# Communication-optimal parallel 2.5D matrix multiplication and LU factorization algorithms

Edgar Solomonik and James Demmel

Department of Computer Science
University of California at Berkeley, Berkeley, CA, USA
solomon@eecs.berkeley.edu, demmel@eecs.berkeley.edu

**Abstract.** One can use extra memory to parallelize matrix multiplication by storing $p^{1/3}$ redundant copies of the input matrices on $p$ processors in order to do asymptotically less communication than Cannon's algorithm [2], and be faster in practice [1]. We call this algorithm "3D" because it arranges the $p$ processors in a 3D array, and Cannon's algorithm "2D" because it stores a single copy of the matrices on a 2D array of processors. We generalize these 2D and 3D algorithms by introducing a new class of "2.5D algorithms". For matrix multiplication, we can take advantage of any amount of extra memory to store $c$ copies of the data, for any $c \in \{1, 2, ..., \lfloor p^{1/3} \rfloor\}$, to reduce the bandwidth cost of Cannon's algorithm by a factor of $c^{1/2}$ and the latency cost by a factor $c^{3/2}$. We also show that these costs reach the lower bounds [13, 3], modulo polylog($p$) factors. We similarly generalize LU decomposition to 2.5D and 3D, including communication-avoiding pivoting, a stable alternative to partial-pivoting [7]. We prove a novel lower bound on the latency cost of 2.5D and 3D LU factorization, showing that while $c$ copies of the data can also reduce the bandwidth by a factor of $c^{1/2}$, the latency must *increase* by a factor of $c^{1/2}$, so that the 2D LU algorithm ($c = 1$) in fact minimizes latency. Preliminary results of 2.5D matrix multiplication on a Cray XT4 machine also demonstrate a performance gain of up to 3X with respect to Cannon's algorithm. Careful choice of $c$ also yields up to a 2.4X speed-up over 3D matrix multiplication, due to a better balance between communication costs.

## 1 Introduction

Goals of parallelization include minimizing communication, balancing the work load, and balancing the memory per processor. In practice there are tradeoffs among these goals. For example, some problems can be made embarrassingly parallel by replicating the entire input on each processor. However, this approach may use much more memory than necessary, and require significant redundant computation. At the other extreme, one stores exactly one copy of the data spread evenly across the processors, and tries to minimize communication and balance the load subject to this constraint.

However, some parallel algorithms do successfully take advantage of limited extra memory to increase parallelism or decrease communication. In this paper, we examine the trade-off between memory usage and communication cost in linear algebra algorithms. We introduce 2.5D algorithms (the name is explained below), which have the property that they can use any available amount of extra memory beyond that needed to store one distributed copy of the input and output, to provably reduce the amount of communication they perform to a theoretical minimum.

We measure costs along the critical path to make sure our algorithms are well load balanced as well as communication efficient. In particular, we measure the following quantities along the critical path of our algorithms (which determines the running time):

- $F$, the computational cost, is the number of flops done along the critical path.
- $W$, the bandwidth cost, is the number of words sent/received along the critical path.
- $S$, the latency cost, is the number of messages sent/received along the critical path.
- $M$, memory size, is the maximum amount of memory, in words, utilized by a processor at any point during algorithm execution.

Our communication model does not account for network topology. However, it does assume that all communication has to be synchronous. So, a processor cannot send multiple messages at the cost of a single message. Under this model a reduction or broadcast among $p$ processors costs $O(\log p)$ messages but a one-to-one permutation requires only $O(1)$ messages. This model aims to capture the behavior of low-dimensional mesh or torus network topologies. Our LU communication lower-bound does is independent of the above collective communication assumptions, however, it does leverage the idea of the critical path.

Our starting point is $n$-by-$n$ dense matrix multiplication, for which there are known algorithms that minimize both bandwidth and latency costs in two special cases:

1. When only enough memory, $M$, for one copy of the input/output matrices is available, evenly spread across all $p$ processors (so $M \approx 3n^2/p$), it is known that Cannon's Algorithm [5] simultaneously balances the load (so $F = \Theta(n^3/p)$), minimizes the bandwidth cost (so $W = \Theta(n^2/p^{1/2})$), and minimizes the latency cost (so $S = \Theta(p^{1/2})$) [13, 3]. We call this the "2D algorithm" because it is naturally expressed by laying out the matrices across a $p^{1/2}$-by-$p^{1/2}$ grid of processors.
2. When enough memory, $M$, for $p^{1/3}$ copies of the input/output matrices is available, evenly spread across all $p$ processors (so $M \approx 3n^2/p^{2/3}$), it is known that algorithms presented in [6, 1, 2] simultaneously balance the load (so $F = \Theta(n^3/p)$), minimize the bandwidth cost (so $W = \Theta(n^2/p^{2/3})$), and minimize the latency cost (so $S = \Theta(\log p)$) [13, 3]. We call this the "3D algorithm" because it is naturally expressed by laying out the matrices across a $p^{1/3}$-by-$p^{1/3}$-by-$p^{1/3}$ grid of processors.

The contributions of this paper are as follows.

1. We generalize 2D and 3D matrix multiplication to use as much memory as available, in order to reduce communication: If there is enough memory for $c$ copies of the input and output matrices, for any $c \in \{1, 2, ..., \lfloor p^{1/3} \rfloor\}$, then we present a new matrix multiplication algorithm that sends $c^{1/2}$ times fewer words than the 2D (Cannon's) algorithm, and sends $c^{3/2}$ times fewer messages. We call the new algorithm *2.5D matrix multiplication*, because it has the 2D and 3D algorithms as special cases, and effectively interpolates between them, by using a processor grid of shape $(p/c)^{1/2}$-by-$(p/c)^{1/2}$-by-$c$.
2. Our 2.5D matrix multiplication algorithm attains lower bounds (modulo polylog($p$) factors) on the number of words and messages communicated that hold for *any* matrix multiplication algorithm that (1) performs the usual $n^3$ multiplications, and (2) uses $c$ times the minimum memory possible.
3. We present a 2.5D LU algorithm that also reduces the number of words moved by a factor of $c^{1/2}$, attaining the same lower bound. The algorithm does *tournament pivoting* as opposed to partial pivoting [7, 10]. Tournament pivoting is a stable alternative to partial pivoting that was used to minimize communication (both number of words and messages) in the case of 2D LU. We will refer to tournament pivoting as communication-avoiding pivoting (CA-pivoting) to emphasize the fact that this type of pivoting attains the communication lower-bounds.
4. 2.5D LU does not, however, send fewer messages than 2D LU; instead it sends a factor of $c^{1/2}$ *more* messages. Under reasonable assumptions on the algorithm, we prove a new lower bound on the number of messages for any LU algorithm that communicates as few words as 2.5D LU, and show that 2.5D LU attains this new lower bound. Further, we show that using extra memory cannot reduce the latency cost of LU below the 2D algorithm, which sends $\Omega(p^{1/2})$ messages. These results hold for LU with and without pivoting.

## 2 Previous work

In this section, we detail the motivating work for our algorithms. First, we recall linear algebra communication lower bounds that are parameterized by memory size. We also detail the main motivating algorithm for this work, 3D matrix multiplication, which uses extra memory but performs less communication. The communication complexity of this algorithm serves as a matching upper-bound for our general lower bound.

### 2.1 Communication lower bounds for linear algebra

Recently, a generalized communication lower bound for linear algebra has been shown to apply for a large class of matrix-multiplication-like problems [3]. The lower bound applies to either sequential or parallel distributed memory, and either dense or sparse algorithms. The distributed memory lower bound is formulated under a communication model identical to that which we use in this paper. This lower bound states that for a fast memory of size $M$ the lower bound on communication bandwidth is

$$W = \Omega\left(\frac{\#arithmetic\ operations}{\sqrt{M}}\right)$$

words, and the lower bound on latency is

$$S = \Omega\left(\frac{\#arithmetic\ operations}{M^{3/2}}\right)$$

messages. On a parallel machine with $p$ processors and a local processor memory of size $M$, this yields the following lower bounds for communication costs of matrix multiplication of two dense $n$-by-$n$ matrices as well as LU factorization of a dense $n$-by-$n$ matrix,

$$W = \Omega\left(\frac{n^3/p}{\sqrt{M}}\right),$$
$$S = \Omega\left(\frac{n^3/p}{M^{3/2}}\right).$$

These lower bounds are valid for $\frac{n^2}{p} < M < \frac{n^2}{p^{2/3}}$ and suggest that algorithms can reduce their communication cost by utilizing more memory. If $M < \frac{n^2}{p}$, the entire matrix won't fit in memory.

### 2.2 3D linear algebra algorithms

Consider we have $p$ processors arranged into a 3D grid as in Figure 1(a), with each individual processor indexed as $P_{i,j,k}$. We replicate input matrices on 2D layers of this 3D grid so that each processor uses $M = \Omega\left(\frac{n^2}{p^{2/3}}\right)$ words of memory. In this decomposition, the lower bound on bandwidth is

$$W_{3d} = \Omega\left(\frac{n^2}{p^{2/3}}\right).$$

According to the general lower bound the lower bound on latency is trivial: $\Omega(1)$ messages. However, for any blocked 2D or 3D layout,

$$S_{3d} = \Omega(\log p)$$

messages are required since all entries of $C$ depend on entries from a block row of $A$, and information from a block row of $A$ can only be propagated to one processor with $\Omega(\log p)$ messages.

**3D matrix multiplication** For matrix multiplication, Algorithm 1 [6, 1, 2] achieves the 3D bandwidth and latency lower bounds.

The amount of memory used in this 3D matrix multiplication algorithm is

$$M = \Theta\left(\frac{n^2}{p^{2/3}}\right)$$

so the 3D communication lower bounds apply. The only communication performed is the reduction of $C$ and possibly a broadcast to spread the input. So the bandwidth cost is

$$W = O\left(\frac{n^2}{p^{2/3}}\right),$$

**Algorithm 1**: $[C]$ = 3D-matrix-multiply($A$,$B$,$n$,$p$)

**Input**: $n$-by-$n$ matrix $A$ distributed so that $P_{ij0}$ owns $\frac{n}{p^{1/3}}$-by-$\frac{n}{p^{1/3}}$ block $A_{ij}$ for each $i, j$
**Input**: $n$-by-$n$ matrix $B$ distributed so that $P_{0jk}$ owns $\frac{n}{p^{1/3}}$-by-$\frac{n}{p^{1/3}}$ block $B_{jk}$ for each $j, k$
**Output**: $n$-by-$n$ matrix $C = A \cdot B$ distributed so that $P_{i0k}$ owns $\frac{n}{p^{1/3}}$-by-$\frac{n}{p^{1/3}}$ block $C_{ik}$ for each $i, k$

```
// do in parallel with all processors
```
**forall** $i, j, k \in \{0, 1, ..., p^{1/3} - 1\}$ **do**
    $P_{ij0}$ broadcasts $A_{ij}$ to all $P_{ijk}$ /* replicate $A$ on each $ij$ layer */
    $P_{0jk}$ broadcasts $B_{jk}$ to all $P_{ijk}$ /* replicate $B$ on each $jk$ layer */
    $C_{ijk} := A_{ij} \cdot B_{jk}$
    $P_{ijk}$ contributes $C_{ijk}$ to a sum-reduction to $P_{i0k}$
**end**

which is optimal, and the latency cost is

$$S = O(\log p),$$

which is a optimal for a blocked layout.

**Memory efficient matrix multiplication** McColl and Tiskin [14] present a memory efficient variation on the 3D matrix multiplication algorithm for a PRAM-style model. They partition the 3D computation graph to pipeline the work and therefore reduce memory in a tunable fashion. However, their theoretical model is not reflective of modern supercomputer architectures, and we see no clear way to reformulate their algorithm to be communication optimal. Nevertheless, their research is in very similar spirit to and serves as a motivating work for the new 2.5D algorithms we present in section 3.

**Previous work on 3D LU factorization** Irony and Toledo [12] introduced a 3D LU factorization algorithm that minimizes total communication volume (sum of the number of words moved over all processors), but does not minimize either bandwidth or latency along the critical path. This algorithm distributes $A$ and $B$ cyclically on each processor layer and recursively calls 3D LU and 3D TRSM routines on sub-matrices.

Neither the 3D TRSM nor the 3D LU base-case algorithms given by Irony and Toledo minimize communication along the critical path which, in practice, is the bounding cost. We define a different 2.5D LU factorization algorithm that does minimize communication along its critical path.

## 3 2.5D lower and upper bounds

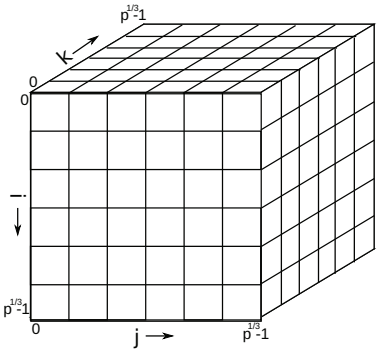
The general communication lower bounds are valid for a range of $M$ in which 2D and 3D algorithms hit the extremes. 2.5D algorithms are parameterized to be able to achieve the communication lower bounds for any valid $M$. Let $c \in \{1, 2, \ldots, \lfloor p^{1/3} \rfloor\}$ be the number of replicated copies of the input matrix. [1] Consider the processor grid in Figure 1(b) (indexed as $P_{i,j,k}$) where each processor has local memory size $M = \Omega\left(\frac{cn^2}{p}\right)$. The lower bounds on communication are

$$W_{2.5d} = \Omega\left(\frac{n^2}{\sqrt{cp}}\right),$$

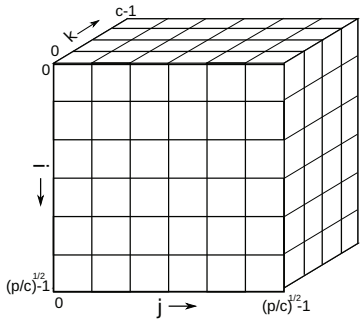
$$S_{2.5d} = \Omega\left(\frac{p^{1/2}}{c^{3/2}}\right).$$

From a performance-tuning perspective, by formulating 2.5D linear algebra algorithms, we are essentially adding an extra tuning parameter to the algorithm. Also, as a sanity check for our 2.5D algorithms, we made sure they reduced to practical 2D algorithms when $c = 1$ and to practical 3D algorithms when $c = p^{1/3}$.

[1] If $c > p^{1/3}$, assuming the initial data layout has no redundant copies, replicating an input matrix would cost more communication than the lower bound.




(a) 3D processor grid of dimension $p^{1/3}$-by-$p^{1/3}$-by-$p^{1/3}$.




(b) 2.5D processor grid for replication factor $c$ of dimension $(p/c)^{1/2}$-by-$(p/c)^{1/2}$-by-$c$.

**Fig. 1.**

### 3.1 2.5D matrix multiplication

For matrix multiplication, Algorithm 2 achieves the 2.5D bandwidth lower bound and gets within a factor of $O(\log p)$ of the 2.5D latency lower bound (likely optimal). Algorithm 2 generalizes Cannon's algorithm (set $c = 1$). At a high level, our 2.5D algorithm does a portion of Cannon's algorithm on with each set of copies of matrices $A$ and $B$, then combines the results. To make this possible, we adjust the initial shift done in Cannon's algorithm to be different for each set of copies.

Our 2.5D algorithm doesn't quite generalize Algorithm 1 since $C$ is reduced in a different dimension (shifting compensates for the difference). However, in terms of complexity, only two extra matrix shift operations are required by the 3D version of our 2.5D algorithm. Further, the 2.5D algorithm has the nice property that $C$ ends up spread over the same processor layer that both $A$ and $B$ started on. The algorithm moves

$$W = O\left(\frac{n^2}{\sqrt{cp}}\right)$$

words and sends

$$S = O\left(\sqrt{p/c^3} + \log c\right)$$

messages. This cost is optimal according to the general communication lower bound. The derivations of these communication costs are in Appendix A.

We also note that if the latency cost is dominated by the intra-layer communication $S = O(\sqrt{p/c^3})$, our 2.5D matrix multiplication algorithm can achieve perfect strong scaling in certain regimes. For matrix size $n^2$ consider the smallest number of processors that can actually fit the problem $p_{\min} = \Theta(n^2/M)$. For this number of processors, a 2D algorithm must be used so the bandwidth cost is $W_{p_{\min}} = O(\frac{n^2}{\sqrt{p_{\min}}})$ and the latency cost is $S_{p_{\min}} = O(\sqrt{p_{\min}})$. Note that if we scale this problem to $p = c \cdot p_{\min}$ processors, we have $M = \Theta(cn^2/p_{\min})$ and can therefore run 2.5D matrix multiplication with a bandwidth cost of $O(\frac{n^2}{\sqrt{c^2 p_{\min}}}) = W_{p_{\min}}/c$ and a latency cost of $O(\sqrt{p_{\min}/c^2}) = S_{p_{\min}}/c$. Thus by using a factor $c$ more processors than necessary we can reduce the entire complexity cost (computation and communication) of matrix multiplication by a factor of $c$.

**Algorithm 2**: $[C]$ = 2.5D-matrix-multiply($A$,$B$,$n$,$p$,$c$)

**Input**: square $n$-by-$n$ matrices $A$, $B$ distributed so that $P_{ij0}$ owns $\frac{n}{\sqrt{p/c}}$-by-$\frac{n}{\sqrt{p/c}}$ blocks $A_{ij}$ and $B_{ij}$ for each $i, j$

**Output**: square $n$-by-$n$ matrix $C = A \cdot B$ distributed so that $P_{ij0}$ owns $\frac{n}{\sqrt{p/c}}$-by-$\frac{n}{\sqrt{p/c}}$ block $C_{ij}$ for each $i, j$

```
  // do in parallel with all processors
forall i,j ∈ {0,1,...,√(p/c)−1}, k ∈ {0,1,...,c−1} do
    P_ij0 broadcasts A_ij and B_ij to all P_ijk                    /* replicate input matrices */
    r := mod (j + i − k√(p/c^3), √(p/c))
      // initial circular shift on A
    s := mod (j − i + k√(p/c^3), √(p/c))
    P_ijk sends A_ij to P_isk
    P_ijk receives block A_ir
      // initial circular shift on B
    s' := mod (i − j + k√(p/c^3), √(p/c))
    P_ijk sends B_ij to P_s'jk
    P_ijk receives block B_rj

    C_ijk := A_ir · B_rj
    s := mod (j + 1, √(p/c))
    s' := mod (i + 1, √(p/c))
      // do steps of Cannon's algorithm
    for t = 1 to √(p/c^3) − 1 do
          // rightwards circular shift on A
        P_ijk sends A_ir to P_isk
          // downwards circular shift on B
        P_ijk sends B_rj to P_s'jk
        r := mod (r − 1, √(p/c))
        P_ijk receives block A_ir
        P_ijk receives block B_rj

        C_ijk := C_ijk + A_ir · B_rj
    end
    P_ijk contributes C_ijk to a sum-reduction to P_ij0
end
```

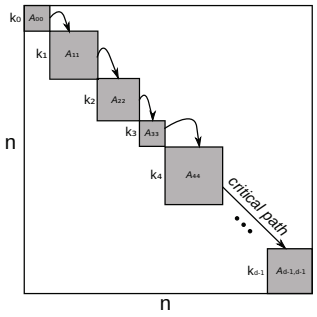



**Fig. 2.** LU diagonal block dependency path. These blocks must be factorized in order and communication is required between each block factorization.

## 4 2.5D LU communication lower bound

We argue that for Gaussian-elimination style LU algorithms that achieve the bandwidth lower bound, the latency lower bound is actually much higher, namely

$$S_{lu} = \Omega\left(\sqrt{cp}\right)$$

Given a parallel LU factorization algorithm, we assume the algorithm must uphold the following properties

1. Consider the largest $k$-by-$k$ matrix $A_{00}$ factorized sequentially such that $A = \begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix}$ (we can always pick some $A_{00}$ since at least the top left element of $A$ is factorized sequentially), the following conditions must hold,
   (a) $\Omega(k^3)$ flops must be done before $A_{11}$ can be factorized (it can be updated but Gaussian elimination cannot start).
   (b) $\Omega(k^2)$ words must be communication before $A_{11}$ can be factorized.
   (c) $\Omega(1)$ messages must be sent before $A_{11}$ can be factorized.
2. The above condition holds recursively (for factorization of $A_{11}$ in place of $A$).

We now lower bound the communication cost for any algorithm that follows the above restrictions. Any such algorithm must compute a sequence of diagonal blocks $\{A_{00}, A_{11}, \ldots, A_{d-1,d-1}\}$. Let the dimensions of the blocks be $\{k_0, k_1, \ldots, k_{d-1}\}$. As done in Gaussian Elimination and as required by our conditions, the factorizations of these blocks are on the critical path and must be done in strict sequence.

Given this dependency path (shown in Figure 2), we can lower bound the complexity of the algorithm by counting the complexity along this path. The latency cost is $\Omega(d)$ messages, the bandwidth cost is $\sum_{i=0}^{d-1} \Omega(k_i^2)$ words and the computational cost is $\sum_{i=0}^{d-1} \Omega(k_i^3)$ flops. Due to the constraint, $\sum_{i=0}^{d-1} k_i = n$, it is best to pick all $k_i = k$, for some $k$, (we now get $d = n/k$) to minimize communication and flops. Now we see that the algorithmic costs are

$$F_{lu} = \Omega(nk^2)$$
$$S_{lu} = \Omega(n/k)$$
$$W_{lu} = \Omega(nk).$$

Evidently, if we want to do $O(n^3/p)$ flops we need

$$k = O\left(\frac{n}{\sqrt{p}}\right),$$

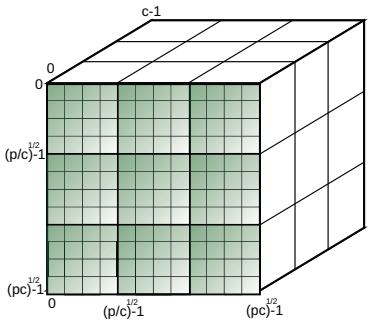



**Fig. 3.** 2.5D LU algorithm processor virtual layout. Each physical processor owns a sub-block of its index in each bigger block.

which would necessitate

$$S = \Omega(\sqrt{p}).$$

Further, the cost of sacrificing flops for latency is large. Namely, if $S = O\left(\frac{\sqrt{p}}{r}\right)$, the computational cost is

$$F = \Omega\left(\frac{r^2 n^3}{p}\right),$$

a factor of $r^2$ worse than optimal. Since we are very unlikely to want to sacrifice so much computational cost to lower the latency cost we will not attempt to design algorithms that achieve a latency smaller than $\Omega(\sqrt{p})$.

If we want to achieve the bandwidth lower bound we need,

$$\begin{aligned} W_{lu} &= O\left(\frac{n^2}{\sqrt{cp}}\right) \\ k &= O\left(\frac{n}{\sqrt{cp}}\right) \\ S_{lu} &= \Omega\left(\frac{n}{O\left(\frac{n}{\sqrt{cp}}\right)}\right) \\ &= \Omega(\sqrt{cp}). \end{aligned}$$

A latency cost of $O(\sqrt{cp}/r)$, would necessitate a factor of $r$ larger bandwidth cost. So, an LU algorithm can do minimal flops, bandwidth, and latency as defined in the general lower bound, only when $c = 1$. For $c > 1$, we can achieve optimal bandwidth and flops but not latency.

Its also worth noting that the larger $c$ is, the higher the latency cost for LU will be (assuming bandwidth is prioritized). This insight is the opposite of that of the general lower bound, which lower bounds the latency as $\Omega(1)$ messages for 3D, while now we have a lower bound of $\Omega(p^{2/3})$ messages for 3D (assuming optimal bandwidth). This tradeoff suggests that $c$ should be tuned to balance the bandwidth cost and the latency cost.

## 5 2.5D communication optimal LU

In order to write down a 2.5D LU algorithm, it is necessary to find a way to meaningfully exploit extra memory. A 2D parallelization of LU typically factorizes a vertical and a top panel of the matrix and updates

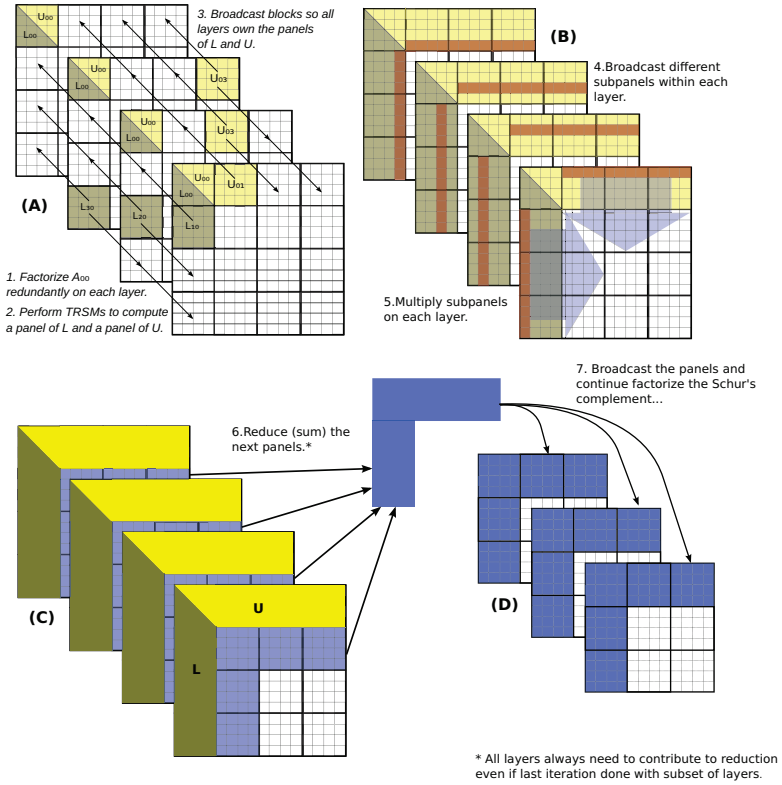



**Fig. 4.** 2.5D LU algorithm work-flow

the remainder (the Schur complement). The dominant cost in a typical parallel LU algorithm is the update to the Schur complement. Our 2.5D algorithm exploits this by accumulating the update over layers. However, in order to factorize each next panel we must reduce the contributions to the Schur complement. We note that only the panel we are working on needs to be reduced and the remainder can be further accumulated. Even so, to do the reductions efficiently, a block-cyclic layout is required. This layout allows more processors to participate in the reductions and pushes the bandwidth cost down to the lower bound.

Algorithm 3 (work-flow diagram in Figure 4) is a communication optimal LU factorization algorithm for the entire range of $c \in \{1, 2, \ldots, \lfloor p^{1/3} \rfloor\}$. The algorithm replicates the matrix $A$ on each layer and partitions it block cyclically across processors with block size $(n/\sqrt{pc})$-by-$(n/\sqrt{pc})$ (see Figure 3). Note that this block dimension corresponds to the lower bound derivations in the previous section. Every processor owns one such block within each bigger block of size $n/c$-by-$n/c$. We will sometimes refer to big blocks (block dimension $n/c$) and small blocks (block dimension $n/\sqrt{pc}$) for brevity.

Algorithm 3 has a bandwidth cost of

$$W = O\left(\frac{n^2}{\sqrt{cp}}\right)$$

words and a latency cost of

$$S = O\left(\sqrt{cp}\log(p)\right)$$

messages. Therefore, it is asymptotically communication optimal for any choice of $c$ (modulo a $\log(p)$ factor for latency). Further, it is also always asymptotically computationally optimal (the redundant work is a low order cost). These costs are derived in Appendix B.

**Algorithm 3**: $[L, U]$ = 2.5D-LU-factorization($A$,$n$,$p$,$c$)

Let `[`$L_{ij}$`, `$U_{ij}$`] = 2D-LU`$_{ij}$`(`$A_{ij}$`,k)` be a function that computes the LU factorization of $A$ using a 2D parallel algorithm with $P_{ijk}$ for each $i$, $j$ on layer $k$.

Let `[`$X_{ij}$`] = 2D-TRSM`$_{ij}$`(`$L_{ij}$`,`$B_{ij}$`,k)` be a function that given triangular $L$ computes the $X$ such that $L \cdot X = B$ using a 2D parallel algorithm with $P_{ijk}$ for each $i$, $j$ on layer $k$.

**Input**: $n$-by-$n$ matrix $A$ distributed so that for each $i, j$, $P_{ij0}$ owns $\frac{n}{\sqrt{pc}}$-by-$\frac{n}{\sqrt{pc}}$ blocks $A_{l_i m_j}$ for $l_i \in \{i, i+\sqrt{p/c}, i+2\sqrt{p/c}, \ldots, i+(c-1)\sqrt{p/c}\}$ and $m_j \in \{j, j+\sqrt{p/c}, j+2\sqrt{p/c}, \ldots, j+(c-1)\sqrt{p/c}\}$.

**Output**: triangular $n$-by-$n$ matrices $L$, $U$ such that $A = L \cdot U$ and for each $i, j$, $P_{ij0}$ owns $\frac{n}{\sqrt{pc}}$-by-$\frac{n}{\sqrt{pc}}$ blocks $L_{l_i m_j}$ and $U_{l_i m_j}$ for each $l_i$ and $m_j$.

*// do in parallel with all processors*

**forall** $i, j \in \{0, 1, ..., \sqrt{p/c} - 1\}$, $k \in \{0, 1, ..., c-1\}$ **do**

$S = 0$

$P_{ij0}$ broadcasts $A_{l_i m_j}$ for each $l_i$, $m_j$ to all $P_{ijk}$ */* replicate $A$ */*

**for** $t = 0$ **to** $c - 1$ **do**

*// redundantly factorize top right $(n/c)$-by-$(n/c)$ block*

`[`$L_{t\sqrt{p/c}+i, t\sqrt{p/c}+j}$`,`$U_{t\sqrt{p/c}+i, t\sqrt{p/c}+j}$`] = 2D-LU`$_{ij}$`(`$A_{t\sqrt{p/c}+i, t\sqrt{p/c}+j}$`,k)`

**if** $t + k < c - 1$ **then** */* perform $(n/c)$-by-$(n/c)$ block TRSMs */*

`[`$L^T_{(t+k+1)\sqrt{p/c}+i, t\sqrt{p/c}+j}$`] = 2D-TRSM`$_{ij}$`(`$U^T_{t\sqrt{p/c}+i, t\sqrt{p/c}+j}$`, `$A^T_{(t+k+1)\sqrt{p/c}+i, t\sqrt{p/c}+j}$`,k)`

$P_{ijk}$ broadcasts $L_{(t+k+1)\sqrt{p/c}+i, t\sqrt{p/c}+j}$ to $P_{ijk'}$ for $k' \in \{0, 1, ..., c-1\}$

`[`$U_{t\sqrt{p/c}+i, (t+k+1)\sqrt{p/c}+j}$`] = 2D-TRSM`$_{ij}$`(`$L_{t\sqrt{p/c}+i, t\sqrt{p/c}+j}$`, `$A_{t\sqrt{p/c}+i, (t+k+1)\sqrt{p/c}+j}$`,k)`

$P_{ijk}$ broadcasts $U_{t\sqrt{p/c}+i, (t+k+1)\sqrt{p/c}+j}$ to $P_{ijk'}$ for $k' \in \{0, 1, ..., c-1\}$

**end**

**if** $t < c - 1$ **then** */* perform trailing matrix update */*

**if** $\lfloor k\sqrt{p/c^3} \rfloor \le j < \lfloor (k+1)\sqrt{p/c^3} \rfloor$ **then**

$P_{ijk}$ broadcasts $L_{m\sqrt{p/c}+i, t\sqrt{p/c}+j}$ for each $m \in \{t+1, t+2, ..., c-1\}$ to each $P_{ij'k}$ for $j' \in \{0, 1, ..., \sqrt{p/c} - 1\}$

**end**

**if** $\lfloor k\sqrt{p/c^3} \rfloor \le i < \lfloor (k+1)\sqrt{p/c^3} \rfloor$ **then**

$P_{ijk}$ broadcasts $U_{t\sqrt{p/c}+i, m\sqrt{p/c}+j}$ for each $m \in \{t+1, t+2, ..., c-1\}$ to each $P_{i'jk}$ for $i' \in \{0, 1, ..., \sqrt{p/c} - 1\}$

**end**

**for** $l = t + 1$ **to** $c - 1$ **do**

**for** $m = t + 1$ **to** $c - 1$ **do**

**for** $r = \lfloor k\sqrt{p/c^3} \rfloor$ **to** $\lfloor (k+1)\sqrt{p/c^3} \rfloor - 1$ **do**

$S_{lm} := S_{lm} + L_{l\sqrt{p/c}+i, t\sqrt{p/c}+r} \cdot U_{t\sqrt{p/c}+r, m\sqrt{p/c}+j}$

**end**

**end**

**end**

*// reduce panels of Schur complement updates and adjust $A$*

$P_{ijk}$ contributes $S_{l\sqrt{p/c}+i, m\sqrt{p/c}+j}$ and for $(l, m) \in \{(t+1, t+1), (t+2, t+1), ..., (c-1, t+1)\} \cup \{(t+1, t+2), (t+1, t+3), ..., (t+1, c-1)\}$ to a sum all-reduction among all $P_{ijk}$

$A_{l\sqrt{p/c}+i, m\sqrt{p/c}+j} := A_{l\sqrt{p/c}+i, m\sqrt{p/c}+j} - S_{l\sqrt{p/c}+i, m\sqrt{p/c}+j}$

**end**

**end**

**end**

## 6 2.5D communication optimal LU with pivoting

Regular partial pivoting is not latency optimal because it requires $O(n)$ messages if the matrix is in a blocked layout. $O(n)$ messages are required by partial pivoting since a pivot needs to be determined for each matrix column which always requires communication unless the entire column is owned by one processor. However, tournament pivoting (CA-pivoting) [7], is a new LU pivoting strategy that can satisfy the general communication lower bound. We will incorporate this strategy into our 2.5D LU algorithm.

CA-pivoting simultaneously determines $b$ pivots by forming a tree of factorizations as follows,

1. Factorize each $2b$-by-$b$ block $[A_{0,2k}, A_{0,2k+1}]^T = P_k^T L_k U_k$ for $k \in [0, \frac{n}{2b} - 1]$ using GEPP.
2. Write $B_k = P_k[A_{0,2k}, A_{0,2k+1}]^T$, and $B_k = [B'_k, B''_k]^T$. Each $B'_k$ represents the 'best rows' of each sub-panel of $A$.
3. Now recursively perform steps 1-3 on $[B'_0, B'_1, ..., B'_{n/(2b)-1}]^T$ until the number of total best pivot rows is $b$.

For a more detailed and precise description of the algorithm and stability analysis see [7, 10].

To incorporate CA-pivoting into our LU-algorithm, we would like to do pivoting with block size $b = n/\sqrt{pc}$. The following modifications need to be made to accomplish this,

1. Previously, we did the big-block side panel Tall-Skinny LU (TSLU) via a redundant top block LU-factorization and TRSMs on lower blocks. To do pivoting, the TSLU factorization needs to be done as a whole rather than in blocks. We can still have each processor layer compute a different 'TRSM block' but we need to interleave this computation with the top block LU factorization and communicate between layers to determine each set of pivots as follows (Algorithm 4 gives the full TSLU algorithm),
   (a) For every small block column, we perform CA-pivoting over all layers to determine the best rows.
   (b) We pivot the rows within the panel on each layer. Interlayer communication is required, since the best rows are spread over the layers (each layer updates a subset of the rows).
   (c) Each $ij$ processor layer redundantly performs small TRSMs and the Schur complement updates in the top big block.
   (d) Each $ij$ processor layer performs TRSMs and updates on a unique big-block of the panel.
2. After the TSLU, we need to pivot rows in the rest of the matrix. We do this redundantly on each layer, since each layer will have to contribute to the update of the entire Schur complement.
3. We still reduce the side panel (the one we do TSLU on) at the beginning of each step but we postpone the reduction of the top panel until pivoting is complete. Basically, we need to reduce the 'correct' rows which we know only after the TSLU.

Algorithm 5 details the entire 2.5D LU with CA-pivoting algorithm and Figure 5 demonstrates the work-flow of the new TSLU with CA-pivoting. Asymptotically, 2.5D LU with CA-pivoting has almost the same communication and computational cost as the original algorithm. Both the flops and bandwidth costs gain an extra asymptotic $\log p$ factor (which can be remedied by using a smaller block size and sacrificing some latency). Also, the bandwidth cost derivation requires a probabilistic argument about the locations of the pivot rows, however, the argument should hold up very well in practice. For the full cost derivations of this algorithm see Appendix C.

## 7 Performance results

We validate the practicality of 2.5D matrix multiplication via performance results on Franklin, a Cray XT4 machine at NERSC. Franklin has 9,572 compute nodes connected in a 3D torus network topology. Each node is a 2.3 GHz single socket quad-core AMD Opteron processor (Budapest) with a theoretical peak performance of 9.2 GFlop/sec per core. More details on the machine can be found on the NERSC website (see http://www.nersc.gov/nusers/systems/franklin/about.php).

**Algorithm 4**: $[V, L, U]$ = 2.5D-LU-pivot-panel-factorization($A$,$n$,$m$,$p$,$c$)

Let `[V] = CA-Pivot`$_l$`(`$A_l$`,n,b)` be a function that performs CA-pivoting with block size $b$ on $A$ of size $n$-by-$b$ and outputs the pivot matrix $V$ to all processors.

**Input**: $n$-by-$m$ matrix $A$ distributed so that for each $i, j$, $P_{ijk}$ owns $\frac{m}{\sqrt{p/c}}$-by-$\frac{m}{\sqrt{p/c}}$ blocks $A_{l_i j}$ for $l_i \in \{i, i+\sqrt{p/c}, i+2\sqrt{p/c}, \ldots, i+(n/m-1)\sqrt{p/c}\}$.

**Output**: $n$-by-$n$ permutation matrix $V$ and triangular matrices $L$, $U$ such that $V \cdot A = L \cdot U$ and for each $i, j$, $P_{ijk}$ owns $\frac{n}{\sqrt{pc}}$-by-$\frac{n}{\sqrt{pc}}$ blocks $L_{l_i j}$ and $U_{ij}$ for each $i$, $l_i$, and $j$.

$V := I$

*// do in parallel with all processors*

**forall** $i, j \in \{0, 1, ..., \sqrt{p/c}-1\}$, $k \in \{0, 1, ..., c-1\}$ **do**

**for** $s = 0$ **to** $\sqrt{p/c}-1$ **do**

**if** $j = s$ *and either* $k \in \{1, 2, ..., n/m-1\}$ *or* $k = 0$ *and* $i \geq s$ **then**

`[`$V_s$`] = CA-Pivot`$_{k\sqrt{p/c}+i}$`(`$A_{k\sqrt{p/c}+i,j}$`,k)`

**end**

$P_{sjk}$ for $j \in \{s, s+1, ...\sqrt{p/c}-1\}$ swaps any rows as required by $V_s$ between $A_{sj}$ and $A_{k\sqrt{p/c}+l,j}$ for $l \in \{0, 1, ..., \sqrt{p/c}-1\}$ and $k \in \{1, 2, ..., n/m-1\}$.

$P_{sjk}$ for $j \in \{s, s+1, ..., \sqrt{p/c}-1\}$ do an all-gather, for all $k$, of the rows they just swapped in.

$P_{sjk}$ for $j \in \{s, s+1, ..., \sqrt{p/c}-1\}$ swaps any rows as required by $V_s$ between $A_{sj}$ and $A_{lj}$ for $l \in \{s+1, s+2, ..., \sqrt{p/c}-1\}$.

$P_{ssk}$ computes $A_{ss} := V_s'^T L_{ss} U_{ss}$.

$P_{ssk}$ broadcasts $V_s'$, $L_{ss}$ to all $P_{sjk}$ and $U_{ss}$ to al $P_{isk}$.

$P_{sjk}$ for $j \in \{s+1, ..., \sqrt{p/c}-1\}$ solves $U_{sj} := L_{ss}^{-1} V_s' A_{sj}$.

$P_{sjk}$ for $j \in \{s+1, ..., \sqrt{p/c}-1\}$ broadcasts $U_{sj}$ to all $P_{ijk}$.

$P_{isk}$ for $i \in \{s+1, ..., \sqrt{p/c}-1\}$ solves $L_{is}^T := U_{ss}^{-T} A_{is}^T$.

$P_{isk}$ for $i \in \{s+1, ..., \sqrt{p/c}-1\}$ broadcasts $L_{is}$ to all $P_{ijk}$.

$P_{ijk}$ for $i, j \in \{s+1, ..., \sqrt{p/c}-1\}$ computes $A_{ij} := A_{ij} - L_{is} \cdot U_{sj}$.

$P_{isk}$ for $k \in \{1, ..., n/m-1\}$ solves $L_{k\sqrt{p/c}+i,s}^T := U_{ss}^{-T} A_{k\sqrt{p/c}+i,s}^T$.

$P_{isk}$ for $k \in \{1, ..., n/m-1\}$ broadcasts $L_{k\sqrt{p/c}+i,s}$ to all $P_{ijk}$.

$P_{ijk}$ for $j \in \{s+1, ..., \sqrt{p/c}-1\}$ computes $A_{k\sqrt{p/c}+i,j} := A_{k\sqrt{p/c}+i,j} - L_{k\sqrt{p/c}+i,s} \cdot U_{sj}$.

$V := \begin{bmatrix} I & 0 \\ 0 & V_s' \end{bmatrix} \cdot \begin{bmatrix} I & 0 \\ 0 & V_s \end{bmatrix} \cdot V$.

**end**

$P_{ijk}$ for $k \in \{1, ..., n/m-1\}$ broadcasts $L_{k\sqrt{p/c}+i,j}$ to $P_{ijk'}$ for $k' \in \{0, 1, ..., c-1\}$.

**end**

**Algorithm 5**: $[V, L, U]$ = 2.5D-LU-pivot-factorization($A$,$n$,$p$,$c$)

Let `[`$X_{ij}$`] = 2D-TRSM`$_{ij}$`(`$L_{ij}$`,`$B_{ij}$`,k)` be defined as in Algorithm 3.

**Input**: $n$-by-$n$ matrix $A$ distributed so that for each $i, j$, $P_{ij0}$ owns $\frac{n}{\sqrt{pc}}$-by-$\frac{n}{\sqrt{pc}}$ blocks $A_{l_i m_j}$ for $l_i \in \{i, i+\sqrt{p/c}, i+2\sqrt{p/c}, \ldots, i+(c-1)\sqrt{p/c}\}$ and $m_j \in \{j, j+\sqrt{p/c}, j+2\sqrt{p/c}, \ldots, j+(c-1)\sqrt{p/c}\}$.

**Output**: $n$-by-$n$ permutation matrix $V$ and triangular matrices $L$, $U$ such that $V \cdot A = L \cdot U$ and for each $i, j$, $P_{ij0}$ owns $\frac{n}{\sqrt{pc}}$-by-$\frac{n}{\sqrt{pc}}$ blocks $L_{l_i m_j}$ and $U_{l_i m_j}$ for each $l_i$ and $m_j$.

**forall** $i, j \in \{0, 1, ..., \sqrt{p/c}-1\}$, $k \in \{0, 1, ..., c-1\}$ **do**

$S = 0$, $V = I$

processor $P_{ij0}$ broadcasts $A_{l_i m_j}$ for each $l_i$, $m_j$ to all $P_{ijk}$ */* replicate $A$ */*

**for** $t = 0$ **to** $c-1$ **do**

`[`$V_t, L_{t\sqrt{p/c}:\sqrt{pc}-1, t\sqrt{p/c}:(t+1)\sqrt{p/c}-1}, U_{t\sqrt{p/c}:(t+1)\sqrt{p/c}-1, t\sqrt{p/c}:(t+1)\sqrt{p/c}-1}$`] =`
`2.5D-LU-pivot-panel-factorization(`$A_{t\sqrt{p/c}:\sqrt{pc}-1, t\sqrt{p/c}:(t+1)\sqrt{p/c}-1}, n-tn/c, n/c, p, c$)

$V := \begin{bmatrix} I & 0 \\ 0 & V_t \end{bmatrix} \cdot V$.

$P_{ijk}$ swaps any rows as required by $V_t$ between $(A, S)_{t\sqrt{p/c}+i, m}$ and $(A, S)_{l, m}$ for $l \in \{t\sqrt{p/c}, t\sqrt{p/c}+1, ..., \sqrt{pc}-1\}$ and $m \in \{j, j+\sqrt{p/c}, ..., j+(t-1)\sqrt{p/c}, j+(t+1)\sqrt{p/c}, ..., j+\sqrt{pc}-\sqrt{p/c}\}$.

$P_{ijk}$ contributes $S_{t\sqrt{p/c}+i, m\sqrt{p/c}+j}$ for $m \in \{t+1, t+2, ..., c-1\}$ to a sum all-reduction among all $P_{ijk}$

$A_{t\sqrt{p/c}+i, m\sqrt{p/c}+j} := A_{t\sqrt{p/c}+i, m\sqrt{p/c}+j} - S_{t\sqrt{p/c}+i, m\sqrt{p/c}+j}$

**if** $t + k < c-1$ **then** */* perform $(n/c)$-by-$(n/c)$ block TRSMs */*

`[`$U_{t\sqrt{p/c}+i, (t+k+1)\sqrt{p/c}+j}$`] = 2D-TRSM`$_{ij}$`(`$L_{t\sqrt{p/c}+i, t\sqrt{p/c}+j}$`,` $A_{t\sqrt{p/c}+i, (t+k+1)\sqrt{p/c}+j}$`,k)`

$P_{ijk}$ broadcasts $U_{t\sqrt{p/c}+i, (t+k+1)\sqrt{p/c}+j}$ to $P_{ijk'}$ for $k' \in \{0, 1, ..., c-1\}$

**end**

**if** $t < c-1$ **then** */* perform trailing matrix update */*

**if** $\lfloor k\sqrt{p/c^3} \rfloor \le j < \lfloor (k+1)\sqrt{p/c^3} \rfloor$ **then**

$P_{ijk}$ broadcasts $L_{m\sqrt{p/c}+i, t\sqrt{p/c}+j}$ for each $m \in \{t+1, t+2, ..., c-1\}$ to each $P_{ij'k}$ for $j' \in \{0, 1, ..., \sqrt{p/c}-1\}$

**end**

**if** $\lfloor k\sqrt{p/c^3} \rfloor \le i < \lfloor (k+1)\sqrt{p/c^3} \rfloor$ **then**

$P_{ijk}$ broadcasts $U_{t\sqrt{p/c}+i, m\sqrt{p/c}+j}$ for each $m \in \{t+1, t+2, ..., c-1\}$ to each $P_{i'jk}$ for $i' \in \{0, 1, ..., \sqrt{p/c}-1\}$

**end**

**for** $l = t+1$ **to** $c-1$ **do**

**for** $m = t+1$ **to** $c-1$ **do**

**for** $r = \lfloor k\sqrt{p/c^3} \rfloor$ **to** $\lfloor (k+1)\sqrt{p/c^3} \rfloor - 1$ **do**

$S_{lm} := S_{lm} + L_{l\sqrt{p/c}+i, t\sqrt{p/c}+r} \cdot U_{t\sqrt{p/c}+r, m\sqrt{p/c}+j}$

**end**

**end**

**end**

*// reduce horizontal panel of Schur complement and adjust $A$*

$P_{ijk}$ contributes $S_{l\sqrt{p/c}+i, t\sqrt{p/c}+j}$ for $l \in \{t+1, t+2, ..., c-1\}$ to a sum all-reduction among all $P_{ijk}$

$A_{l\sqrt{p/c}+i, t\sqrt{p/c}+j} := A_{l\sqrt{p/c}+i, t\sqrt{p/c}+j} - S_{l\sqrt{p/c}+i, t\sqrt{p/c}+j}$

**end**

**end**

**end**

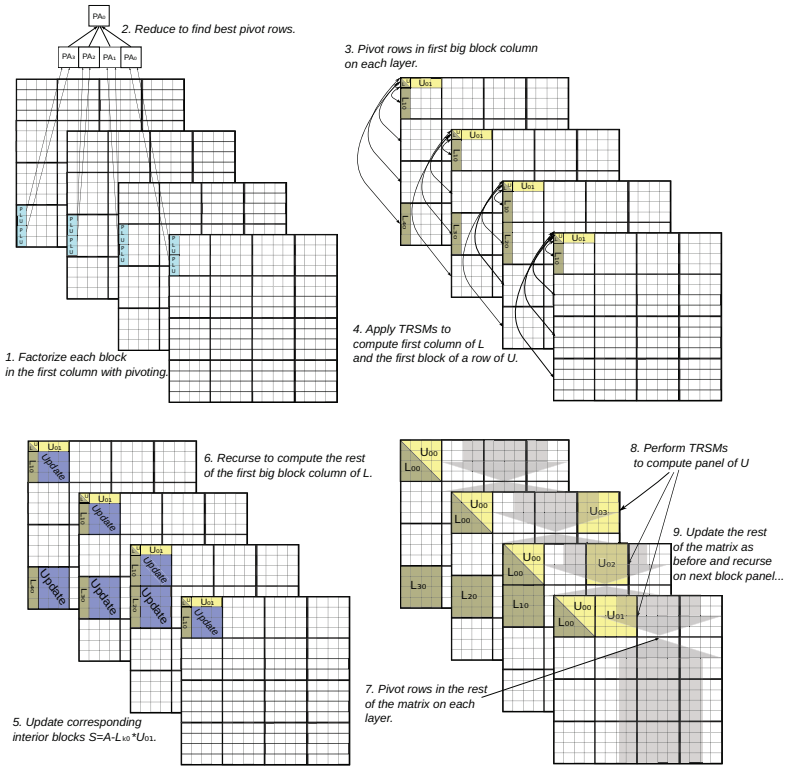



**Fig. 5.** 2.5D LU with pivoting panel factorization (step A in Figure 4).

We implemented 2.5D matrix multiplication using MPI [11] for inter-processor communication and calls to the BLAS for the sequential sub-matrix multiplications. For simplicity, our implementation makes the assumption that the matrices are square and processor counts as well as the matrix dimension are powers of two. We benchmarked versions with blocking communication as well as overlapped (when possible) communication. Both versions were competitive and we report results for the best performing version at any data-point.

We omit comparison with the ScaLAPACK (PBLAS) [4] as it is unfair to compare a general purpose implementation with a very limited one (for the problem sizes tested, the performance of our implementation was strictly and significantly better).

Figure 6 shows weak scaling (static dataset size per processor) results with 32-by-32 sized matrix blocks of 64-bit elements per processor. We used a such a small problem size in order to expose the communication bottlenecks at small processor counts and use less supercomputer resources. Larger problem sizes run at high efficiency when using so few nodes, therefore, inter-processor communication avoidance does not yield significant benefit to overall performance.

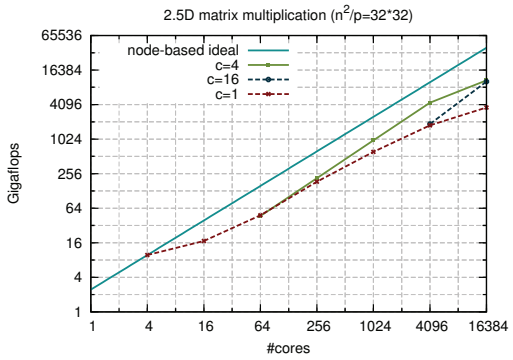



**Fig. 6.** Performance of 2.5D square matrix multiplication with various $c$ on Franklin (XT4).

The results in Figure 6 demonstrate that replicating memory improves the scalability and performance of dense matrix multiplication when many processors are utilized. When using $p = 4,096$ cores, picking $c = 4$ boosts performance by a factor of 2.5 over the 2D version and a factor of 2.4 over the 3D version ($c = 16$). Further, when using $p = 16,384$ cores, we get a factor of 3.0 speed-up over the 2D version, suggesting that even larger speed-ups are possible if more cores/nodes are utilized. We are currently working on implementation of 2.5D LU and more in-depth performance analysis of 2.5D algorithms. However, we believe these preliminary results serve as a strong early indicator of the validity and practicality of the theoretical analysis presented in this paper.

## 8 Exascale performance predictions

We analyze the new 2.5D algorithms and compare them to their 2D counterparts via a performance model. In particular, we look at their performance given an exascale interconnect architecture model.

### 8.1 Predicted architectural parameters

Table 1 details the parameters we use in the model. We avoid looking at intra-node performance because a two-level algorithm would likely be used in practice and our 2.5D algorithms are designed for the highest level (network level). This assumption is a bit flawed because the node-level operations performed are of different size or function depending on the higher level algorithm.

| | |
|---|---|
| *Total flop rate ($f_r$)* | 1E18 flops |
| *Total memory* | 32 PB |
| *Node count ($p$)* | 1,000,000 |
| *Node interconnect bandwidth ($\beta$)* | 100 GB/s |
| *Network latency ($\alpha$)* | 500 ns |

**Table 1.** Estimated architecture characteristics of an exaflop machine

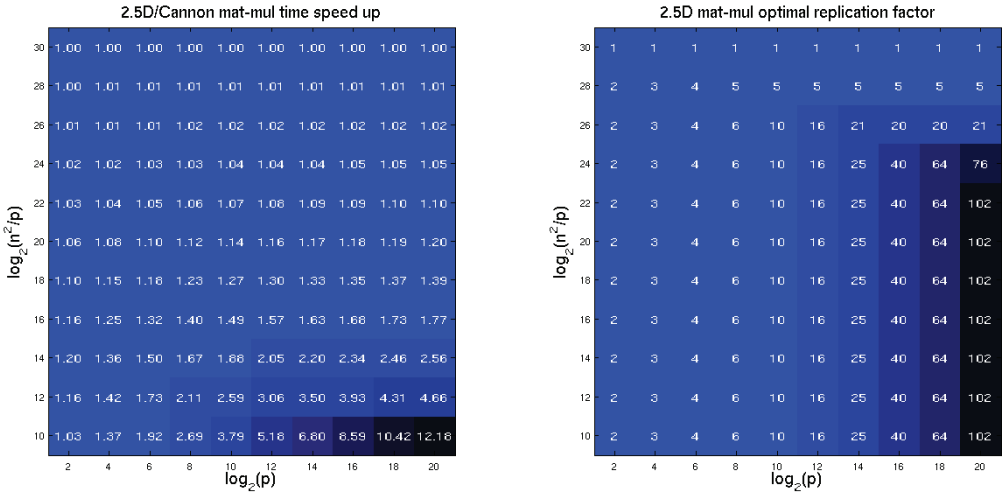



**Fig. 7.** Speed up for best choice of $c$ in comparison to Cannon's algorithm.

### 8.2 Performance estimation

We use the exact costs derived in Appendices A and B to model the performance of the new 2.5D matrix multiplication and LU factorization algorithms. We estimate the bandwidth achieved as 1/3 of the predicted total node interconnect bandwidth. The actual bandwidth achieved on a machine depends heavily on the interconnect topology and implementation of communication collectives. The best current implementations of collectives on 3D torus topologies can achieve the full node bandwidth, but if the broadcasts are single dimensional as they are in our algorithms, at most 1/3 of the total node bandwidth can be achieved [9]. We estimate the execution time by summing the computational, bandwidth, and latency costs

$$\begin{aligned} t &= t_f + t_w + t_s \\ &= F/f_r + \frac{W}{3 \cdot 8 \cdot \beta} + S \cdot \alpha. \end{aligned}$$

We calculate the performance of our algorithms according to this model for different node counts (partitions of the exascale machine) and for different values of $\frac{n^2}{p}$ (weak scaling analysis). We will analyze the algorithm performance in relation to other algorithms rather than as absolute performance since our model is only analyzing inter-node communication.

### 8.3 2.5D matrix multiplication speed-ups

For each data point (choice of $p$ and $\frac{n^2}{p}$) we compare the performance of Cannon's algorithm with the performance achieved using the optimal choice of $c$ (sampled over valid values). Figure 7 displays the factor of speed-up and the corresponding choice of $c$.

First, we notice that the 2.5D algorithm always outperforms Cannon's algorithm since if $c$ is picked to be 1 they are equivalent. When using the full machine on a small problem, we get as much as a factor of 12.18X speed up and $c$ gets up to 102 (101 redundant copies of the matrices are used). The speed-up is most ample when Cannon's algorithm is at low efficiency and is bound by bandwidth or latency.

### 8.4 2.5D LU factorization speed-ups

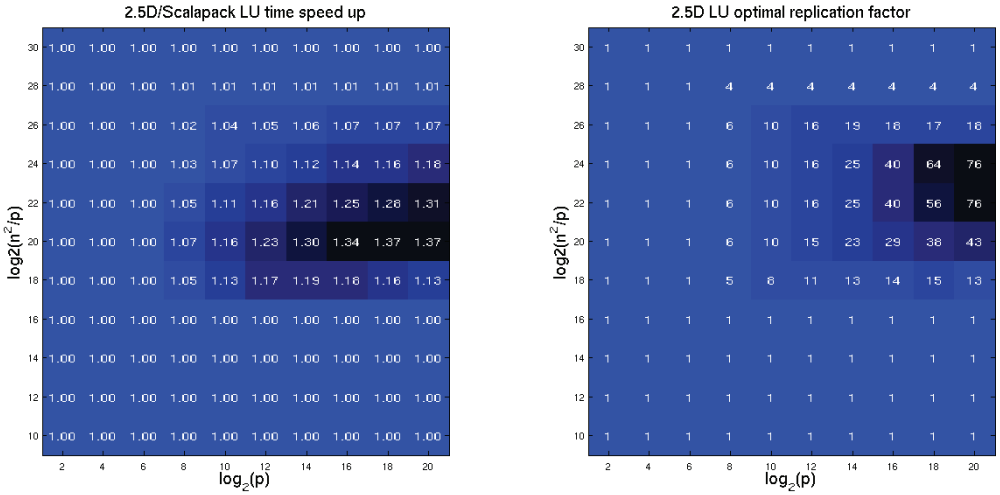



**Fig. 8.** Speed up for best choice of $c$ in comparison to a 2D LU algorithm.

For each data point (choice of $p$ and $\frac{n^2}{p}$) we compare the performance of a 2D LU algorithm with the performance achieved using the optimal choice of $c$ (sampled over valid values) of the new 2.5D LU-factorization algorithm. Figure 8 displays the factor of speed-up and the corresponding choice of $c$.

In this case, we only get a speed-up up of to 1.37X on a mid-size, bandwidth-bound problem. For matrix multiplication, we reduced both the bandwidth and latency costs but for LU we reduced only the bandwidth cost. So the speed-up is not as significant for LU. However, in practice, LU factorization is likely to be more bandwidth bound on a supercomputer and the bandwidth number we are using is quite optimistic. Further, our algorithm can be better mapped to cuboid topologies and, therefore, might be able to operate with significantly less network contention and efficient topology-aware collectives in practice.

## 9 Future work

Preliminary analysis suggests that a 2.5D algorithm for TRSM can be written using a very similar parallel decomposition to what we present in this paper for LU. We will formalize this analysis.

Our 2.5D LU algorithm can also be modified to do Cholesky. Thus, using Cholesky-QR we plan to formulate many other numerical linear algebra operations with minimal communication. As an alternative, we are also looking into adjusting the algorithms for computing QR, eigenvalue decompositions, and the SVD which use Strassen's algorithm [8] to using our 2.5D matrix multiplication algorithm instead. Further, we plan to look for the most efficient and stable 2.5D QR factorization algorithms. In particular, the 2D parallel Householder algorithm for QR has a very similar structure to LU, however, we have not found a way to accumulate Householder updates across layers. The Schur complement updates are subtractions and therefore commute,

however, each step of Householder QR orthogonalizes the remainder of the matrix with the newly computed panel of Q. This orthogonalization is dependent on the matrix remainder and is a multiplication, which means the updates do not commute and therefore seem to be difficult to accumulate.

Finally, we are working with preliminary implementations of 2.5D matrix multiplication and LU factorization. We aim to analyze the performance these algorithms on modern supercomputers and determine how much performance benefit our data replication techniques might yield. In addition to the benefits of communication reduction, our 2.5D algorithms should be able to efficiently utilize topology-aware collective communication operations [9] by matching the algorithmic processor layout to the actual physical network topology.

## 10 Appendix A

In this appendix we derive costs for the 2.5D Matrix multiplication algorithm. We break down the bandwidth/latency/flops ($W/S/F$) costs by the steps of the algorithm

1. We assume the matrices start on one layer of the 2D grid and the other $c-1$ copies need to be created. If $c > 1$ we require two broadcasts for each matrix ($A$, $B$). The costs are

$$W_1 = \frac{2n^2}{p/c}$$
$$S_1 = 2\log(c)$$

2. Shift $A$ and $B$ in each processor layer. All processors send and receive one message so this communication can all be done simultaneously. The costs are

$$W_2 = \frac{2n^2}{p/c}$$
$$S_2 = 2$$

3. Perform $\frac{\sqrt{p/c}}{c} - 1 = \sqrt{p/c^3} - 1$ steps of Cannon's algorithm on each layer. Each step requires a shift of both $A$ and $B$, so the costs over all iterations are

$$W_3 = \frac{2n^2}{p/c} \cdot (\sqrt{p/c^3} - 1)$$
$$= \frac{2n^2}{\sqrt{pc}} - \frac{2cn^2}{p}$$
$$S_3 = 2(\sqrt{p/c^3} - 1)$$

4. If $c > 1$, we reduce the contributions to $C$ between processor layers. This single reduction costs

$$W_4 = \frac{n^2}{p/c}$$
$$S_4 = \log(c)$$

No extra computation is done by this algorithm. We can calculate the total communication costs by summing over all previously enumerated steps,

$$
\begin{aligned}
W_{2.5D\_MM} &= W_1 + W_2 + W_3 + W_4 \\
&= \frac{2n^2}{p/c} + \frac{2n^2}{p/c} + \frac{2n^2}{\sqrt{pc}} - \frac{2cn^2}{p} + \frac{n^2}{p/c} \\
&= \frac{3n^2}{p/c} + \frac{2n^2}{\sqrt{pc}} \\
&\approx \frac{2n^2}{\sqrt{pc}} \\
&= O\left(\frac{n^2}{\sqrt{pc}}\right) \\
S_{2.5D\_MM} &= S_1 + S_2 + S_3 + S_4 \\
&= 2\log(c) + 2 + 2(\sqrt{p/c^3} - 1) + \log(c) \\
&= 3\log(c) + 2\sqrt{p/c^3} \\
&= O\left(\sqrt{p/c^3} + \log(c)\right).
\end{aligned}
$$

So, the number of words moved is optimal. The number of messages has an extra $\log(p)$ term but is otherwise optimal (the $\log(p)$ term probably can't be eliminated). If we pay attention to the fact that no broadcast/reduction needs to be done in the case of $c = 1$, the costs are also the same as for Cannon's algorithm.

## 11 Appendix B

Here we derive precise bandwidth/latency/flop ($W/S/F$) costs for the 2.5D LU algorithm. For generality, the small block size will now be $\frac{n}{r\sqrt{pc}}$, where $r$ is some small constant blocking parameter.

1. Factorize the top big block redundantly on each active layer. This is a 2D LU on a matrix of size $n/c$-by-$n/c$ on a $\sqrt{p/c}$-by-$\sqrt{p/c}$ processor grid with a block-cyclic distribution defined by $r$. The costs over $c$ iterations are

$$
\begin{aligned}
F_1 &= \frac{(2/3)(n/c)^3}{p/c} \cdot c \\
&= \frac{(2/3)n^3}{cp} \\
W_1 &= \frac{3(n/c)^2}{\sqrt{p/c}} \cdot c \\
&= \frac{3n^2}{\sqrt{pc}} \\
S_1 &= 4r\sqrt{p/c}\log(\sqrt{p/c}) \cdot c \\
&= 2r\sqrt{pc}\log(p/c)
\end{aligned}
$$

2. Perform a different big block TRSM on different layers. This is a 2D TRSM of size $n/c$-by-$n/c$ on a $\sqrt{p/c}$-by-$\sqrt{p/c}$ processor grid. We do not perform this TRSM at the last iteration. The costs over $c$ iterations are

$$
\begin{aligned}
F_2 &= \frac{(n/c)^3}{p/c} \cdot (c-1) \\
&= \frac{n^3}{cp} - \frac{n^3}{pc^2} \\
W_2 &= \frac{(n/c)^2}{\sqrt{p/c}} \cdot (c-1) \\
&= \frac{n^2}{\sqrt{pc}} - \frac{n^2}{p^{1/2}c^{3/2}} \\
S_2 &= 2r\sqrt{p/c}\log(\sqrt{p/c}) \cdot (c-1) \\
&= r\sqrt{pc}\log(p/c) - r\sqrt{p/c}\log(p/c)
\end{aligned}
$$

3. Broadcast the big blocks of $L$ and $U$ we just computed via TRSMs so that each layer gets the entire panels. All processors are involved in each big block broadcast so we can do them one by one efficiently. We don't need to do this at the last iteration. So, over $c$ iterations the costs are

$$
\begin{aligned}
W_3 &= \sum_{i=1}^{c} (n/\sqrt{pc})^2 \cdot 2(i-1) \\
&= \frac{n^2 c}{p} - \frac{n^2}{p} \\
S_3 &= \sum_{i=1}^{c} \log(\sqrt{p/c}) \cdot 2(i-1) \\
&= (1/2)c^2\log(p/c) - (1/2)c\log(p/c)
\end{aligned}
$$

   Its worthy to note that these communication costs are slightly excessive for the required task. For one, we can be smarter and interleave the big block broadcasts to reduce the latency cost. Also, we don't actually need the entire panels on each layer but rather different sub-panels. We use this approach for brevity and because it has the nice property that all layers end up with the solution.
4. At iteration $i$ for $i \in \{1, 2, \ldots, c-1\}$, broadcast a sub-panel of $L$ of size $(n/c^2)$-by-$(n - in/c)$ and a sub-panel of $U$ of size $(n - in/c)$-by-$(n/c^2)$ along a single dimension of each 2D processor layer. So for both $L$ and $U$ we are doing $(c-i)r\sqrt{p/c}$ broadcasts of $r\sqrt{p/c^3}$ small blocks. The blocks are distributed cyclically among $\sqrt{p/c}$ sets of $\sqrt{p/c^3}$ processors and each of these processor sets communicates with a disjoint set of processors. So each processor can combine the broadcasts and broadcasts can be done simultaneously among rows for $L$ and columns for $U$. Thus, since each disjoint broadcasts is of size $\frac{(c-i)n^2}{pc}$, the total communication costs is

$$
\begin{aligned}
W_4 &= \sum_{i=1}^{c-1} \frac{2(c-i)n^2}{cp} \cdot \sqrt{p/c^3} \\
&= \frac{n^2}{\sqrt{cp}} - \frac{n^2}{\sqrt{c^3 p}} \\
S_4 &= 2\log(\sqrt{p/c}) \cdot \sqrt{p/c^3} \cdot (c-1) \\
&= \sqrt{p/c}\log(p/c) - \sqrt{p/c^3}\log(p/c)
\end{aligned}
$$

5. Multiply the panels we just broadcasted in blocks to compute the Schur complement distributed over all layers and processors. We already distributed the matrices and no more inter-processor communication is required for this step. At iteration $i$, each processor needs to compute $r\sqrt{p/c^3} \cdot (r(c-i))^2$ small block multiplications. The computational cost over all iterations is

$$
\begin{aligned}
F_5 &= \sum_{i=1}^{c-1} r\sqrt{p/c^3} \cdot (r(c-i))^2 \cdot 2\left(\frac{n}{r\sqrt{pc}}\right)^3 \\
&= \frac{2n^3(c-1)^3}{3c^3p} \\
&= \frac{2n^3}{3p} - \frac{2n^3}{3cp}
\end{aligned}
$$

6. Reduce side and top panels of $A$. At iteration $i$, the panels we are reducing are of size $(n - in/c)$-by-$(n/c)$ and $(n/c)$-by-$(n - (i+1)n/c)$. All processors are involved in the panel reduction. Each processor is responsible for reducing $2(c-i)$ small blocks but each of these reductions involves the same set of processors and can be combined into one bigger one. Thus, the communication costs are

$$
\begin{aligned}
W_6 &= \sum_{i=1}^{c-1} 2(c-i-1) \cdot (n/\sqrt{pc})^2 \\
&= \frac{cn^2}{p} - \frac{n^2}{p} \\
S_6 &= \log(c) \cdot (c-1) \\
&= c\log(c) - \log(c)
\end{aligned}
$$

We can sum the above costs to derive the total cost of the algorithm.

$$
\begin{aligned}
F_{2.5D\_LU} &= F_1 + F_2 + F_5 \\
&= \frac{2n^3}{3p} - \frac{2n^3}{3cp} + \frac{n^3}{cp} - \frac{n^3}{pc^2} + \frac{(2/3)n^3}{cp} \\
&= \frac{2n^3}{3p} + \frac{n^3}{cp} - \frac{n^3}{pc^2} \\
&\approx \frac{2n^3}{3p} \\
&= O\left(\frac{n^3}{p}\right) \\
W_{2.5D\_LU} &= W_1 + W_2 + W_3 + W_4 + W_6 \\
&= \frac{3n^2}{\sqrt{pc}} + \frac{n^2}{\sqrt{pc}} - \frac{n^2}{p^{1/2}c^{3/2}} + \frac{n^2 c}{p} - \frac{n^2}{p} + \frac{n^2}{\sqrt{cp}} - \frac{n^2}{\sqrt{c^3 p}} + \frac{cn^2}{p} - \frac{n^2}{p} \\
&= \frac{5n^2}{\sqrt{pc}} - \frac{n^2}{p^{1/2}c^{3/2}} + \frac{2cn^2}{p} - \frac{2n^2}{p} - \frac{n^2}{\sqrt{c^3 p}} \\
&\approx \frac{5n^2}{\sqrt{pc}} + \frac{2cn^2}{p} \\
&= O\left(\frac{n^2}{\sqrt{pc}}\right) \\
S_{2.5D\_LU} &= S_1 + S_2 + S_3 + S_4 + S_6 \\
&= 2r\sqrt{pc}\log(p/c) + r\sqrt{pc}\log(p/c) - r\sqrt{p/c}\log(p/c) + (1/2)c^2\log(p/c) \\
&\quad - (1/2)c\log(p/c) + \sqrt{p/c}\log(p/c) - \sqrt{p/c^3}\log(p/c) + c\log(c) - \log(c) \\
&= 3r\sqrt{pc}\log(p/c) + \sqrt{p/c}\log(p/c) + (1/2)c^2\log(p/c) + c\log(c) \\
&\quad - \log(c) - r\sqrt{p/c}\log(p/c) - (1/2)c\log(p/c) - \sqrt{p/c^3}\log(p/c) \\
&\approx 3r\sqrt{pc}\log(p/c) + (1/2)c^2\log(p/c) \\
&= O(r\sqrt{pc}\log(p/c)).
\end{aligned}
$$

All of these costs are asymptotically optimal according to our lower bounds (with the exception of the latency cost which, if we set $r = 1$ is a factor of $\log(p/c)$ higher than the lower bound but still likely optimal). The costs also reduce to the optimal 2D LU costs when $c = 1$.

## 12 Appendix C

Here we derive precise bandwidth/latency/flop ($W/S/F$) costs for the 2.5D LU with CA-pivoting algorithm. The algorithm does a superset of the operations of the no-pivoting 2.5D LU algorithm. We will not re-derive all the steps but rather only the extra steps. For generality, the small block size will now be $\frac{n}{r\sqrt{pc}}$, where $r$ is some small constant blocking parameter.

1. Perform CA-pivoting to determine pivots for each small block column. Every processor layer finds the best rows within a big block, then we reduce to find the best rows over the entire small block column. If we do a binary reduction and perform each $\left(\frac{2n}{r\sqrt{pc}}\right)$-by-$\left(\frac{n}{r\sqrt{pc}}\right)$ factorization sequentially on some processor, there are $\log(\frac{r\sqrt{pc}}{2})$ factorizations along the critical path. So the costs over all iterations are

$$
\begin{aligned}
F_1 &= \sum_{i=1}^{r\sqrt{pc}} (5/3) \left(\frac{n}{r\sqrt{pc}}\right)^3 \log(i) \\
&\leq \frac{(5/3)n^3 \log(r\sqrt{pc})}{r^2 pc} \\
W_1 &= \sum_{i=1}^{r\sqrt{pc}} \left(\frac{n}{r\sqrt{pc}}\right)^2 \log(i/r) \\
&\leq \frac{(1/2)n^2 \log(pc)}{r\sqrt{pc}} \\
S_1 &\leq (1/2) r\sqrt{pc} \log(pc)
\end{aligned}
$$

If $r$ is a constant, the flops cost is suboptimal by a factor of $\Theta(\log(p))$ when $c = 1$ and the bandwidth cost is suboptimal by a factor of $\Theta(\log(p))$ for any $c$, however they can be brought down by raising $r = \Omega(\log(p))$ at the expense of latency. In fact, 2D LU with CA-pivoting has to sacrifice a factor of $\Theta(\log(p))$ in flops or in latency so this is to be expected. We actually get an advantage because the the flops cost of this step has an extra factor of $c$ in the denominator.

2. Once the correct pivot rows are determined we broadcast the pivots to all processors as they may all be involved in the pivoting. Over all iterations this costs

$$
\begin{aligned}
W_2 &= \left(\frac{n}{r\sqrt{pc}}\right)^2 \cdot r\sqrt{pc} \\
&= \frac{n^2}{r\sqrt{pc}} \\
S_2 &= \log(p) \cdot r\sqrt{pc} \\
&= r\sqrt{pc}\log(p)
\end{aligned}
$$

3. After calculating the pivots and broadcasting them for a single small-block column, we pivot within the entire panel. This is really a many-to-many communication since the pivot rows are distributed over all layers and every layer needs all the rows. The communication is disjoint among columns of processors over all layers. However, we will write it as a gather within each layer and an all-reduce across layers. So, we gather the pivot rows local to each layer and then we implement the all-gather as an all-reduction (the reduction concatenates contributions). Over all iterations, the costs are

$$
\begin{aligned}
W_3 &\leq 2\left(\frac{n}{r\sqrt{pc}}\right)^2 \cdot r\sqrt{pc} \\
&\leq \frac{2n^2}{r\sqrt{pc}} \\
S_3 &\leq (\log(\sqrt{p/c}) + \log(c)) \cdot r\sqrt{pc} \\
&\leq (1/2) r\sqrt{pc}\log(pc)
\end{aligned}
$$

4. After we finish factorizing the entire big block panel we must pivot the rest of the rows redundantly on each layer. In the worst case, we need to collect an entire panel. The communication is disjoint between processor columns. The communication pattern is essentially a partial transpose among the processors in each column.

   We know that no processor will receive more row blocks than it partially owns within then panel. On average, each processor will also send no more blocks than it owns within the panel. However, in the worst case, a single processor may have to pivot all of the rows it owns to the panel. For any realistic matrix the pivot rows should be distributed randomly among the processors (especially since the distribution

is block-cyclic). So, applying a simple probabilistic argument (balls in bins), we argue that no processor should contribute more than a factor of $\log(\sqrt{p/c})$ pivot rows than the average. Given this assumption, the costs are

$$\begin{aligned} W_4 &\leq \sum_{i=1}^{c} (c-1) \cdot (n/\sqrt{pc})^2 \log(\sqrt{p/c}) \\ &\leq \sum_{i=1}^{c} (c-1) \cdot \frac{n^2}{pc} \log(\sqrt{p/c}) \\ &\leq \frac{cn^2}{2p} \log(p/c) \\ S_4 &\leq (c/2) \log(p/c) \end{aligned}$$

Further, we argue that the indices of pivot rows for most realistic matrices can be treated as independent random events. In this case, we can apply a Chernoff bound to say that for large enough matrices, no processor should contribute more than a small constant factor, $\epsilon$, of pivot rows than the average, with high probability $(1 - e^{-\frac{\epsilon^2 n^2}{pc}})$. Under these assumptions, the bandwidth cost goes down to

$$\begin{aligned} W_4 &\leq \sum_{i=1}^{c} (c-1) \cdot (n/\sqrt{pc})^2 (1+\epsilon) \\ &\approx \sum_{i=1}^{c} c \cdot \frac{n^2}{pc} \\ &\leq \frac{cn^2}{p} \end{aligned}$$

However, in the worst case (which seems very impractical), the bandwidth cost here is

$$\begin{aligned} W_4' &\leq \sum_{i=1}^{c} (c-i) \cdot c(n/\sqrt{pc})^2 \\ &\leq \frac{c^2 n^2}{2p}. \end{aligned}$$

5. We also have to account for the fact that we need to postpone the reduction of the top panel. The amount of bandwidth we need stays the same since we move the same amount of data, however, the latency cost for this step doubles. So we must send double the messages for this step, for an extra latency cost of

$$S_5 = c \log(c) - \log(c)$$

We can sum the above costs and add them to the 2D LU costs to derive the total cost of the algorithm.

$$
\begin{aligned}
F_{2.5D\_LU\_P} &= F_{2.5D\_LU} + F_1 \\
&\leq F_{2.5D\_LU} + \frac{(5/3)n^3 \log(r\sqrt{pc})}{r^2 pc} \\
&\approx \frac{2n^3}{3p} + \frac{(5/6)n^3 \log(pc)}{r^2 pc} \\
&= O\left(\frac{n^3}{p} + \frac{n^3 \log(p)}{r^2 pc}\right) \\
W_{2.5D\_LU\_P} &= W_{2.5D\_LU} + W_1 + W_2 + W_3 + W_4 \\
&\leq W_{2.5D\_LU} + \frac{(1/2)n^2 \log(pc)}{r\sqrt{pc}} + \frac{n^2}{r\sqrt{pc}} + \frac{2n^2}{r\sqrt{pc}} + \frac{cn^2}{p} \\
&\leq W_{2.5D\_LU} + \frac{(1/2)n^2 \log(pc)}{r\sqrt{pc}} + \frac{3n^2}{r\sqrt{pc}} + \frac{cn^2}{p} \\
&\approx \frac{5n^2}{\sqrt{pc}} + \frac{3cn^2}{p} + \frac{(1/2)n^2 \log(pc)}{r\sqrt{pc}} + \frac{3n^2}{r\sqrt{pc}} \\
&= O\left(\frac{n^2 \log(p)}{r\sqrt{pc}} + \frac{n^2}{\sqrt{pc}}\right) \\
S_{2.5D\_LU\_P} &= S_{2.5D\_LU} + S_1 + S_2 + S_3 + S_4 + S_5 \\
&= S_{2.5D\_LU} + (1/2)r\sqrt{pc}\log(pc) + r\sqrt{pc}\log(p) \\
&\quad + (1/2)r\sqrt{pc}\log(pc) + (c/2)\log(p/c) + c\log(c) - \log(c) \\
&\approx S_{2.5D\_LU} + 2r\sqrt{pc}\log(pc) \\
&\approx 3r\sqrt{pc}\log(p/c) + (1/2)c^2\log(p/c) + 2r\sqrt{pc}\log(pc) \\
&\approx 5r\sqrt{pc}\log(p/c) + (1/2)c^2\log(p/c) \\
&= O\left(r\sqrt{pc}\log(p)\right)
\end{aligned}
$$

All of these costs are within $O(\log(p))$ of the lower bound if $r$ is a constant. However, if we pick $r = \Omega(\log(p))$ the computational cost and the bandwidth costs become optimal, while the latency cost is suboptimal by a factor of $\Theta(\log^2(p))$. However, a probabilistic argument was required for the bandwidth cost to reach this bound.

## References


1. Agarwal, R.C., Balle, S.M., Gustavson, F.G., Joshi, M., Palkar, P.: A three-dimensional approach to parallel matrix multiplication. IBM J. Res. Dev. 39, 575–582 (September 1995)
2. Aggarwal, A., Chandra, A.K., Snir, M.: Communication complexity of PRAMs. Theoretical Computer Science 71(1), 3 – 28 (1990)
3. Ballard, G., Demmel, J., Holtz, O., Schwartz, O.: Minimizing communication in linear algebra (2010)
4. Blackford, L.S., Choi, J., Cleary, A., D'Azeuedo, E., Demmel, J., Dhillon, I., Hammarling, S., Henry, G., Petitet, A., Stanley, K., Walker, D., Whaley, R.C.: ScaLAPACK user's guide. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA (1997)
5. Cannon, L.E.: A cellular computer to implement the Kalman filter algorithm. Ph.D. thesis, Bozeman, MT, USA (1969)
6. Dekel, E., Nassimi, D., Sahni, S.: Parallel matrix and graph algorithms. SIAM Journal on Computing 10(4), 657–675 (1981)
7. Demmel, J., Grigori, L., Xiang, H.: A Communication Optimal LU Factorization Algorithm. EECS Technical Report EECS-2010-29, UC Berkeley (March 2010)
8. Demmel, J., Dumitriu, I., Holtz, O.: Fast linear algebra is stable. Numerische Mathematik 108, 59–91 (2007)
9. Faraj, A., Kumar, S., Smith, B., Mamidala, A., Gunnels, J.: MPI collective communications on the Blue Gene/P supercomputer: Algorithms and optimizations. In: High Performance Interconnects, 2009. HOTI 2009. 17th IEEE Symposium on. pp. 63 –72 (2009)

10. Grigori, L., Demmel, J.W., Xiang, H.: Communication avoiding Gaussian elimination. In: Proceedings of the 2008 ACM/IEEE conference on Supercomputing. pp. 29:1–29:12. SC '08, IEEE Press, Piscataway, NJ, USA (2008)
11. Gropp, W., Lusk, E., Skjellum, A.: Using MPI: portable parallel programming with the message-passing interface. MIT Press, Cambridge, MA, USA (1994)
12. Irony, D., Toledo, S.: Trading replication for communication in parallel distributed-memory dense solvers. Parallel Processing Letters 71, 3–28 (2002)
13. Irony, D., Toledo, S., Tiskin, A.: Communication lower bounds for distributed-memory matrix multiplication. Journal of Parallel and Distributed Computing 64(9), 1017 – 1026 (2004)
14. McColl, W.F., Tiskin, A.: Memory-efficient matrix multiplication in the BSP model. Algorithmica 24, 287–297 (1999)