A Communication-Avoiding 3D LU Factorization Algorithm for Sparse Matrices

Piyush Sao
Georgia Institute of Technology
email: piyush3@gatech.edu

Xiaoye S. Li
Lawrence Berkeley National Lab
email: xsli@lbnl.gov

Richard Vuduc
Georgia Institute of technology
email: richie@gatech.edu

***Abstract*—We propose a new algorithm to improve the strong scalability of right-looking sparse LU factorization on distributed memory systems. Our *3D sparse LU algorithm* uses a three-dimensional MPI process grid, aggressively exploits elimination tree parallelism and trades off increased memory for reduced per-process communication. We also analyze the asymptotic improvements for planar graphs (e.g., from 2D grid or mesh domains) and certain non-planar graphs (specifically for 3D grids and meshes). For planar graphs with n vertices, our algorithm reduces communication volume asymptotically in n by a factor of $\mathcal{O}\left(\sqrt{\log n}\right)$ and latency by a factor of $\mathcal{O}(\log n)$. For non-planar cases, our algorithm can reduce the per-process communication volume by $3\times$ and latency by $\mathcal{O}\left(n^{\frac{1}{3}}\right)$ times. In all cases, the memory needed to achieve these gains is a constant factor.**

We implemented our algorithm by extending the 2D data structure used in SuperLU_DIST. Our new 3D code achieves speedups up to 27× for planar graphs and up to 3.3× for non-planar graphs over the baseline 2D SuperLU_DIST when run on 24,000 cores of a Cray XC30.

第一部分 I. Introduction

We wish to improve the strong scalability of sparse direct solvers, which solve a system of linear equations $Ax = b$, where coefficient matrix A is large and sparse, by Gaussian elimination or *sparse LU factorization*. They are notorious for their complex data dependencies, irregular memory access patterns, and highly dynamic arithmetic intensity, which in turn depends on the sparsity pattern of A. Compared to its dense matrix counterpart, communication in a sparse solver can quickly dominate at even relatively small core counts. While techniques like overlapping computation and communication can be effective, they only work well when the computation and communication costs are comparable. In the strong scaling regime, communication must eventually become relatively more expensive.

Thus, we are motivated to redesign algorithms to reduce communication, as the recent flurry of research on *communication-avoiding algorithms* suggests. There, one critical strategy is to shrink the amount of data transferred through redundant computation, data replication, or both. There are several examples for dense linear algebra [10], [11], [18], including some for dense LU [21], [34]. However, precisely how to apply communication-avoiding methods to sparse LU has been open.

In this paper, we describe our design and implementation of the first such method, which we refer to as a *3D sparse LU factorization algorithm*. It is so-named for two reasons, both inspired by the 2.5D *dense* LU algorithm [34]. First, it uses a 3D logical process grid, instead of the 2D process grid that is the state-of-the-art in sparse LU. Second, it replicates data to reduce both the number of messages and the volume of communication. In addition, all sparse LU methods have an *elimination tree structure*, which our method uses to efficiently map the problem to the 3D process grid. As a result, our algorithm not only reduces communication but also reduces the critical path of the factorization—a feature that does not apply to the 2.5D dense LU case. For matrices with planar graph structure (e.g., planar grids and meshes), our 3D sparse LU algorithm's critical path is $\mathcal{O}(n/\log n)$ whereas a state-of-the-art 2D algorithm's is $\mathcal{O}(n)$.

Briefly, here is how 3D sparse LU works. First, consider the 3D process grid as a collection of 2D grids. We divide the elimination tree into independent subtrees and a common ancestor tree of all the subtrees. Factoring each subtree is independent, but each factorization updates the common ancestor tree. We map the factorization of each subtree to a 2D grid and replicate the common ancestor on all process grids. Each 2D grid factorizes its subtree and uses its copy of the common ancestors to perform Schur-complement updates. We then reduce these copies onto a single grid, where it is factored in a 2D fashion.

We implement this scheme on top of SuperLU_DIST, using a hybrid MPI+OpenMP programming model. We measure performance on a wide range of matrices in both 2D and 3D process grid configurations. (The baseline is 2D SuperLU_DIST.) In the best case, we observe speedups of 27× over the best 2D process grid configuration using 1.7× the memory. We observe that our new algorithm can use up to 16× more processors for the same problem size with continued time reduction, which confirms its potential to strongly scale. We also derive performance models to help understand how the performance of the new algorithm depends on the sparsity structure of the matrix and process grid configuration.

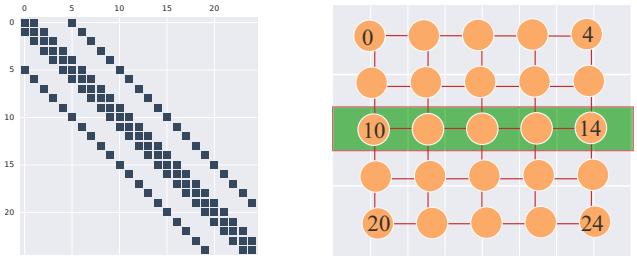
第二部分 II. Background

This section provides some of the relevant background on sparse direct solver and existing algorithm needed to under-

1530-2075/18/$31.00 ©2018 IEEE
DOI 10.1109/IPDPS.2018.00100

Table I: List of symbols used

Symbol type	Symbol	Description
Process	P	#MPI processes
	P_x, P_y, P_z	Process grid dimensions
	P_{xy}	$P_x \times P_y$ # processes in xy plane
	$P_x(k)$	($k \mod P_x$)-th process row
Graphs	E	Elimination tree of A
	E_f	Elimination tree-forest (section III-C)
	S	Top level separator of E
	C_1, C_2	Children etrees of E
Misc.	n	Dimension of the matrix A
	$nlevel$	Height of E $\mathcal{O}(\log n)$
	l	$\log_2 P_z$
	M	Per-process memory
	W	Per-process communication
	L	Latency of factorization
	$T(v)$	*Cost* of factoring node$v$



(a) A 5×5 sparse matrix

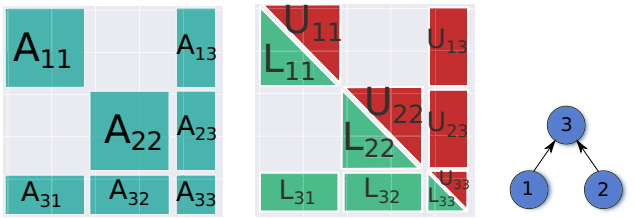
(b) The associated graph and separator

Figure 1: A sparse matrix (Fig. 1a), its associated graph (Fig. 1b), and a separator (highlighted in green).

stand the new 3D algorithm. The most important concepts include the elimination tree, which guides parallelism, as well as the baseline algorithm, which is SuperLU_DIST.

A. Introduction to sparse direct solvers

A sparse direct solver solves a system of linear equation $Ax = b$ in two steps. First, it factors the matrix A into the product $A = LU$, where L is a unit lower triangular matrix and U is an upper triangular matrix. It then solves two triangular systems, $Ly = b$ and $Ux = y$, by forward and backward substitution. Calculating the L and U factors usually takes much more time than substitution. When A is sparse, the factor matrices L and U tend to *fill-in*, meaning they have many more non-zeros than A. Usually, before factorization, the A matrix is permuted to reduce the amount of fill-in in L and U.



(a) A 3×3 block sparse matrix

(b) The L and U factors

(c) The elimination tree

Figure 2: In Fig. 2a, we show the block sparse matrix A obtained from nested dissection of the adjacency graph of A. The L and U factors overwrite A after the factorization, as shown in Fig. 2b. The elimination tree (Fig. 2c) captures the dependencies between the factorization of A_{11}, A_{22}, and A_{33}.

B. A sparse matrix, its associated graph, and separators

Any sparse matrix A of dimension n has a corresponding graph G with n vertices. For any non-zero element a_{ij} in A, there is a directed edge in G from i to j with weight a_{ij}. In Fig. 1a, we show a pentadiagonal matrix, which might arise from a finite difference discretization of a PDE on a 2D square grid, as shown in Fig. 1b.

A *separator* S of the graph G is a subgraph which partitions G into three disjoint subgraphs (C_1, S, C_2) so that C_1 and C_2 are disconnected. A *good* separator is small and the partitions C_1 and C_2 are balanced. In Fig. 1b, we highlight a separator in green. Using this partition, we order the sparse matrix A so that vertices in C_1 and C_2 come first, followed by the vertices in S. For instance, the block sparse matrix in Fig. 2a shows one such ordering, where A_{11}, A_{22}, and A_{33} correspond to C_1, C_2, and S respectively, with remaining submatrices representing the edges that connect these partitions. Then, C_1 and C_2 can be recursively partitioned to get more disjoint subgraphs of A, a process known as *nested dissection* (ND). Graph partitioning tools like METIS can compute ND partitions [25].

C. Sequential Sparse LU factorization

Consider the LU factorization of the 3×3 block sparse matrix shown in Fig. 2a. The L and U factors are computed iteratively. There are three main steps involved in the factorization:

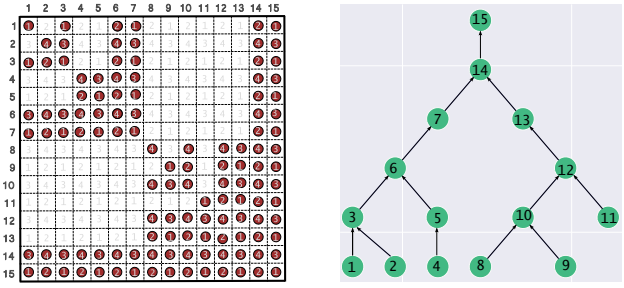
1) *Diagonal factorization*: $A_{ii} \rightarrow L_{ii}U_{ii}$
2) *Panel update*: $U_{ij} = L_{ii}^{-1}A_{ij}$ and $L_{ji} = A_{ji}U_{ii}^{-1}$
3) *Schur-complement update*: $A_{jk} = A_{jk} - L_{ji}U_{jk}$

D. Dependency tree in sparse LU factorization

Factoring the diagonal blocks proceeds sequentially in the dense LU factorization, but not so in sparse LU. In the 3×3 block sparse matrix example, block 1 (A_{11}) and block 2 (A_{22}) may be factored independently and so in any order. But the factorization of block 1 and 2 updates the block 3 (A_{33}). Thus, block 3's factorization must follow that of 1 and 2. This dependency is represented by a tree called *elimination tree* or *etree* for short, as shown in Fig. 2c. An etree will have multiple levels since blocks 1 and 2 are also recursively partitioned. In Fig. 3a and Fig. 3b, we show a larger sparse matrix and its etree.

E. A distributed algorithm: SuperLU_DIST

We build our new algorithm on top of SuperLU_DIST's data structure. SuperLU_DIST is a widely used sparse direct solver library which uses a right-looking scheme



(a) (b)

Figure 3: A distributed sparse matrix and its elimination tree. Suppose the block sparse matrix of Fig. 3a is distributed in a 2×2 process grid. Each circle represents a non-zero block and number denotes the process-id that owns the block. Fig. 3b shows the etree.

and static pivoting. It uses the supernodal approach to find and exploit dense substructures in the sparse LU factorization. SuperLU_DIST uses MPI for inter-process parallelism and OpenMP for intra-process parallelism. We also recently demonstrated GPU and Xeon-Phi acceleration for SuperLU_DIST [32], [33]. In this paper, we focus on reducing inter-node communication and therefore only consider SuperLU_DIST for non-accelerated systems.

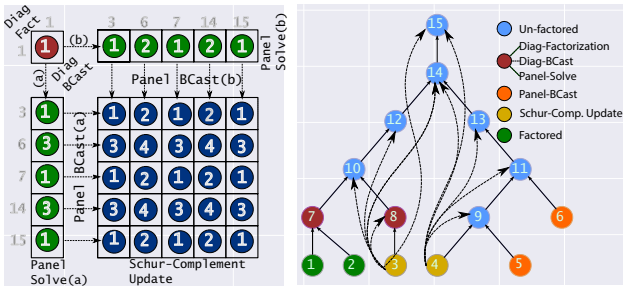
1) Data structure: SuperLU_DIST arranges MPI processes in a 2D logical grid. In this grid, the sparse matrix is distributed in a block-cyclic fashion. In Fig. 3a, we show a sparse matrix distributed in 2×2 process grid.

2) Factorization algorithm: SuperLU_DIST factorizes supernodes following a bottom-up order of the etree. We divide the factorization of a supernode into two steps: panel-factorization and Schur-complement update.

The panel-factorization step computes L and U panels of the current supernode and broadcasts it to all the processes to perform the Schur-complement update. It involves following kernels:

1) *Diagonal factorization*: The process P_{kk}, which owns block A_{kk}, factors it into $L_{kk}U_{kk}$.
2) *Diagonal braodcast*: The process P_{kk} broadcasts L_{kk} across its process row $P_x(k)$ and U_{kk} across its process column $P_y(k)$.
3) *Panel Solve*: Each process in the $P_x(k)$, which owns any block of $A_{k:}$, performs triangular solves to get the corresponding block of $U_{k:}$. Similarly, each process in $P_y(k)$, which owns any block of $A_{:k}$, performs triangular solves to get the corresponding block of $L_{:k}$
4) *Panel broadcast*: Each process in $P_x(k)$ broadcasts blocks of $U_{k:}$ to its process column, and each process in the $P_y(k)$ broadcasts blocks of $L_{:k}$ to its process row.

Qualitatively, the panel-factorization step is the communication phase of the factorization. Panel-factorization involves data transfers, synchronizations, and only a tiny fraction of total floating-point operations. 分解的通信阶段



(a) (b)

Figure 4: Fig. 4a shows the kernels and data involved in factoring supernode 1. Fig. 4b shows how SuperLU_DIST uses the etree for pipelining panel-factorization and Schur-complement update.

Following panel-factorization, each process updates its part of the trailing matrix, also known as Schur-complement. If a process owns A_{ij} in the trailing matrix, then it updates it using the received L and U panels by

$$A_{ij} = A_{ij} - L_{ik}U_{kj}.$$

The L_{ik} and U_{kj} are sparse blocks. To perform the above update, we first pack L_{ik} and U_{kj} into dense BLAS compliant format. Then, we use dense BLAS level 3 routines to compute the product $V = -L_{ik}U_{kj}$. Finally we compute the mapping from V back to A_{ij} and update A_{ij} element-wise.

The Schur-complement update is the main computational step in the factorization. It accounts for most of the floating point operations in the factorization. It also involves a lot of local indirect memory accesses.

In Fig. 4a, we show the regions of the matrix that participate in the different steps when we factor the first supernode. Interested readers can find detailed pseudocode and descriptions of the inner workings of the algorithm elsewhere [32], [36].

F. Task scheduling and the elimination tree

SuperLU_DIST uses the etree's parallelism to overlap computation and communication. It concurrently performs the Schur-complement update of a supernode and panel factorization of nodes in a so-called *lookahead window* [36]. In the bottom-up ordering of factorization of the etree, leaf nodes are factored first. So panel-factorization of the next several nodes do not depend on the panel-factorization or Schur-complement update of the current node. As such, SuperLU_DIST performs panel factorization of the supernodes ahead of their Schur-complement update. But the Schur-complement update of the nodes in the lookahead window cannot be performed in parallel, because the Schur-complements of the leaf nodes may share common blocks of the matrix A. Therefore, SuperLU_DIST sequentially performs the Schur-complement update of each supernode.

Usually a large lookahead window creates too many in-flight messages and requires too much buffer space for the incoming messages. So the lookahead window typically has a fixed size in the range 8-20 steps.

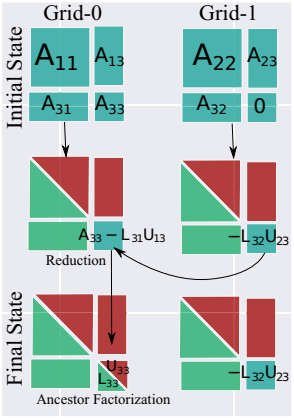


Figure 5: How 3D sparse LU works for a block sparse matrix A (Fig. 2a) using 2 process grids. The sparse blocks 1 and 2, and their panels, reside on grid 1 and grid 2, respectively. Block 3 is replicated in both the grids and is initialized with A_{33} and 0 on grid 0 and grid 1, respectively (the initial state). The two grids factor their respective blocks and Schur-update their copies of the block 3. Then, we reduce the 3rd block from both grids onto grid 0, which is then factored on grid 1. Lastly, the L and U factors are distributed among the two process grids (final state).

G. *Limitations of 2D Sparse LU*

The 2D algorithm scales well up to a certain point, beyond which the cost of data transfer starts to dominate the cost of computation. Moreover, at a large number of processes, the effect of load-imbalance becomes more prominent. So after a certain number of processes, we see that adding more processes can cause a slowdown in the factorization time. Fundamentally, the 2D algorithm suffers from the following two major limitations.

Sequential Schur-complement update: For a given block, only one process can perform the Schur-complement update in the 2D algorithm. So despite abundant tree-level parallelism, the 2D algorithm must perform all Schur-complement updates sequentially.

Fixed latency cost: Almost all processes participate in the factorization of all the supernodes. So the latency of various communication kernels does not decrease with increasing number of processors.

第三部分 III. A 3D Sparse LU Factorization Algorithm

How can we perform the updates on a given block A_{ij} in parallel by two different processes? The 2D algorithm uses an owner-only update policy. So, the Schur-complement update on a given block is sequential. This motivates our approach of replicating some blocks of A on different processes. Doing so allows the Schur-complement updates on those blocks to proceed in parallel. But how do we choose such blocks and processes to replicate?

A. *The 3×3 block sparse case*

We can use the etree to decide how to replicate data. Consider the 3×3 block sparse matrix shown in Fig. 2a and its etree. After factoring blocks 1 and 2, the block A_{33} needs

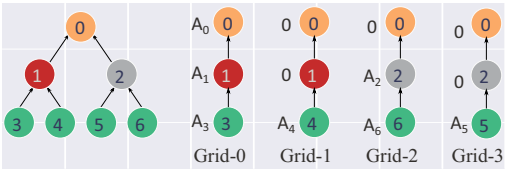


Figure 6: A two-level partition of the elimination tree and its mapping into 4 process grids. Here $A_i = \{A_{ii} \cup A_{i,\ i+1:n} \cup A_{i+1:n,\ i}\}$, represents the set of three sub-matrices a) the diagonal block matrix A_{ii}; b) horizontal off diagonal panel $A_{i,\ i+1:n}$; and c) the vertical off diagonal panel $A_{i+1:n,\ i}$.

updates from both blocks 1 and 2, according to

$$A_{33} = A^0{}_{33} - L_{31}U_{13} - L_{32}A_{23}.$$

We can replicate and keep two copies of the block A_{33}. The first copy accumulates $A^0{}_{33} - L_{31}U_{13}$ from the factorization of the block 1; the second copy accumulates $-L_{32}U_{23}$ from block 2. We then sum the two copies to get final form of A_{33} before factoring it. Thus, the replication of A_{33} allows parallel Schur-complement update of block A_{33}. Fig. 5 shows the timeline of this process.

Formally, we carry out this process as follows. Let E be the etree of the matrix A. We partition E into two independent subtrees, C_1 and C_2, and a common parent S (Fig. 7a). We partition A into $A^0 = A(C_1) \cup A(S)$ and $A^1 = A(C_2) \cup A(S)$ (Fig. 7c and Fig. 7e). We factor A^1 and A^2 in two 2D process grids, grid-0 and grid-1. In grid-1, we initialize the blocks of $A(S)$ with zeros. Grid-0 and grid-1 factor C_1 and C_2 in parallel and update their copy of $A(S)$. After the factorization, they synchronize, and grid-1 sends its copy of $A(S)$ to grid-0:

$$A^0(S) = A^0(S) + A^1(S)$$

Then the grid-0 factors the updated copy of $A(S)$.

The two process grids only need to communicate once. In section IV, we show that this is a small fraction of the total communication. Furthermore, now each process factors a smaller number of supernodes, which reduces latency.

B. *General Case*

Suppose we want to use four 2D grids, instead of two. We can divide the etree in one more level. For instance, in Fig. 6, we have a two-level etree that we divide into four partial etrees. The root (node-0) of the etree is replicated in all the grids. In the first level, we replicate node 1 on grids 0 and 1, and node 2 on grids 3 and 4. In the second level, all the nodes lie on only one grid.

Process grids 0 and 1 synchronize after they have factored nodes 3 and 4, respectively. Then process grids 0 and 1, reduce all the common ancestor nodes, namely nodes 1 and 0 on grid-0. Similarly, process grids 2 and 3 synchronize after they have factored nodes 5 and 6 respectively. Then process grids 2 and 3 reduce all the common ancestor nodes, namely nodes 2 and 0 on grid-2.

In the second step, only grid-0 and grid-2 are active. They factor nodes 1 and 2, and they reduce the updates on node 0 to grid-0. And in the last step, grid-0 factors node 1. We can generalize this process for any $P_z = 2^l$, which is Algorithm 1.

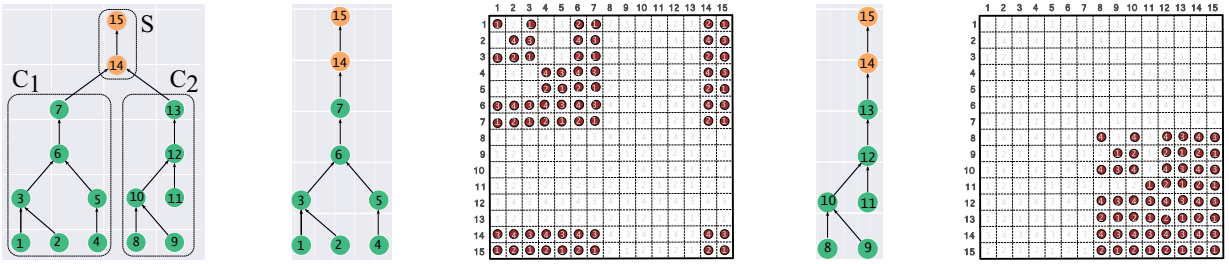


Figure 7: Data distribution in 3D sparse LU algorithm. The elimination tree of the block sparse matrix in Fig. 3a is divided into common ancestor C and subtrees S_1 and S_2 as shown in Fig. 7a. The C_1 and C_2 subtree reside and are factored in process grid-0 and grid-1 respectively, whereas S is replicated in both the $2D$ grids. Fig. 7b and Fig. 7c show the local elimination tree and the data distributed in grid-0, respectively; and Fig. 7d and Fig. 7e show the same for grid-1.

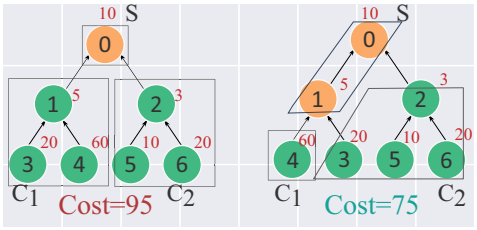


Figure 8: Inter-grid load balancing: an unbalanced elimination tree with 2 ways of mapping nested dissection and a greedy heuristic, and the cost of factorization in the critical path ($T(E) = T(S) + \max\{T(C_1),\ T(C_2)\}$). The cost of factorization of each node is shown in red.

C. Inter-grid Load Balancing

When the sub-trees at the top level are unbalanced, we may further divide the subtrees to another level to get better balance. For instance, in Fig. 8, we show an elimination tree with the cost of factorization of each node. This tree is unbalanced at the top level. The ND ordering (shown Fig. 8 in left) to partition the etree is sub-optimal. We show a better partition of the etree, obtained by dividing the subtrees another level, in the right of Fig. 8. This partition has a smaller critical path of *cost* 75 units versus the ND partition that gives a critical path of length 95 units. In some cases, we may need to divide one of the subtrees even further to obtain the desired balance. We use a greedy heuristic to find a partition so that $T(S) + \max\{T(C_1),\ T(C_2)\}$ is minimized, where $T(C)$ is the cost of factoring nodes in the subtree C. However, we do not know the cost of factoring each node. We use the number of floating-point operations in factoring of a node as a heuristic cost function $T(C)$.

Elimination tree-forest E_f: Our greedy heuristic gives a partition of the etree E that can have multiple disjoint subtrees as a *node*. For instance, in the right partition of Fig. 8, the second child C_2 consists of two unconnected components. So the final partition of the etree is a tree of forests, which we call *Elimination tree-forest* E_f. The E_f obeys the same dependency rules as E. The previous discussions of etree partitioning and mapping to grids remains the same for E_f as well.

The elimination tree-forest has $l = \log_2 P_z$ levels. Each grid only stores the *local* elimination tree-forest. The local elimination tree-forest stores the forests for each level of E_f. For example, for the partition shown in the right on fig. 8, local E_f for grid-0 and 1 is $[S, C_1]$ and $[S, C_2]$, respectively. Each forest is stored as a list of nodes in bottom-up order. So $S = [0]$, $C_1 = [4]$ and $C_2 = [3, 5, 6, 2]$.

D. The Pseudocode of the 3D Sparse LU Factorization Algorithm

Algorithm 1 3D Sparse LU factorization algorithm

```
function dSparseLU3D(A, E_f):
# All process grids execute this function in parallel. P_z is the number of 2D grids
# P_z ← 2^l. Each process-grid has a unique p_z ∈ {0,... P_z − 1}. E_f is the local elimination tree-forest (section III-C).
for lvl in l:0 :
  if p_z = k2^(l−lvl) for some integer k:
#At lvl-th level the only grids that participate are those numbered as a multiple of 2^(l−lvl).
# The following call factors all supernodes of this level E_f[lvl] in the 2D grid, and
# performs the Schur-complement update on their copy of ancestor blocks.
    dSparseLU2D(A,E_f[lvl])
    if lvl > 0:
      if k(mod 2) ≡ 0:        # Note p_z = k2^(l−lvl)
        dest = p_z
        src = p_z + 2^(l−lvl)
      else:
        src = p_z
        dest = p_z − 2^(l−lvl)
      for l_a in lvl − 1: 0:
# Any supernode s in E_f[l_a] consists of blocks A_s = {A_ss ⋃ A_s, s+1:n ⋃ A_s+1:n, s}. If any process with co-ordinate (p_x,p_y,src) owns any block of A_s, then it will send that block to the process with coordinate (p_x,p_y,dest), which then reduce the two copies.
        for s ∈ E_f[l_a]:
          if p_z = src:
            Send A_s^src to dest
          else :
            Receive A_s^src from src
            A_s^dest = A_s^dest + A_s^src
```

The pseudocode of the 3D sparse LU factorization appears in Algorithm 1. The parameter $P_z = 2^l$ is the number of 2D process grids, i.e., P_z is the number of processes in the "z-dimension" of the 3D process grid. And E_f is the elimination tree-forest, the output of our load-balance heuristic. Each process grid only stores forests that resides the grid, and the each forest is stored as list of nodes. The factorization progresses from leaves $lvl = l$ to the root $lvl = 0$. We use another variable $ilvl = l - lvl$ to simplify lengthy index expressions in Algorithm 1. The two main subroutines invoked at any level are $dSparseLU2D$ and Ancestor-Reduction.

1) $dSparseLU2D(A, nList)$: *My* process grid performs the $2D$ factorization of nodes in the $nList$ on my copy of the matrix A. The forest $E_f[tr]$ is passed on to $dSparseLU2D$ as a list of supernodes. Since we use SuperLU_DIST as the baseline data structure, in our implementation $dSparseLU2D$ is a call to modified factorization routine of SuperLU_DIST.
2) *Ancestor-Reduction*: After the factorization of level-i, we reduce the nodes of the ancestor matrix before factoring the next level. In the i-th level's reduction, the receiver is $k2^{l-i+1}$-th process grid and the sender is $(2k+1)2^{l-i}$-th process grid, for some integer k. The process in the 2D grid which owns a block $A_{i,j}$ has the same (x,y) coordinate in both sender and receiver grids. So communication in the ancestor-reduction step is point-to-point and takes places along the z-axis in the 3D process grid.

Aside from these two steps in Algorithm 1, the rest are index calculations.

IV. Analysis of Memory and Communication Costs

How well Algorithm 1 performs relative to the baseline depends on the sparsity pattern of the matrix. However, we can derive analytical expressions of performance on certain model problems, and thereby gain some insight into the algorithm's behavior. Our analysis considers two types of input matrices. The first are associated with planar graphs, such as those that arise when discretizing partial differential equations (PDEs) on 2D domains. The second type are those that arise with 3D PDEs, which have a "well-shaped" geometry but are non-planar.

Below, we derive expressions specifically for memory use, communication volume, and message latency (number of messages) for the baseline SuperLU_DIST algorithm when using a 2D process grid, given a general input matrix. Then, we give expressions for both the 2D and 3D algorithms, specifically for the planar (2D geometry) and non-planar (3D geometry) model problems.

To help distinguish the 2D and 3D *algorithms*, which use 2D and 3D process grids, from the 2D and 3D *model problems*, which have 2D or 3D geometries, we will use "planar" and "non-planar" to refer to the model problems.

A. 2D Sparse LU with a generic sparse matrix

Consider the factorization of a sparse matrix A of dimension n and its elimination tree E. For simplicity, assume that E is balanced at each level. Also assume that the levels in E are indexed from top to bottom. Thus, the root of the tree has level or index 0, and the later levels are indexed from 1 to $nlevel$, where $nlevel+1$ is the number of levels in E. Let the supernode size in level-i has dimension n_i. The i-th level has 2^i nodes.

1) Per-process memory: In sparse LU factorization, typically the LU factors of separator nodes, which are usually dense, account for most of the storage. Thus, each node in level-i requires a memory of n_i^2. Further suppose that SuperLU_DIST, which uses a 2D block cyclic scheme for distributing the LU factors, distributes the factors evenly across P processors. So, the per-process memory, M, required to store all the LU factors is

$$M \approx \frac{1}{P}\sum_{i=0}^{nlevel} 2^i n_i^2. \tag{1}$$

2) Per-process communication volume: The per-process communication volume in the factorization for a dense matrix of size n in a $2D$ process grid, without any data replication, is given by $\mathcal{O}\left(n^2/\sqrt{P}\right)$ [34][1].

To estimate the communication involved in the sparse factorization, we only consider the factorization of the separator nodes. Then the per-process communication of sparse LU on a $2D$ process grid is

$$W \approx \sum_{i=0}^{nlevel} 2^i \frac{n_i^2}{\sqrt{P}} = \mathcal{O}\left(\sqrt{P}M\right). \tag{2}$$

3) Latency: In the $2D$ sparse LU algorithm, each process participates in the factorization of every supernode of the sparse matrix. Thus, the number of steps for factorization is $\mathcal{O}(n)$, and the latency L (number of messages in the critical path) must also scale that way, i.e.,

$$L = \mathcal{O}(n). \tag{3}$$

B. Planar input graphs

For a planar graph with n vertices, we can find a separator of size $\mathcal{O}\left(\sqrt{n}\right)$ in $\mathcal{O}(n)$ time [30]. This result also holds for other classes of graphs, like graphs with bounded genus and graphs with excluded minors [4], [13].

The separator divides the graph into two almost equal halves with $n/2$ vertices each. These subgraphs can further be divided into two almost equal halves with a separator of size $\sqrt{n/2}$. So the separator in the first level is of size $\sqrt{n/2}$ and subsequently, the size of separator in i-th level is $\sqrt{n/2^i}$. This approximation is good when $n/2^i \gg 1$. The number of levels in the elimination tree is $\sim \log n$.

1) Per-process memory :

[1]The network topology and the underlying MPI implementation may increase the asymptotic complexity

Table II: Asymptotic memory, communication and latency costs for 2D and 3D Sparse LU algorithm

	2D PDE			3D PDE	
Parameter	dSparseLU2D	dSparseLU3D	dSparseLU3D $P_z = \mathcal{O}(\log n)$	dSparseLU2D	dSparseLU3D
Per-process Memory (M)	$\mathcal{O}\left(\frac{n}{P}\log n\right)$	$\mathcal{O}\left(\frac{n}{P}(\log(\frac{n}{P_z})+P_z)\right)$	$\mathcal{O}\left(\frac{n}{P}\log n\right)$	$\mathcal{O}\left(\frac{n^{\frac{4}{3}}}{P}\right)$	$\mathcal{O}\left(\frac{n^{4/3}}{P}\left(\kappa P_z+\frac{1}{P_z^{1/3}}\right)\right)$
Pre-process Communication(W)‡	$\mathcal{O}\left(\frac{n}{\sqrt{P}}\log n\right)$	$\mathcal{O}\left(\frac{n}{\sqrt{P}}(2\sqrt{P_z}+\frac{\log n}{\sqrt{P_z}})+\frac{nP_z}{P}\right)$	$\mathcal{O}\left(\frac{n}{\sqrt{P}}\sqrt{\log n}\right)$†	$\mathcal{O}\left(\frac{n^{4/3}}{\sqrt{P}}\right)$	$\mathcal{O}\left(\frac{n^{4/3}}{\sqrt{P}}\left(\kappa_1\sqrt{P_z}+\frac{1-\kappa_1}{P_z{}^{4/3}}\right)\right)$
Latency	$\mathcal{O}(n)$	$\mathcal{O}\left(\frac{n}{P_z}+\sqrt{n}\right)$	$\mathcal{O}\left(\frac{n}{\log n}\right)$	$\mathcal{O}(n)$	$\mathcal{O}\left(\frac{n}{P_z^{2/3}}+\kappa_0 n^{2/3}\right)$

† when $P \gg \log n$

‡ on the critical path of factorization. Average per-process communication is $\mathcal{O}\left(\frac{n\log n}{P_z\sqrt{P_{XY}}}\right)$. For $P_z = \mathcal{O}(\log n)$, both are asymptotically same and equal to $\mathcal{O}\left(\frac{n}{\sqrt{P_{XY}}}\right) = \mathcal{O}\left(\frac{n}{\sqrt{P}}\sqrt{\log n}\right)$.

2D algorithm: We calculate per-process memory using Equation (1). For a planar graph, $n_i = \sqrt{n/2^i}$, so the per-process memory required is

$$M = \frac{1}{P}\sum_{i=0}^{\log n} 2^i n_i^2 = \frac{1}{P}\sum_{i=0}^{\log n} 2^i\left(\sqrt{\frac{n}{2^i}}\right)^2 = \frac{n}{P}\log n \qquad (4)$$

3D algorithm: We assume $P = P_{XY} \times P_z$, where P_z is the number of $2D$ grids of size $P_{XY} = P_x \times P_y$ and $P_z = 2^l$ for some integer l; thus, $l = \log P_z$.

The root node is replicated in all the process layers. Thus it requires $n \cdot 2^l$ memory. Similarly, if $i < l$, level-i will be replicated on 2^{l-i} grids and will require $(\sqrt{n/2^i})^2 \cdot 2^i \cdot 2^{l-i} = n \cdot 2^{l-i}$ words. If instead $i > l$, there will be no replication as each subtree resides in only a single 2D grid. Therefore, for $i > l$, the LU factors will require n memory in each level. Altogether, per-process memory required can be written as:

$$\begin{aligned} M_{3D}(n,\ P,\ P_z) &= \frac{1}{P}\left(\sum_{i=0}^{l} n2^{l-i} + \sum_{i=l+1}^{\log n} n\right) \\ &\approx \frac{1}{P}\left(2nP_z + n\log\frac{n}{P_z}\right). \end{aligned} \qquad (5)$$

2) Per-process communication:

2D algorithm: From Equations (2) and (4), the per-process communication volume of the 2D algorithm on planar graphs is

$$W_{2D} = \frac{n\log n}{\sqrt{P}}. \qquad (6)$$

3D algorithm: We separately calculate the communication in $SuperLU\_DIST2D$ step, denoted as W_{3D}^{xy}, and the communication in Ancestor-Reduction step, denoted as W_{3D}^{z} in Algorithm 1.[2]

Per-process communication in factorization (W_{3D}^{xy}): For the factorization of supernodes in the etree level $i > l$, each process grid works on a $1/2^l$ fraction of the matrix at level-i. Thus, the per-process communication at level-i is $\frac{n}{2^l\sqrt{P_{XY}}}$. However, in level $i < l$, only 2^i process grids participate. Thus, the per-process communication at level-i is $\frac{n}{2^i\sqrt{P_{XY}}}$ for the processes participating in this level.[3] Thus, the total communication across the critical path is given by:

$$W_{3D}^{xy}(n,\ P,\ P_z) = \sum_{i=0}^{l-1}\frac{1}{2^i}\frac{n}{\sqrt{P_{XY}}} + \sum_{i=l}^{\log n}\frac{1}{2^l}\frac{n}{\sqrt{P_{XY}}}$$

We can substitute $2^l = P_z$ and $P_{XY} = \frac{P}{P_z}$. Then, assuming that $\log n \gg l = \log P_z$, this expression simplifies to

$$W_{3D}^{xy}(n,\ P,\ P_z) \approx \frac{n}{\sqrt{P}}\left(2\sqrt{P_z} + \frac{\log n}{\sqrt{P_z}}\right). \qquad (7)$$

Equation (7) is per-process communication for any general 3D process grid. W_{3D}^{xy} has a minimum at

$$P_z = \frac{1}{2}\log n. \qquad (8)$$

Thus, the minimum amount of communication is

$$W_{3D}^{xy}(n,\ P) \approx 2\sqrt{2}\frac{n}{\sqrt{P}}\sqrt{\log n}. \qquad (9)$$

Per-process communication in Ancestor-Reduction (W_{3D}^{z}): We calculate W_{3D}^{z} for grid-0 as it is the only grid that participates in all the levels. Grid-0 receives the root node, distributed among all P_{XY} processes, in each iteration of Algorithm 1. The combined per-process data it receives just for the root is $\frac{l\cdot n}{P_{XY}}$. This expression for the i-th level is $\frac{(l-i)\cdot n}{2^i P_{XY}}$. We sum this expression over all i to get per-process communication in the ancestor-reduction step along the critical path of 3D sparse LU, which is

$$W_{3D}^{z}(n,\ P,\ P_z) = \frac{nl}{P_{XY}} = n\frac{P_z\log P_z}{P}. \qquad (10)$$

Total per-process communication on the critical path: The total per-process communication is $W_{3D} = W_{3D}^{z} + W_{3D}^{xy}$. From Equation (7) and Equation (10),

$$W_{3D}(n,\ P,\ P_z) = \frac{n}{\sqrt{P}}\left(2\sqrt{P_z} + \frac{\log n}{\sqrt{P_z}}\right) + n\frac{P_z\log P_z}{P}.$$

When we choose P_z by Equation (8), this becomes

$$W_{3D}(n,\ P,\ P_z) = \mathcal{O}\left(\frac{n\sqrt{\log n}}{\sqrt{P}} + n\frac{\log n\log\log n}{P}\right). \qquad (11)$$

For any practical $n, P \gg \log n$ even for modest values of P.

[2]All communications in $SuperLU\_DIST2D$ occur in the XY plane.

[3]Note that average per-process communication across all the processes will still be $\frac{n}{2^l\sqrt{P_{XY}}}$, but we are more interested in total communication in the critical path of the factorization.

Thus, for fixed n the first term of Equation (11) dominates.

3) Latency: Latency for the 2D algorithm is $\mathcal{O}(n)$. The latency for 3D algorithm is the dimension of sparse matrix described by the local elimination tree of grid-0. We can show that the latency in the 3D algorithm is:

$$L_{3D}(n,\ P,\ P_z) = \mathcal{O}\left(\frac{n}{P_z} + \sqrt{n}\right) \tag{12}$$

When $P_z = \mathcal{O}(\log n)$, L_{3D} is a $\log n$ factor smaller than L_{2D}.

C. Non-planar input graphs

For the non-planar sparse matrices with a strongly 3D geometry, the 3D factorization algorithm does not reduce the asymptotic communication costs, yet it can reduce the communication and latency by a constant factor. For such matrices, the dimension of the top separator is $n^{2/3}$. Asymptotically, the size of LU factors of the matrix is $\mathcal{O}\left(n^{4/3}\right)$ and almost 20% of it is concentrated in the top separator. So the 3D algorithm cannot reduce the asymptotic complexity of the algorithm. A large separator also means that replicating the top-level root among many 2D grids will rapidly increase the additional per-process memory. However, it can still reduce the communication and latency of the factorization. We summarize the expressions for memory, communication, and latency in Table II. If we choose a P_z that minimizes per process communication, then we can reduce the per-process communication by a factor of 2.89 in the best case. If we choose a P_z that minimizes the latency, then we can reduce the latency by $\mathcal{O}\left(n^{1/3}\right)$ compared to the 2D algorithm, but this P_z will significantly increase per-process memory and the per-process communication.

V. Experimental Results

We evaluate 3D sparse LU against the baseline 2D algorithm. The main results show performance gains from the 3D algorithm at both small and large core counts on a variety of sparse matrices taken from real applications. In addition, we estimate the scaling limits of the 3D algorithm. Beyond measured performance, we quantify the effects of the 3D algorithm on the communication volume and memory usage.

A. Setup

We use SuperLU_DIST's default parameters in our experiments. We ran our experiments on Edison cluster at NERSC. Each node of the Edison contains dual-socket 12-core Intel Ivy Bridge processors. We chose 4 OpenMP threads per MPI process after trying various MPI×OpenMP configurations for different test matrices on 16 nodes. The code was compiled with the Intel C compiler version 18.0.0 and linked with Intel MKL version 2017.2.174 for BLAS operations. We use the same parameters for 3D that we obtained by tuning the 2D code.

Table III: Test sparse matrices used in experiments

Name	Application	n	$\frac{nnz}{n}$	#Flop†	$T^{\ddagger}_{fact}$
audikw_1	Structural	9.4e+5	82.0	1.17e+13	5.70
CoupCons3D	Structural	4.2e+5	53.6	9.09e+11	1.10
dielFilterV3real	FEM/EM	1.1e+6	81.0	2.00e+12	3.80
ldoor	Structural	9.5e+5	44.6	1.69e+11	1.97
nlpkkt80	KKT matrices	1.1e+6	26.5	3.14e+13	10.48
G3_circuit	Circuit Sim.	1.6e+6	4.8	1.21e+11	3.33
Ecology1	Ecology/Circuit	1.0e+6	5.0	4.49e+10	1.36
K2D5pt4096	PDE	1.6e+7	5.0	3.26e+12	59.81
S2D9pt3072	PDE	9.4e+6	9.0	2.47e+12	26.02
Serena	Structural	1.4e+6	46.1	5.97e+13	19.49

† #Floating point operations in the baseline SuperLU_DIST (dSparseLU2D)
‡ Factorization time in seconds for baseline algorithm on 16 nodes.

1) Test matrices: We used four planar and six non-planar matrices, summarized in Table III. The planar matrices come from the discretization of two-dimensional PDEs (*K2D5pt4096*, *S2D9pt3072*) and circuit analysis (*g3_circuit*, *ecology1*). Five of the six non-planar matrices are from the discretization of 3D PDEs and one, matrix *nlpkkt80*, comes from non-linear optimization. The factorization time of the test matrices ranges from 10-55 seconds on 16 nodes when using the baseline 2D SuperLU_DIST.

B. Performance of the 3D algorithm on 16 nodes

The 3D sparse LU configurations achieve 2-11.6× and 0.33-4.9× speedup with respect to 2D SuperLU_DIST for planar and non-planar matrices, respectively. The results appear in Fig. 9, which shows the factorization time normalized by the baseline 2D SuperLU_DIST for each matrix and process configuration. Columns correspond to different 3D process configurations. The leftmost column, P_z = 1, is the 2D algorithm; subsequent columns correspond to P_z values of 2, 4, 8, and 16. The factorization time is divided into two components, T_{scu} and T_{comm}. The T_{scu} is the time spent in Schur-complement update on the critical path of the 3D factorization, and T_{comm} is the non-overlapped communication and synchronization time.

Planar graphs achieve better performance when P_z is large and the 2D grid size is small. Planar matrices have already very high communication cost at 16 nodes. We can see that T_{comm} decreases as we increase P_z. The profiling of *K2d5pt4096* for the 2D algorithm shows severe load imbalance, which also has a cascading effect on the synchronization time. The 3D algorithm at $P_z = 2$ shows less time spent at synchronization points as it has roughly half synchronization point as the 2D algorithm. Some 3D matrices also achieve better performance when P_z is large and 2D grid size is small. For instance, *ldoor* comes from finite element discretization of a large door using a tetrahedral mesh. A "large door" is a very thin, or nearly planar, 3D object, and thus partitions like a 2D object.

We also see a slowdown by up to 4× at P_z = 16 for

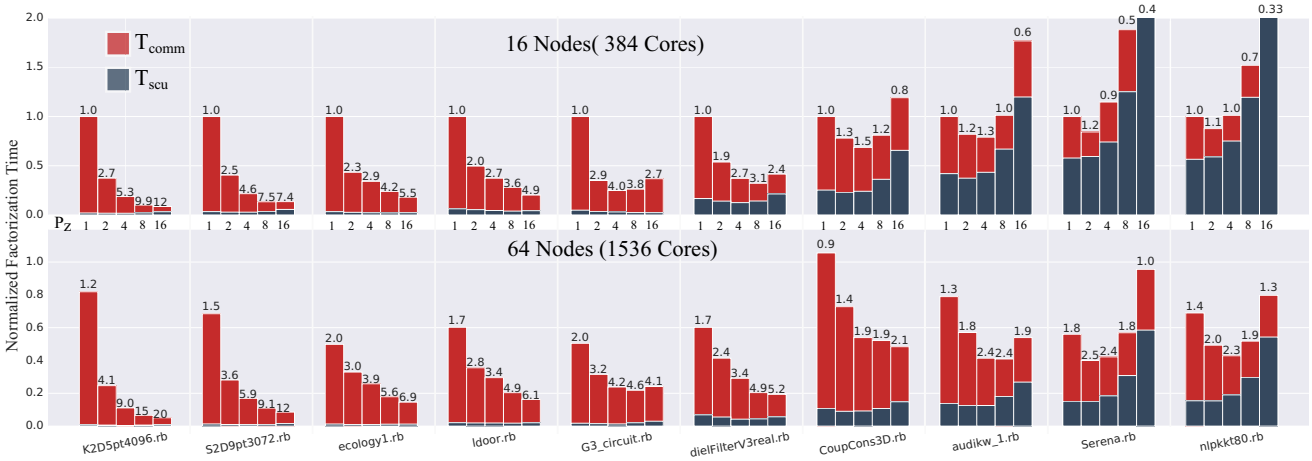
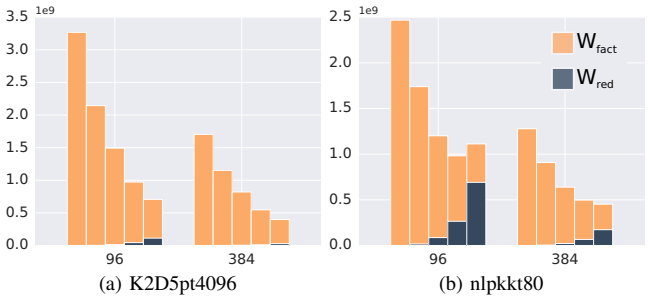


Figure 9: Performance comparison of various $P_x \times P_y \times P_z$ grids on 16 nodes (upper plot) and 64 nodes (lower) on the Edison system at NERSC. For each matrix, each column represents a different value of $P_z = 1, 2, 4, 8, 16$ from left to right. Thus, the leftmost column is the $2D$ algorithm, and when moving right, the $2D$ grids become smaller as P_z increases. For each data set, the time shown is normalized with respect to $2D$ SuperLU_DIST on 16 nodes. T_{scu} is the time spent in the Schur-complement update on the critical path, whereas T_{comm} is the non-overlapped time spent in communication and synchronization.



(a) K2D5pt4096 (b) nlpkkt80

Figure 10: Per-process communication volume (in bytes) for different process grid configurations on 16 and 64 nodes (or 96 and 384 MPI processes, respectively). Each column in a group represents a $P_{XY} \times P_z$ process grid configuration, where $P_z \in 1, 2, 4, 8, 16$ from left to right (leftmost being a purely $2D$ configuration). Here, W_{fact} is number of words sent during the local factorization along the $2D$ grid, whereas W_{red} is number of words sent in the ancestor-reduction step along the z-axis.

extremely non-planar matrices *Serena* and *nlpkkt80*. For these matrices, computation is still a large fraction of factorization time for the baseline 2D algorithm at 384 cores. Most of those computations are concentrated in the top few levels of the etree. So reducing the 2D process grid size increases T_{scu}, which masks any gains from reduced communication.

C. Results on 64 Nodes

On 64 nodes, the 3D sparse LU configurations achieve 2-16.6× and 1.0-3.6× speedup with respect to 2D SuperLU_DIST for planar and non-planar matrices, respectively. On 64 nodes the factorization time is qualitatively similar to the 16 nodes. Except now for all the matrices T_{comm} dominates the factorization time for the baseline 2D algorithm. Therefore, even for extremely non-planar matrices *Serena* and *nlpkkt80*, 3D configurations achieve speed-up of 1.7 and 1.9×, respectively.

D. Effects on per-process communication

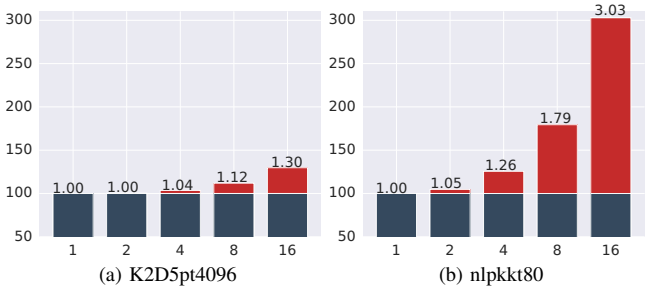
For the planar graphs, 3D algorithm reduces per-process communication by 3-4.6× on 16 nodes and by 4-4.7× on 64 nodes. For non-planar graphs, 3D algorithm reduces per-process communication by 2.5-3.2× on 16 nodes and by 2.9-3.7× on 64 nodes. Fig. 10 shows the per-process communication volume along the critical path of the 3D algorithm, for 16 and 64 nodes, and a planar and a non-planar matrix. We distinguish the number of words sent during the 2D factorization step (W^{fact}) and that of ancestor reduction (W^{red}) of Algorithm 1.

The W^{fact} decreases with increasing P_z. Yet at large P_z, W^{total} can increase. For instance, W^{total} increases for *nlpkkt80* at 16 nodes when going from $P_z = 8$ to 16. It is so as W^{red} increases almost linearly with P_z. Yet for planar graphs, this increase isn't much as they have very small separators at the top level. We estimate that for *K2d5pt4096*, W^{total} will increase with P_z after $P_z > 64$ at 96 processes.

Nevertheless, W^{red} decreases as $1/P_{XY}$ and W^{fact} does decrease as $1/\sqrt{P_{XY}}$ with increasing P_{XY}. So for larger P_{XY}, the cross-over P_z will be even larger.

E. Memory overhead

The 3D algorithm needs 30% more memory for the planar graph *K2D5pt4096* and 200% more for the non-planar graph *nlpkkt80*, at $P_z = 16$ (see Fig. 11). Memory overhead comes from replicating the dense separator blocks on all the process grids. Since the planar graphs have small separators, the memory overhead grows slowly with increasing P_z. Our model suggests that $P_z = \mathcal{O}(\log n)$ before memory overhead becomes comparable to memory of the LU factors. But non-planar graphs, like *nlpkkt80*, do not have good separators. Therefore, the memory overhead increases quickly. At $P_z =$



(a) K2D5pt4096 (b) nlpkkt80

Figure 11: The relative memory overhead of 3D sparse LU algorithm over 2D (in percent).

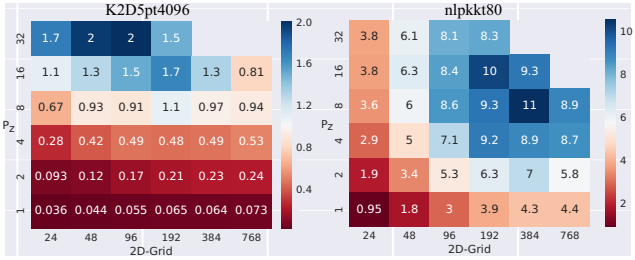


Figure 12: The performance (in TFLOP/s) of the 3D algorithm for different $P_{XY} \times P_z$ combinations. Here, we increase P_{XY} and P_z by a factor of 2 along x and y-axis respectively. Thus, bottommost row ($P_z = 1$) is the 2D algorithm. The performance is calculated using number of floating point operations in baseline 2D factorization (table III).

16, *nlpkkt80* already needs 200% more memory. Overall at $P_z = 16$, memory overhead ranged between 18% to 245% for all matrices we tried.

F. Performance at Large Number of Cores

The best case speed-up: for a matrix, is the speed-up of the best $P_{XY} \times P_z$ configuration relative to the best possible 2D process configuration. The best case speed-up is 5-27.4× for the planar graphs and 2.1-3.3× for non-planar graphs. We show a *heatmap* plot of performance for *K2D5pt4096* and *nlpkkt80* in Fig. 12 for different combinations of $P_{XY} \times P_z$. The performance is shown in Tera-floating point operations per second (TFLOP/s).

Depending on their geometry and size, different matrix achieves the best performance on different $P_{XY} \times P_z$. For a given $P = P_{XY} \times P_z$, planar graph *K2D5pt4096* achieves best performance along the line $P_{XY} = 24$. Strongly non-planar graph *nlpkkt80* achieves best performance along the line $P_z = P_{XY}/24$(↗) for a constant $P = P_{XY} \times P_z$. For all the other matrices achieved the best performance between the two lines[4]. In the best case, we achieved 27.4× speed-up for graph *K2D5pt4096*. And on average the best 3D configuration was 6.5× faster than the best 2D configuration among all the matrices.

VI. Related Work

The idea of using data replication to reduce communication in LU factorization goes back to Ashcraft, who described the first dense LU factorization based on a three-dimensional logical partitioning of the grid [5]. Ashcraft later presented the *fan-both* family of Cholesky factorization algorithm [6], which is a generalization of his 3D LU factorization algorithm. Later, Irony and Toledo [21] and Solomonik and Demmel [34] also described LU factorization algorithms using logical 3D partitionings of MPI processes. The central idea of all work above and ours is same, i.e., using multiple copies of the matrix to perform multiple Schur-complement updates in parallel. The total communication volume of all the above algorithms is $\mathcal{O}\left(n^2\sqrt[3]{P}\right)$, an asymptotic improvement over $\mathcal{O}\left(n^2\sqrt{P}\right)$ for 2D algorithms. However, these algorithms also increase the latency costs. Solomonik and Demmel showed that for such algorithms, communication costs are inversely proportional to the latency costs. Thus, despite the lower asymptotic communication complexity, the performance gains of these algorithms are limited even on communication bound problems. In contrast, our 3D algorithm reduces both bandwidth and latency by using the elimination tree parallelism. It is possible to use these algorithms for factoring dense nodes at the top levels of the etree. But we would like to avoid using them at the lower levels because of the increased latency.

Hulbert and Zmijewski [20] presented a column-oriented distributed sparse Cholesky. It can be considered as a special case of our 3D algorithm with $P_{XY} = 1$. For planar graphs, the per process communication volume in their case is $\mathcal{O}(n \log P)$, as oppose to $\mathcal{O}\left(\frac{n\sqrt{\log n}}{\sqrt{P}}\right)$ in our case.[5] However, their approach can only use $\mathcal{O}(\log n)$ processes for planar problems as opposed to $\mathcal{O}(n \log n)$ processes in the our 3D sparse LU algorithm. For sparse matrices with non-planar associated graph, $P_{XY} = 1$ will be extremely inefficient.

Multifrontal methods also use additional data to improve the locality and communication. A notable such example is from Gupta et al. [16]. The per-process communication volume in their multifrontal sparse Cholesky algorithm for planar graphs is asymptotically $\mathcal{O}\left(\frac{n}{\sqrt{P}}\right)$, which is lower than $\mathcal{O}\left(\frac{n}{\sqrt{P_{XY}}}\right)$ of our 3D algorithm by a factor of $\sqrt{P_z} = \sqrt{\log n}$ (see table II). Yet, their algorithm can use only $\mathcal{O}(n)$ processes in comparison to $\mathcal{O}(n \log n)$ processes for our 3D algorithm. Consequently, for achieving same parallel efficiency, the per-process memory requirement for their algorithm increases with increasing n, whereas it remains constant for the 3D sparse LU algorithm. It's worth noting that, for achieving similar parallel efficiency among their and our algorithm, their algorithm will use $\mathcal{O}(\log n)$ more memory M than our algorithm and reduce communication W by a factor of $\mathcal{O}\left(\sqrt{\log n}\right)$ to our algorithm. Thus, for such a scenrio, the two algorithms have the same $WM^{1/2} = \mathcal{O}\left(\frac{n^{3/2}\log n}{P}\right)$. For

[4]If we had a completely dense matrix the best performance would have occurred along the line $P_z = 1$.

[5]We get the same expression if we substitute $P = P_z$ in eq. (10).

matrix multiplication-like dense linear algebra algorithms, it is known that [9], [22], [24], [34]

$$W = \Omega\left(\frac{\#\text{ Arithmetic Operations}}{\sqrt{M}}\right). \quad (13)$$

The number of arithmetic operations for sparse LU factorization for the planar graph is $\mathcal{O}\left(n^{3/2}/P\right)$. Thus, if eq. (13) holds also for sparse matrices then our 3D algorithm and Gupta's multifrontal method are not optimal by a factor of $\mathcal{O}(\log n)$. However, its likely that eq. (13) is not the same for sparse LU factorization algorithm. That's because dense computations perform $\mathcal{O}\left(n^3\right)$ operations on n^2 data, whereas sparse LU factorization of planar graphs performs $\mathcal{O}\left(n^{3/2}\right)$ operations on $n \log n$ data. Establishing similar lower bounds for sparse LU factorization as eq. (13) warrants further investigation. In addition, the dynamic memory requirement of the multifrontal method can be prohibitive and does not scale well with increasing number of processors, i.e., per-process memory requirement may increase with increasing number of processors. Therefore, significant effort has been on improving the memory scalability [1], [12]. So, such methods trade-off parallelism and performance to reduce memory requirements.

Similar to the multifrontal method, our 3D algorithm also uses elimination tree parallelism to reduce communication. Our mapping of subtrees to process layers is very similar to tree-based mapping algorithm for multifrontal methods. Also, our 3D LU factorization remains predominantly right-looking, which algorithmically is very different from the multifrontal methods. A comprehensive discussion on differences in right-looking and multifrontal methods can be found elsewhere [17], [31].

The use of the elimination tree parallelism to improve the scalability of the right-looking direct solver has also been explored previously, albeit, with a focus on hiding communication by pipelining and overlapping with the computation, and as such, did not reduce communication volume [36].

Researchers have also proposed communication-avoiding pivoting strategies to make LU factorization more scalable [7], [15], [26]. Since SuperLU_DIST uses static pivoting with iterative refinement, these techniques are not needed.

Among sparse direct solvers, prior work has studied efficient scheduling [2], [3], [14], [23], [27], [29], [36]. To improve the overlap of communication and computation, efficient lookahead techniques are part of state-of-practice for both dense and sparse direct solvers [8], [35], [36]. Lacoste [28] and Hugo [19] have also addressed memory and compute resource management for scaling multifrontal sparse direct solvers. The baseline SuperLU_DIST incorporates similar techniques.

VII. Conclusions and Future Work

Our new 3D algorithm shows precisely how communication-avoiding techniques, namely the use of data replication, can be extended from the dense to the sparse case. Our discussion was limited to right-looking LU factorization and static pivoting. However, we believe these principles could be applied to other variants of sparse factorization, such as Cholesky or QR decomposition.

In previous work, we proposed techniques for SuperLU_DIST that can exploit manycore co-processors (e.g., GPU [33] and Xeon Phi [32]). Our "HALO" algorithm for accelerator offload can be seen as an instance of the 3D sparse LU algorithm presented in this paper where accelerators form another parallel 2D grid in addition to the host multicore CPUs. Despite that, HALO works much better for matrices that have large dense blocks; while 3D sparse LU factorization performs better for sparser matrices with small dense separators. We plan to add HALO to the 3D algorithm for hybrid clusters. We believe that by combining the two, we can potentially improve performance across a wider spectrum of matrices and platforms.

To improve the performance of the 3D algorithm for matrices with large dense blocks, we can in principle use a dense 2.5D LU algorithm to factor the supernodes on levels where we only use a subset of 2D grids. Alternatively, for those levels, we can merge two 2D grids to make a larger 2D grid to factor denser blocks. However, doing so would require significant changes to the data structure. Consequently, we plan to pursue this idea as part of our future work.

References

[1] E. Agullo, P. R. Amestoy, A. Buttari, A. Guermouche, J.-Y. L'Excellent, and F.-H. Rouet. Robust memory-aware mappings for parallel multifrontal factorizations. *SIAM Journal on Scientific Computing*, 38(3):C256–C279, 2016.

[2] E. Agullo, A. Buttari, A. Guermouche, and F. Lopez. Implementing multifrontal sparse solvers for multicore architectures with sequential task flow runtime systems. *ACM Transactions on Mathematical Software (TOMS)*, 43(2):13, 2016.

[3] E. Agullo, L. Giraud, and S. Nakov. Task-based sparse hybrid linear solver for distributed memory heterogeneous architectures. In *European Conference on Parallel Processing*, pages 83–95. Springer, 2016.

[4] N. Alon, P. Seymour, and R. Thomas. A separator theorem for graphs with an excluded minor and its applications. In *Proceedings of the twenty-second annual ACM symposium on Theory of computing*, pages 293–299. ACM, 1990.

[5] C. Ashcraft. A taxonamy of distributed dense LU factorization methods. *Engineering Computing and Analysis Technical Report ECA-TR-161, Boeing Computer Services*, 1991.

[6] C. Ashcraft. The fan-both family of column-based distributed cholesky factorization algorithms. In *Graph Theory and Sparse Matrix Computation*, pages 159–190. Springer, 1993.

[7] M. Baboulin, X. S. Li, and F.-H. Rouet. Using random butterfly transformations to avoid pivoting in sparse direct methods. In *International Conference on High Performance Computing for Computational Science*, pages 135–144. Springer, 2014.

[8] M. Bach, M. Kretz, V. Lindenstruth, and D. Rohr. Optimized hpl for amd gpu and multi-core cpu usage. *Computer Science-R&D*, 26(3-4):153–164, 2011.

[9] G. M. Ballard. *Avoiding communication in dense linear algebra*. PhD thesis, 2013.

[10] J. Demmel, M. Hoemmen, M. Mohiyuddin, and K. Yelick. Avoiding communication in sparse matrix computations. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–12. IEEE, 2008.

[11] J. W. Demmel, L. Grigori, M. Gu, and H. Xiang. Communication avoiding rank revealing qr factorization with column pivoting. *SIAM Journal on Matrix Analysis and Applications*, 36(1):55–89, 2015.

[12] L. Eyraud-Dubois, L. Marchal, O. Sinnen, and F. Vivien. Parallel scheduling of task trees with limited memory. *ACM Transactions on Parallel Computing*, 2(2):13, 2015.

[13] J. R. Gilbert, J. P. Hutchinson, and R. E. Tarjan. A separator theorem for graphs of bounded genus. *Journal of Algorithms*, 5(3):391–407, 1984.

[14] D. Goik, K. Jopek, M. Paszyński, A. Lenharth, D. Nguyen, and K. Pingali. Graph grammar based multi-thread multi-frontal direct solver with galois scheduler. *Procedia Computer Science*, 29:960–969, 2014.

[15] L. Grigori, J. W. Demmel, and H. Xiang. CALU: a communication optimal LU factorization algorithm. *SIAM Journal on Matrix Analysis and Applications*, 32(4):1317–1350, 2011.

[16] A. Gupta, G. Karypis, and V. Kumar. Highly scalable parallel algorithms for sparse matrix factorization. *IEEE Transactions on Parallel and Distributed Systems*, 8(5):502–520, 1997.

[17] M. T. Heath, E. Ng, and B. W. Peyton. Parallel algorithms for sparse linear systems. *SIAM review*, 33(3):420–460, 1991.

[18] M. Hoemmen. *Communication-avoiding Krylov subspace methods*. University of California, Berkeley, 2010.

[19] A.-E. Hugo, A. Guermouche, P.-A. Wacrenier, and R. Namyst. A runtime approach to dynamic resource allocation for sparse direct solvers. In *Parallel Processing (ICPP), 2014 43rd International Conference on*, pages 481–490. IEEE, 2014.

[20] L. Hulbert and E. Zmijewski. Limiting communication in parallel sparse cholesky factorization. *SIAM Journal on Scientific and Statistical Computing*, 12(5):1184–1197, 1991.

[21] D. Irony and S. Toledo. Trading replication for communication in parallel distributed-memory dense solvers. *Parallel Processing Letters*, 12(01):79–94, 2002.

[22] D. Irony, S. Toledo, and A. Tiskin. Communication lower bounds for distributed-memory matrix multiplication. *Journal of Parallel and Distributed Computing*, 64(9):1017–1026, 2004.

[23] M. Jacquelin, Y. Zheng, E. Ng, and K. Yelick. An asynchronous task-based fan-both sparse cholesky solver. *arXiv preprint arXiv:1608.00044*, 2016.

[24] H. Jia-Wei and H.-T. Kung. I/o complexity: The red-blue pebble game. In *Proceedings of the thirteenth annual ACM symposium on Theory of computing*, pages 326–333. ACM, 1981.

[25] G. Karypis and V. Kumar. Family of graph and hypergraph partitioning software. http://glaros.dtc.umn.edu/gkhome/views/metis. Accessed: 2014-01-26.

[26] A. Khabou, J. Demmel, L. Grigori, and M. Gu. Communication avoiding LU factorization with panel rank revealing pivoting. *SIAM Journal on Matrix Analysis and Applications*, 34(3):1401–1429, 2013.

[27] K. Kim and V. Eijkhout. A parallel sparse direct solver via hierarchical dag scheduling. *ACM Transactions on Mathematical Software (TOMS)*, 41(1):3, 2014.

[28] X. Lacoste. *Scheduling and memory optimizations for sparse direct solver on multi-core/multi-gpu duster systems*. PhD thesis, Bordeaux, 2015.

[29] X. Lacoste, M. Faverge, G. Bosilca, P. Ramet, and S. Thibault. Taking advantage of hybrid systems for sparse direct solvers via task-based runtimes. In *Parallel & Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International*, pages 29–38. IEEE, 2014.

[30] R. J. Lipton and R. E. Tarjan. A separator theorem for planar graphs. *SIAM Journal on Applied Mathematics*, 36(2):177–189, 1979.

[31] E. Rothberg. Exploiting the memory hierarchy in sequential and parallel sparse cholesky factorization. Technical report, STANFORD UNIV CA DEPT OF COMPUTER SCIENCE, 1992.

[32] P. Sao, X. Liu, R. Vuduc, and X. Li. A sparse direct solver for distributed memory Xeon Phi-accelerated systems. In *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*, pages 71–81. IEEE, 2015.

[33] P. Sao, R. Vuduc, and X. S. Li. A distributed cpu-gpu sparse direct solver. In *European Conference on Parallel Processing*, pages 487–498. Springer, 2014.

[34] E. Solomonik and J. Demmel. Communication-optimal parallel 2.5d matrix multiplication and LU factorization algorithms. In *Euro-Par 2011 Parallel Processing*, pages 90–109. Springer, 2011.

[35] P. Strazdins et al. A comparison of lookahead and algorithmic blocking techniques for parallel matrix factorization. 1998.

[36] I. Yamazaki and X. S. Li. New scheduling strategies and hybrid programming for a parallel right-looking sparse LU factorization algorithm on multicore cluster systems. In *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 619–630. IEEE, 2012.