of $L(I,K)$ using the shared memory, similar to MAGMA's GEMM implementation [40].

Based on the single-GPU L-solve implementation, we briefly summarize the NVSHMEM-based 2D L-solve algorithm for NVIDIA GPUs in Algorithm 5. Just like Subsection 3.3, we only describe the case of $P_x \times 1$ 2D grids for simplicity and the complete 2D L-solve algorithm can be found in work [12]. NVSHMEM has a major limitation which is that the number of thread blocks that can be concurrently scheduled on a GPU equals the number of SMs. Such a design is to avoid potential deadlocks when using point-to-point synchronization in the CUDA kernel. However, that limitation would significantly restrict SpTRSV concurrency. To address this limitation, two kernels (SOLVE and WAIT) that run on two CUDA streams are utilized. The SOLVE kernel assigns one thread block (with ID bid) per supernode column K that my GPU/CPU owns. If the GPU is the diagonal process for handling K and dependency $fmod(K)$ has been met, the GEMV/GEMM operation involving $L(K,K)^{-1}$ is parallelized over the threads of the thread block (Line 7). Otherwise, if the GPU is an off-diagonal process (Line 11), thread 0 will spin wait for the $flag\_y(K)$ indicating whether $y(K)$ (the buffer $ready\_y(K)$) has been received (Line 12). If received, $y(K)$ will be forwarded along the binary broadcast tree of K using NVSHMEM_SEND (Line 13). Then the GEMV/GEMM involving $L(I,K)$, $I > K$ are parallelized over the threads and $fmod(I)$ is updated, just like the single-GPU algorithm of Algorithm 4. Recall that the number of thread blocks of the SOLVE kernel equals the number of supernode columns one GPU/CPU handles. The WAIT kernel has only one thread block, where all threads are waiting for incoming messages using nvshmem_init_wait_until_any (Line 22).

Algorithm 5 Multi-GPU SpTRSV for L on a 2D grid of size $P_x \times 1$

procedure SOLVE_L_MULTI_GPU(L,b,y)
  NVSHMEM launch WAIT($flag\_y$, stream[0]) ▷ $flag\_y(K) = 1$ indicates that $y(K)$ has been received.
  CUDA launch SOLVE(L,b,y,stream[1])
end procedure

procedure SOLVE(L,b,y,stream[1])
  $K = bid$ ▷ one thread block handles one block column
  if I am the diagonal process in charge of K **then**
    while(fmod(K)!=0); ▷ spin wait, called by thread 0
    $y(K) += lsum(K)$
    $y(K) = L(K,K)^{-1} \cdot y(K)$ ▷ parallelize TRSV/TRSM over threads
  else
    while($flag\_y(K)$!=1); ▷ spin wait, called by thread 0
    NVSHMEM SEND $ready\_y(K)$ to my children's $ready\_y$ buffer ▷ called by all threads
    for each $L(I,K) != 0, I > K$ **do** ▷ parallelize I and GEMV/GEMM over threads
      $lsum(I) = lsum(I) + L(I,K) \cdot ready\_y(K)$
      $fmod(I) = fmod(I) - 1$
    end for
  end if
end procedure

procedure WAIT(stream[0]) ▷ probe messages, this kernel only requires one thread block
  while expecting more messages **do**
    idx=nvshmem_int_wait_until_any($flag\_y$) ▷ message arrived in block column idx
  end while
end procedure

Table 1: Test matrices. Density := {nonzeros in LU} / n^2

Matrix	Size n	Nonzeros in LU	Density	Description
nlpkkt80	1,062,400	1,928,132,340	0.17%	Optimization
Ga19As19H42	133,123	1,565,515,001	9.15%	Chemistry
s1_mat_0_253872	253,872	425,394,978	0.66%	Fusion
s2D9pt2048	4,194,304	810,605,750	0.005%	Poisson
ldoor	952,203	319,022,661	0.035%	Structural
dielFilterV3real	1,102,824	1,138,910,076	0.094%	Wave

Each thread has a unique waiting entry. Note that the binary broadcast trees and $fmod$ array have been precomputed on CPUs and transferred to GPUs, and the 2D L-solve algorithm requires no CPU interference during the execution.

4 NUMERICAL RESULTS

In this section, we present several numerical experiments to demonstrate the superior performance of the proposed 3D SpTRSV algorithm compared with the baseline 3D SpTRSV algorithm [39] and the existing 2D SpTRSV algorithms [12, 29] using three leadership supercomputing systems: Cori Haswell, Perlmutter and Crusher.

The CPU-only experiments are performed on the Cori Haswell system at NERSC. Cori Haswell is a Cray XC40 system and consists of 2388 dual-socket nodes with Intel Xeon E5-2698v3 processors running 16 cores per socket. The nodes are equipped with 128 GB of DDR4 memory. The nodes are connected through the Cray Aries interconnect. The GPU experiments (including the reference CPU experiments) are performed on the Crusher system at OLCF and the Perlmutter system at NERSC. Crusher is a testbed system for the Frontier exascale machine and each node consists of a 64-core AMD EPYC 7A53 CPU processor and 4 AMD MI250X GPUs (8 Graphics Compute Dies) each with 64 GB of HBM2 memory. The nodes are connected with HPE Slingshot interconnect with a 25 GB/s bandwidth. Perlmutter (the GPU partition) is a HPE Cray EX system and consists of 1536 GPU nodes each with a 64-core AMD EPYC 7763 CPU processor, 4 NVIDIA A100 GPUs, and 40 GB HBM memory per GPU. The nodes are connected with the HPE Slingshot 11 interconnect with a 25 GB/s bandwidth.

All the benchmark matrices are listed in Table 1, which arise from various applications such as optimization problems, structural computation, quantum chemistry calculation, fusion plasma simulation, finite-difference discretization of Poisson equations, and finite element discretization of Maxwell equations. All these matrices except for s1_mat_0_253872 and s2D9pt2048 are publicly available via the SuiteSparse Matrix Collection [4]. The LU factors are generated by running the 3D numerical factorization algorithm in SuperLU_DIST [25] with the METIS ordering for fill-in reduction [17]. The size and nonzeros of the LU factors are also listed in Table 1.

4.1 Performance of 3D SpTRSV on CPU clusters

First, we demonstrate the improved scalability and efficiency of the proposed 3D SpTRSV algorithm using the Cori Haswell CPU nodes. We test the runtime of the baseline and proposed 3D SpTRSV algorithms by varying the total MPI count $P = P_x \times P_y \times P_z$ from 128 to 2048 and changing P_z from 1 to 32. When fixing P and P_z, the 2D grid sizes are set such that $P_x \approx P_y$. Fig. 4 shows the runtime of the two algorithms for four matrices s2D9pt2048, nlpkkt80, ldoor and dielFilterV3real. It is worth noting that the proposed algorithm with $P_z = 1$ reduces to the 2D SpTRSV algorithm [29] with latency optimization, corresponding to the red solid curves. For both the baseline and the proposed algorithms, increasing P_z until 16 leads to improved runtime. However, the proposed algorithm constantly overperforms the baseline 3D SpTRSV [39] and 2D communication-optimized SpTRSV [29]. For the four matrices s2D9pt2048, nlpkkt80, ldoor and dielFilterV3real, the proposed algorithm respectively achieves up to 3.45x, 1.87x, 1.13x and 1.98x speedups compared to the baseline 3D algorithm. Moreover, it respectively achieves up to 2.2x, 1.1x, 2.1x and 1.43x speedups compared to the 2D communication-optimized SpTRSV. It's worth noting that without further communication optimization, the baseline 3D algorithm can be worse than the communication-optimized 2D algorithms. For example, see the solid red curves and dashed green curves for matrices ldoor and nlpkkt80.

To better understand the performance gains, we show the time breakdown (averaged over all MPI ranks) in inter-grid communication, intra-grid communication and FP operations in Fig. 5 and Fig. 6 for matrices s2D9pt2048 and nlpkkt80, respectively. We compare the results of the baseline 3D SpTRSV algorithm [39] and the proposed 3D SpTRSV algorithm. In Fig. 5, one can see that the optimization algorithm in Subsections 3.1 and 3.2 significantly reduce the intra-grid communication time particularly when P_z is large; the optimization algorithm in Subsection 3.3 significantly reduces the inter-grid communication time, particularly when $P_x \times P_y$ is large; the algorithm in Subsection 3 introduces extra FP operations, however they do not contribute to the overall runtime much as the replicated FP operations are performed in parallel in different grids. Unlike the 2D PDE discretized matrix s2D9pt2048 in Fig. 5, the 3D PDE discretized matrix nlpkkt80 (see Fig. 6) introduces asymptotically more replicated computation and intra-grid communication when P_z is large, as can be seen from the bottom middle figure. The increased intra-grid communication for large P_z leads to worse 3D SpTRSV performance.

For the s2D9pt2048 matrix, we also measured the load balance when varying P_z. In Fig. 7, we plotted the time in the L and U solve across all MPI ranks given $P = 128$ and $P = 1024$. The load imbalance is indicated by the error bars where both the baseline and proposed 3D SpTRSV exhibit reasonable load balance.

For the nlpkkt80 matrix, we measured the load balance when varying P_z. In Fig. 8, we plotted the time in the L and U solve across all MPI ranks given $P = 128$ and $P = 1024$. When P_z is large, the baseline code shows large imbalance, while the proposed code shows good balance. Although the proposed code shows increased CPU time averaged over the ranks due to duplicated computation, it still achieves decreased overall CPU time, which is the maximum over the ranks.

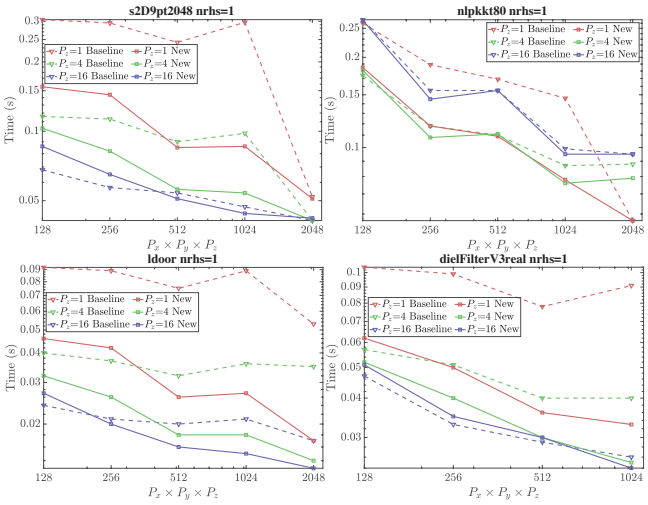

Figure 4: SpTRSV time using the Cori Haswell machine as the total MPI counts $P_x \times P_y \times P_z$ vary. For each P_z value (one curve), the 2D grid (P_x, P_y) is set as square as possible. "Baseline" denotes the baseline 3D SpTRSV [39] and "New" denotes the proposed 3D SpTRSV.

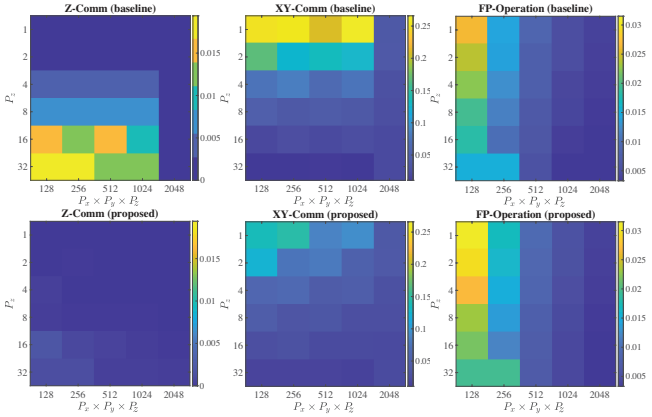

Figure 5: Time breakdown in seconds (averaged over all MPI ranks) of the s2D9pt2048 matrix using the baseline and proposed 3D SpTRSV algorithms on the Cori Haswell machine. Z-Comm denotes inter-grid communication time, XY-Comm denotes intra-grid communication time, and FP-Operation denotes the floating-point operation time.

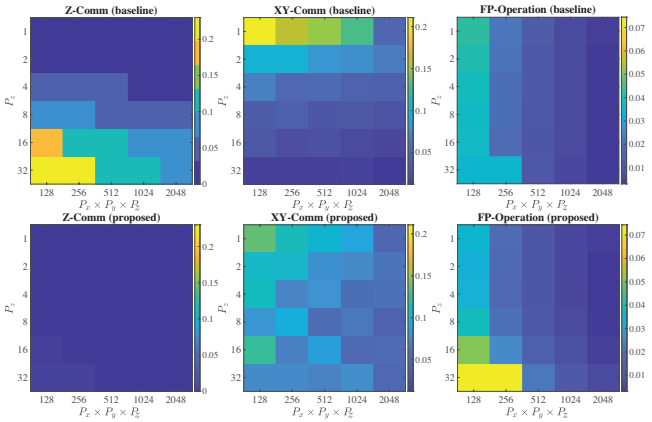

Figure 6: Time breakdown in seconds (averaged over all MPI ranks) of the nlpkkt80 matrix using the baseline and proposed 3D SpTRSV algorithms on the Cori Haswell machine. Z-Comm denotes inter-grid communication time, XY-Comm denotes intra-grid communication time, and FP-Operation denotes the floating-point operation time.

4.2 Performance of 3D SpTRSV on GPU clusters

This subsection provides several numerical examples to demonstrate the enhanced scalability of the proposed 3D SpTRSV algorithm on both AMD and NVIDIA GPUs, compared with the proposed 3D SpTRSV algorithm on CPUs and existing multi-GPU 2D SpTRSV algorithm [12]. As mentioned in Subsection 3.4, ROC-SHMEM for AMD GPUs does not yet support the use of MPI sub-communicators and hence the proposed 3D SpTRSV cannot use more than 1 GPUs per 2D grid. Therefore, we shows scalability results for both AMD and NVIDIA GPUs for $1 \times 1 \times P_z$ layouts, but

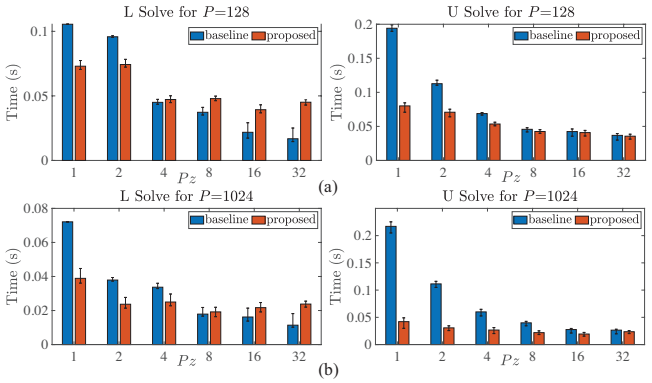

Figure 7: Load balance for the s2D9pt2048 matrix of the baseline and proposed 3D SpTRSV algorithms using (a) $P = 128$ and (b) $P = 1024$. Here the colored bars show the mean value across all MPI ranks and the error bars show the maximum and minimum values across all MPI ranks. The left and right subfigures show the data for the L and U solve phases, respectively. Note that the Z-Comm time is not included.

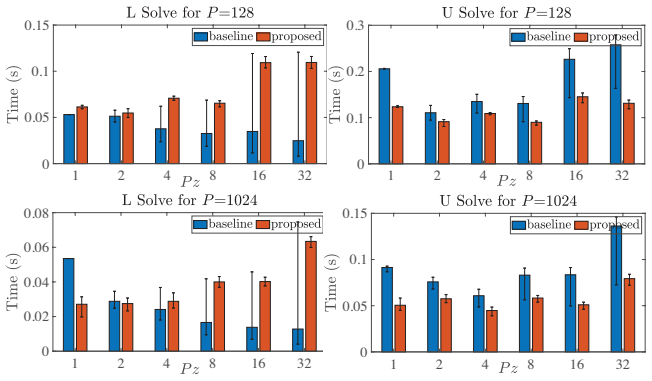

Figure 8: Load balance for the nlpkkt80 matrix of the baseline and proposed 3D SpTRSV algorithms using (a) $P = 128$ and (b) $P = 1024$. Here the colored bars show the mean value across all MPI ranks and the error bars show the maximum and minimum values across all MPI ranks. The left and right subfigures show the data for the L and U solve phases, respectively. Note that the Z-Comm time is not included.

show scalability results only for NVIDIA GPUs for more general $P_x \times P_y \times P_z$ layouts. In all experiments below, we assume one GPU is assigned to one MPI. For each configuration with GPU solves, we also include the results with CPU solves (using the proposed 3D SpTRSV algorithm instead of the baseline 3D SpTRSV algorithm) obtained from the same system using only CPU cores.

4.2.1 $1 \times 1 \times P_z$ layouts on Crusher and Perlmutter. We first test the performance of the proposed 3D SpTRSV algorithms on CPUs and GPUs using the Crusher system with AMD MI250X GPUs. We tested the runtime of CPU and GPU SpTRSV algorithms using the matrices s1_mat_0_253872, s2D9pt2048 and ldoor by changing P_z for 1 to 64 (i.e., 8 compute nodes), with both 1 RHS and 50 RHSs. Fig. 9 shows that total runtime, L-solve time, U-solve time and inter-grid communication time for each configuration. For most matrices, the inter-grid communication time is negligible due to the efficient sparse allreduce operation in Subsection 3.2. The CPU 3D SpTRSV algorithms exhibits good scalability until $P_z = 64$ while the GPU 3D SpTRSV algorithm exhibits good scalability until around $P_z = 16$. The deteriorated scalability for larger P_z is largely due to the increased FP operations as there is no intra-grid communication. The CPU-GPU speedups for these three matrices are up to 1.6x, 1.8x, and 1.7x for 1 RHS and 2.9x, 2.2x and 2.2x for 50 RHSs. Moreover, the L-solve shows higher CPU-GPU speedups compared with the U-solve. This is largely due to reversed computation order and less coalesced memory access patterns of the U-solve compared to the L-solve.

Next, we perform the similar tests of the proposed 3D SpTRSV algorithms on CPUs and GPUs using the Perlmutter system with NVIDIA A100 GPUs. We tested the runtime of CPU and GPU SpTRSV algorithms using the matrices s1_mat_0_253872, s2D9pt2048, nlpkkt80 and dielFilterV3real by changing P_z for 1 to 64 (i.e., 16 compute nodes), with both 1 RHS and 50 RHSs. Fig. 10 shows that total runtime, L-solve time, U-solve time and inter-grid communication time for each configuration. The observations are similar to those on Crusher: Both CPU and GPU 3D SpTRSV algorithms exhibit scalability until $P_z = 64$. For example when P_z changes from 1 to 16 for matrix dielFilterV3real with 1 RHS, the CPU and GPU 3D SpTRSV algorithms exhibit a speedup of 9.5x and 7x, respectively. Moreover, the CPU-GPU speedups for these four matrices are up to 6.5x, 4.6x, 4.8x and 5x for 1 RHS, and 5.2x, 3.7x, 4.1x and 4x for 50 RHSs. These speedup numbers are much higher than those on Crusher.

4.2.2 $P_x \times 1 \times P_z$ layouts on Perlmutter. Finally, we demonstrate the scalability of the proposed 3D SpTRSV algorithm using the Perlmutter system with NVIDIA A100 GPUs with $P_z >= 1$, and $P_x \times P_y > 1$, which requires the use of NVSHMEM-based 2D L- and U-solves described in Subsection 3.4. As reported in [12], for NVSHMEM-based 2D solves, best performance can be obtained with $P_y = 1$ due to the slower reduction performance compared to the broadcast performance. Therefore in this subsection we set $P_y = 1$ and only vary P_x and P_z.

We tested four matrices s1_mat_0_253872, nlpkkt80, Ga19As19H42 and dielFilterV3real with 1 RHS by changing P_z from 1 to 64 and P_y varying from 1 to 4, as shown in Fig. 11. First, it is worth noting that when $P_z = 1$, the 3D GPU SpTRSV algorithm reduces to the NVSHMEM-enhanced 2D GPU SpTRSV algorithm [12]. As shown with the solid red curves in most sub-figures of Fig. 11, the 2D SpTRSV stops scaling at $P = P_y = 8$ which requires NVSHMEM to send data across multiple nodes (recall each Perlmutter GPU node has 4 GPUs). This is expected as the peak intra-node and inter-node bandwidth per direction per GPU on Perlmutter are 300 GB/s and 12.5 GB/s, respectively. Such a big bandwidth performance difference suggests that 2D GPU SpTRSV may not benefit from using more than one GPU node. Therefore, all other data points in Fig. 11 only vary P_x from 1 to 4 to ensure all NVSHMEM communications are confined within one GPU node. Next, we observe that as P_x and P_z increase, both CPU and GPU 3D SpTRSV algorithms exhibit

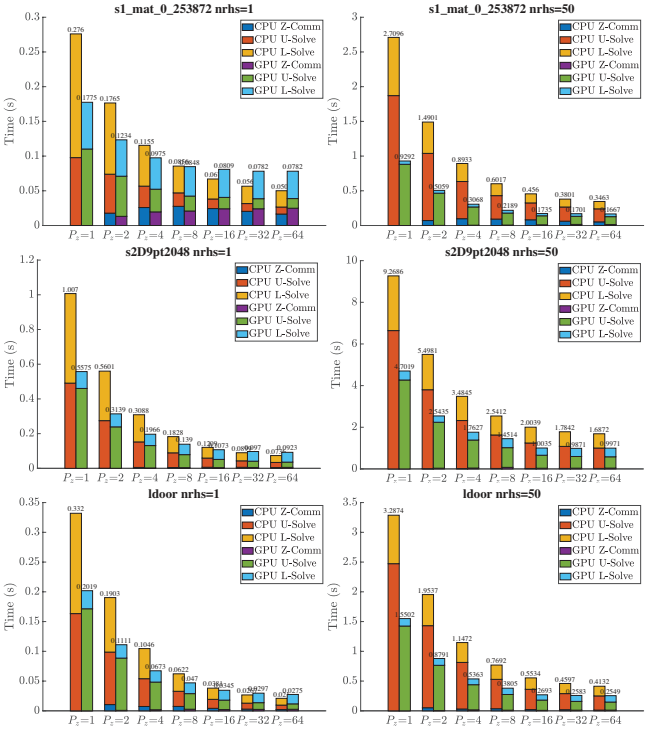

Figure 9: Time breakdown (averaged over all MPI ranks) of the proposed GPU 3D SpTRSV algorithm with 1 and 50 RHSs on the Crusher machine as $P_x = P_y = 1$ and P_z varies. "L-Solve" and "U-solve" denote the time spent in each 2D grid and $P_x = P_y = 1$ leads to no intra-grid communication.

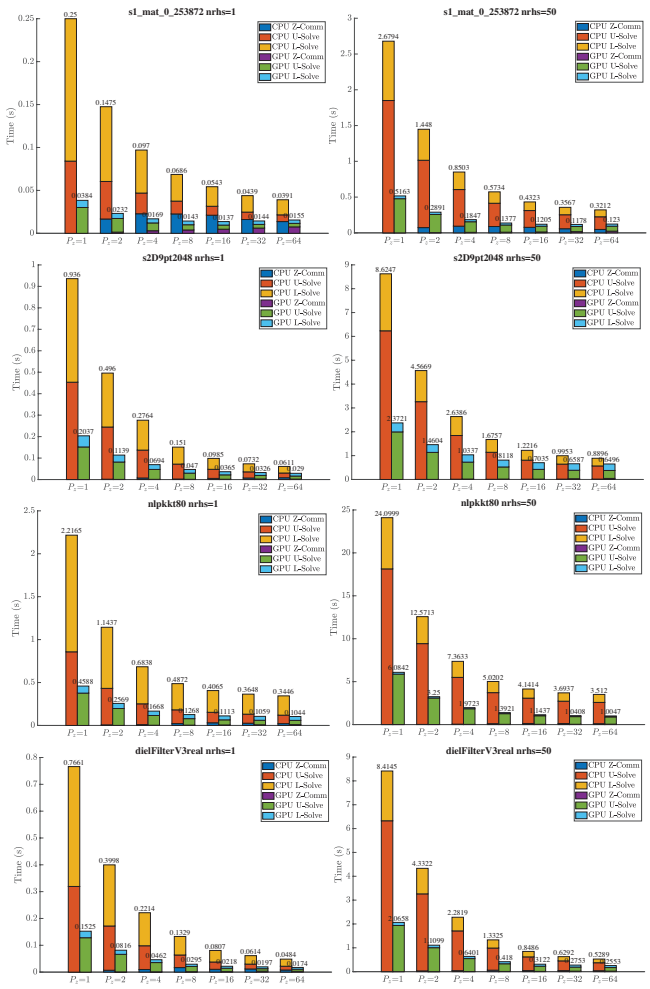

Figure 10: Time breakdown (averaged over all MPI ranks) of the proposed GPU 3D SpTRSV algorithm with 1 and 50 RHSs on the Perlmutter machine as $P_x = P_y = 1$ and P_z varies. "L-Solve" and "U-solve" denote the time spent in each 2D grid and $P_x = P_y = 1$ leads to no intra-grid communication.

good scalability. That said, given a fixed number of $P = P_x \times 1 \times P_z$ GPUs, larger P_z can yield better performance than larger P_x meaning the 3D GPU SpTRSV scales better with increasing P_z. Finally, we remark that as opposed to 2D GPU SpTRSV [12] that only scales up to 4 GPUs, the proposed 3D GPU SpTRSV can scale to 256 GPUs. To our best knowledge, this represents the most scalable GPU SpTRSV implementation to date.

5 CONCLUSIONS

This paper presents a novel communication optimization framework for enhancing scalability of the SpTRSV algorithms on CPU and GPU clusters. The framework builds upon a 3D communication-avoiding parallel layout which decomposes the sparse matrix into submatrices mapped to multiple 2D process grid. Each grid handles a judiciously selected submatrix. The proposed framework advances the existing 3D SpTRSV algorithm with four improvements in communication: an inter-grid synchronization reduced process layout, an efficient inter-grid sparse allreduce communication scheme, integration of a latency reduction scheme for intra-grid communication for CPU clusters, and integration of a GPU-initiated one-sided communication scheme for GPU clusters. The resulting 3D SpTRSV algorithm exhibits up to 3.45x speedups compared to the baseline 3D SpTRSV algorithm using up to 2048 Cori Haswell CPU cores, and scales up to 256 GPUs compared to existing 2D multi-GPU SpTRSV algorithms that can only scale to 4 GPUs using Perlmutter GPU nodes.

Although the unified communication optimization framework is only applied to the SpTRSV algorithm in this paper, the general take-away message is that the proposed domain-decomposition-type hierarchical communication optimization framework applies to many 3D communication avoiding algorithms and their GPU implementations. For each 2D subdomain or subproblem, GPU can boost fine-grained and localized computation, but lacks MPI scalability due to excessive communication. GPU-initiated one-sided communication and CPU-based communication optimization can improve their scalability to some extent. On the other hand, adding the third dimension in the parallel layout can increase coarse-level parallelism at the expense of limited replicated memory and computation. Moreover, it can be very beneficial to avoid intertwining

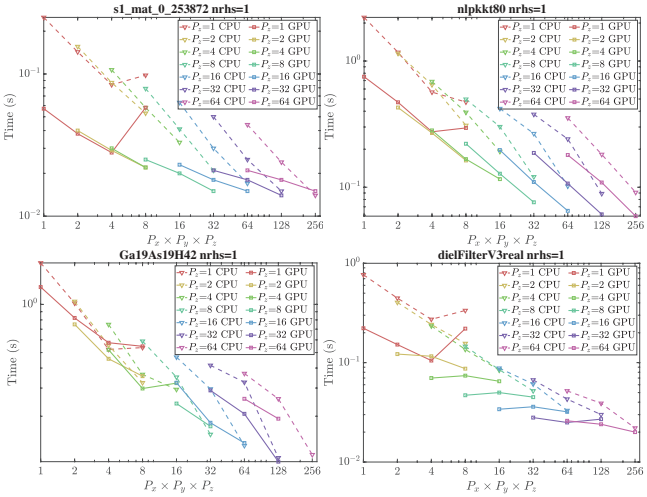

Figure 11: Runtime of the proposed 3D SpTRSV algorithm using the Perlmutter machine as the total MPI counts $P_x \times P_y \times P_z$ vary. For each P_z value (one curve), the 2D grid (P_x, P_y) is set to $P_y = 1$. "CPU" denotes the CPU 3D SpTRSV and "GPU" denotes the GPU 3D SpTRSV with one GPU per MPI rank.

intra-grid and inter-grid communication such that the subdomain can be handled independently as mush as possible.

ACKNOWLEDGMENTS

This research was supported in part by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration, and in part by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Scientific Discovery through Advanced Computing (SciDAC) program through the FASTMath Institute under Contract No. DE-AC02-05CH11231 at Lawrence Berkeley National Laboratory. This research used resources of the National Energy Research Scientific Computing Center (NERSC), a U.S. Department of Energy Office of Science User Facility operated under Contract No. DE-AC02-05CH11231. This research also used resources of the Oak Ridge Leadership Facility which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

REFERENCES

[1] Najeeb Ahmad, Buse Yilmaz, and Didem Unat. 2020. A prediction framework for fast sparse triangular solves. In *Euro-Par 2020: Parallel Processing: 26th International Conference on Parallel and Distributed Computing, Warsaw, Poland, August 24–28, 2020, Proceedings*. Springer, 529–545.

[2] Andrew M. Bradley. 2016. A hybrid multithreaded direct sparse triangular solver. In *Proceedings of SIAM Workshop on Combinatorial Scientific Computing*. 13–22. https://doi.org/10.1137/1.9781611974690.ch2 arXiv:http://epubs.siam.org/doi/pdf/10.1137/1.9781611974690.ch2

[3] Indranil Chowdhury and Jean-Yves L'Excellent. 2010. *Some Experiments and Issues to Exploit Multicore Parallelism in a Distributed-Memory Parallel Sparse Direct Solver*. Research Report RR-7411. INRIA. https://hal.inria.fr/inria-00524249

[4] Timothy A. Davis and Yifan Hu. 2011. The University of Florida sparse matrix collection. *ACM Trans. Math. Softw.* 38, 1, Article 1 (Dec. 2011), 25 pages. https://doi.org/10.1145/2049662.2049663

[5] James Demmel, Laura Grigori, Mark Hoemmen, and Julien Langou. 2008. Communication-avoiding parallel and sequential QR factorizations. *CoRR abs/0806.2159* (2008).

[6] James Demmel, Mark Hoemmen, Marghoob Mohiyuddin, and Katherine Yelick. 2008. Avoiding communication in sparse matrix computations. In *2008 IEEE International Symposium on Parallel and Distributed Processing*. IEEE, 1–12.

[7] J. W. Demmel, S. C. Eisenstat, J. R. Gilbert, X. S. Li, and J. W. H. Liu. 1999. A supernodal approach to sparse partial pivoting. *SIMAX* 20, 3 (1999), 720–755.

[8] J. W. Demmel, J. R. Gilbert, and X. S. Li. 1997. SuperLU and SuperLU_MT. http://www.netlib.org/scalapack/prototype/.

[9] J. W. Demmel, J. R. Gilbert, and X. S. Li. 1997. *SuperLU Users' Guide*. Technical Report UCB//CSD-97-944. Computer Science Division, U.C. Berkeley.

[10] J. W. Demmel, J. R. Gilbert, and X. S. Li. 1999. An Asynchronous Parallel Supernodal Algorithm for Sparse Gaussian Elimination. *SIMAX* 20, 4 (1999), 915–952.

[11] N. Ding, Y. Liu, X. S. Li, and S. Williams. 2019. Leveraging One-Sided Communication for Sparse Triangular Solvers — A Pathway to Exascale Solvers. In *Proceedings of SC19*. Denver, CO.

[12] Nan Ding, Yang Liu, Samuel Williams, and Xiaoye S Li. 2021. A Message-Driven, Multi-GPU Parallel Sparse Triangular Solver. In *SIAM Conference on Applied and Computational Discrete Algorithms (ACDA21)*. SIAM, 147–159.

[13] Nan Ding, Samuel Williams, Yang Liu, and Xiaoye S Li. 2020. Leveraging one-sided communication for sparse triangular solvers. In *Proceedings of the 2020 SIAM Conference on Parallel Processing for Scientific Computing*. SIAM, 93–105.

[14] Ernesto Dufrechou and Pablo Ezzatti. 2018. Solving Sparse Triangular Linear Systems in Modern GPUs: A Synchronization-Free Algorithm. In *2018 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*. 196–203. https://doi.org/10.1109/PDP2018.2018.00034

[15] Ernesto Dufrechou, Pablo Ezzatti, Manuel Freire, and Enrique S. Quintana-Ortí. 2021. Machine learning for optimal selection of sparse triangular system solvers on GPUs. *J. Parallel and Distrib. Comput.* 158 (2021), 47–55. https://doi.org/10.1016/j.jpdc.2021.07.013

[16] Pieter Ghysels and Ryan Synk. 2022. High performance sparse multifrontal solvers on modern GPUs. *Parallel Comput.* 110 (2022), 102897. https://doi.org/10.1016/j.parco.2022.102897

[17] George Karypis and Vipin Kumar. 1998. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.* 20, 1 (1998), 359–392. https://doi.org/10.1137/S1064827595287997 arXiv:https://doi.org/10.1137/S1064827595287997

[18] X. S. Li. 2003. *An Overview of SuperLU: Algorithms, Implementation, and User Interface*. Technical Report LBNL-53848. Lawrence Berkeley National Laboratory. http://crd.lbl.gov/~xiaoye/LBNL-53848.pdf.

[19] X. S. Li. 2005. An Overview of SuperLU: Algorithms, Implementation, and User Interface. *ACM Trans. Math. Software* 31, 3 (September 2005), 302–325.

[20] X. S. Li. 2008. Evaluation of sparse factorization and triangular solution on multicore architectures. In *Proceedings of VECPAR'08 8th International Meeting High Performance Computing for Computational Science*. Toulouse, France.

[21] X. S. Li. 2008. Evaluation of SuperLU on multicore architectures. In *Proceedings of SciDAC 2008 Conference, Journal of Physics: Conference Series 125 (2008) 012079. Institute of Physics Publishing*. Seattle.

[22] Xiaoye S Li and James W Demmel. 1998. Making sparse Gaussian elimination scalable by static pivoting. In *SC'98: Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*. IEEE, 34–34.

[23] X. S. Li and J. W. Demmel. 1999. A Scalable Sparse Direct Solver using Static Pivoting. In *Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing*. San Antonio, Texas.

[24] X. S. Li and J. W. Demmel. 2003. SuperLU_DIST: A Scalable Distributed-Memory Sparse Direct Solver for Unsymmetric Linear Systems. *ACM Trans. Math. Software* 29, 2 (June 2003), 110–140.

[25] Xiaoye S Li, Paul Lin, Yang Liu, and Piyush Sao. 2022. Newly Released Capabilities in Distributed-memory SuperLU Sparse Direct Solver. *ACM Trans. Math. Software* (2022).

[26] X. S. Li and M. Shao. 2009. *A Supernodal approach to imcomplete LU factorization with partial pivoting*. Technical Report LBNL-2178E. Lawrence Berkeley National Laboratory. ACM Trans. Mathematical Software (submitted).

[27] X. S. Li and M. Shao. 2010. A Supernodal approach to imcomplete LU factorization with partial pivoting. *ACM Trans. Math. Software* 37, 4 (2010).

[28] Weifeng Liu, Ang Li, Jonathan D. Hogg, Iain S. Duff, and Brian Vinter. 2017. Fast synchronization-free algorithms for parallel sparse triangular solves with multiple right-hand sides. *Concurrency and Computation: Practice and Experience* 29, 21 (2017), e4244–n/a. https://doi.org/10.1002/cpe.4244 e4244 cpe.4244.

[29] Yang Liu, Mathias Jacquelin, Pieter Ghysels, and Xiaoye S Li. 2018. Highly scalable distributed-memory sparse triangular solution algorithms. In *2018 Proceedings of the Seventh SIAM Workshop on Combinatorial Scientific Computing*. SIAM, 87–96.

[30] Sirine Marrakchi and Mohamed Jemni. 2017. Fine-Grained Parallel Solution for Solving Sparse Triangular Systems on Multicore Platform Using OpenMP Interface. In *2017 International Conference on High Performance Computing & Simulation (HPCS)*. 659–666. https://doi.org/10.1109/HPCS.2017.102

[31] Jan Mayer. 2009. Parallel algorithms for solving linear systems with sparse triangular matrices. *Computing* 86, 4 (16 Sep 2009), 291. https://doi.org/10.1007/s00607-009-0066-3
[32] Padma Raghavan. 1998. Efficient parallel sparse triangular solution using selective inversion. *Parallel Processing Letters* 08, 01 (1998), 29–40. https://doi.org/10.1142/S0129626498000067 arXiv:http://www.worldscientific.com/doi/pdf/10.1142/S0129626498000067
[33] Bram Reps, P Ghysels, O Schenk, K Meerbergen, and W Vanroose. 2015. Communication Avoiding and Hiding in preconditioned Krylov solvers. In *High Performance Computing in Science and Engineering: HPCSE'15*.
[34] F.-H. Rouet. 2012. *Memory and performance issues in parallel multifrontal factorization and triangular solutions with sparse right-hand sides.* Theses. Université de Toulouse.
[35] P. Sao, X. Liu, R. Vuduc, and X.S. Li. 2015. A Sparse Direct Solver for Distributed Memory Xeon Phi-accelerated Systems. In *29th IEEE International Parallel & Distributed Processing Symposium (IPDPS)*. Hyderabad, India.
[36] P. Sao, R. Vuduc, and X. Li. 2014. A Distributed CPU-GPU Sparse Direct Solver. In *Proc. of Euro-Par 2014, LNCS Vol. 8632, pp. 487-498*. Porto, Portugal.
[37] P. Sao, R. Vuduc, and X.S. Li. 2018. A Communication-Avoiding 3D Factorization for Sparse Matrices. In *32nd IEEE International Parallel & Distributed Processing Symposium (IPDPS)*. Vancouver, Canada.
[38] P. Sao, R. Vuduc, and X. Li. 2019. A communication-avoiding 3D algorithm for sparse LU factorization on heterogeneous systems. *J. Parallel and Distributed Computing* (September 2019). https://doi.org/10.1016/j.jpdc.2019.03.004 https://www.sciencedirect.com/science/article/abs/pii/S0743731518305197.
[39] P. Sao, R. Vuduc, and X.S. Li. 2019. A Communication-Avoiding 3D Sparse Triangular Solver. In *ICS 2019: International Conference on Supercomputing*. Phoenix, AZ.
[40] Stanimire Tomov, Rajib Nath, Hatem Ltaief, and Jack Dongarra. 2010. Dense Linear Algebra Solvers for Multicore with GPU Accelerators. In *Proc. of the IEEE IPDPS'10*. IEEE Computer Society, Atlanta, GA, 1–8. DOI: 10.1109/IPDPSW.2010.5470941.
[41] Ehsan Totoni, Michael T. Heath, and Laxmikant V. Kale. 2014. Structure-adaptive parallel solution of sparse triangular linear systems. *Parallel Comput.* 40, 9 (2014), 454 – 470. https://doi.org/10.1016/j.parco.2014.06.006
[42] Xinliang Wang, Weifeng Liu, Wei Xue, and Li Wu. 2018. swSpTRSV: a fast sparse triangular solve with sparse level tile layout on Sunway architectures. In *Proceedings of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)* (Vienna, Austria). 338–353. https://doi.org/10.1145/3178487.3178513
[43] T. Wicky, E. Solomonik, and T. Hoefler. 2017. Communication-avoiding parallel algorithms for solving triangular systems of linear equations. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 678–687. https://doi.org/10.1109/IPDPS.2017.104
[44] Markus Wittmann, Georg Hager, Radim Janalik, Martin Lanser, Axel Klawonn, Oliver Rheinbach, Olaf Schenk, and Gerhard Wellein. 2018. Multicore Performance Engineering of Sparse Triangular Solves Using a Modified Roofline Model. In *2018 30th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. 233–241. https://doi.org/10.1109/CAHPC.2018.8645938
[45] Feng Zhang, Jiya Su, Weifeng Liu, Bingsheng He, Ruofan Wu, Xiaoyong Du, and Rujia Wang. 2021. YuenyeungSpTRSV: A Thread-Level and Warp-Level Fusion Synchronization-Free Sparse Triangular Solve. *IEEE Transactions on Parallel and Distributed Systems* 32, 9 (2021), 2321–2337. https://doi.org/10.1109/TPDS.2021.3066635
[46] İlke Çuğu and Murat Manguoğlu. 2020. A parallel multithreaded sparse triangular linear system solver. *Computers & Mathematics with Applications* 80, 2 (2020), 371–385. https://doi.org/10.1016/j.camwa.2019.09.012 Numerical Methods for Scientific Computations and Advanced Applications II.

A ARTIFACT DESCRIPTION & EVALUATION

A.1 Artifact Description

Machines:

The CPU-only experiments are performed on the Cori Haswell system at NERSC (Fig. 4-8). The GPU experiments (including the reference CPU experiments) are performed on the Crusher system at OLCF (Fig. 9) and the Perlmutter system at NERSC (Fig. 10-11).

- **Cori Haswell** is a Cray XC40 system and consists of 2388 dual-socket nodes with Intel Xeon E5-2698v3 processors running 16 cores per socket. The nodes are equipped with 128 GB of DDR4 memory. The nodes are connected through the Cray Aries interconnect. Please be advised that Cori will retire at the end of April 2023.
- **Crusher** is a testbed system for the Frontier exascale machine and each node consists of a 64-core AMD EPYC 7A53 CPU processor and 4 AMD MI250X GPUs (8 Graphics Compute Dies) each with 64 GB of HBM2 memory. The nodes are connected with HPE Slingshot interconnect with a 25 GB/s bandwidth.
- **Perlmutter** (the GPU partition) is a HPE Cray EX system and consists of 1536 GPU nodes each with a 64-core AMD EPYC 7763 CPU processor, 4 NVIDIA A100 GPUs, and 40 GB HBM memory per GPU. The nodes are connected with the HPE Slingshot 11 interconnect with a 25 GB/s bandwidth.

Software: The proposed SpTRSV algorithm is implemented in the gpu_trisolve_new branch of SuperLU_DIST github repository https://github.com/xiaoyeli/superlu_dist/tree/gpu_trisolve_new. Most software dependencies are available on those three machines, except for parmetis-4.0.3 which can be installed by:

```
wget https://launchpad.net/ubuntu/+archive/primary/+sourcefiles/parmetis/4.0.3-4/parmetis_4.0.3.orig.tar.gz
tar -xf parmetis_4.0.3.orig.tar.gz
cd parmetis-4.0.3/
export CRAYPE_LINK_TYPE=dynamic
mkdir -p install
make config shared=1 cc=cc cxx=CC prefix=$PWD/install
make install
```

Once parmetis has been installed, SuperLU_DIST can be installed in the three test machines as follows:

- **Cori Haswell** The loaded modules are:
  - gcc/11.2.0
  - cray-mpich/7.7.19
  - cray-libsci/20.09.1
  - PrgEnv-gnu/6.0.10
  - cmake/3.22.2

  SuperLU_DIST can be installed on Cori with https://github.com/xiaoyeli/superlu_dist/blob/gpu_trisolve_new/example_scripts/run_cmake_build_haswell_gnu.sh and set the parmetis path accordingly.
- **Crusher** The loaded modules are:
  - gcc/12.2.0
  - cray-mpich/8.1.23
  - cray-libsci/22.12.1.1
  - PrgEnv-gnu/8.3.3
  - cmake/3.23.2
  - rocm/5.3.0

  SuperLU_DIST can be installed on Crusher with https://github.com/xiaoyeli/superlu_dist/blob/gpu_trisolve_new/example_scripts/run_cmake_build_crusher_gnu_gpu_hip_amd.sh and set the parmetis path accordingly.
- **Perlmutter GPUs** The loaded modules are:
  - nvidia/22.7
  - cray-mpich/8.1.25
  - cray-libsci/23.02.1.1
  - PrgEnv-nvidia/8.3.3
  - libfabric/1.15.2.0

– cmake/3.24.3
– cudatoolkit/11.7

In addition, SuperLU_DIST with fully 3D GPU SpTRSV support depends on NVSHMEM 2.8.0, which can be found at https://developer.download.nvidia.com/compute/redist/nvshmem/2.8.0/source/ and installed with https://github.com/xiaoyeli/superlu_dist/blob/gpu_trisolve_new/example_scripts/install_nvshmem_perlmutter.sh. SuperLU_DIST can be installed on Perlmutter with https://github.com/xiaoyeli/superlu_dist/blob/gpu_trisolve_new/example_scripts/run_cmake_build_perlmutter_nvidia_nvshmem.sh and set the parmetis path accordingly.

Data collection:

Most of the test matrices including nlpkkt80, ldoor and dielFilterV3real and Ga19As19H42 are publicly available from the SuiteSparse Matrix Collection at http://sparse.tamu.edu/. The following scripts (after allocating the appropriate numbers of compute nodes) have been used to collect performance data from the three machines.

- **Cori Haswell** The CPU runs with the baseline and proposed 3D SpTRSV algorithms used the script at https://github.com/xiaoyeli/superlu_dist/blob/gpu_trisolve_new/example_scripts/batch_script_mpi_runit_cori_haswell_gcc.sh.
- **Crusher** The CPU and GPU runs on Crusher used the script at https://github.com/xiaoyeli/superlu_dist/blob/gpu_trisolve_new/example_scripts/batch_script_mpi_runit_crusher_gcc_hip_3dsolve.sh.
- **Perlmutter GPUs** The CPU and GPU runs on the Perlmutter GPU partition used the script at https://github.com/xiaoyeli/superlu_dist/blob/gpu_trisolve_new/example_scripts/batch_script_mpi_runit_perlmutter_3dsolve_nvidia_nvshmem.sh.

A.2 Artifact Evaluation

Please follow instructions at https://zenodo.org/record/8260545 and use the docker image liuyangzhuan/superlu3dsptrsv to evaluate the artifact on local machines.

Received 07 April 2023; accepted 16 June 2023