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

Edgar Solomonik
James Demmel

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2011-72
http://www.eecs.berkeley.edu/Pubs/TechRpts/2011/EECS-2011-72.html

June 7, 2011

Copyright © 2011, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

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. Extra memory allows parallel matrix multiplication to be done with asymptotically less communication than Cannon's algorithm and be faster in practice. "3D" algorithms arrange the p processors in a 3D array, and store redundant copies of the matrices on each of $p^{1/3}$ layers. '2D" algorithms such as Cannon's algorithm store 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, modulo polylog(p) factors. We introduce a novel algorithm for 2.5D LU decomposition. To the best of our knowledge, this LU algorithm is the first to minimize communication along the critical path of execution in the 3D case. Our 2.5D LU algorithm uses communication-avoiding pivoting, a stable alternative to partial-pivoting. 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. We provide implementations and performance results for 2D and 2.5D versions of all the new algorithms. Our results demonstrate that 2.5D matrix multiplication and LU algorithms strongly scale more efficiently than 2D algorithms. Each of our 2.5D algorithms performs over 2X faster than the corresponding 2D algorithm for certain problem sizes on 65,536 cores of a BG/P supercomputer.

1 Introduction

Goals of parallelization include minimizing communication, balancing the work load, and reducing the memory footprint. 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, tries to balance the load, and minimize communication 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 utilize any available amount of extra memory beyond the memory needed to store one distributed copy of the input and output. 2.5D algorithms use this extra memory 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, the memory footprint, is the maximum amount of memory, in words, utilized by any 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 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. Most algorithms assume that the amount of available memory, M, is enough for one copy of the input/output matrices to be evenly spread across all p processors (so $M \approx 3n^2/p$). If this is the case, it is known that Cannon's Algorithm [7] 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 $S = \Theta(p^{1/2})$) [15, 5]. We call Cannon's algorithm a "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. "3D algorithms" assume the amount of available memory, M, is enough for $p^{1/3}$ copies of the input/output matrices to be evenly spread across all p processors (so $M \approx 3n^2/p^{2/3}$). Given this much memory, it is known that algorithms presented in [8, 1, 2, 16] 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 $S = \Theta(\log p)$) [15, 5]. These algorithms are called "3D" because they are 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 present a new matrix multiplication algorithm that uses $M \approx 3cn^2/p$ memory for $c \in \{1, 2, ..., \lfloor p^{1/3} \rfloor\}$, sends $c^{1/2}$ times fewer words than the 2D (Cannon's) algorithm, and sends $c^{3/2}$ times fewer messages than Cannon's algorithm. 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. Our 2.5D matrix multiplication algorithm attains lower bounds (modulo polylog(p) factors) on the number of words and messages communicated.
   Our implementation of 2.5D matrix multiplication achieves better strong scaling and efficiency than Cannon's algorithm and ScaLAPACK's PDGEMM [6]. On 2048 nodes of BG/P, our 2.5D algorithm multiplies square matrices of size $n = 65,536$ 5.3X faster than PDGEMM and 1.2X faster than Cannon's algorithm. On 16,384 nodes of BG/P, our 2.5D algorithm multiplies a small square matrix ($n = 8192$), 2.6X faster than Cannon's algorithm.
2. We present a 2.5D LU algorithm that also reduces the number of words moved by a factor of $c^{1/2}$ in comparison with standard 2D LU algorithms. 2.5D LU attains the same lower bound on the number of words moved as 2.5D matrix multiplication Our 2.5D LU algorithm uses *tournament pivoting* as opposed to partial pivoting [9, 12]. 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.
   We present 2.5D LU implementations without pivoting and with CA-pivoting. Our results demonstrate that 2.5D LU reduces communication and runs more efficiently than 2D LU or ScaLAPACK's PDGETRF [6]. For an LU factorization of a square matrix of size $n = 65,536$, on 2048 nodes of BG/P, 2.5D LU with CA-pivoting is 3.4X faster than PDGETRF with partial pivoting. Further, on 16384 nodes of BG/P, 2.5D LU without pivoting and with CA-pivoting are over 2X faster than their 2D counterparts.
3. 2.5D LU does not, however, send fewer messages than 2D LU; instead it sends a factor of $c^{1/2}$ *more* messages. Under minor assumptions on the algorithm, we demonstrate an inverse relationship among the latency and bandwidth costs of any LU algorithms. This relation yields a lower bound on the latency cost of an LU algorithm with a given bandwidth cost. We 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 CA-pivoting 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 [5]. 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 (e.g. cache size or size of memory space local to processor) the lower bound on communication bandwidth is

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

words, and the lower bound on latency is

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

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:

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

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. As explained in [5], conventional algorithms, for example those in ScaLAPACK [6], mostly do not attain both these lower bounds, so it is of interest to find new algorithms that do.

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

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

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

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

This cost arises from the row and column dependencies of dense matrix-multiplication-like problems. Information from a block row or block column of can only be propagated to one processor with $\Omega(\log p)$ messages.

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 ∈ {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 · B_jk
    P_ijk contributes C_ijk to a sum-reduction to P_i0k
end
```

3D matrix multiplication. For matrix multiplication, Algorithm 1 [8, 1, 2, 16] 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, if necessary, a broadcast to spread the input. So the bandwidth cost is $W = O\left(\frac{n^2}{p^{2/3}}\right)$, 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 [18] 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 later sections.

Previous work on 3D LU factorization. Irony and Toledo [14] 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.

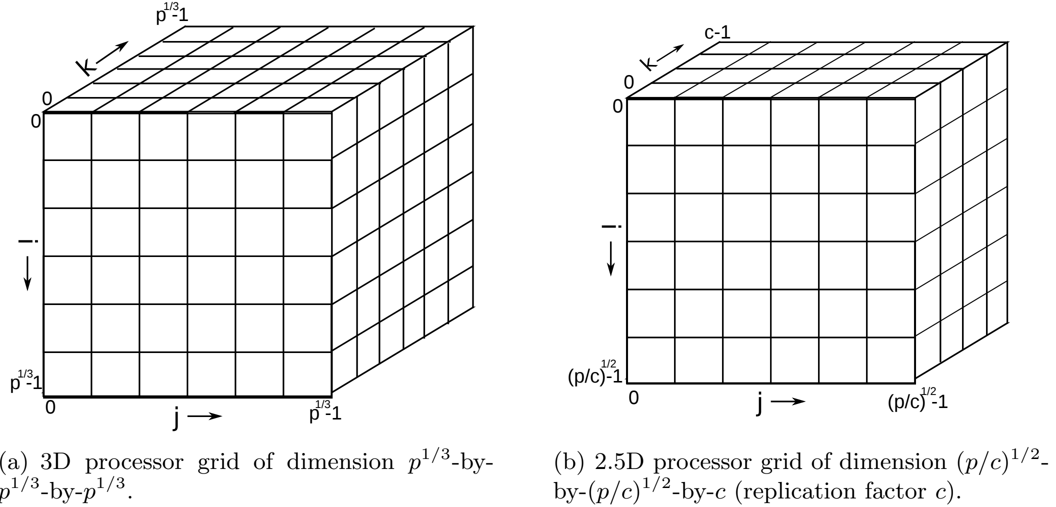
Ashcraft [4, 3] suggested that total communication volume can be reduced for LU and Cholesky via the use of *aggregate data*. Aggregate data is a partial sum of updates, rather than simply the matrix entries. Our 2.5D LU algorithm uses aggregate data to reduce communication by the amount Ashcraft predicted.

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. 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

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

The lower bound in Section 6 of [5] is valid while $c <= p^{1/3}$. When $c = p^{1/3}$, the latency lower bound is trivial, $\Omega(1)$ messages, and the bandwidth lower bound is $\Omega(n^2/p^{2/3})$ words. If the initial data is not replicated, we



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

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

Fig. 1.

claim the $\Omega(n^2/p^{2/3})$ bandwidth lower bound also holds for $c > p^{1/3}$. A total of $\Omega(cn^2 - n^2) = \Omega(cn^2)$ words must be communicated to produce the replicated copies without local entry duplicates. Therefore, some processor must communicate $\Omega(cn^2/p)$ words. When $c > p^{1/3}$, this replication bandwidth cost is bound from below by $cn^2/p = \Omega(n^2/p^{2/3})$ words.

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 $c = p^{1/3}$.

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 each set of copies of matrices A and B, then combines the results. To make this possible, we adjust the initial shift done by Cannon's algorithm to be different for each set of copies of matrices A and B.

Our 2.5D algorithm doesn't quite generalize Algorithm 1 since C is reduced in a different dimension and shifted initially. 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 costs are in Appendix A in [19].

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. Suppose we want to multiply $n \times n$ matrices, and the maximum memory available per processor is $M_{\max}$. Then we need to use at least $p_{\min} = \Theta(n^2/M_{\max})$ processors to store one copy of the matrices. The 2D algorithm uses only one copy of the matrix and has a bandwidth cost of $W_{p_{\min}} = O(n^2/\sqrt{p_{\min}})$ words and latency cost of $S_{p_{\min}} = O(\sqrt{p_{\min}})$ messages. If we use $p = c \cdot p_{\min}$ processors, with a total available memory of $p \cdot M_{\max} = c \cdot p_{\min} \cdot M_{\max}$, we can afford to have c copies of the matrices. The 2.5D algorithm can store a matrix copy on each of c layers of the p processors. Utilizing c copies reduces the bandwidth cost to $W_p = O(n^2/\sqrt{cp}) = O(n^2/(c\sqrt{p_{\min}})) = O(W_{p_{\min}}/c)$ words, and the latency cost to $S_p = O(\sqrt{p/c^3}) = O(\sqrt{p_{\min}}/c) = O(S_{p_{\min}}/c)$ messages. This strong scaling is perfect because all three costs (flops, bandwidth and latency) fall by a factor of c. (up to a factor of $c = p^{1/3}$, and ignoring the $\log(c)$ latency term).

Algorithm 2: $[C] = 2.5\text{D-matrix-multply}(A,B,n,p,c)$

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 */
    s := mod (j − i + k√(p/c³), √(p/c))                   /* initial circular shift on A */
    P_ijk sends A_ij to A_local on P_isk
    s' := mod (i − j + k√(p/c³), √(p/c))                  /* initial circular shift on B */
    P_ijk sends B_ij to B_local on P_s'jk
    C_ijk := A_local · B_local
    s := mod (j + 1, √(p/c))
    s' := mod (i + 1, √(p/c))
    for t = 1 to √(p/c³) − 1 do
        P_ijk sends A_local to P_isk                      /* rightwards circular shift on A */
        P_ijk sends B_local to P_s'jk                     /* downwards circular shift on B */
        C_ijk := C_ijk + A_local · B_local
    end
    P_ijk contributes C_ijk to a sum-reduction to P_ij0
end
```

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 bandwidth and flop costs. Now we see that the algorithmic costs are

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

Evidently, if we want to do $O(n^3/p)$ flops we need $k = O\left(\frac{n}{\sqrt{p}}\right)$, 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

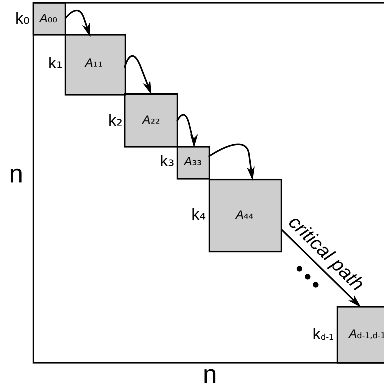


Fig. 2. LU diagonal block dependency path. These blocks must be factorized in order and communication is required between each block factorization.

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

$$W_{lu} = O\left(n^2/\sqrt{cp}\right) \qquad k = O\left(n/\sqrt{cp}\right) \qquad S_{lu} = \Omega(\sqrt{cp}).$$

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.

It is 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 ($c = p^{1/3}$). However, if a 3D LU algorithm minimizes the number of words communicated, it must send $\Omega(p^{2/3})$ messages. 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 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 3) 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-$(n/\sqrt{pc})$. 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: $[L, U] = \text{2.5D-LU-factorization}(A,n,p,c)$

Input: n-by-n matrix A distributed so that for each l, m, (n/c)-by-(n/c) block A_{lm} is spread over P_{ij0} in $(n/\sqrt{pc})$-by-$(n/\sqrt{pc})$ blocks.
Output: triangular n-by-n matrices L, U such that $A = L \cdot U$ and for each l, m, (n/c)-by-(n/c) blocks L_{lm}, U_{lm} are spread over P_{ij0}.
P_{ij0} broadcasts its portion of A to each P_{ijk}
for $t = 0$ **to** $c-1$ **do**
  $[L_{tt}, U_{tt}] = \text{2D-LU}(A_{tt})$ `/* redundantly factorize top right` (n/c)`-by-`(n/c) `block */`
  $[L^T_{t+k+1,t}] = \text{2D-TRSM}(U^T_{tt}, A^T_{t+k+1,t})$ `/* perform TRSMs on` (n/c)`-by-`(n/c) `blocks */`
  $[U_{t,t+k+1}] = \text{2D-TRSM}(L_{tt}, A_{t,t+k+1})$
  P_{ijk} broadcasts its portions of $L_{t+k+1,t}$ and $U_{t,t+k+1}$ to $P_{ijk'}$ for all k' `/* all-gather panels */`
  if $\lfloor k\sqrt{p/c^3}\rfloor \le j < \lfloor (k+1)\sqrt{p/c^3}\rfloor$ **then** `/* broadcast sub-panels of` L `*/`
    P_{ijk} broadcasts its portion of $L_{t+1:c-1,t}$ to each $P_{ij'k}$ for all j'
  end
  if $\lfloor k\sqrt{p/c^3}\rfloor \le i < \lfloor (k+1)\sqrt{p/c^3}\rfloor$ **then** `/* broadcast sub-panels of` U `*/`
    P_{ijk} broadcasts its portion of $U_{t,t+1:c-1}$ to each $P_{i'jk}$ for all i'
  end
  P_{ijk} computes and accumulates its portion of the Schur complement update S `/* multiply sub-panels */`
  All-reduce (sum and subtract from A) $S_{t+1:c-1,t+1}$, $S_{t+1,t+2:c-1}$ `/* reduce next big block panels */`
end

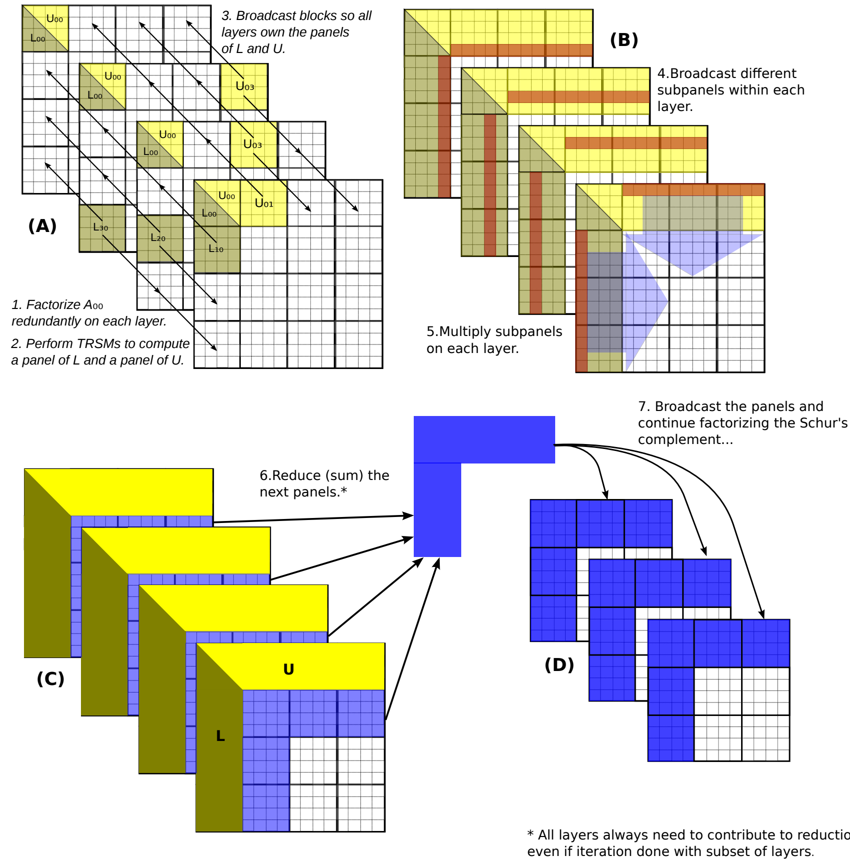


Fig. 3. 2.5D LU algorithm work-flow

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 in [19].

6 2.5D communication optimal LU with pivoting

Regular partial pivoting is not latency optimal because it requires $\Omega(n)$ messages if the matrix is in a 2D blocked layout. $\Omega(n)$ 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) [9], 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 [9, 12].

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 4 demonstrates the workflow 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 in [19].

Algorithm 4: $[V, L, U]$ = 2.5D-TSLU-pivot-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.

for $s = 0$ **to** $\sqrt{p/c} - 1$ **do**
- P_{isk} compute `[`V_s`] = CA-Pivot`$_{k\sqrt{p/c}+i}$`(`$A_{k\sqrt{p/c}+i,s}$`,`n`,`m`)`
- P_{ijk} pivots rows between $A_{k\sqrt{p/c}+i,j}$ and each copy of A_{sj} stored on each P_{sjk} according to V_s
- $A_{ss} := V_s'^T L_{ss} U_{ss}$ `/* factorize top left small block redundantly using GEPP */`
- $U_{sj} := L_{ss}^{-1} V_s' A_{sj}$ for $j > s$ `/* do TRSMs on top small block row redundantly */`
- $L_{is}^T := U_{ss}^{-T} A_{is}^T$ for $i > s$ `/* do TRSMs on the top part of a small block column redundantly */`
- $L_{k\sqrt{p/c}+i,s}^T := U_{ss}^{-T} A_{k\sqrt{p/c}+i,s}^T$ `/* do TRSMs on rest of small block column */`
- P_{isk} broadcasts L_{is} and $L_{k\sqrt{p/c}+i,s}$ to all P_{ijk}
- P_{sjk} broadcasts U_{sj} to all P_{ijk}
- $A_{ij} := A_{ij} - L_{is} \cdot U_{sj}$ for $i, j > s$ `/* update top big block redundantly */`
- $A_{k\sqrt{p/c}+i,j} := A_{k\sqrt{p/c}+i,j} - L_{k\sqrt{p/c}+i,s} \cdot U_{sj}$ for $j > s$ `/* update remaining big blocks */`
- Update V with V_s

end

P_{ijk} broadcasts $L_{k\sqrt{p/c}+i,j}$ to $P_{ijk'}$ for all k'

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

Input: n-by-n matrix A distributed so that for each l, m, (n/c)-by-(n/c) block A_{lm} is spread over P_{ij0} in $(n/\sqrt{pc})$-by-$(n/\sqrt{pc})$ blocks.

Output: n-by-n matrices V and triangular L, U such that $V \cdot A = L \cdot U$ and for each l, m, (n/c)-by-(n/c) blocks L_{lm}, U_{lm} are spread over P_{ij0}.

$S_{1:n,1:n} := 0$ `/* S will hold the accumulated Schur complemented updates to A */`

P_{ij0} broadcasts its portion of A to each P_{ijk}

for $t = 0$ **to** $c - 1$ **do**
- `[`V_t`, `$L_{t:n/c-1,t}$`, `U_{tt}`] = 2.5D-TSLU-pivot-factorization(`$A_{t:c-1,t}$`,`$n - tn/c$`,`n/c`,`p`,`c`)`
- Update V with V_t
- Swaps any rows as required by V_t to $(A, S)_{t,1:c-1}$ `/* pivot remainder of matrix redundantly */`
- All-reduce (sum and subtract from A) $S_{t,t+1:c-1}$ `/* reduce big block top panel */`
- $[U_{t,t+k+1}]$ = 2D-TRSM(L_{tt},$A_{t,t+k+1}$) `/* perform TRSMs on (n/c)-by-(n/c) blocks */`
- P_{ijk} broadcasts its portion of $U_{t,t+k+1}$ to each $P_{ijk'}$ for all k' `/* all-gather top panel */`
- **if** $\lfloor k\sqrt{p/c^3} \rfloor \le j < \lfloor (k+1)\sqrt{p/c^3} \rfloor$ **then** `/* broadcast sub-panels of L */`
  - P_{ijk} broadcasts its portion of $L_{t+1:c-1,t}$ to each $P_{ij'k}$ for all j'
- **end**
- **if** $\lfloor k\sqrt{p/c^3} \rfloor \le i < \lfloor (k+1)\sqrt{p/c^3} \rfloor$ **then** `/* broadcast sub-panels of U */`
  - P_{ijk} broadcasts its portion of $U_{t,t+1:c-1}$ to each $P_{i'jk}$ for all i'
- **end**
- P_{ijk} computes and accumulates its portion the Schur complement update S `/* multiply sub-panels */`
- All-reduce (sum and subtract from A) $S_{t+1:c-1,t+1}$ `/* reduce next big block vertical panel */`

end

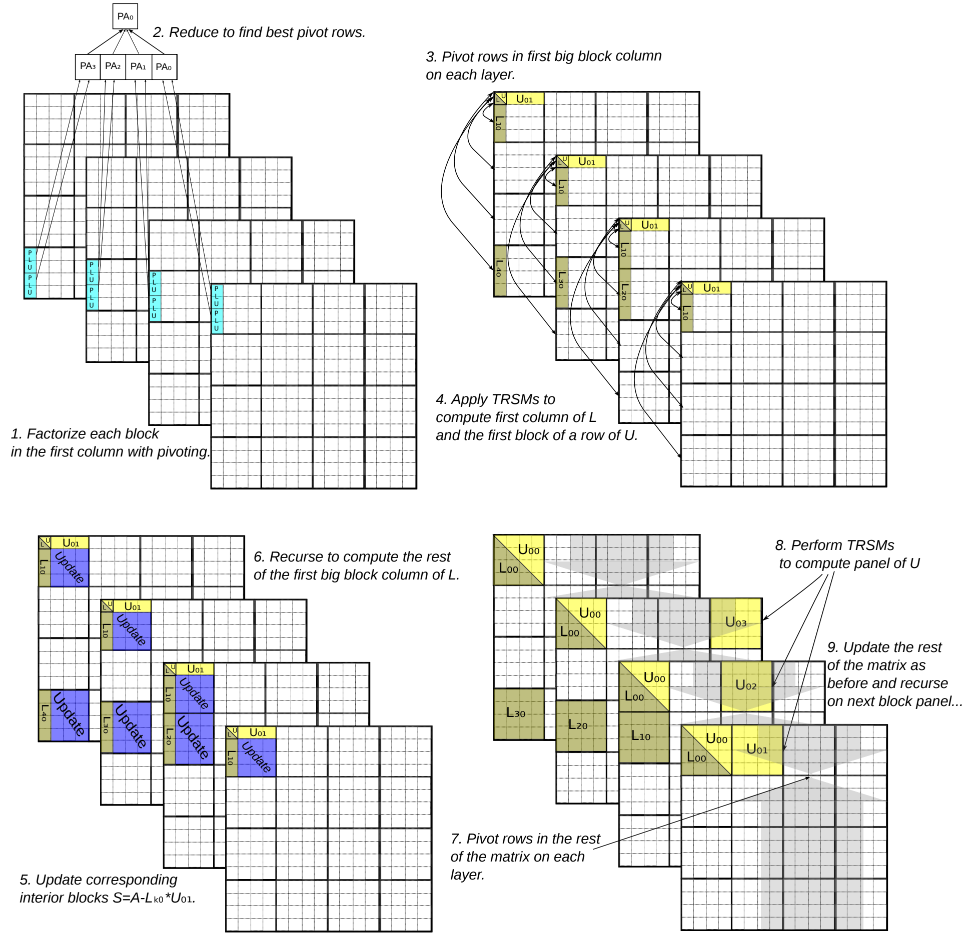


Fig. 4. 2.5D LU with pivoting panel factorization (step A in Figure 3).

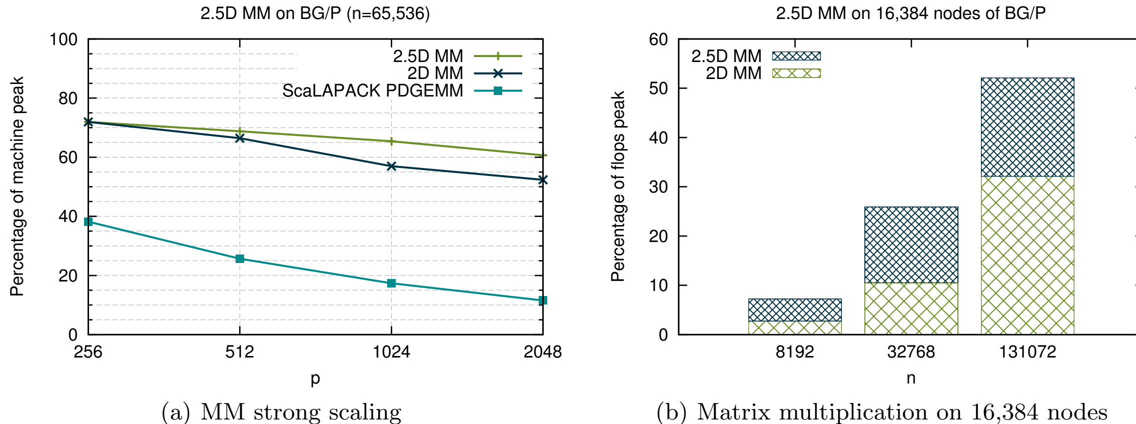
7 Performance results

We implemented 2.5D matrix multiplication and LU factorization using MPI [13] for inter-processor communication. We perform most of the sequential work using BLAS routines: DGEMM for matrix multiplication, DGETRF, DTRSM, DGEMM, for LU. We found it was fastest to use provided multi-threaded BLAS libraries rather than our own threading. All the results presented in this paper use threaded ESSL routines.

We benchmarked our implementations on a Blue Gene/P (BG/P) machine located at Argonne National Laboratory (Intrepid). We chose BG/P as our target platform because it uses few cores per node (four 850 MHz PowerPC processors) and relies heavily on its interconnect (a bidirectional 3D torus with 375 MB/sec of achievable bandwidth per link). On this platform, reducing inter-node communication is vital for performance.

BG/P also provides topology-aware partitions, which 2.5D algorithms are able to exploit. For node counts larger than 16, BG/P allocates 3D cuboid partitions. Since 2.5D algorithms have a parameterized 3D virtual topology, a careful choice of c allows them to map precisely to the allocated partitions (provided enough memory).

Topology-aware mapping can be very beneficial since all communication happens along the three dimensions of the 2.5D virtual topology. Therefore, network contention is minimized or, in certain scenarios, completely eliminated. Topology-aware mapping also allows 2.5D algorithms to utilize optimized line multicast and line reduction collectives provided by the DCMF communication layer [11, 17].



(a) MM strong scaling

(b) Matrix multiplication on 16,384 nodes

Fig. 5. Performance of 2.5D MM on BG/P

We study the strong scaling performance of 2.5D algorithms on a 2048 node partition (Figures 5(a), 6(a), 7(a)). The 2048 node partition is arranged in a 8-by-8-by-32 torus. In order to form square layers, our implementation uses 4 processes per node (1 process per core) and folds these processes into the X dimension. Now, each XZ virtual plane is 32-by-32. We strongly scale 2.5D algorithms from 256 nodes $c = Y = 1$ to 2048 nodes $c = Y = 8$. For ScaLAPACK we use smp or dual mode on these partitions, since it is not topology-aware.

We also compare performance of 2.5D and 2D algorithms on 16,384 nodes (65,536 cores) of BG/P. The 16,384 node partition is a 16-by-32-by-32 torus. We run both 2D and 2.5D algorithms in SMP mode. For 2.5D algorithms, we use $c = 16$ YZ processor layers.

7.1 2.5D matrix multiplication performance

Our 2.5D matrix multiplication implementation is a straight-forward adjustment of Cannon's algorithm. We assume square and correctly padded matrices, as does Cannon's algorithm. A more general 2.5D matrix multiplication algorithm ought to be built on top of a more general 2D algorithm (e.g. the SUMMA algorithm [20]). However, our algorithm and implementation provide an idealistic and easily reproducible proof of concept.

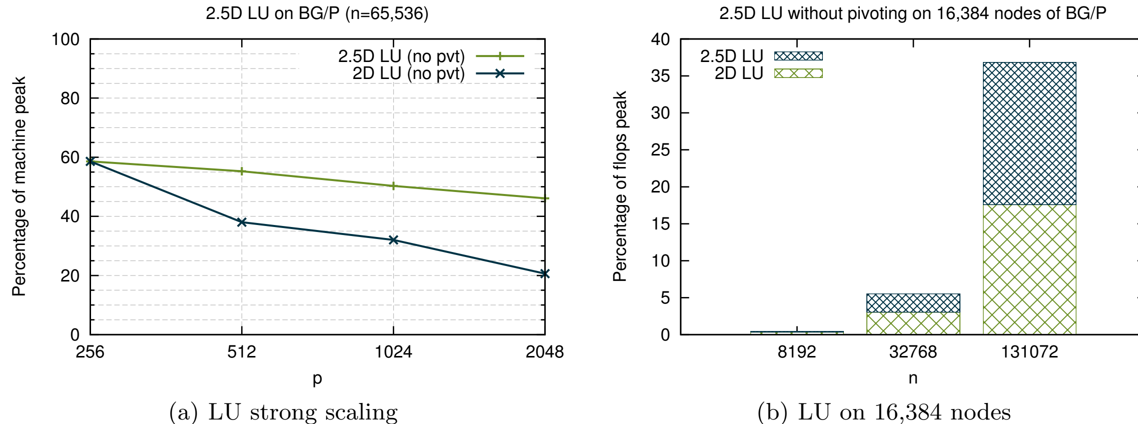
Figure 5(a) demonstrates that 2.5D matrix multiplication achieves better strong scaling than its 2D counterpart. However, both run at high efficiency (over 50%) for this problem size, so the benefit is minimal. The performance of the more general ScaLAPACK implementation lags behind the performance of our code by a large factor.

Figure 5(b) shows that 2.5D matrix multiplication outperforms 2D matrix multiplication significantly for small matrices on large partitions. The network latency and bandwidth costs are reduced, allowing small problems to execute much faster (up to 2.6X for the smallest problem size).

7.2 2.5D LU performance

We implemented a version of 2.5D LU without pivoting. While this algorithm is not stable for general dense matrices, it provides a good upper-bound on the performance of 2.5D LU with pivoting. The performance of 2.5D LU is also indicative of how well a 2.5D Cholesky implementation might perform.

Our 2.5D LU implementation has a structure closer to that of Algorithm 5 rather than Algorithm 3. Processor layers perform updates on different big-block subpanels of the matrix as each corner small block gets factorized.



(a) LU strong scaling

(b) LU on 16,384 nodes

Fig. 6. Performance of 2.5D LU without pivoting on BG/P

Our 2.5D LU implementation made heavy use of subset broadcasts (multicasts). All communication is done in the form of broadcasts or reductions along axis of the 3D virtual topology. This design allowed our code to utilize efficient line broadcasts on the BG/P supercomputer.

Figure 6(a) shows that 2.5D LU achieves more efficient strong scaling than 2D LU. 2D LU maps well to the 2D processor grid on 256 nodes. However, the efficiency of 2D LU suffers when we use more nodes, since the network partition becomes 3D. On 3D partitions, the broadcasts within 2D LU are done via topology-oblivious binomial trees and suffer from contention. For this problem configuration, 2.5D LU achieves a 2.2X speed-up over the 2D algorithm on 2048 nodes.

Figure 6(b) demonstrates that 2.5D LU is also efficient and beneficial at a larger scale. However, the efficiency of both 2D and 2.5D LU falls off for small problem sizes. The best efficiency and relative benefits are seen for the largest problem size ($n = 131,072$). We did not test larger problem sizes, since execution becomes too time consuming. However, we expect better performance and speed-ups for larger matrices.

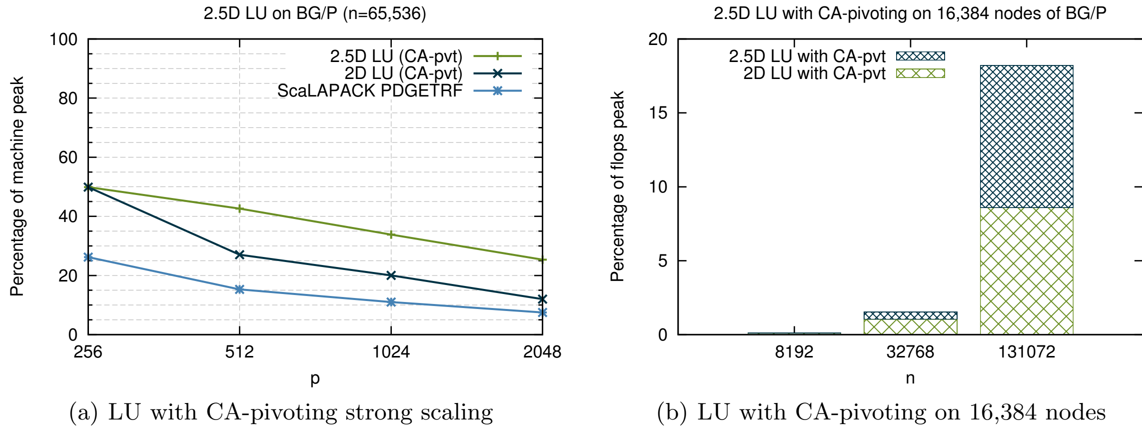
7.3 2.5D LU with CA-pivoting performance

2.5D LU performs pivoting in two stages. First, pivoting is performed only in the big-block panel. Then the rest of the matrix is pivoted according to a larger, accumulated pivot matrix. We found it most efficient to perform the subpanel pivoting via a broadcast and a reduction, which minimize latency. For the rest of the matrix, we performed scatter and gather operations to pivot, which minimize bandwidth. We found that this optimization can also be used to improve the performance of 2D LU and used it accordingly.

Figure 7(a) shows that 2.5D LU with CA-pivoting strongly scales with higher efficiency than its 2D counterpart. It also outperforms the ScaLAPACK PDGETRF implementation. Though, we note that ScaLAPACK uses partial pivoting rather than CA-pivoting and therefore computes a different answer.

On 16,384 nodes, 2D and 2.5D LU run efficiently only for larger problem sizes (see Figure 7(b)). The latency costs of pivoting heavily deteriorate the performance of the algorithms when the matrices are small. Since, 2.5D LU does not reduce latency cost, not much improvement is achieved for very small matrix sizes. However, for medium sized matrices ($n = 131,072$) over 2X gains in efficiency are achieved. We expect similar trends and better efficiency for even larger matrices.

The absolute efficiency achieved by our 2.5D LU with CA-pivoting algorithm is better than ScaLAPACK and can be improved even further. Our implementation does not exploit overlap between communication and computation and does not use prioritized scheduling. We observed that, especially at larger scales, processors spent most of their time idle (waiting to synchronize). Communication time, on the other hand, was heavily reduced by our techniques and was no longer a major bottleneck.



(a) LU with CA-pivoting strong scaling

(b) LU with CA-pivoting on 16,384 nodes

Fig. 7. Performance of 2.5D LU with pivoting on BG/P

8 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 [10] 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. Therefore it seems to be difficult to accumulate Housholder updates onto multiple buffers.

We plan to implement a more general 2.5D MM algorithm based on SUMMA [20]. We also plan to further tune our 2.5D LU algorithms. Incorporating better scheduling and overlap should improve the absolute efficiency of our implementation. We hope to apply these implementations to accelerate scientific simulations that solve distributed dense linear algebra problems. Our motivating scientific domain has been quantum chemistry applications, which spend a significant fraction of execution time performing small distributed dense matrix multiplications and factorizations.

Acknowledgements

The first author was supported by a Krell Department of Energy Computational Science Graduate Fellowship, grant number DE-FG02-97ER25308. Research was also supported by Microsoft (Award #024263) and Intel (Award #024894) funding and by matching funding by U.C. Discovery (Award #DIG07-10227). This material is supported by U.S. Department of Energy grants numbered DE-SC0003959, DE-SC0004938, and DE-FC02-06-ER25786. This research used resources of the Argonne Leadership Computing Facility at Argonne National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-06CH11357.

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. Ashcraft, C.: A taxonomy of distributed dense LU factorization methods. Boeing Computer Services Technical Report ECA-TR-161 (March 1991)
4. Ashcraft, C.: The fan-both family of column-based distributed Cholesky factorization algorithms. In: Alan George, J.R.G., Liu, J.W.H. (eds.) Graph Theory and Sparse Matrix Computation. IMA Volumes in Mathematics and its Applications, Springer-Verlag, vol. 56, pp. 159–190 (1993)
5. Ballard, G., Demmel, J., Holtz, O., Schwartz, O.: Minimizing communication in numerical linear algebra. To appear in SIAM J. Mat. Anal. Appl., UCB Technical Report EECS-2009-62 (2010)
6. 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)
7. Cannon, L.E.: A cellular computer to implement the Kalman filter algorithm. Ph.D. thesis, Bozeman, MT, USA (1969)
8. Dekel, E., Nassimi, D., Sahni, S.: Parallel matrix and graph algorithms. SIAM Journal on Computing 10(4), 657–675 (1981)
9. Demmel, J., Grigori, L., Xiang, H.: A Communication Optimal LU Factorization Algorithm. EECS Technical Report EECS-2010-29, UC Berkeley (March 2010)
10. Demmel, J., Dumitriu, I., Holtz, O.: Fast linear algebra is stable. Numerische Mathematik 108, 59–91 (2007)
11. 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)
12. 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)
13. Gropp, W., Lusk, E., Skjellum, A.: Using MPI: portable parallel programming with the message-passing interface. MIT Press, Cambridge, MA, USA (1994)
14. Irony, D., Toledo, S.: Trading replication for communication in parallel distributed-memory dense solvers. Parallel Processing Letters 71, 3–28 (2002)
15. 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)
16. Johnsson, S.L.: Minimizing the communication time for matrix multiplication on multiprocessors. Parallel Comput. 19, 1235–1257 (November 1993)
17. Kumar, S., Dozsa, G., Almasi, G., Heidelberger, P., Chen, D., Giampapa, M.E., Michael, B., Faraj, A., Parker, J., Ratterman, J., Smith, B., Archer, C.J.: The deep computing messaging framework: generalized scalable message passing on the Blue Gene/P supercomputer. In: Proceedings of the 22nd annual international conference on Supercomputing. pp. 94–103. ICS '08, ACM, New York, NY, USA (2008)
18. McColl, W.F., Tiskin, A.: Memory-efficient matrix multiplication in the BSP model. Algorithmica 24, 287–297 (1999)
19. Solomonik, E., Demmel, J.: Communication-optimal parallel 2.5D matrix multiplication and LU factorization algorithms. Tech. Rep. UCB/EECS-2011-10, EECS Department, University of California, Berkeley (Feb 2011), http://www.eecs.berkeley.edu/Pubs/TechRpts/2011/EECS-2011-10.html
20. Van De Geijn, R.A., Watts, J.: SUMMA: scalable universal matrix multiplication algorithm. Concurrency: Practice and Experience 9(4), 255–274 (1997)