Journal of Parallel and Distributed Computing 131 (2019) 218–234

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

A communication-avoiding 3D algorithm for sparse LU factorization on heterogeneous systems

Piyush Sao [a,*,1], Xiaoye S. Li [b], Richard Vuduc [c]

[a] Oak Ridge National Laboratory, United States
[b] Lawrence Berkeley National Laboratory, United States
[c] Georgia Institute of Technology, United States

HIGHLIGHTS

- The 3D sparse LU factorization algorithm reduces the asymptotic communication complexity by $(\sqrt{\log n})$ for planar sparse matrices of dimension n and by a constant factor for non-planar sparse matrices.
- The 3D sparse LU factorization algorithm improves the strong scaling from 1000 cores to 32,000 cores on problems of interest, resulting in a speedup of up to 27X on planar and up to 3.3X on non-planar sparse matrices.
- On smaller node counts of heterogeneous clusters, the 3D algorithm with the co-processor acceleration improves the performance up to additional 3.5X.

ARTICLE INFO

Article history:
Received 23 July 2018
Received in revised form 8 January 2019
Accepted 11 March 2019
Available online 24 April 2019

ABSTRACT

We propose a new algorithm to improve the strong scalability of right-looking sparse LU factorization on distributed memory systems. Our *3D algorithm for sparse LU* uses a three-dimensional MPI process grid, 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., those arising from 2D grid or mesh discretizations) and certain non-planar graphs (specifically for 3D grids and meshes). For a planar graph 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 empirical speedups up to $27\times$ for planar graphs and up to $3.3\times$ for non-planar graphs over the baseline 2D SuperLU_DIST when run on 24,000 cores of a Cray XC30. We extend the 3D algorithm for heterogeneous architectures by adding the Highly Asynchronous Lazy Offload (Halo) algorithm for co-processor offload [44]. On 4096 nodes of a Cray XK7 with 32,768 CPU cores and 4096 Nvidia K20x GPUs, the 3D algorithm achieves empirical speedups up to $24\times$ for planar graphs and $3.5\times$ for non-planar graphs over the baseline 2D SuperLU_DIST with co-processor acceleration.

1. Introduction

We wish to improve the strong scalability of sparse direct solvers, which given a large and sparse coefficient matrix A solve the system of linear equations $Ax = b$ by Gaussian elimination or *sparse LU factorization*. Such solvers 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 eventually becomes relatively more expensive.

* Corresponding author.
E-mail addresses: saopk@ornl.gov (P. Sao), xsli@lbnl.gov (X.S. Li), richie@gatech.edu (R. Vuduc).
URLs: http://crd-legacy.lbl.gov/~xiaoye (X.S. Li), https://vuduc.org (R. Vuduc).
[1] During period of this research, Piyush Sao was affiliated with Georgia Institute of Technology.

https://doi.org/10.1016/j.jpdc.2019.03.004
0743-7315/

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 [12,13,21], including some for dense LU [24,46]. 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 [46]. 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 the Schur-complement updates. We then reduce these copies onto a single grid, where it is factorized in a 2D fashion. Because of ancestor replication, this 3D scheme requires more memory than a pure 2D algorithm; however, the independent 2D subgrids of the 3D scheme operate on smaller subproblems, which can reduce communication. In Section 4, we derive analytical performance models for broad classes of sparsity patterns to explain this trade-off more precisely.

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 improve strong scaling.

We then extend the 3D algorithm for hybrid architecture consisting of both multicore CPU and co-processors such as GPU or Xeon-Phi. We augment the 3D algorithm with the Halo algorithm [44] for offloading the compute-intensive Schur-complement update computations to the co-processor. The Halo algorithm, like the 3D factorization algorithm, uses data replication to reduce the communication between host multicore and the co-processor. We evaluate the 3D algorithm augmented with Halo on the GPU accelerated Cray XK7 system. On smaller 2D process grids, the GPU acceleration can improve the performance by a 1.4–3.5× over the unaccelerated 3D sparse LU factorization. Thus, by using the 3D algorithm, we can use larger node counts where we attain meaningful performance gains due to GPU acceleration.

We wish to clarify the novelty of this paper. It combines the main ideas of two prior papers, one that introduces the CPU-only distributed 3D algorithm [43] and one that considers manycore co-processor acceleration for the distributed 2D algorithm [44]. Indeed, as noted in Section 5, the 3D algorithm generalizes the Halo technique; and all results for the combined method, which appear in Section 6.7, are completely new. Relative to these past efforts, the present one serves as the "definitive" reference for a distributed 3D algorithm with manycore co-processor acceleration.

Table 1
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
Matrix & Indexing	A, L, U	The input matrix A and LU factors
	$L(k)$, $U(k)$	kth L and U panel
	$A(:, k)$, $A(k, :)$	$A(k:n, k)$ and $A(k, k+1:n)$: kth A panels
Graphs	G	The graph associated with sparse matrix A
	E	Elimination tree of A
	E_f	Elimination tree-forest (Section 3.3)
	S	Top level separator of G
	C_1, C_2	Children etrees of S
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

2. Background

This section briefly summarizes the relevant background on sparse direct solvers needed to understand our new 3D algorithm. The most important concepts include the elimination tree, which guides parallelism, as well as the baseline 2D algorithm, which underlies SuperLU_DIST (see Table 1).

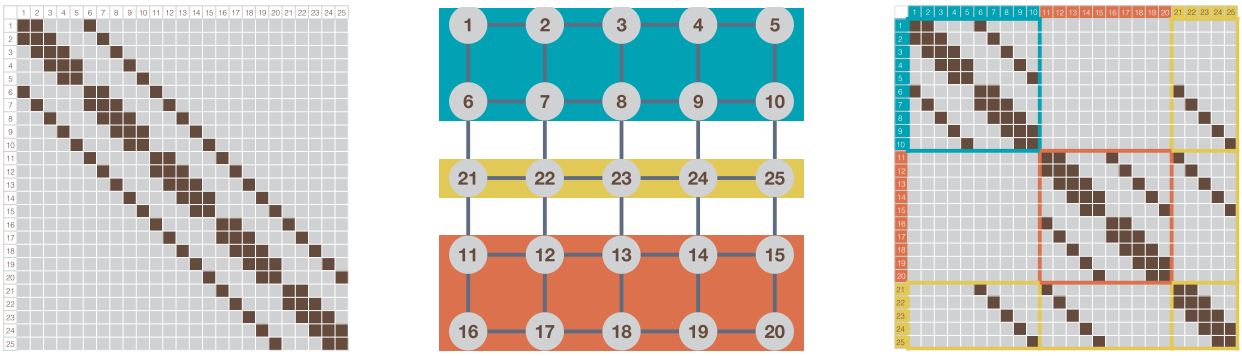
2.1. Introduction to sparse direct solvers

A sparse direct solver solves a system of linear equations $Ax = b$ 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 factored 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.

2.2. 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. In Fig. 1a, we show a penta-diagonal 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. Informally, a *good* separator is small and yields partitions C_1 and C_2 that are balanced, meaning approximately equal in size. In Fig. 1b, we highlight one such separator in yellow. 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, Fig. 1c shows one such ordering for the matrix shown in Fig. 1a. Fig. 2a shows a simplified block representation of the reordered



(a) A 25×25 sparse matrix

(b) The associated graph and separator

(c) Reordered sparse matrix using ND ordering

Fig. 1. A sparse matrix (Fig. 1a), its associated graph (Fig. 1b), and a separator (highlighted in yellow); and the re-ordered matrix (Fig. 1c) using nested dissection (ND) ordering. The ND orders the variables so that the variables corresponding to the separator are numbered last.

matrix Fig. 1c 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 and SCOTCH can compute ND partitions [10,28,41].

2.3. Sequential sparse LU factorization

Consider the LU factorization of the 3×3 block sparse matrix shown in Fig. 2a. Conceptually, one can compute the factors L and U by iterating along the diagonal of A, where at each iteration i one carries out the following three steps:

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

2.4. Dependency tree in sparse LU factorization

Factoring the diagonal blocks would proceed sequentially in the case of a dense A, but not so in the sparse case. In the 3×3 block sparse matrix example, block 1 (A_{11}) and block 2 (A_{22}) may be factorized independently and so in any order. But the factorization of blocks 1 and 2 requires updates to the same block 3 (A_{33}). Thus, factorization of block 3 must follow that of 1 and 2. This dependency structure is represented by the *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 Figs. 3a and 3b, we show a larger sparse matrix and its etree.

2.5. 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 and static pivoting [35]. In contrast to partial pivoting, static pivoting allows *a priori* determination of the non-zero structure of factored L and U matrix, that makes SuperLU_DIST more scalable. 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 [44,45].

2.5.1. 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 using a 2×2 process grid.

2.5.2. 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 them to all the processes to perform the Schur-complement update. It involves the following kernels:

1. *Diagonal factorization*: The process P_{kk}, which owns block A_{kk}, factorizes it into $L_{kk}U_{kk}$.
2. *Diagonal broadcast*: 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.

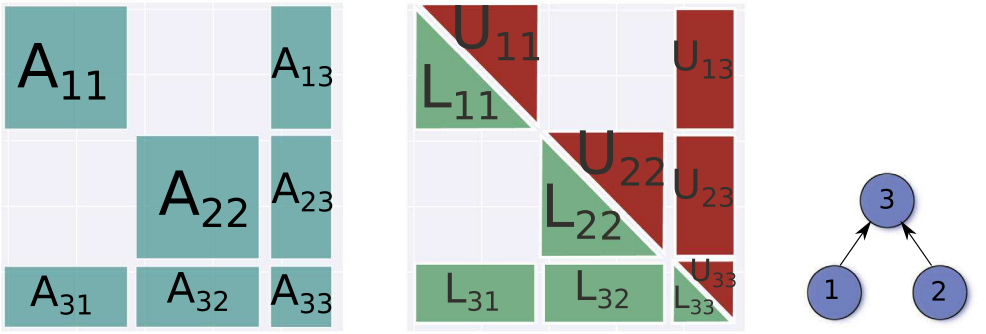
Qualitatively, the panel-factorization step is the main communication phase of the factorization. Panel-factorization involves data transfers and synchronizations, and it usually incurs a relatively small fraction of the total floating-point operations.

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

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

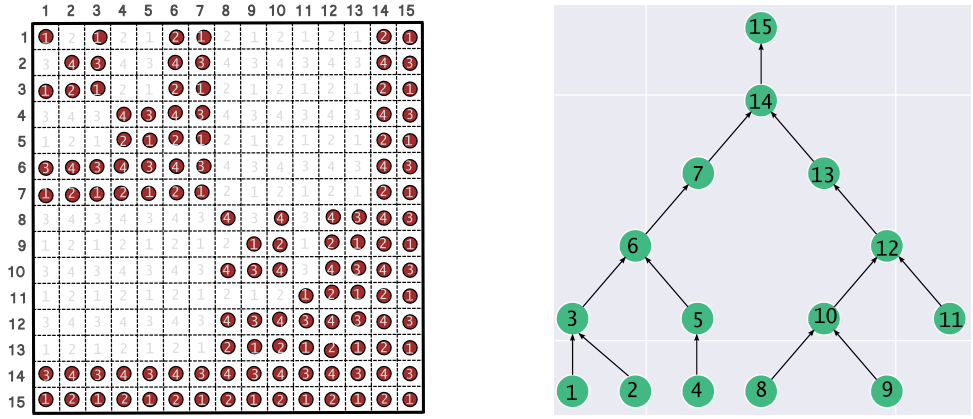
The blocks L_{ik} and U_{kj} are sparse. Therefore, to perform the above update, we first pack L_{ik} and U_{kj} into a dense BLAS-compliant format. Then, we use dense Level 3-BLAS routines to compute the product $V = -L_{ik}U_{kj}$. Finally we compute the mapping from V back to A_{ij} and update A_{ij}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



(a) A 3×3 block sparse matrix (b) The L and U factors (c) The elimination tree

Fig. 2. In Fig. 2a, we show the block sparse matrix A obtained from nested dissection ordering (e.g. the 25 × 25 sparse matrix shown in Fig. 1). 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}.



(a) (b)

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

In Fig. 4a, we show the regions of the matrix that participate in the different steps when we factorize the first supernode. Interested readers can find detailed pseudocode and descriptions of the inner workings of the algorithm elsewhere [11,34,36,37,44,49].

2.6. 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* [49]. In the bottom-up ordering of factorization of the etree, leaf nodes are factored first. Thus, the 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 the 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 performs the Schur-complement update of each supernode sequentially.

Furthermore, the larger the lookahead window, the more in-flight messages and incoming buffer space is required. Therefore, the lookahead window is fixed to lie in the range of 8–20 steps.

2.7. Limitations of 2D sparse LU

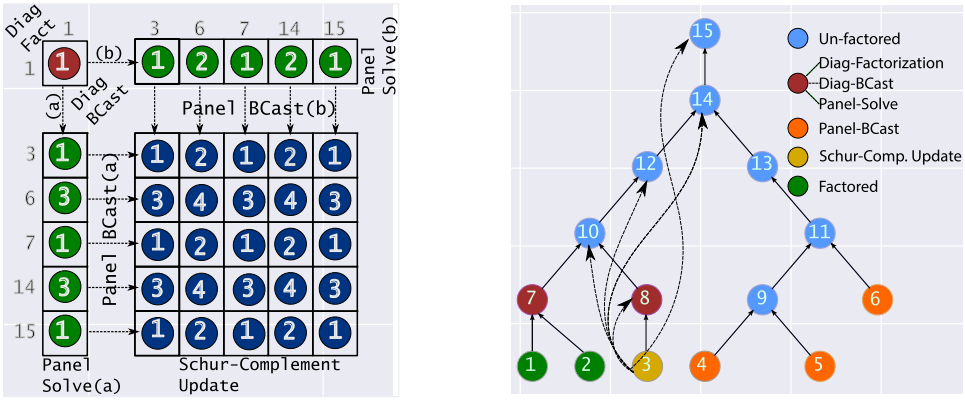
The 2D algorithm scales well up to a certain point, beyond which the cost of data transfer starts to dominate the cost of computation. Moreover, at a large number of processes, the effect of load imbalance becomes more prominent. In practice, after a certain number of processes, adding more processes 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. Therefore, the latency of various communication kernels cannot decrease with increasing numbers of processors.

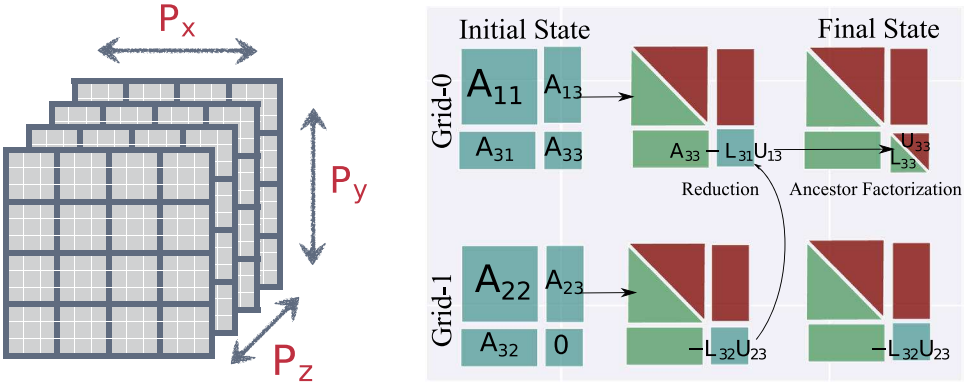
3. A 3D sparse LU factorization algorithm

Since the 2D algorithm uses an owner-computes update policy, the Schur-complement update on a given block A_{ij} must proceed sequentially. 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) (b)

Fig. 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.



(a) 3D process grid (b) The 3D algorithm with two 2D grids

Fig. 5. How 3D sparse *LU* works. Consider the 3D process grid as a collection of 2D subgrids (Fig. 5a). We show the working of the 3D sparse *LU* for a block sparse matrix A (Fig. 2a) using 2 process grids in Fig. 5b. The sparse blocks 1 and 2, and their panels, reside on grid 0 and grid 1, respectively. Block 3 is replicated in both 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).

3.1. The 3 × 3 block sparse case

One way is to use the structure of the etree to help 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, from its initial value A^0_{33}, the updated block A_{33} must accumulate from both blocks 1 and 2 according to

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

Suppose we replicate A_{33}, keeping two copies. The first copy accumulates $A^0{}_{33} - L_{31}U_{13}$ from the factorization of 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 factorizing it. Thus, the replication of A_{33} allows the 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. 6a). We partition A into $A^0 = A(C_1)\bigcup A(S)$ and $A^1 = A(C_2)\bigcup A(S)$ (Figs. 6c and 6e). We factorize 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 factorize 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 factorizes the updated copy of $A(S)$.

The two process grids only need to communicate once. In Section 4, we show that this communication accounts for a small fraction of the total. Furthermore, now each process factorizes a smaller number of supernodes, thereby potentially reducing latency.

3.2. General Case

Suppose we want to use *four* 2D grids, instead of two. We can divide the etree by one more level. For instance, in Fig. 7, 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[2] At the first level, we replicate node 1 on grids 0 and 1, and node 2 on grids 2 and 3. At the second level, all the nodes lie on only one grid.

Process grids 0 and 1 synchronize after they have factorized nodes 3 and 4, respectively. Then process grids 0 and 1, reduce

[2] Note that we index subtrees in the etree in the top-down order (Fig. 7). That is, the root subtree has an index zero and leaf subtrees have the largest indices. This is different from the indices of the blocks of the sparse matrix (e.g. Figs. 2 and 3 and 4b), where the indices correspond to an order of factorization.

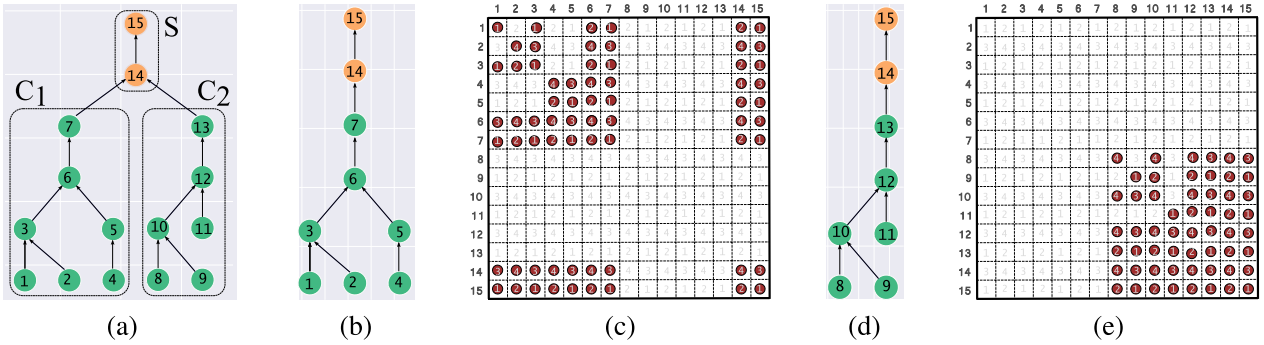


Fig. 6. Data distribution in 3D sparse LU algorithm. The elimination tree of the block sparse matrix in Fig. 3a is divided into common ancestor S and subtrees C_1 and C_2 as shown in Fig. 6a. The C_1 and C_2 subtree reside and are factored in process grid-0 and grid-1 respectively, whereas S is replicated in both the 2D grids. Figs. 6b and 6c show the local elimination tree and the data distributed in grid-0, respectively; and Figs. 6d and 6e show the same for grid-1.

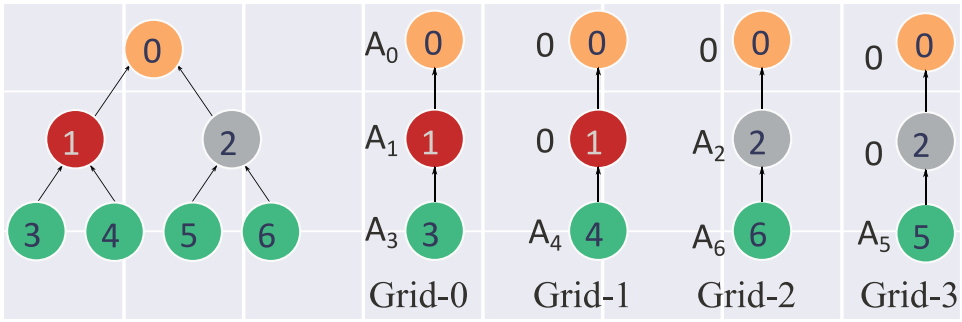


Fig. 7. A two-level partition of the elimination tree and its mapping into 4 process grids. Here $A_i = \{A_{ii} \bigcup A_{i,\, i+1:n} \bigcup A_{i+1:n,\, i}\}$, represents the set of three sub-matrices a) the diagonal block matrix A_{ii}; b) horizontal off-diagonal panel $A_{i,\, i+1:n}$; and c) the vertical off-diagonal panel $A_{i+1:n,\, i}$.

all the common ancestor nodes, namely nodes 1 and 0 on grid-0. Similarly, process grids 2 and 3 synchronize after they have factorized 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 factorize 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.

3.3. 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, consider the etree in Fig. 8. Suppose the cost of factorization of each node is known. Observe that this tree is unbalanced at the top level. The ND ordering (shown in Fig. 8 on the left) to partition the etree is sub-optimal. We show a better partition of the etree, obtained by dividing the subtrees another level, on the right of Fig. 8. This partition has a shorter 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.

One issue is that we do not know the exact cost of factorizing each node in the distributed fashion—partly due to its dependence on the process grid dimension, network parameters, sparsity pattern of the matrix, and distribution of block sizes, all of which can be difficult to estimate efficiently during runtime. Instead, we heuristically use the number of floating-point operations needed to factorize a given node for cost model $T(C)$, calculating which requires the dimension of the node and number of non-zero rows and columns in the node's L and U panel respectively. This estimation of the number of floating point operation in the factorization of a node does not take the parallel distribution of L and U panels, load balance, and sparsity into account, so it may not be very accurate in all cases, but performs well in most of the test matrices we tried.

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 $S = [1, 0]$, $C_1 = [4]$ and $C_2 = [3, 5, 6, 2]$.

3.4. The pseudocode of the 3D sparse LU factorization algorithm

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 2D subgrids in the "z-dimension" of the 3D process grid. And E_f is the etree-forest, the output of our load-balance heuristic. Each process subgrid only stores forests that reside on the grid, and each forest is stored as a list of nodes. The factorization progresses from leaves lvl $= l$ to the root lvl $= 0$. The two main subroutines invoked at any level are dSparseLU2D and Ancestor-Reduction.

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 3.3).
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
```

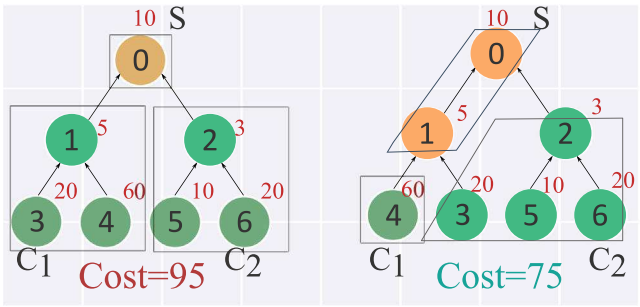


Fig. 8. Inter-grid load balancing: an unbalanced elimination tree with 2 ways of mapping, nested dissection and a greedy heuristic, and the cost of factorization in the critical path ($T(E) = T(S) + \max\{T(C_1),\ T(C_2)\}$). The cost of factorization of each node is shown in red. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

1. dSparseLU2D(A, nList): *My* process grid performs the 2*D* 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 the modified factorization routine of SuperLU_DIST.
2. *Ancestor-Reduction*: After the factorization of level-i, we reduce the nodes of the ancestor matrix before factorizing the next level. In the ith level's reduction, the receiver is the $k2^{l-i+1}$th process grid and the sender is the $(2k+1)2^{l-i}$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 pair-wise and takes places along the z-axis in the 3D process grid.

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

4. 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 the analytical expressions of performance on certain model problems, and thereby gain some insight into the algorithm's behavior, including a precise characterization of the memory-communication tradeoff.

Our analysis considers two types of input matrices. The first is associated with planar graphs, such as those arising from discretizing partial differential equations (PDEs) on 2D domains. The second type is associated with those from 3D PDEs, which have a "well-shaped" geometry but are non-planar.

Below, we derive the expressions specifically for memory use, communication volume, and number of messages (message latency) for the baseline SuperLU_DIST algorithm when using a 2D process grid, given a general matrix. Then, we give the 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' geometries and try to reserve "2D" and "3D" for referencing the logical organization of processes in the algorithms.

4.1. 2d sparse LU with a generic sparse matrix

Consider the factorization of a sparse matrix A of dimension n and its etree 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 supernodes in the level-i have the dimension n_i. The ith level has 2^i nodes.

4.1.1. 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

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

4.1.2. 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 $\mathcal{O}\left(n^2/\sqrt{P}\right)$ [46].[3]

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

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

4.1.3. 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.,

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

4.2. 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 [38]. This result also holds for other classes of graphs, like graphs with bounded genus and graphs with excluded minors [4,16].

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 ith 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 $\sim \log n$.

4.2.1. Per-process memory

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

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

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 has a size $\sqrt{n}\times\sqrt{n}$, and it is replicated on all the $P_z = 2^l$ 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 at each level. Altogether, per-process memory required can be written as:

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

4.2.2. Per-process communication

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

$$W_{2D} = \frac{n\log n}{\sqrt{P}}. \tag{6}$$

3d algorithm. We separately calculate the communication in the *SuperLU_DIST2D* step, denoted as W_{3D}^{xy}, and the communication in the Ancestor-Reduction step, denoted as W_{3D}^{z} in Algorithm 1.[4]

Per-process communication in factorization (W_{3D}^{xy}): For the factorization of supernodes in the etree level $i \geq l$, each process grid works on $1/2^l$ fraction of the submatrix of 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.[5] 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

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

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

$$P_z = \frac{1}{2}\log n. \tag{8}$$

Thus, the minimum amount of communication is

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

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 ith level is $\frac{(l-i)\cdot n}{2^i P_{xy}}$. We sum this expression over all i to get

[3] The network topology and the underlying MPI implementation may increase the asymptotic complexity if, for instance, communication increases with some function of P.

[4] All communications in the *SuperLU_DIST2D* occur in the *xy* plane.

[5] Note that the 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 on the critical path of the factorization.

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

Parameter	2D PDE		
	dSparseLU2D	dSparseLU3D	dSparseLU3D $P_z = \mathcal{O}(\log n)$
Memory per process (M)	$\mathcal{O}\left(\frac{n}{P}\log n\right)$	$\mathcal{O}\left(\frac{n}{P}(\log(\frac{n}{P_z}) + P_z)\right)$	$\mathcal{O}\left(\frac{n}{P}\log n\right)$
Communication per process (W)[b]	$\mathcal{O}\left(\frac{n}{\sqrt{P}}\log n\right)$	$\mathcal{O}\left(\frac{n}{\sqrt{P}}(2\sqrt{P_z} + \frac{\log n}{\sqrt{P_z}}) + \frac{2nP_z \log P_z}{P}\right)$	$\mathcal{O}\left(\frac{n}{\sqrt{P}}\sqrt{\log n}\right)$[a]
Latency	$\mathcal{O}(n)$	$\mathcal{O}\left(\frac{n}{P_z} + \sqrt{n}\right)$	$\mathcal{O}\left(\frac{n}{\log n}\right)$

[a]When $P \gg \log n$.

[b]On the critical path of factorization. The 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 the same and equal to $\mathcal{O}\left(\frac{n}{\sqrt{P_{xy}}}\right) = \mathcal{O}\left(\frac{n}{\sqrt{P}}\sqrt{\log n}\right)$.

per-process communication in the ancestor-reduction step along the critical path of 3D sparse LU, which is

$$W^z_{3D}(n,\ P,\ P_z) = \sum_{i=0}^{l-1} \frac{n(l-i)}{2^i P_{xy}} \approx \frac{n}{P_{xy}}(2l - \frac{1}{4}) \tag{10}$$

$$\approx \frac{2nl}{P_{xy}} = 2n\frac{P_z \log P_z}{P} \tag{11}$$

Total per-process communication on the critical path: The total per-process communication is $W_{3D} = W^z_{3D} + W^{xy}_{3D}$. From Eqs. (7) and (11),

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

When we choose P_z by Eq. (8), this becomes

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

For any practical n, $P \gg \log n$ even for modest values of P. Thus, for fixed n the first term of Eq. (12) dominates.

4.2.3. Latency

Latency for the 2D algorithm is $\mathcal{O}(n)$. Since the elimination of a single node amounts to a $\mathcal{O}(1)$ latency cost, hence the total latency cost of the 3D algorithm is the total number of nodes in the critical path of the factorization. When $l \ll \log n$, each subtree contains approximately $\frac{n}{P_z}$ nodes. On the other hand, any subtree at $i < l$ level will have $\sqrt{\frac{n}{2^i}}$ nodes. Therefore, the total latency cost of the factorization using the 3D algorithm will be:

$$L_{3D}(n, P_z) = \frac{n}{P_z} + \sum_{i=0}^{l-1}\sqrt{\frac{n}{2^i}} = \mathcal{O}\left(\frac{n}{P_z} + \sqrt{n}\right). \tag{13}$$

The second term of expression in Eq. (13) follows from $\sum_{i=0}^{l-1}\sqrt{\frac{n}{2^i}} < \sum_{i=0}^{\infty}\sqrt{\frac{n}{2^i}} = \sqrt{n}(2+\sqrt{2})$. When $P_z = \mathcal{O}(\log n)$, L_{3D} is a $\log n$ factor smaller than L_{2D}.

4.3. Non-planar input graphs

For non-planar sparse matrices, we give the expressions for memory, communication, and latency in Table 3. These expressions are derived from simplifying Eqs. (14) to (17), (20) and (21) using a symbolic-algebra package SymPy [40].

4.3.1. Per-process memory

The memory for the 2D and the 3D algorithm for the non-planar case is given by the following expressions:

$$M_{2D} = \sum_{i=0}^{\log n}\frac{2^i n_i^2}{P};\ \text{and} \tag{14}$$

$$M_{3D} = \sum_{i=0}^{l-1}\frac{n_i^2}{P_{xy}} + \sum_{i=l}^{\log n}\frac{2^i n_i^2}{P}; \tag{15}$$

where $n_i = \left(\frac{n}{2^i}\right)^{2/3}$ is the size of separator at the i-the level and $l = \log P_z$. We give the final simplified forms for M_{2D} and M_{3D} in Table 3. The dimension of the top-level separator is $n_0 = n^{2/3}$, and it requires storage of size $n_0^2 = n^{4/3}$. Asymptotically, the size of all the LU factors of the matrix is also $\mathcal{O}\left(n^{4/3}\right)$. Since the top level separator is replicated on all process grids, the asymptotic storage requirement for the 3D algorithm increases linearly with P_z.

4.3.2. Per-process communication

Similarly, communication in the factorization of the top-level separator is asymptotically equal to the communication for the total LU factorization. So the 3D algorithm cannot reduce the asymptotic complexity of the algorithm. The total communication W_{3D} is the sum of communication in the xy grid W^{xy}_{3D}, and the communication along the z-dimension W^z_{3D}

$$W_{3D} = W^{xy}_{3D} + W^z_{3D} \tag{16}$$

The expression for W^{xy}_{3D} is given by:

$$W^{xy}_{3D} = \sum_{i=0}^{l-1}\frac{n_i^2}{\sqrt{P_{xy}}} + \sum_{i=l}^{\log n}\frac{2^{i-l}n_i^2}{\sqrt{P_{xy}}}, \tag{17}$$

where $n_i = \left(\frac{n}{2^i}\right)^{2/3}$. The first term in the expression for W^{xy}_{3D} denotes the communication in the non-leaf subtrees i.e., from level 0 to $l-1$; and the second term denotes the communication in a leaf subtree.

The expression for W^z_{3D} is given by:

$$W^z_{3D} = \sum_{j=0}^{l-1}\sum_{i=0}^{j}\frac{n_i^2}{P_{xy}} \tag{18}$$

We combined and simplified Eqs. (16)–(18) and arrived at the following expression for W_{3D}:

$$W_{3D} = W_{2D}\left(\kappa_1 P_z^{1/2} + \frac{1-\kappa_1}{P_z^{5/6}}\right) + W_{2D}\left(\kappa_1\frac{P_z \log P_z}{\sqrt{P}}\right), \tag{19}$$

where $\kappa_1 = \frac{2(\alpha-1)}{\alpha^4-1}$, where $\alpha = 2^{1/3}$.

Consider the expression for W^{xy}_{3D}, which is the dominant term in W_{3D} at large P. To find the P_z that minimizes W^{xy}_{3D}, we should have $\frac{dW^{xy}_{3D}}{dP_z} = 0$. This occurs for $P_z = \left(\frac{5(1-\kappa_1)}{3\kappa_1}\right)^{3/4} \approx 2.4$. Hence,

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

Parameter	3D PDE	
	dSparseLU2D	dSparseLU3D
Memory per process (M)	$\mathcal{O}\left(\frac{n^{\frac{4}{3}}}{P}\right)$	$\mathcal{O}\left(M_{2D}\left(P_z + \frac{1}{P_z^{4/3}}\right)\right)$
Communication per process (W)	$\mathcal{O}\left(\frac{n^{4/3}}{\sqrt{P}}\right)$	$\mathcal{O}\left(W_{2D}\left(\kappa_1\sqrt{P_z} + \frac{1-\kappa_1}{P_z^{5/6}}\right) + W_{2D}\left(\kappa_1 \frac{P_z \log P_z}{\sqrt{P}}\right)\right)$ [a]
Latency	$\mathcal{O}(n)$	$\mathcal{O}\left(L_{2D}\left(\kappa_2 n^{-1/3} + \frac{1}{P_z}\right)\right)$ [b]

[a] The first and second term in the expression for W_{3D} denotes W_{3D}^{xy} and W_{3D}^{z}, respectively. $\kappa_1 = \frac{2(\alpha-1)}{\alpha^4-1}$, where $\alpha = 2^{1/3}$.
[b] $\kappa_2 = \frac{\alpha}{\alpha+1}$.

in the case of non-planar graphs, the 3D algorithm can use $\mathcal{O}(1)$ processes in the z-dimension and the reduction in per-process communication volume is also $\mathcal{O}(1)$. In practice, we see that 3D algorithm can reduce communication volume for up to $P_z = 16$ (see Section 6.4).

4.3.3. Latency

The latency for the 2D and the 3D algorithm for the non-planar case is given by the following expressions:

$$L_{2D} = \sum_{i=0}^{\log n} 2^i n_i; \text{ and} \tag{20}$$

$$L_{3D} = \sum_{i=0}^{l-1} n_i + \sum_{i=l}^{\log n} 2^{i-l} n_i. \tag{21}$$

We give the final simplified forms for L_{2D} and L_{3D} in Table 3.

5. The 3D sparse LU factorization on hybrid clusters

To further improve single node performance, we consider acceleration by manycore co-processors (e.g., GPUs), as we had done in prior work to accelerate SuperLU_DIST [44,45]. In particular, we had previously discussed how to accelerate sparse LU factorization by offloading dense BLAS-3 calls to the GPU [45]; and, later, proposed a novel offload scheme called the Halo algorithm [44], which went beyond BLAS-3 call offload to include Scatter computations. Here, we extend the 3D sparse LU factorization algorithm of Section 3 for heterogeneous clusters by combining it with Halo.

Indeed, the original Halo algorithm may be regarded as an instance of the 3D algorithm that uses data replication between the host and co-processor, thereby reducing communication between them. However, a critical difference is that the Halo algorithm offloads only the Schur-complement update computation to the co-processor. So the addition of co-processor offload using the Halo algorithm improves the factorization performance when the local Schur-complement update computation dominates the factorization cost.

5.1. Halo algorithm

We begin by briefly discussing the Schur-complement update step and the Halo algorithm for offloading the Schur-complement update to the co-processors. Recall that in the kth Schur-complement update, each block A_{ij} is updated as by

$$A_{ij} \leftarrow A_{ij} - L_{ik}U_{kj},$$

where A_{ij}, L_{ik} and U_{kj} are sparse blocks. To perform the update, we first pack L_{ik} and U_{kj} into dense BLAS-compliant format and then call dense matrix-multiplication subroutine (matrix-matrix multiply (Gemm)) to compute the product $V_{ij} = L_{ik}U_{kj}$. We then compute a map from the dense V_{ij} block back to the sparse destination block A_{ij}, performing an element-wise update referred to as a Scatter step. A Scatter often involves a lot of index arithmetic and indirect memory accesses. Typically, Gemm and Scatter can be both equally costly in their execution times.

We had previously proposed a scheme that only involved offloading of Gemm to an attached GPU [45]. Such an approach is a natural first step towards adding GPU acceleration to sparse direct solver. Indeed, it mirrors much of the existing work, which – until our GPU cluster work – considered only the *single-node* case [15,31,39,48,50].

This approach had two main limitations. First, when Gemm is offloaded to the GPU, the Scatter computation becomes the performance bottleneck. This fact results in a maximum performance gain from GPU acceleration of about 2× for many realistic sparse matrices. The second limitation is that, in this approach, we need to send a large dense matrix from GPU to the host CPU in each iteration via Peripheral Component Interface Express (PCIe). We used software pipelining to hide such PCIe transfer costs. However, in newer architectures there is a larger bandwidth gap between a host's DRAM and PCIe, meaning it becomes harder to hide a large PCIe transfer cost.

To overcome these limitations, our Halo algorithm also offloads the Scatter computation to the co-processor and, furthermore, significantly reduces the PCIe transfer between the host and the co-processor.

The Halo scheme works as follows; for details, refer to the original description thereof [44]. It maintains an independent copy of the sparse matrix A on the co-processor. Denote this copy by A_ϕ. At the beginning of the factorization, A_ϕ is initialized with zeros. In each iteration k, we divide the Schur-complement update computation between the host CPU and the co-processor. To do so, we send the kth L and U panels from the host CPU to the co-processor. We partition the Schur-complement update computation between the CPU and the co-processor along U, as shown in Fig. 9. The host and co-processor update their respective parts of the Schur-complement matrix. Again, this behavior is illustrated by Fig. 9.

In the Halo algorithm, the host sends the $L(k)$ and $U(k)$ panels to the co-processor in the kth iteration and receives $(k+1)$st A panels (Fig. 9). In the BLAS offload method, we need to send the $L(k)$ and $U(k)$ panels to the co-processor like in the Halo algorithm, but the co-processor sends a large product matrix $V = L(k)U(k)$ to the host. The V matrix is typically much bigger than $(k+1)$st A panels. Hence, the Halo algorithm does significantly less data transfer than the BLAS offload method.

We use a model-driven load balance heuristic to achieve a good load balance between the host CPU and the co-processor.

5.2. Combining Halo algorithm with 3D sparse LU

Our Halo algorithm on top of the 3D works as follows.

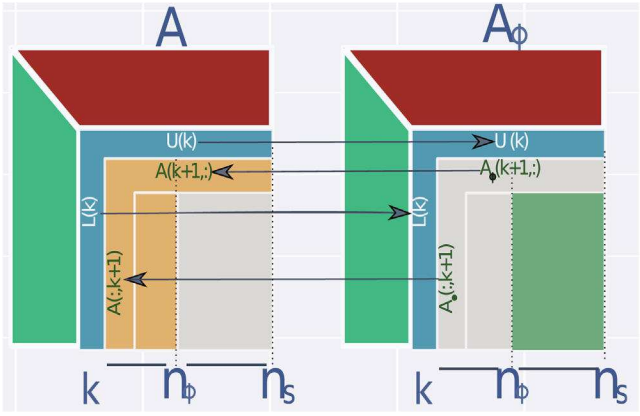


Fig. 9. HALO: Schur-complement update in the kth iteration. The $L(k)$ and $U(k)$ panels – calculated in the kth panel-factorization on the CPU – are sent to the co-processor. The co-processor sends $(k+1)$st A-panels to the CPU. The CPU and co-processor update parts of the kth Schur-complement, shown in yellow for the CPU and in green for co-processor. The CPU merges the received co-processor's $(k+1)$st A-panels with its own $(k+1)$st A-panels, before $(k+1)$st iteration starts . (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

Before factorization begins, we initialize to zero the sparse data structure A_ϕ on the co-processor, corresponding to all the local blocks of the 2D processes grid. If the co-processor does not have enough memory to hold all the blocks, we keep only the blocks corresponding to the top portion of the elimination tree-forest E_f. In other words, we traverse E_f in breadth-first fashion starting from the root, and include in A_ϕ all the top portion that the co-processor can accommodate.

The copy of A_ϕ remains persistent throughout the factorization. When factorizing a node, if the Schur-complement update may offload some update computation to the co-processor, it sends the corresponding L and U panels to the co-processor, and the co-processor performs those updates on A_ϕ. Before panel factorization of a node, if there are any updates accumulated in A_ϕ, the co-processor sends the corresponding blocks back to the host CPU. Then we reduce the A_ϕ and A_{cpu} before performing panel factorization on A_{cpu}.

Finally, in the Ancestor-Reduction step, if a process is the sender, it first collects all the ancestor blocks in A_ϕ before sending it to the receiving process.

6. Experimental results

We evaluate 3D sparse LU against the baseline 2D algorithm, first for the CPU-only case (Sections 6.2 to 6.6) and then when adding GPUs (Section 6.7). 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.

6.1. Setup

We use SUPERLU_DIST's default parameters in our experiments. For our CPU-only runs, we rely primarily on the "Edison" cluster at NERSC.[6] 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.

For our GPU experiments, we use the "Titan" Cray XK7 system at OLCF.[7] Each node in our GPU-testbed consists of a 16-core 2.2 GHz AMD Opteron 6274 "Interlagos" processor (8 cores with hyperthreading disabled) and a Nvidia K20X "Kepler" accelerator connected via PCI express 2.0 interface. The host multicore has 32 GB of DDR3 memory divided into two NUMA nodes and the K20X has 6 GB of DDR5 memory. On each node, we run 2 MPI processes, one on each NUMA node, and each MPI process spawns 4 OpenMP threads. The two MPI processes on a node share the on-node GPU.

Table 4
Test sparse matrices used in experiments.

Name	Application	n	$\frac{nnz}{n}$	#Flop[a]	T_{fact}[b]
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

[a]# of floating point operations in the baseline SUPERLU_DIST (dSparseLU2D)
[b]Factorization time in seconds for the baseline on 16 nodes.

6.1.1. Test matrices

We used four planar and six non-planar matrices, summarized in Table 4. 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 s on 16 nodes when using the baseline 2D SUPERLU_DIST.

6.2. Results 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. 10, 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 the number of synchronization point as the 2D algorithm. Some 3D matrices also achieve better performance when P_z is large

6 http://www.nersc.gov/users/computational-systems/edison/.

7 https://www.olcf.ornl.gov/olcf-resources/compute-systems/titan/.

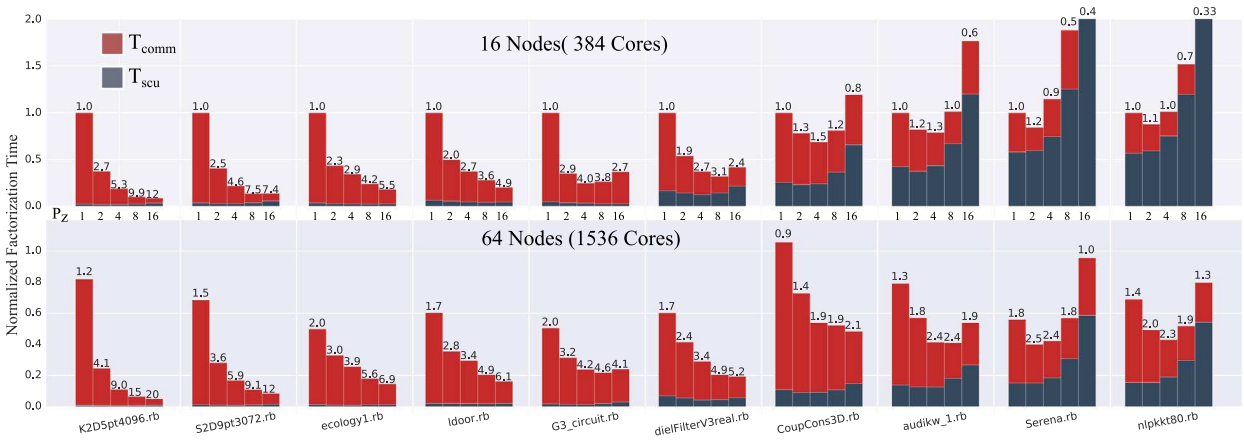


Fig. 10. Performance comparison of various $P_x \times P_y \times P_z$ grids on 16 nodes (upper plot) and 64 nodes (lower) on the Edison system at NERSC. For each matrix, each column represents a different values of $P_z \in \{1, 2, 4, 8, 16\}$ from left to right. Thus, the leftmost column is the 2*D* algorithm, and when moving right, the 2*D* grids become smaller as P_z increases. For each data set, the time shown is normalized with respect to 2*D* SuperLU_DIST on 16 or 64 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.

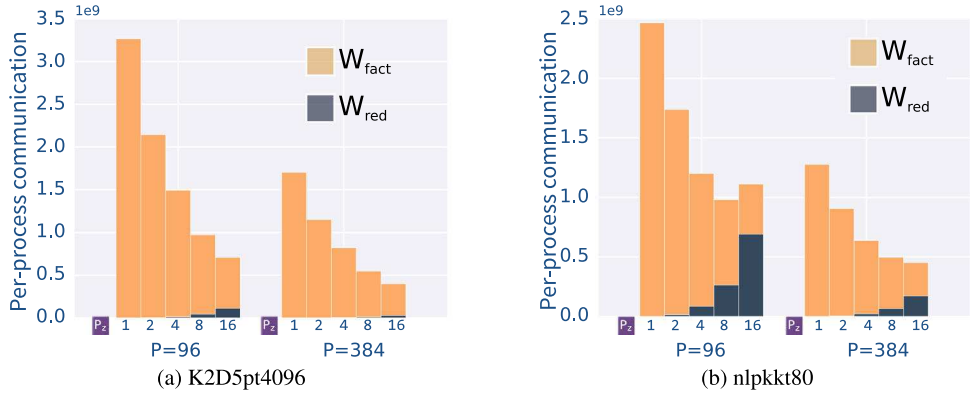


Fig. 11. Per-process communication volume (in bytes) for different process grid configurations on 16 and 64 nodes (or 96 and 384 MPI processes, respectively). Each column in a group represents a $P_{xy} \times P_z$ process grid configuration, where $P_z \in 1, 2, 4, 8, 16$ from left to right (leftmost being a purely 2*D* configuration). Here, W_{fact} is the number of words sent during the local factorization along the 2*D* grid, whereas W_{red} is the number of words sent in the ancestor-reduction step along the z-axis.

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 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.

6.3. 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 speedup of 1.7 and 1.9×, respectively.

6.4. Effects on per-process communication

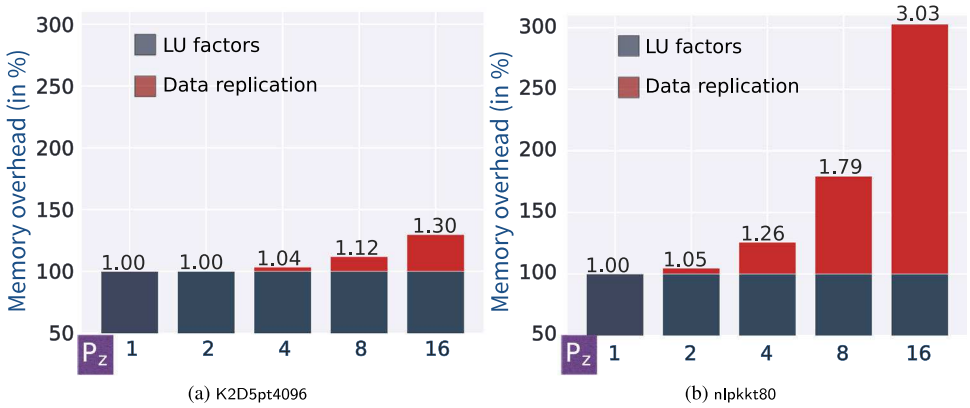
For the planar graphs, the 3D algorithm reduces per-process communication by 3–4.6× on 16 nodes and by 4–4.7× on 64 nodes. For non-planar graphs, the 3D algorithm reduces per-process communication by 2.5–3.2× on 16 nodes and by 2.9–3.7× on 64 nodes. Fig. 11 shows the per-process communication volume 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. This is because W^{red} increases almost linearly with P_z. Yet for planar graphs, this increase is not 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.

6.5. Memory overhead

The 3D algorithm needs 30% more memory for the planar graph K2D5pt4096 and 200% more for the non-planar graph nlpkkt80,



(a) K2D5pt4096 (b) nlpkkt80

Fig. 12. The relative memory overhead of 3D sparse LU algorithm over 2D (in percent).

at $P_z = 16$ (see Fig. 12). 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 (small) separators. Therefore, the memory overhead increases quickly. At $P_z = 16$, nlpkkt80 already needs 200% more memory. Overall at $P_z = 16$, memory overhead ranged between 18% to 245% for all matrices we tried.

6.6. Performance at large numbers of cores

The best case speedup for a matrix is the speedup of the best $P_{xy} \times P_z$ configuration relative to the best possible 2D process configuration. The best case speedup is 5–27.4× for the planar graphs and 2.1–3.3× for non-planar graphs. We show a heat-map of performance for K2D5pt4096 and nlpkkt80 in Fig. 13 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 matrices achieve the best performance on a 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 $P_z = P_{xy}/24(\nearrow)$ for a constant $P = P_{xy} \times P_z$. All the other matrices achieved the best performance between the two lines.[8] In the best case, we achieved 27.4× speedup for graph K2D5pt4096. And on average the best 3D configuration was 6.5× faster than the best 2D configuration among all the matrices.

6.7. Performance of GPU accelerated 3D sparse LU factorization

Lastly, we present the results of adding the HALO-based co-processor acceleration method to the 3D sparse LU factorization algorithm.

6.7.1. Impact of GPU acceleration on the 3D configurations using 32 nodes of cray XK7

In Fig. 14, we show the performance of the 3D algorithm for dielfilterv3real and nlpkkt80 with and without GPU acceleration on 32 nodes of Cray XK7. For this experiment, we omit the extremely planar cases such as K2D5pt4096, since Schur-complement update is a very small fraction of the total factorization cost. This fact can be easily determined *a priori*, and for such matrices, GPU acceleration will not yield any tangible performance improvement.

In the case of matrix dielfilterv3real, the communication cost dominates in the baseline 2D algorithm. Thus, adding GPU acceleration to the 2D algorithm results in only 1.1× improvement. As we go towards the 3D configurations, time spent in communication decreases and thus overall time decreases and it makes T_{scu} again dominant. When T_{scu} is dominant, adding GPU acceleration improves the performance further. Thus, at $P_z = 16$, GPU acceleration improves the performance by an additional 1.5×.

In case of the matrix nlpkkt80, GPU acceleration improves the performance by 1.7× for the baseline 2D algorithm. For 3D configurations, the cost of Schur-complement update increases linearly with P_z, thus eclipsing any benefit from reduced communication. Adding GPU acceleration reduces the relative cost of Schur-complement update; thus, we see an improvement of 1.8× by using 3D configuration with GPU acceleration with respect to the baseline 2D algorithm with GPU acceleration.

6.7.2. Strong scaling

In Fig. 15a, we show the performance (in TFLOP/s) of GPU accelerated 3D sparse LU factorization algorithm for nlpkkt80. The GPU accelerated 3D algorithm achieves 2.6× speedup over the best case GPU accelerated 2D configuration. The variation of performance of GPU accelerated 3D sparse LU follows a similar pattern as for the unaccelerated case (Fig. 13).

In Fig. 15a, we show the speedup of the GPU accelerated 3D sparse LU over the CPU-only 3D sparse LU on the Cray XK7. Observe that when P_{xy} increases, the speedup obtained due to GPU-acceleration decreases. At smaller 2D grids, the per iteration cost of the Schur-complement update computation is relatively large; thus we can offload a significant fraction of Schur-complement update to the GPU. Therefore, at smaller 2D grids, we obtain a speedup of up to 3.5× with GPU acceleration, and at the largest 2D grid size, we do not see any advantage from GPU acceleration. The 3D algorithm allows us to use a large number of processes while keeping the 2D grid size small. Therefore, the 3D algorithm can use a relatively larger number of GPU accelerated nodes effectively. In short, GPU acceleration using the HALO algorithm and the 3D algorithm behave synergistically, improving the performance for a wider range of matrices with respect to the baseline or when either is used standalone.

7. Related work

The idea of using data replication to reduce communication in LU factorization goes back to Ashcraft, who described the first

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

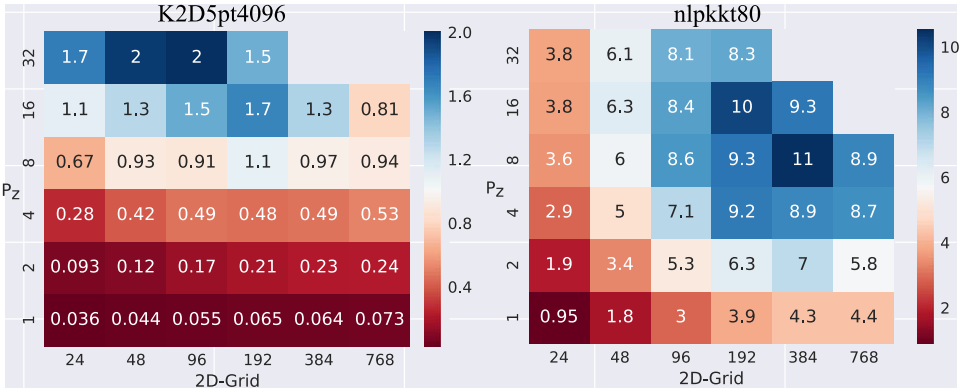
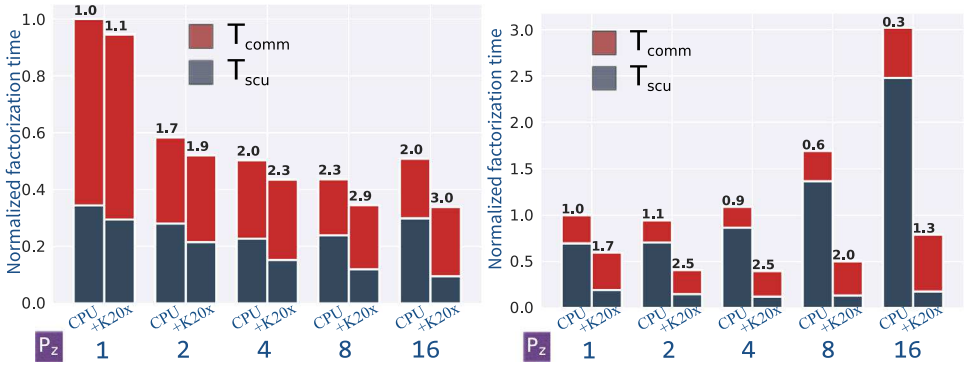
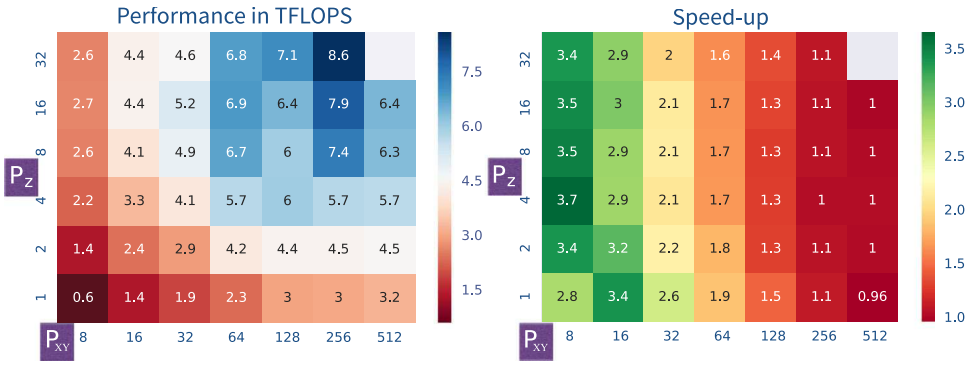


Fig. 13. The performance (in TFLOP/s) of the 3D algorithm for different $P_{xy} \times P_z$ combinations. Here, we increase P_{xy} and P_z by a factor of 2 along the x and y-axis respectively. Thus, bottommost row ($P_z = 1$) is the 2D algorithm. The performance is calculated using the number of floating point operations in baseline 2D factorization (Table 4). For K2D5pt4096, our new 3D code achieves speedups up to 27x.



(a) dielFilterV3real (b) nlpkkt80

Fig. 14. The normalized factorization time for CPU-only ("*CPU*") and GPU accelerated ("+*K*20*x*") variants of the 3D algorithm for different $P_z \in 1, 2, 4, 8, 16$ on 32 nodes of XK7. The factorization time is normalized with respect to unaccelerated 2D version ($P_z = 1$).



(a) GPU accelerated 3D sparse LU (b) Speedup over CPU based 3D sparse LU

Fig. 15. Fig. 15a shows the TFLOP/s performance rate of GPU accelerated 3D sparse LU algorithm on the Titan Cray XK7 cluster for different $P_{xy} \times P_z$ combinations. The largest configuration uses 4096 XK7 nodes with total 32,768 cores and 4096 Nvidia K20x GPUs. Fig. 15b shows the speedup of GPU accelerated 3D sparse LU algorithm over the baseline CPU based 3D algorithm for different $P_{xy} \times P_z$ combinations .

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 [24] and Solomonik and Demmel [46] also described LU factorization algorithms using logical 3D partitionings of MPI processes.

The central idea of all work above and ours is the 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 [23] 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 opposed to $\mathcal{O}\left(\frac{n\sqrt{\log n}}{\sqrt{P}}\right)$ in our case.[9] However, their approach can only use $\mathcal{O}(\log n)$ processes for planar problems as opposed to $\mathcal{O}(n \log n)$ processes in 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. [19]. 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 2). 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 is 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 scenario, the two algorithms have the same $WM^{1/2} = \mathcal{O}\left(\frac{n^{3/2} \log n}{P}\right)$. For matrix multiplication-like dense linear algebra algorithms, it is known that [9,25,27,46]

$$W = \Omega\left(\frac{\#\text{ Arithmetic Operations}}{\sqrt{M}}\right). \tag{22}$$

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. (22) 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, it is likely that Eq. (22) is not the same for sparse LU factorization algorithm: dense computations perform $\mathcal{O}\left(n^3\right)$ operations on $\mathcal{O}\left(n^2\right)$ data, whereas sparse LU factorization of planar graphs performs $\mathcal{O}\left(n^{3/2}\right)$ operations on $\mathcal{O}(n \log n)$ data. Establishing similar lower bounds for sparse LU factorization as Eq. (22) 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 [1,14]. 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 algorithms 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 [20,42].

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 [49].

[9] We get the same expression if we substitute $P = P_z$ in Eq. (11).

Researchers have also proposed communication-avoiding pivoting strategies to make LU factorization more scalable [7,18,29]. Since SuperLU_DIST uses static pivoting with iterative refinement, these techniques are not needed.

Among sparse direct solvers, prior work has studied efficient scheduling [2,3,17,26,30,33,49]. To improve the overlap of communication and computation, efficient lookahead techniques are part of state-of-practice for both dense and sparse direct solvers [8,47,49]. Lacoste [32] and Hugo [22] have also addressed memory and compute resource management for scaling multifrontal sparse direct solvers. The baseline SuperLU_DIST incorporates similar techniques.

8. Conclusions and future work

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

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 currently only use a subset of 2D grids. Alternatively, for those levels, we could merge two 2D grids to make a larger 2D grid for factoring denser blocks. However, doing so would require significant changes to the current data structure implementation in SuperLU_DIST. Consequently, we have deferred this idea to future work.

Perhaps one of the more interesting theoretical open problems is how to reason about lower bounds on communication in the sparse case. They are likely to require different analysis techniques from the dense case. The fundamental reasons stem from the discussion in Section 7 with respect to Eq. (22), as well as the intriguing fact that dense LU methods require trading higher latency for reduced communication, whereas certain sparse patterns, like planar graphs, do not (Section 4).

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 (DOE) Office of Science and the National Nuclear Security Administration (NNSA). It 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, as well as those of the Oak Ridge Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC05-00OR22725. Lastly, portions of this material are based upon work supported by the U.S. National Science Foundation (NSF) Award Numbers 1710371. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect those of DOE, NNSA, NERSC, or the NSF.

References

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

[2] E. Agullo, A. Buttari, A. Guermouche, F. Lopez, Implementing multifrontal sparse solvers for multicore architectures with sequential task flow runtime systems, ACM Trans. Math. Softw. (TOMS) 43 (2) (2016) 13.

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

[4] N. Alon, P. Seymour, 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, ACM, 1990, pp. 293–299.

[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, Springer, 1993, pp. 159–190.

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

[8] M. Bach, M. Kretz, V. Lindenstruth, D. Rohr, Optimized HPL for AMD gpu and multi-core CPU usage, Comput. Sci.-R&D 26 (3–4) (2011) 153–164.

[9] G.M. Ballard, Avoiding communication in dense linear algebra (PhD thesis) 2013 URL http://www2.eecs.berkeley.edu/Pubs/TechRpts/2013/EECS-2013-151.html.

[10] C. Chevalier, F. Pellegrini, PT-SCOTCH: A tool for efficient parallel graph ordering, Parallel Comput. 34 (6–8) (2008) 318–331.

[11] J.W. Demmel, S.C. Eisenstat, J.R. Gilbert, X.S. Li, J.W. Liu, A supernodal approach to sparse partial pivoting, SIAM J. Matrix Anal. Appl. 20 (3) (1999) 720–755.

[12] J.W. Demmel, L. Grigori, M. Gu, H. Xiang, Communication avoiding rank revealing QR factorization with column pivoting, SIAM J. Matrix Anal. Appl. 36 (1) (2015) 55–89.

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

[14] L. Eyraud-Dubois, L. Marchal, O. Sinnen, F. Vivien, Parallel scheduling of task trees with limited memory, ACM Trans. Parallel Comput. 2 (2) (2015) 13.

[15] T. George, V. Saxena, A. Gupta, A. Singh, A. Choudjury, Multifrontal factorization of sparse SPD matrices on GPUs, in: Proc. of IEEE International Parallel and Distributed Processing Symposium (IPDPS 2011), Anchorage, Alaska, 2011.

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

[17] D. Goik, K. Jopek, M. Paszyński, A. Lenharth, D. Nguyen, K. Pingali, Graph grammar based multi-thread multi-frontal direct solver with galois scheduler, Procedia Comput. Sci. 29 (2014) 960–969.

[18] L. Grigori, J.W. Demmel, H. Xiang, CALU: a communication optimal LU factorization algorithm, SIAM J. Matrix Anal. Appl. 32 (4) (2011) 1317–1350.

[19] A. Gupta, G. Karypis, V. Kumar, Highly scalable parallel algorithms for sparse matrix factorization, IEEE Trans. Parallel Distrib. Syst. 8 (5) (1997) 502–520.

[20] M.T. Heath, E. Ng, B.W. Peyton, Parallel algorithms for sparse linear systems, SIAM Rev. 33 (3) (1991) 420–460.

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

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

[23] L. Hulbert, E. Zmijewski, Limiting communication in parallel sparse cholesky factorization, SIAM J. Sci. Stat. Comput. 12 (5) (1991) 1184–1197.

[24] D. Irony, S. Toledo, Trading replication for communication in parallel distributed-memory dense solvers, Parallel Process. Lett. 12 (01) (2002) 79–94.

[25] D. Irony, S. Toledo, A. Tiskin, Communication lower bounds for distributed-memory matrix multiplication, J. Parallel Distrib. Comput. 64 (9) (2004) 1017–1026.

[26] M. Jacquelin, Y. Zheng, E. Ng, K. Yelick, An asynchronous task-based fan-both sparse Cholesky solver arXiv preprint arXiv:1608.00044.

[27] H. Jia-Wei, H.-T. Kung, I/o complexity: The red-blue pebble game, in: Proceedings of the Thirteenth Annual ACM Symposium on Theory of Computing, ACM, 1981, pp. 326–333.

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

[29] A. Khabou, J. Demmel, L. Grigori, M. Gu, Communication avoiding LU factorization with panel rank revealing pivoting, SIAM J. Matrix Anal. Appl. 34 (3) (2013) 1401–1429.

[30] K. Kim, V. Eijkhout, A parallel sparse direct solver via hierarchical DAG scheduling, ACM Trans. Math. Softw. (TOMS) 41 (1) (2014) 3.

[31] G. Krawezik, G. Poole, Accelerating the ANSYS direct sparse solver with GPUs, in: Proc. Symposium on Application Accelerators in High Performance Computing (SAAHPC), Urbana-Champaign, IL, NCSA, 2009.

[32] X. Lacoste, Scheduling and memory optimizations for sparse direct solver on multi-core/multi-GPU duster systems, (Ph.D. thesis) Bordeaux 2015.

[33] X. Lacoste, M. Faverge, G. Bosilca, P. Ramet, 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, IEEE, 2014, pp. 29–38.

[34] X.S. Li, An overview of superlu: Algorithms, implementation, and user interface, ACM Trans. Math. Softw. (TOMS) 31 (3) (2005) 302–325.

[35] X.S. Li, J.W. Demmel, Making sparse Gaussian elimination scalable by static pivoting, in: Proceedings of the 1998 ACM/IEEE Conference on Supercomputing (CDROM), IEEE Computer Society, 1998, pp. 1–17.

[36] X.S. Li, J.W. Demmel, Superlu_dist: A scalable distributed-memory sparse direct solver for unsymmetric linear systems, ACM Trans. Math. Softw. (TOMS) 29 (2) (2003) 110–140.

[37] X.S. Li, J.W. Demmel, J.R. Gilbert, L. Grigori, P. Sao, M. Shao, I. Yamazaki, SuperLU Users' Guide, Tech. rep., Lawrence Berkeley National Laboratory 2016.

[38] R.J. Lipton, R.E. Tarjan, A separator theorem for planar graphs, SIAM J. Appl. Math. 36 (2) (1979) 177–189.

[39] R. Lucas, G. Wagenbreth, D. Davis, R. Grimes, Multifrontal computations on GPUs and their multi-core hosts, in: VECPAR'10: Proc. 9th Intl. Meeting for High Performance Computing for Computational Science, Berkeley, CA, 2010.

[40] A. Meurer, C.P. Smith, M. Paprocki, O. Čertík, S.B. Kirpichev, M. Rocklin, A. Kumar, S. Ivanov, J.K. Moore, S. Singh, T. Rathnayake, S. Vig, B.E. Granger, R.P. Muller, F. Bonazzi, H. Gupta, S. Vats, F. Johansson, F. Pedregosa, M.J. Curry, A.R. Terrel, v. Roučka, A. Saboo, I. Fernando, S. Kulal, R. Cimrman, A. Scopatz, Sympy: symbolic computing in python, PeerJ Comput. Sci. 3 (2017) e103, http://dx.doi.org/10.7717/peerj-cs.103.

[41] F. Pellegrini, J. Roman, SCOTCH : A software package for static mapping by dual recursive bipartitioning of process and architecture graphs, in: International Conference on High-Performance Computing and Networking, Springer, 1996, pp. 493–498.

[42] E. Rothberg, Exploiting the memory hierarchy in sequential and parallel sparse cholesky factorization, Tech. rep., Stanford University, Department of Computer Science 1992.

[43] P. Sao, X.S. Li, R. Vuduc, A communication-avoiding 3D LU factorization algorithm for sparse matrices, in: Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS), Vancouver, BC, Canada, 2018.

[44] P. Sao, X. Liu, R. Vuduc, X. Li, A sparse direct solver for distributed memory xeon phi-accelerated systems, in: Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International, IEEE, 2015, pp. 71–81.

[45] P. Sao, R. Vuduc, X.S. Li, A distributed CPU-gpu sparse direct solver, in: European Conference on Parallel Processing, Springer, 2014, pp. 487–498.

[46] E. Solomonik, J. Demmel, Communication-optimal parallel 2.5d matrix multiplication and LU factorization algorithms, in: Euro-Par 2011 Parallel Processing, Springer, 2011, pp. 90–109.

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

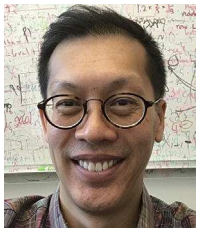
[48] R. Vuduc, A. Chandramowlishwaran, J. Choi, M. Guney, A. Shringarpure, On the limits of GPU acceleration, in: Proc. of the 2nd USENIX Conference on Hot Topics in Parallelism, HotPar'10, Berkeley, CA, 2010.

[49] I. Yamazaki, 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, IEEE, 2012, pp. 619–630.

[50] C. Yu, W. Wang, D. Pierce, A CPU-gpu hybrid approach for the unsymmetric multifrontal method, Parallel Comput. 37 (2011) 759–770.

Piyush Sao is a Ph.D. Candidate at Georgia Institute of Technology. He received his B. Tech. and M. Tech. in electrical engineering from Indian Institute of Technology (IIT), Madras (India) in 2011, and M.S. in computational science and engineering from Georgia Tech in 2016. He is interested in high-performance computing, fault-tolerant algorithms and sparse linear algebra.

Xiaoye S. Li earned Ph.D. in Computer Science from UC Berkeley, and is now a Senior Scientist at Lawrence Berkeley National Laboratory. She has worked on diverse problems in high performance scientific computations, including parallel computing, sparse matrix computations, high precision arithmetic, and combinatorial scientific computing. She has (co)authored over 100 publications, and has led and contributed to the development of several open source mathematical libraries. She is a SIAM Fellow and an ACM Senior Member.



Richard (Rich) Vuduc is an associate professor in the School of Computational Science and Engineering at the Georgia Institute of Technology ("Georgia Tech"). He received his Ph.D. in computer science from the University of California, Berkeley, and his B.S. in computer science from Cornell University. His research lab, The HPC Garage (https://twitter.com/hpcgarage@hpcgarage), is interested in high-performance computing, with an emphasis on algorithms, performance analysis, and performance engineering. Among other activities, he currently serves as the Chair of the Society of Industrial and Applied Mathematics (SIAM) Activity Group on Supercomputing.