Communication-optimal Parallel and Sequential Cholesky decomposition

Grey Ballard
James Demmel
Olga Holtz
Oded Schwartz

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2009-29
http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-29.html

February 14, 2009

Communication-optimal Parallel and Sequential Cholesky decomposition

Grey Ballard [*], James Demmel [†], Olga Holtz [‡], and Oded Schwartz [§]

February 13, 2009

Abstract

Numerical algorithms have two kinds of costs: arithmetic and communication, by which we mean either moving data between levels of a memory hierarchy (in the sequential case) or over a network connecting processors (in the parallel case). Communication costs often dominate arithmetic costs, so it is of interest to design algorithms minimizing communication. In this paper we first extend known lower bounds on the communication cost (both for bandwidth and for latency) of conventional ($O(n^3)$) matrix multiplication to Cholesky factorization, which is used for solving dense symmetric positive definite linear systems. Second, we compare the cost of various Cholesky decomposition implementations to this lower bound, and draw the following conclusions:

(1) "Naïve" sequential algorithms for Cholesky attain neither the bandwidth nor latency lower bounds.

(2) The sequential blocked algorithm in LAPACK (with the right block size), as well as various recursive algorithms [AP00, GJ01, AGW01, ST04], and one based on work of Toledo [Tol97], can attain the bandwidth lower bound.

(3) The LAPACK algorithm can also attain the latency bound if used with blocked data structures rather than column-wise or row-wise matrix data structures, though the Toledo algorithm cannot.

(4) The recursive sequential algorithm due to [AP00] attains the bandwidth and latency lower bounds at *every level* of a multi-level memory hierarchy, in a "cache-oblivious" way.

(5) The parallel implementation of Cholesky in the ScaLAPACK library (again with the right block-size) attains both the bandwidth and latency lower bounds to within a poly-logarithmic factor.

Combined with prior results in [DGHL08a, DGHL08b, DGX08] this gives a complete set of communication-optimal algorithms for $O(n^3)$ implementations of three basic factorizations of dense linear algebra: LU with pivoting, QR and Cholesky. But it goes beyond this prior work on sequential LU and QR by optimizing communication for any number of levels of memory hierarchy.

[*]Computer Science Department, University of California, Berkeley, CA 94720. Research supported by Microsoft and Intel funding (Award #20080469) and by matching funding by U.C. Discovery (Award #DIG07-10227). email:ballard@eecs.berkeley.edu

[†]Mathematics Department and CS Division, University of California, Berkeley, CA 94720. email:demmel@cs.berkeley.edu

[‡]Departments of Mathematics, University of California, Berkeley and Technische Universität Berlin. O. Holtz acknowledges support of the Sofja Kovalevskaja programm of Alexander von Humboldt Foundation. email:oholtz@eecs.berkeley.edu

[§]Departments of Mathematics, Technische Universität Berlin, 10623 Berlin, Germany. This work was done while visiting University of California, Berkeley. email:odedsc@math.tu-berlin.de.

1 Introduction

Let A be a real symmetric and positive definite matrix. Then there exists a real lower triangular matrix L so that $A = L \cdot L^T$ (L is unique if we restrict its diagonal elements to be positive). This is called the Cholesky decomposition. We are interested in finding efficient parallel and sequential algorithms for the Cholesky decomposition. Efficiency is measured both by the number of arithmetic operations, and by the amount of communication, either between levels of a memory hierarchy on a sequential machine, or between processors communicating over a network on a parallel machine. Since the time to move one word of data typically exceeds the time to perform one arithmetic operation by a large and growing factor, our goal will be to minimize communication.

We model communication costs in more detail as follows. In the sequential case, with two levels of memory hierarchy (fast and slow), communication means reading data items (*words*) from slow memory to fast memory and writing data from fast memory to slow memory. If words are stored contiguously, they can be read or written in a bundle which we will call a *message*. We assume that a message of n words can be communicated between fast and slow memory in time $\alpha + \beta n$ where α is the *latency* (seconds per message) and β is the *inverse bandwidth* (seconds per word). We assume that the matrix being factored initially resides in slow memory, and is too large to fit in the smaller fast memory. Our goal is to minimize the total number of words and the total number of messages communicated between fast and slow memory.

In the parallel case, we are interested in the communication among the processors. As in the sequential case, we assume that a message of n consecutively stored words can be communicated in time $\alpha + \beta n$. We assume that the matrix is initially evenly distributed among all P processors, and that there is only enough memory to store about $1/P$-th of a matrix per processor. As before, our goal is to minimize the number of words and messages communicated. In order to measure the communication complexity of a parallel algorithm, we will count the number of words and messages communicated along the critical path of the algorithm.

We consider *classical* algorithms for Cholesky decomposition, i.e., those that perform "the usual" $O(n^3)$ arithmetic operations, possibly reordered by associativity and commutativity of addition. That is, our results do not apply when using distributivity to reorganize the algorithm (such as Strassen-like algorithms); we also assume no pivoting is performed. We define "classical" more carefully later. We show that the communication complexity of any such Cholesky algorithm shares essentially the same lower bound as does the classical matrix multiplication (i.e., using the usual $2n^3$ arithmetic operations possibly reordered using associativity and commutativity of addition).

Theorem 1 (Main Theorem). *Any sequential or parallel classical algorithm for the Cholesky decomposition of n-by-n matrices can be transformed into a classical algorithm for $\frac{n}{3}$-by-$\frac{n}{3}$ matrix multiplication, in such a way that the bandwidth of the matrix multiplication algorithm is at most a constant times the bandwidth of the Cholesky algorithm.*

Therefore any bandwidth lower bound for classical matrix multiplication applies to classical Cholesky, in a Big-O sense:

$$bandwidth = \Omega(\#arithmetic\_operations/fast\_memory\_size^{1/2})$$

Similarly, the latency lower bound for Cholesky is

$$latency = \Omega(\#arithmetic\_operations/fast\_memory\_size^{3/2})$$

In particular, since a sequential classical n-by-n matrix multiplication algorithm has a bandwidth lower bound of $\Omega(n^3/M^{1/2})$ where M is the fast memory size [HK81, ITT04], classical Cholesky has the same lower bound (we discuss the parallel case later).

To get the latency lower bound, we use the simple observation [DGHL08a] that the number of messages is at least the bandwidth lower bound divided by the maximum message size, and that the maximum message size is at most fast memory size in the sequential case (or the local memory size in the parallel case). So for sequential matrix multiplication this means the latency lower bound is $\Omega(n^3/M^{3/2})$.

Attainability of the latency lower bound depends on the data structure more strongly than does attainability of the bandwidth lower bound. As a simple example, consider computing the sum of $n \leq M$ numbers in slow memory, which obviously requires reading these n words. If they are in consecutive memory locations, this can be done in 1 read operation, the minimum possible latency. But if they are not consecutive, say they are separated by at least $M-1$ words, this may require n read operations, the maximum possible latency.

In the case of matrix multiplication, the well-known blocked matrix multiplication algorithm for $C = A \cdot B$ that multiplies and accumulates $\sqrt{\frac{M}{3}}$-by-$\sqrt{\frac{M}{3}}$ submatrices of A, B and C attains the bandwidth lower bound. But only if each matrix is stored so that the $\frac{M}{3}$ entries of each of its submatrices are contiguous (not the case with columnwise or rowwise storage) can the latency lower bound be reached; we call such a data structure *contiguous block storage* and describe it in more detail below. Alternatively, one could try to copy A and B from their input format (say columnwise) to contiguous block storage doing (asymptotically) no more communication than the subsequent matrix multiplication; we will see this is possible provided $M = O(n)$. There will be analogous requirements for Cholesky to attain its latency lower bound.

In particular, we will draw the following conclusions [1] about the communication costs of sequential classical Cholesky, as summarized in Table 1:

1. "Naïve" sequential variants of Cholesky that operate on single rows and columns (be they left-looking, right-looking, etc.) attain neither the bandwidth nor the latency lower bound.

2. A sequential blocked algorithm used in LAPACK (with the correct block size) attains the bandwidth lower bound, as do the recursive algorithms in [AP00, GJ01, AGW01, ST04]. A recursive algorithm analogous to Toledo's LU algorithm [Tol97] attains the bandwidth lower bound in nearly all cases, expect possibly for an $O(\log n)$ factor in the narrow range $\frac{n^2}{\log^2 n} < M < n^2$.

3. Whether the LAPACK algorithm also attains the latency lower bound depends on the matrix layout: If the input matrix is given in row-wise or column-wise format, and this is not changed by the algorithm, then the latency lower bound cannot be attained. But if the input matrix is given in contiguous block storage, or $M = \Omega(n)$ so that it can be copied quickly to contiguous block format, then the latency lower bound can be attained by the $LAPACK$ algorithm[2]. Toledo's algorithm cannot minimize latency (at least when $M > n^{2/3}$).

So far we have discussed a two-level memory hierarchy, with fast and slow memory. It is natural to ask about more than 2 levels, since most computers have multiple levels (e.g., L1, L2, L3 caches, main memory, and disk). In this case, an optimal algorithm should simultaneously minimize communication between *all* pairs of adjacent levels of memory hierarchy (e.g., minimize bandwidth and latency between L1 and L2, between L2 and L3, etc.).

[1] We number our main conclusions consecutively from 1 to 6.

[2] This can be done by reading M elements at a time, in a columnwise order (which requires one message) then writing each of these elements to the right location of the new matrix. We write these words using $\sqrt{M}$ messages (one per each relevant block). Thus, the total number of messages is $O\left(\frac{n^2}{\sqrt{M}}\right)$ which is asymptotically dominated by $\frac{n^3}{M^{3/2}}$ for $M \geq n$.

In the case of sequential matrix multiplication, bandwidth is minimized in this sense by simply applying the usual blocked algorithm recursively, where each level of recursion multiplies matrices that fit at a particular level of the memory hierarchy, by using the blocked algorithm to multiply submatrices that fit in the next smaller level. This is easy since matrix multiplication naturally breaks into smaller matrix multiplications.

For matrix multiplication to minimize latency across all memory hierarchy levels, it is necessary for all submatrices of all sizes to be stored contiguously. This leads to a data structure variously referred to as *recursive block storage* or storage using *space-filling curves*, and described in [FLPR99, AP00, EGJK04].

Finally, sequential matrix multiplication can achieve communication optimality as just described in one of two ways: (1) We can choose the number of recursion levels and sizes of the subblocks with prior knowledge of the number of levels and sizes of the levels of memory hierarchy, a *cache-aware* process called tuning. (2) We can simply always recur down to 1-by-1 blocks (or some other small constant size), repeatedly dividing block sizes by 2 (perhaps padding submatrices to have even dimensions as needed). Such an algorithm is called *cache-oblivious* [FLPR99], and has the advantage of simplicity and portability compared to a cache-aware algorithm, though it might also have more overhead in practice.

It is indeed possible for sequential Cholesky to be organized to be optimal across multiple memory hierarchy levels in all the senses just described, assuming we use recursive block storage:

4. The recursive algorithm modelled on Toledo's LU can be implemented in a cache-oblivious way so as to minimize bandwidth, but not latency [3].

5. The cache-oblivious recursive Cholesky algorithm of Ahmed and Pingali [AP00] minimizes both bandwidth and latency for all matrices across all memory hierarchy levels. None of the other algorithms do so.

Finally, we address the case of parallel Cholesky, where there are P processors connected by a network with latency α and reciprocal bandwidth β. We consider only the memory-scalable case, where each processor's local memory is of size $M = O(n^2/P)$, so that only $O(1)$ copies of the matrix are stored overall (the so-called "2D case", see [ITT04] for the general case, including 3D, for matrix multiplication). The consequence of our Main Theorem is again a bandwidth lower bound of the form $\Omega(n^3/(PM^{1/2})) = \Omega(n^2/P^{1/2})$, and a latency lower bound of the form $\Omega(n^3/(PM^{3/2})) = \Omega(P^{1/2})$.

ScaLAPACK attains a matching upper bound. It does so by partitioning the matrix into submatrices and distributing them to the processors in a block cyclic manner. See Section 3.3.1 for details.

6. With the right choice of block size b, namely the largest possible value $b = n/\sqrt{P}$, the Cholesky algorithm in ScaLAPACK [BJCD+97] attains the above bandwidth and latency lower bounds to within a factor of $\log P$. This is summarized in Table 2.

A 'real' computer may be more complicated than any model we have discussed so far, with both parallelism and multiple levels of memory hierarchy (where each sequential processor making up a parallel computer has multiple levels of cache), or with multiple levels of parallelism (i.e., where each 'parallel processor' itself consists of multiple processors), etc. And it may be 'heterogenous', with functional units and communication channels of greatly differing speeds. We leave lower and upper communication bounds on such processors for future work.

[3] Toledo's algorithm is designed to retain numerical stability for the LU case. The [AP00] algorithm deals with

	Bandwidth	Latency	Cache Oblivious
Lower bound	$\Omega\left(\frac{n^3}{\sqrt{M}}\right)$	$\Omega\left(\frac{n^3}{M^{3/2}}\right)$	
Naïve: left/right looking			
Column-major	$\Theta(n^3)$	$\Theta\left(n^2+\frac{n^3}{M}\right)$	✓
LAPACK [ABB+92]			
Column-major	$O\left(\frac{n^3}{\sqrt{M}}\right)$	$O\left(\frac{n^3}{M}\right)$	×
Contiguous blocks	$O\left(\frac{n^3}{\sqrt{M}}\right)$	$O\left(\frac{n^3}{M^{3/2}}\right)$	×
Rectangular Recursive [Tol97]			
Column-major	$\Theta\left(\frac{n^3}{\sqrt{M}}+n^2\log n\right)$	$\Omega\left(\frac{n^3}{M}\right)$	✓
Contiguous blocks	$\Theta\left(\frac{n^3}{\sqrt{M}}+n^2\log n\right)$	$\Omega\left(n^2\right)$	✓
Square Recursive [AP00]			
"Recursive Packed Format" [AGW01]	$O\left(\frac{n^3}{\sqrt{M}}\right)$	$\Omega\left(\frac{n^3}{M}\right)$	✓
Column-major [AP00]	$O\left(\frac{n^3}{\sqrt{M}}\right)$	$O\left(\frac{n^3}{M}\right)$	✓
Contiguous blocks [AP00]	$O\left(\frac{n^3}{\sqrt{M}}\right)$	$O\left(\frac{n^3}{M^{3/2}}\right)$	✓

Table 1: Sequential bandwidth and latency: lower bound vs. algorithms. M denotes the size of the fast memory. FLOPs count of all is $O(n^3)$. Refer to Section 3 for definitions of terms and details.

	Bandwidth	Latency	FLOPS
Lower-bound			
General	$\Omega\left(\frac{n^3}{P\sqrt{M}}\right)$	$\Omega\left(\frac{n^3}{PM^{3/2}}\right)$	$\Omega\left(\frac{n^3}{P}\right)$
2D layout: $M=O\left(\frac{n^2}{P}\right)$	$\Omega\left(\frac{n^2}{\sqrt{P}}\right)$	$\Omega\left(\sqrt{P}\right)$	$\Omega\left(\frac{n^3}{P}\right)$
ScaLAPACK [BJCD+97]			
General	$O\left(\left(\frac{n^2}{\sqrt{P}}+nb\right)\log P\right)$	$O\left(\frac{n}{b}\log P\right)$	$O\left(\frac{n^3}{P}+\frac{n^2b}{\sqrt{P}}+nb^2\right)$
Choosing $b=\frac{n}{\sqrt{P}}$	$O\left(\frac{n^2}{\sqrt{P}}\log P\right)$	$O(\sqrt{P}\log P)$	$O\left(\frac{n^3}{P}\right)$

Table 2: Parallel, lower bound vs. algorithms. M denotes the size of the memory of each processor. P is the number of processors. b is the block size. Refer to Section 3 for definitions of terms and details.

The rest of this paper is organized as follows. In Section 2 we show the reduction from matrix multiplication to Cholesky decomposition, thus extending the bandwidth lower bounds of [HK81] and [ITT04] to a bandwidth lower bound for the sequential and parallel implementations of Cholesky decomposition. We also discuss latency lower bounds. In Section 3 we recall known Cholesky decomposition algorithms and compare their bandwidth and latency costs with the lower bounds.

the Cholesky case, therefore requires no pivoting for numerical stability. Thus a simpler recursion suffices, and the latency improves.

2 Communication Lower Bounds

Consider an algortihm for a parallel computer with P processors that multiplies matrices in the 'classical' way (the usual $2n^3$ arithmetic operations possibly reordered using associativity and commutativity of addition) and each of the processors has memory of size M. Irony *et al.* [ITT04] showed that at least one of the processors has to send or receive this minimal number of words:

Theorem 2 ([ITT04]). *Any 'classical' implementation of matrix-multiplication of $n \times n$ matrices on a P processor machine, each equipped with memory of size M, requires that one of the processors sends or receives at least $\frac{n^3}{2\sqrt{2}PM^{\frac{1}{2}}} - M$ words. These can be entries of A, of B or of $A \cdot B$.*

If A and B are of size $n \times m$ and $m \times r$ respectively, then the corresponding bound is $\frac{nmr}{2\sqrt{2}PM^{\frac{1}{2}}} - M$

As any processor has memory of size M, any message it sends or receives may deliver at most M words. Therefore we deduce the following:

Corollary 2.1. *Any 'classical' implementation of matrix-multiplication on a P processor machine, each processor equipped with memory of size M, requires that one of the processors sends or receives at least $\frac{n^3}{2\sqrt{2}PM^{\frac{3}{2}}} - 1$ messages.*

If A and B are of size $n \times m$ and $m \times r$ respectively, then the corresponding bound is $\frac{nmr}{2\sqrt{2}PM^{\frac{3}{2}}} - 1$

For the case of $P = 1$ these give lower bounds for the bandwidth and the latency of the sequential case. These lower bounds for bandwidth for the sequential case were previously shown (up to some multiplicative constant factor) by Hong and Kung [HK81].

It is easy to reduce matrix multiplication to LU decomposition of a slightly larger order, as the following identity shows:

$$\begin{pmatrix} I & 0 & -B \\ A & I & 0 \\ 0 & 0 & I \end{pmatrix} = \begin{pmatrix} I & & \\ A & I & \\ 0 & 0 & I \end{pmatrix} \cdot \begin{pmatrix} I & 0 & -B \\ & I & A \cdot B \\ & & I \end{pmatrix} \tag{1}$$

This identity means that LU factorization can be used to perform matrix multiplication; to accomodate pivoting A and/or B can be scaled down to be too small to be chosen as pivots, and $A \cdot B$ can be scaled up accordingly. Thus an $O(n^3)$ implementation of LU that only uses associativity and commutativity of addition to reorganize its operations (thus eliminating Strassen-like algorithms) must perform at least as much communication as a correspondingly reorganized implementation of $O(n^3)$ matrix multiplication.

We wish to mimic this lower bound construction for Cholesky. Consider the following reduction from matrix multiplication to Cholesky decomposition. Let T be the matrix defined below, composed of 9 square blocks each of dimension n; then the Cholesky decomposition of T is:

$$T \equiv \begin{pmatrix} I & A^T & -B \\ A & I + A \cdot A^T & 0 \\ -B^T & 0 & D \end{pmatrix} = \begin{pmatrix} I & & \\ A & I & \\ -B^T & (A \cdot B)^T & X \end{pmatrix} \cdot \begin{pmatrix} I & A^T & -B \\ & I & A \cdot B \\ & & X^T \end{pmatrix} \equiv L \cdot L^T \tag{2}$$

where X is the Cholesky factor of $D' \equiv D - B^T B - B^T A^T A B$, and D can be any symmetric matrix such that D' is positive definite.

Thus $A \cdot B$ is computed via this Cholesky decomposition. Intuitively this seems to show that the communication complexity needed for computing matrix multiplication is a lower bound to that of computing the Cholesky decomposition (of matrices 3 times larger) as $A \cdot B$ appears in L^T, the decomposition of T. Note however that $A \cdot A^T$ appears in T.

One has to consider the possibility that all the communication that is guaranteed by [ITT04] is in fact performed when computing $A \cdot A^T$ and so we have no non-trivial lower bound for the Cholesky decomposition of T.[4] Stated otherwise, maybe computing $A \cdot B$ from A and B incurs less communication cost if we are also given $A \cdot A^T$.[5] So let us instead consider the following approach to prove the lower bound.

In addition to the real numbers $\mathbb{R}$, consider new "starred" numerical quantities, called 1^* and 0^*, with arithmetic properties detailed in the following tables. 1^* and 0^* mask any real value in addition/substraction operation, but behave similarly to $1 \in \mathbb{R}$ and $0 \in \mathbb{R}$ in multiplication and division operations.

$\pm$	1^*	0^*	y
1^*	1^*	1^*	1^*
0^*	1^*	0^*	0^*
x	1^*	0^*	$x \pm y$

$\cdot$	1^*	0^*	y
1^*	1^*	0^*	y
0^*	0^*	0	0
x	x	0	$x \cdot y$

$/$	1^*	0^*	$y \neq 0$
1^*	1^*	$-$	$1/y$
0^*	0^*	$-$	0
x	x	$-$	x/y

$\sqrt{\cdot}$	
1^*	1^*
0^*	0^*
$x \geq 0$	$\sqrt{x}$

Table 3: Arithmetic Operations Tables. The variables x and y stand for any *real* values. For consistency, $-0^* \equiv 0^*$ and $-1^* \equiv 1^*$.

Consider this set of values and arithmetic operations.

- It is commutative with respect to addition and to multiplication (by the symmetries of the corresponding tables).
- It is associative with respect to addition: regardless of ordering of summation, the sum is 1^* if one of the addends is 1^*, otherwise it is 0^* if one of the addends is 0^*.
- The set is also associative with respect to multiplication: $(a \cdot b) \cdot c = a \cdot (b \cdot c)$. This is trivial if all factors are in $\mathbb{R}$. As 1^* is a multiplicative identity, it is also immediate if some of the factors equal 1^*. Otherwise, at least one of the factors is 0^*, and the product is 0.
- Distributivity, however, does not hold: $1 \cdot (1^* + 1^*) = 1 \neq 2 = (1 \cdot 1^*) + (1 \cdot 1^*)$

Let us return to the construction. We set T' to be:

$$T' \equiv \begin{pmatrix} I & A^T & -B \\ A & C & 0 \\ -B^T & 0 & C \end{pmatrix}$$

where C has 1^* on the diagonal and 0^* everywhere else:

$$C \equiv \begin{pmatrix} 1^* & 0^* & & \cdots & 0^* \\ 0^* & 1^* & 0^* & & \vdots \\ & & & \ddots & \\ \vdots & & & & 0^* \\ 0^* & \cdots & & 0^* & 1^* \end{pmatrix}$$

[4] Note that computing $A \cdot A^T$ is asymptotically as hard as matrix multiplication: take $A = [X, 0; Y^T, 0]$. Then $A \cdot A^T = [*, XY; *, *]$

[5] Note that the result of [ITT04] does not mean that both A and B are communicated a lot, as one can communicate each of the entries of B only once, and shift all other entries many times, resulting in an inefficient algorithm, but such that no non-trivial lower bound on the communication of the elements of B can be given.

One can verify that the (unique) Cholesky decomposition of C is[6]

$$C = \begin{pmatrix} 1^* & 0 & \dots & 0 \\ 0^* & 1^* & & \vdots \\ \vdots & & \ddots & 0 \\ 0^* & \cdots & 0^* & 1^* \end{pmatrix} \cdot \begin{pmatrix} 1^* & 0^* & \cdots & 0^* \\ & \ddots & \ddots & \vdots \\ \vdots & & 1^* & 0^* \\ 0 & \dots & & 1^* \end{pmatrix} \equiv C' \cdot C'^T \tag{3}$$

Note that if a matrix X does not contain any "starred" values 0^* and 1^* then $X = C \cdot X = X \cdot C = C' \cdot X = X \cdot C' = C'^T \cdot X = X \cdot C'^T$ and $C + X = C$. Therefore, one can confirm that the Cholesky decomposition of T' is is:

$$T' \equiv \begin{pmatrix} I & A^T & -B \\ A & C & 0 \\ -B^T & 0 & C \end{pmatrix} = \begin{pmatrix} I & & \\ A & C' & \\ -B^T & (A \cdot B)^T & C' \end{pmatrix} \cdot \begin{pmatrix} I & A^T & -B \\ & C'^T & A \cdot B \\ & & C'^T \end{pmatrix} \equiv L \cdot L^T \tag{4}$$

One can think of C as masking the $A \cdot A^T$ previously appearing in the central block of T, therefore allowing the lower bound of computing $A \cdot B$ to be accounted for by the Cholesky decomposition, and not by the computation of $A \cdot A^T$. More formally, let Alg be any 'classical' algorithm for Cholesky factorization. We convert it to a matrix multiplication algorithm as follows:

Algorithm 1 Matrix Multiplication by Cholesky-Decomposition

Input: Two $n \times n$ matrices, A and B.

1: Let Alg' be Alg updated to correctly handle the new $0^*, 1^*$ values. {note that Alg' can be constructed off-line.}
2: $T' = T'(A, B)$ {constructed as in Equation (4).}
3: $L = Alg'(T')$
4: **return** $(L_{32})^T$

The simplest conceptual way to do step (1) is to attach an extra bit to every numerical value, indicating whether it is "starred" or not, and modify every arithmetic operation to first check this bit before performing an operation. This increases the bandwidth by at most a constant factor. Alternatively, we can use Signalling NaNs as defined in the IEEE Floating Point Standard [IEE08] to encode 1^* and 0^* with no extra bits.

If the instructions implementing Cholesky are scheduled deterministically, there is another alternative: one can run the algorithm "symbolically", propagating 0^* and 1^* arguments from the inputs forward, simplifying or eliminating arithmetic operations whose inputs contain 0^* or 1^*, and also eliminating operations for which there is no path in the directed acyclic graph (describing how outputs of each operation propagate to inputs of other operations) to the desired output $A \cdot B$. The resulting Alg' performs a strict subset of the arithmetic and memory operations of the original Cholesky algorithm.

We note that updating Alg to form Alg' is done off-line, so that step (1) does not actually take any time to perform when Algorithm 1 is called.

We next verify the correctness of this reduction: that the output of this procedure on input A, B is indeed the multiplication $A \cdot B$, as long as Alg is a classical algorithm, in a sense we now define carefully.

[6]By writing $X \cdot Y$ we mean the resulting matrix assuming the straightforward n^3 matrix multiplication algorithm. This had to be stated clearly, as the distributivity does not hold for the starred values.

Let $T' = L \cdot L^T$ be the Cholesky decomposition of T'. Then we have the following formulas:

$$L(i,i) = \sqrt{T'(i,i) - \sum_{k \in [i-1]} (L(i,k))^2} \tag{5}$$

$$L(i,j) = \frac{1}{L(j,j)} \left(T'(i,j) - \sum_{k \in [j-1]} L(i,k) \cdot L(j,k) \right) \ , \ i > j \tag{6}$$

where $[t] = \{1, ..., t\}$. By the no-pivoting and no-distributivity restrictions to Alg, when an entry of L is computed, all the entries on which it depends have already been computed and combined by the above formulas, with the sums occurring in any order. These dependencies form a dependency graph which is a DAG (directed acyclic graph), and therefore impose a partial ordering on the computation of the entries of L (see Figure 1). That is, when an entry $L(i,i)$ is computed, by Equation (5), all the entries $\{L(i,k)\}_{k \in [i-1]}$ have already been computed. Denote this set by $S_{i,i}$, namely,

$$S_{i,i} \equiv \{L(i,k)\}_{k \in [i-1]} \tag{7}$$

Similarly, when an entry $L(i,j)$ (for $i > j$) is computed, by Equation (6), all the entries $\{L(i,k)\}_{k \in [j-1]}$ and all the entries $\{L(j,k)\}_{k \in [j]}$ have already been computed. Denote this set by $S_{i,j}$ namely,

$$S_{i,j} \equiv \{L(i,k)\}_{k \in [j-1]} \cup \{L(j,k)\}_{k \in [j]} \tag{8}$$

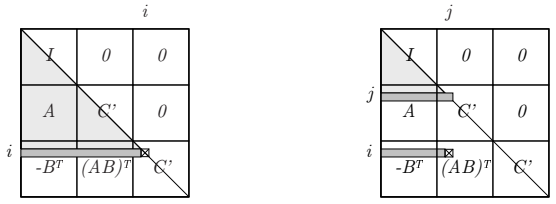

Figure 1: Dependencies of $L(i,j)$, for diagonal entries (left) and other entries (right).
Dark grey represents the sets $S_{i,i}$ (left) and $S_{i,j}$ (right). Light grey represents indirect dependencies.

Lemma 2.2. *Any ordering of the computation of the elements of L that respects the partial ordering induced by the above mentioned directed acyclic graph results in a correct computation of $A \cdot B$.*

Proof. We need to confirm that the starred entries 1^* and 0^* of T' do not somehow "contaminate" the desired entries of L_{32}^T. The proof is by induction on the partial order on pairs (i,j) implied by (7) and (8). The base case —the correctness of computing $L(1,1)$— is immediate. Assume by induction that all elements of $S_{i,j}$ are correctly computed and consider the computation of $L(i,j)$ according to the block in which it resides:

- If $L(i,j)$ resides in block L_{11}, L_{21} or L_{31} then $S_{i,j}$ contains only real values, and no arithmetic operations with 0^* or 1^* occur (recall Figure 1 or Equations (4),(7) and (8)). Therefore, the correctness follows from the correctness of the original Cholesky decomposition algorithm.
- If $L(i,j)$ resides in L_{22} or L_{33} then $S_{i,j}$ may contain "starred" value (elements of C'). We treat separately the case where $L(i,j)$ is on the diagonal and the case where it is not.

If $i = j$ then by Equation (5) $L(i,i)$ is determined to be 1^* since $T'(i,i) = 1^*$ and since adding to, subtracting from and taking the square root of 1^* all result in 1^* (recall Table 3 and Equation (5)).

If $i > j$ then by the inductive assumption the divisor $L(j,j)$ of Equation (6) is correctly computed to be 1^* (recall Figure 1 and the definition of C' in Equation (3)). Therefore, no division by 0^* is performed. Moreover, $T'(i,j)$ is 0^*. Then $L(i,j)$ is determined to be the correct value 0^*, unless 1^* is subtracted (recall Equation (6)). However, every subtracted product (recall Equation (6)) is composed of two factors of the same column but of different rows. Therefore, by the structure of C', none of them is 1^* so their product is not 1^* and the value is computed correctly.

- If $L(i,j)$ resides in L_{32} then $S_{i,j}$ may contain "starred" values (see Figure 1, right-hand side, row j). However, every subtraction performed (recall Equation (6)) is composed of a product of two factors, of which one is on the ith row (and on a column $k < j$). Hence, by induction (on i, j), the (i,k) element has been computed correctly to be a real value, and by the multiplication properties so is the product. Therefore no masking occurs.

This completes the proof of Lemma 2.2. □

We now know that Algorithm 1 correctly multiplies matrices 'classically', and so has known communication lower bounds given by Theorem 2 and Corollary 2.1. But it remains to confirm that step 2 (setting up T') and step 4 (returning L_{32}^T) do not require much communication, so that these lower bounds apply to step 3, running Alg' (recall that step 1 may be performed off-line and so doesn't count). Since Alg' is either a small modification of Cholesky to add "star" labels to all data items (at most doubling the bandwidth), or a subset of Cholesky with some operations omitted (those with starred arguments, or not leading to the desired output L_{32}), a lower bound on communication for Alg' is also a lower bound for Cholesky.

Theorem 1 (Main Theorem). *Any sequential or parallel classical algorithm for the Cholesky decomposition of n-by-n matrices can be transformed into a classical algorithm for $\frac{n}{3}$-by-$\frac{n}{3}$ matrix-multiplication, in such a way that the bandwidth of the matrix-multiplication algorithm is at most a constant times the bandwidth of the Cholesky algorithm.*

Therefore any bandwidth or latency lower bound for classical matrix multiplication applies to classical Cholesky in a Big-O sense:

Corollary 2.3. *In the sequential case, with a fast memory of size M, the bandwidth lower bound for Cholesky decomposition is $\Omega(n^3/M^{1/2})$, and the latency lower bound is $\Omega(n^3/M^{3/2})$.*

Proof. Constructing T' (in any data format) requires bandwidth of at most $18n^2$ (copying a $3n$-by-$3n$ matrix, with another factor of 2 if each entry has a flag indicating whether it is "starred" or not), and extracting L_{32}^T requires another n^2 of bandwidth. Furthermore, we can assume $n^2 < n^3/M^{1/2}$, i.e., that $M < n^2$, i.e., that the matrix is too large to fit entirely in fast memory (the only case of interest). Thus the bandwidth lower bound $\Omega(n^3/M^{1/2})$ of Algorithm 1 dominates the bandwidth costs of Steps 2 and 4, and so must apply to Step 3 (Cholesky). Finally, the latency lower bound for Step 3 is by a factor of M smaller than its bandwidth lower bound, as desired. □

Corollary 2.4. *In the parallel case (with a 2D layout on P processors as described earlier), the bandwidth lower bound for Cholesky decomposition is $\Omega(n^2/P^{1/2})$, and the latency lower bound is $\Omega(P^{1/2})$.*

Proof. The argument in the parallel case is analogous to that of Corollary 2.3. The construction of input and retrieval output at steps 2 and 4 of Algorithm 1 contribute bandwidth of $O\left(\frac{n^2}{P}\right)$. Therefore the lower bound of the bandwidth $\Omega\left(\frac{n^3}{P\sqrt{M}}\right)$ is determined by Step 3, the Cholesky decomposition. The lower bound on the latency of Step 3 is therefore $\Omega\left(\frac{n^3}{PM^{3/2}}\right)$, as each message delivers at most M words. Plugging in $M = \Theta\left(\frac{n^2}{P}\right)$ yields $B = \Omega(P^{1/2})$. □

3 Upper Bounds

In this section we discuss known algorithms for Cholesky decomposition, and their bandwidth and latency analysis, in sequential two memory levels model, in the of hierarchical memory model and in the parallel computing model.

Recall that the input is an $n \times n$ matrix and there are P processors, each equipped with memory of size M. The output is given on the same matrix. That is, the matrix A is overwritten to contain the lower triangular factor L when the algorithm halts.

3.1 Sequential Algorithms

Before reviewing the various sequential algorithms let us first consider the underlying data-structure which is used for storing the matrix. Although it does not affect the bandwidth, it may have a significant influence on the latency of the algorithm: it may or may not allow retrieving many relevant words using only a single message.

3.1.1 Data Storage

The various data-storage (some of which are implemented in $LAPACK$) appear in Figure 2. We can partition these data-structures into two classes: column-major and block-contiguous.

Column-Major Storage. The column-major data-structures store the items column-wise. This means that they are most fit for algorithms that access a column at a time (e.g., the left and right looking naïve algorithms). However, the latency of algorithms that access a block at time (e.g., $LAPACK$'s implementation) suffer, as retrieving a block of size $b \times b$ requires at least b messages, even if a single message can deliver b^2 words. This means a loss of the factor b in the latency (where b is typically the order of $\sqrt{M}$).

As the matrices in interest are symmetric, storing the entire matrix (known as 'Full-storage') wastes space. About half of the space can be saved by using the 'old packed' or the 'rectangular full packed' storages. The latter has the advantage of a more uniform indexing, which allows faster addressing.

Block-Contiguous Storage. The block-contiguous storages store the matrix entries in a way that allows a read or write of a block using a single message. This may improve that latency of Cholesky decomposition algorithm that accesses a $b \times b$ block at a time by a factor of b. The 'blocked' storage stores each block of the matrix in a contiguous space of the memory. For this, one has to know in advance the size of blocks to be used (which is a machine-specific parameter).

The elements of each block may be sorted by contiguous sub-blocks, where each sub-block is of a predefined (cache-aware) size. This can go on for several such layers of sub-blocks. This 'layered' data structure may fit a machine with several types of memories, ranging from slow and large to

fast and small (see subsection 3.2 for further discussion of this model). The next data structure allows the benefits of blocks-contiguously storage. Nevertheless, it is machine-independent.

The 'recursive format' [FLPR99, EGJK04] (also known as the bit interleaved layout, space-filling curve storage format and block recursive structure) is 'cache-oblivious'. This means that it allows access to a single block using a single message (or a constant number of messages), without knowing in advance the size of the blocks. This is done by storing the element of the matrix according to a space-filling curve (see Figure 2). Both of these block-contiguous formats have packed versions, where about half of the space is saved by using the symmetry of the matrices (see [EGJK04] for Recursive-Full-Packed data structures). The algorithm in [AGW01] uses a hybrid data structure called the 'recursive packed format' in which only half the matrix is stored and recursive ordering is used on triangular sub-matrices and column-major ordering is used on square sub-matrices.

Other Variants of these Data Structures. Similar to the column-major data structures, there are row-major data structures that store the matrix entries row-wise. All the above-mentioned packed data structures have versions that are indexed to efficiently store the lower (as in Figure 2) and upper triangular part of a matrix.

In the algorithms below, we only consider the application of algorithm to column-major (or block-contiguous) data structures. Taking any of the algorithms below, and changing it to work well with other compatible data structures (row-major vs. column-major, upper triangular vs. lower triangular, full storage vs. packed storage) is straightforward.

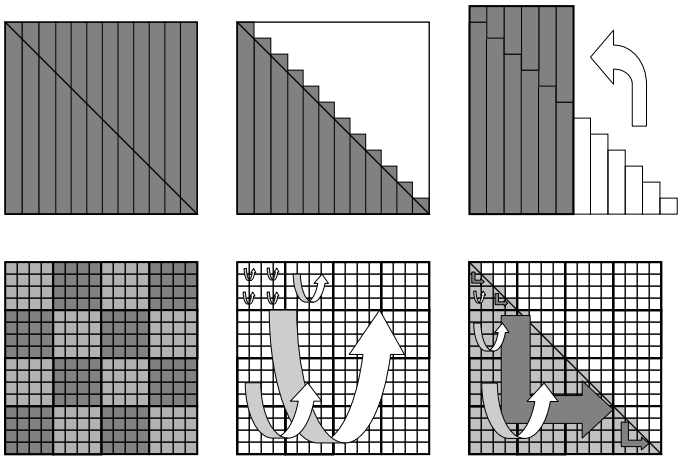

Figure 2: Underlying Data Structure.
Top (column-major): Full, Old Packed, Rectangular Full Packed.
Bottom (block-contiguous): Blocked, Recursive Format, Recursive Full Packed.

3.1.2 Accuracy and Stability of Cholesky Decomposition Algorithms.

In Section 10.1.1 of [Hig02], standard error analyses of Cholesky decomposition algorithm are given. These hold for any ordering of the summation of Equations (5) and (6), and therefore apply to all Cholesky decomposition algorithms below.

3.1.3 Arithmetic Count of Cholesky Decomposition Algorithms.

The arithmetic operations of all the sequential algorithms considered below are exactly the same, up to reordering (and correspond to Equations (5),(6)). The arithmetic count of all these algorithm is therefore the same, and is given only once.

By Equations (5),(6) the number of FLOPS for computing the (i, j) element of L, given that all the elements it depends on have already been computed, is $i + 2$. Therefore, the total arithmetic count is:

$$A(n) = \sum_{i,j \ | \ 1 \le j \le i \le n} i + 2 = \frac{n^3}{3} + \Theta(n^2)$$

3.1.4 The Naïve Left-Looking Cholesky Algorithm

We start with revisiting the two naïve algorithms (left-looking and right-looking), and see that both have non-optimal bandwidth and latency, as stated in Conclusion 1 of the Introduction.

The naïve left-looking Cholesky decomposition algorithm works as follows (see Algorithm 2 and Figure 3).

Algorithm 2 Naïve left-looking Cholesky algorithm

```
1:  for j = 1 to n do
2:    read A(j : n, j) from slow memory
3:    for k = 1 to j − 1 do
4:      read A(j : n, k) from slow memory
5:      update diagonal element: A(j, j) ← A(j, j) − A(j, k)^2
6:      for i = j + 1 to n do
7:        update j^th column element: A(i, j) ← A(i, j) − A(i, k)A(j, k)
8:      end for
9:    end for
10:   calculate final value of diagonal element: A(j, j) ← √A(j, j)
11:   for i = j + 1 to n do
12:     calculate final value of j^th column element: A(i, j) ← A(i, j)/A(j, j)
13:   end for
14:   write A(j : n, j) to slow memory
15: end for
```

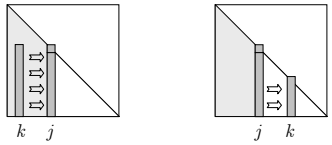

Figure 3: Naïve algorithms. Left: left-looking. Right: right-looking.
j is the currently computed column (both algorithms). Light grey is the area already computed.
k is the column being read from (left-looking) or written to (right-looking).

Analysis. Algorithm 2 assumes that two columns of the matrix can fit into fast memory simultaneously (i.e., $M > 2n$). Communication occurs at lines 2, 4, and 14, so the total number of words

transferred between fast and slow memory while executing the entire algorithm is given by

$$\sum_{j=1}^{n}\left[2(n-j+1)+\left(\sum_{k=1}^{j-1}(n-j+1)\right)\right]=\frac{1}{6}n^3+n^2+\frac{5}{6}n.$$

Assuming the matrix is stored in column-major order, each column is contiguous in memory, so the total number of messages is given by

$$\sum_{j=1}^{n}\left[2+\left(\sum_{k=1}^{j-1}1\right)\right]=\frac{1}{2}n^2+\frac{3}{2}n.$$

In the case when $M < 2n$, each column j is read into fast memory in segments of size $M/2$. For each segment of column j, the corresponding segments of previous columns k are read into fast memory individually to update the current segment. In this way, the total number of words transferred between fast and slow memory does not change. Since an entire column cannot be read/written in one message as before, the column must be read/written in multiple messages of size $O(M)$. Again, assuming the matrix is stored in column-major order, segments of columns will be contiguous in memory, and the latency is given by the bandwidth divided by the message size: $O\left(\frac{n^3}{M}\right)$. The algorithm can be adjusted to work one row at a time (up-looking) if the matrix is stored in row-major order (with no change in bandwidth or latency), but a blocked storage format will increase the latency.

3.1.5 The Naïve Right-Looking Cholesky Algorithm

The naïve right-looking Cholesky decomposition algorithm works as follows (see Algorithm 3 and Figure 3).

Algorithm 3 Naive right-looking Cholesky algorithm

```
1:  for j = 1 to n do
2:    read A(j : n, j) from slow memory
3:    calculate final value for diagonal element: A(j, j) ← sqrt(A(j, j))
4:    for i = j + 1 to n do
5:      calculate final value for j^th column element: A(i, j) ← A(i, j)/A(j, j)
6:    end for
7:    for k = j + 1 to n do
8:      read A(k : n, k) from slow memory
9:      for i = k to n do
10:       update k^th column element: A(i, k) ← A(i, k) − A(i, j)A(k, j)
11:     end for
12:     write A(k : n, k) to slow memory
13:   end for
14:   write A(j : n, j) to slow memory
15: end for
```

Analysis. Algorithm 3 assumes that two columns of the matrix can fit into fast memory simultaneously (i.e., $M > 2n$). Communication occurs at lines 2, 8, 12, and 14, so the total number of

words transferred between fast and slow memory while executing the entire algorithm is given by

$$\sum_{j=1}^{n}\left[2(n-j+1)+\sum_{k=1}^{j-1}2(n-k+1)\right]=\frac{1}{3}n^3+n^2+\frac{2}{3}n.$$

Assuming the matrix is stored in column-major order, the total number of messages is given by

$$\sum_{j=1}^{n}\left[2+\left(\sum_{k=j+1}^{n}2\right)\right]=n^2+n$$

In the case when $M < 2n$, more communication is required. In order to perform the column factorization, the column is read into fast memory in segments of size $M-1$, updated with the diagonal element (which must remain in fast memory), and written back to slow memory. In order to update the trailing k^{th} column, the k^{th} column is read into fast memory in segments of size $(M-1)/2$ along with the corresponding segment of the j^{th} column. In order to compute the update for the entire column, the k^{th} element of the j^{th} column must remain in fast memory. After updating the segment of the k^{th} column, it is written back to slow memory. Thus, Algorithm 3 requires more reads from slow memory when two columns of the matrix cannot fit into fast memory, but this increases the bandwidth only by a constant factor. As in the case of the left-looking naïve algorithm, the size of a message is no longer the entire column. Assuming the matrix is stored in column-major order, message sizes are of the order equal to the size of fast memory. Thus, for $O(n^3)$ bandwidth, the latency is $O\left(\frac{n^3}{M}\right)$. This right-looking algorithm can be easily transformed into a down-looking row-wise algorithm in order to handle row-major data storages, with no change in bandwidth or latency.

3.1.6 *LAPACK*

We next consider an implementation available in *LAPACK* (see [ABB+92]) and show that it is bandwidth-optimal (as stated in Conclusion 2 of the Introduction) and can also be made latency-optimal, assuming the correct data structure is used (as stated in Conclusion 3 of the Introduction).

Algorithm 4 LAPACK `POTRF` (blocked left-looking algorithm)

1: **for** $j = 1$ to n/b **do**
2: partition matrix so that diagonal block (j,j) is A_{22}: $\begin{pmatrix} A_{11} & * & * \\ A_{21} & A_{22} & * \\ A_{31} & A_{32} & A_{33} \end{pmatrix}$
3: update diagonal block (j,j) (`SYRK`): $A_{22} \leftarrow A_{22} - A_{21}A_{21}^T$
4: factor diagonal block (j,j) (`POTF2`): $A_{22} \leftarrow \text{Chol}(A_{22})$
5: update block column (`GEMM`): $A_{32} \leftarrow A_{32} - A_{31}A_{21}^T$
6: triangular solve for block column (`TRSM`): $A_{32} \leftarrow A_{32}A_{22}^{-T}$
7: **end for**

Analysis. The total number of words transferred between slow and fast memory while executing Algorithm 4 depends on the subroutines which perform symmetric rank-b update, matrix multiply, and triangular solve, respectively. For simplicity, we will assume that the symmetric rank-b update is computed with the same bandwidth and latency as a general matrix multiply. Although general

matrix multiply requires more communication, this assumption will not affect the asymptotic count of the communication required by the algorithm. We will also assume that the block size b is chosen so that three blocks can fit into fast memory simultaneously; that is, we assume that

$$1 \leq b \leq \sqrt{\frac{M}{3}}.$$

In this case, the factorization of the diagonal block (line 4) requires only $\Theta(b^2)$ words to be transferred since the entire block fits into fast memory. Also, in the triangular solve for the block column (line 6), the triangular matrix is only one block, so the computation can be performed by reading the blocks of the column individually and solving in fast memory. Thus, the amount of communication required by that subroutine is $\Theta(b^2)$ times the number of blocks in the column. Finally, the dimensions of the matrices in the matrix multiply of line 3 during the j^{th} iteration of the loop are $b \times b$, $b \times (j-1)b$, and $(j-1)b \times b$, and the dimensions of the matrices in the matrix multiply of line 5 during the j^{th} iteration are $(n/b-j)b \times b$, $(n/b-j)b \times (j-1)b$, and $(j-1)b \times b$. Thus, an upper bound on the number of words transferred between slow and fast memory while executing Algorithm 4 is given by

$$B(n) \leq \sum_{j=1}^{n/b} \left[B_{MM}(b,(j-1)b,b) + \Theta(b^2) + B_{MM}((n/b-j)b,(j-1)b,b) + (n/b-j)\Theta(b^2) \right]$$

where $B_{MM}(m,n,r)$ is the bandwidth required to execute a matrix multiplication of matrices of size $m \times n$ and $n \times r$ in order to update a matrix of size $n \times r$. Since B_{MM} is nondecreasing in each of its variables, we have

$$\begin{aligned} B(n) &\leq \frac{n}{b}\left[B_{MM}(b,n,b) + \Theta(b^2) + B_{MM}(n,n,b) + \frac{n}{b}\Theta(b^2) \right] \\ &\leq \frac{n}{b}\left[2B_{MM}(n,n,b) + \Theta(b^2) + \frac{n}{b}\Theta(b^2) \right]. \end{aligned}$$

Assuming the matrix–multiply algorithm used by $LAPACK$ achieves the same bandwidth as the one given in Section 3.2.4, namely $B_{MM}(n,m,r) = \Theta\left(\frac{nmr}{\sqrt{M}} + nm + mr + nr\right)$, we have,

$$B_{MM}(n,n,b) = \Theta\left(\frac{n^2 b}{\sqrt{M}} + n^2 + nb\right)$$

and since $b \leq \sqrt{M}$ and $b \leq n$, $B_{MM}(n,n,b) = O(n^2)$. Thus,

$$B(n) = O\left(\frac{n^3}{b} + n^2\right)$$

and choosing a blocksize $b = \Theta(\sqrt{M})$ gives $B(n) = O\left(\frac{n^3}{\sqrt{M}} + n^2\right)$ and achieves the lower bound (as stated in Conclusion 2 of the Introduction). Note that choosing a blocksize $b = 1$ reduces the blocked algorithm to the naïve left-looking algorithm (see Algorithm 2) which has bandwidth $O(n^3)$.

Similarly, since all reads and writes are done block by block, the optimal latency $L(n) = O\left(\frac{n^3}{M^{3/2}}\right)$ is achieved if a block-contiguous data structure is used with block size of $\Theta(\sqrt{M})$. However, as the current implementations of $LAPACK$ use column-major data structures, the latency guarantee is only $L(n) = O\left(\frac{n^3}{M} + \frac{n^2}{\sqrt{M}}\right)$ (as stated in Conclusion 3 of the Introduction).

3.1.7 Recursive Algorithm for the Sequential Model

The next recursive algorithm for Cholesky decompositionand its bandwidth analysis follow the recursive LU-decomposition algorithm of Toledo (see [Tol97]). The algorithm here is in fact a simplified version of Toledo's: there is no pivoting, and $L = U^T$. For completeness we repeat the bandwidth analysis and provide a latency analysis. The bandwidth proves to be optimal (as stated in Conclusion 2 of the Introduction), but the latency does not (as stated in Conclusion 3 of the Introduction).

Algorithm 5 $L = RectangularRChol(A)$: A Recursive (Right-Looking) Cholesky Algorithm

Input: A, an $m \times n$ section of positive semidefinite matrix ($m \geq n$). See Figure 4 for block partitioning.

Output: L, a lower triangular matrix, so that $A = L \cdot L^T$

1: **if** $n = 1$ **then**
2: $L = A/\sqrt{A(1,1)}$
3: **else**
4: $\begin{pmatrix} L_{11} \\ L_{21} \\ L_{31} \end{pmatrix} = RectangularRChol(\begin{pmatrix} A_{11} \\ A_{21} \\ A_{31} \end{pmatrix})$
5: $\begin{pmatrix} A_{22} \\ A_{32} \end{pmatrix} = \begin{pmatrix} A_{22} \\ A_{32} \end{pmatrix} - \begin{pmatrix} L_{21} \\ L_{31} \end{pmatrix} \cdot L_{2,1}^T$ \ \ Multiplication done recursively. See Section 3.2.4.
6: $\begin{pmatrix} L_{22} \\ L_{32} \end{pmatrix} = RectangularRChol(\begin{pmatrix} A_{22} \\ A_{32} \end{pmatrix})$
7: $L_{12} = 0$
8: **end if**
9: **return** L

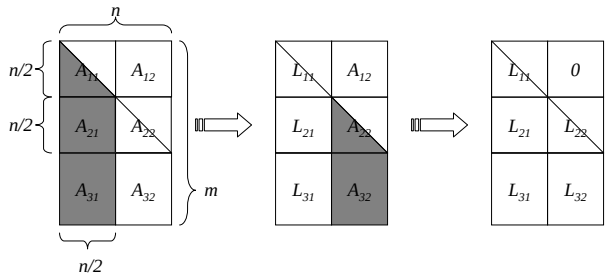

Figure 4: Rectangular Recursive Cholesky Algorithm.

Analysis. The analysis of the bandwidth follows that of Toledo [Tol97]. By the recursive algorithm we either read/write one column (if $n = 1$) or make two recursive calls and perform one matrix multiplication. Therefore,

$$B(m,n) = \begin{cases} B\left(m, \frac{n}{2}\right) + B_{MM}(m - \frac{n}{2}, \frac{n}{2}, \frac{n}{2}) + B\left(m - \frac{n}{2}, \frac{n}{2}\right) & \text{if } n > 1 \\ 2m & \text{if } n = 1 \end{cases} \tag{9}$$

Claim 3.1. $B(m,n) = \Theta\left(\frac{mn^2}{\sqrt{M}} + mn\log n\right)$

Proof. We start with upper bound for B. As $B_{MM}(n,m,r) = \Theta\left(\frac{nmr}{\sqrt{M}} + nm + mr + nr\right)$, (see 3.2.4) we have

$$B(m,n) \leq \begin{cases} 2B\left(m, \frac{n}{2}\right) + O\left(\frac{mn^2}{\sqrt{M}} + mn + n^2\right) & \text{if } n > 1 \\ 2m & \text{if } n = 1. \end{cases} \tag{10}$$

Consider the recursion tree T given by Equation (10). Since T is a binary tree of depth $\log n$, it has n leaves. Each leaf contributes $2m$, so the total contribution of the leaves is $2mn$.

Now consider the contribution of the internal nodes of T. Note that the contribution of an internal node at depth d is

$$O\left(\frac{m(\frac{n}{2^d})^2}{\sqrt{M}} + \frac{mn}{2^d} + \left(\frac{n}{2^d}\right)^2\right),$$

and there are 2^d nodes at depth d. Therefore, the total contribution of internal nodes is

$$\begin{aligned} &\sum_{d=1}^{\log n} 2^d \cdot O\left(\frac{m(\frac{n}{2^d})^2}{\sqrt{M}} + 2m\frac{n}{2^d} + \left(\frac{n}{2^d}\right)^2\right) \\ &= O\left(\frac{mn^2}{\sqrt{M}} \sum_{d=1}^{\log n} \frac{1}{2^d} + 2mn\log n + n^2 \sum_{d=1}^{\log n} \frac{1}{2^d}\right) \\ &= O\left(\frac{mn^2}{\sqrt{M}} + mn\log n\right) \end{aligned}$$

Thus, the total contribution of T is $B(m,n) = O\left(\frac{mn^2}{\sqrt{M}} + mn\log n\right)$.

In order to find a lower bound for $B(m,n)$, we consider the recursion tree for Equation (9). Assuming without loss of generality that $n > 1$ (so that the root node has two children), we have $B(m,n) \geq B\left(m, \frac{n}{2}\right)$ (the contribution of the subtree with root node having arguments m and $n/2$). Since $m \geq n$, the first argument of every node in this subtree is at least $\frac{m}{2}$. Thus, there are $\frac{n}{2}$ leaves each contributing at least $\frac{m}{2}$ words for a total contribution of $\Omega(mn)$. The contribution of an internal node of the subtree at depth d is at least $B_{MM}\left(\frac{m}{2}, \frac{n}{2^d}, \frac{n}{2^d}\right)$, so the total contribution of the internal nodes is at least

$$\begin{aligned} &\sum_{d=2}^{\log n - 1} 2^{d-1} \cdot B_{MM}\left(\frac{m}{2}, \frac{n}{2^d}, \frac{n}{2^d}\right) \\ &= \sum_{d=2}^{\log n - 1} 2^{d-1} \cdot \Omega\left(\frac{(\frac{m}{2})(\frac{n}{2^d})^2}{\sqrt{M}} + \left(\frac{m}{2}\right)\left(\frac{n}{2^d}\right) + \left(\frac{n}{2^d}\right)^2\right) \\ &= \Omega\left(\frac{mn^2}{\sqrt{M}}\left(\sum_{d=2}^{\log n - 1} \frac{1}{2^{d-1}}\right) + mn\log n + n^2\left(\sum_{d=2}^{\log n - 1} \frac{1}{2^{d-1}}\right)\right) \\ &= \Omega\left(\frac{mn^2}{\sqrt{M}} + mn\log n\right). \end{aligned}$$

Thus we have $B(m,n) = \Omega\left(\frac{mn^2}{\sqrt{M}} + mn\log n\right)$.

□

Thus, the application of the algorithm to an n-by-n matrix (where $n^2 > M$) yields a bandwidth of $B(n,n) = \Theta\left(\frac{n^3}{\sqrt{M}} + n^2 \log n\right)$. This is optimal, provided that $M \geq \frac{n^2}{\log^2 n}$, or, since any algorithm is bandwidth optimal when the whole matrix fits into fast memory, when $n^2 \leq M$. Here the range of non-optimality is very small, so we consider Algorithm 5 to have optimal bandwidth.

Latency. For the latency $L(n)$, we consider the column-major and recursive data storage formats. In order to show that this algorithm does not attain optimal latency, we provide lower bounds in each case.

In the case of column-major storage, the latency of the entire algorithm is bounded below by the multiplication of the lower left quarter of the matrix with its transpose (computed after the left half of the matrix is factored). This is a square multiplication of size $\frac{n}{2}$, so the latency with column major storage is $\Omega\left(\frac{n^3}{M}\right)$ (from Section 3.2.4). Thus, using column major storage, Algorithm 5 can never be latency optimal.

In the case of block-recursive storage, the latency of the entire algorithm is bounded below by the latency required to resolve the base cases of the recursion. The base case occurs when $n = 1$, and the algorithm factors a single column. However, the block-recursive data structure does not store columns contiguously (up to 2 elements of the same column could be stored consecutively, depending on the recursive pattern), so reading a column requires $\Omega(n)$ messages. Since a base case is reached for each column, the latency contribution of all the base cases is $\Omega(n^2)$, a lower bound for the total latency of Algorithm 5. Since n^2 asymptotically dominates $\frac{n^3}{M^{3/2}}$ for $M > n^{2/3}$, this algorithm cannot be latency-optimal in this case. It is possible that the algorithm does perform optimally when $M < n^{2/3}$, although we make no claim here.

3.2 Sequential Algorithm for more than Two Memory Levels

In real life, there are usually more than two types of memory, and in fact there is a hierarchy of memories, ranging from a very fast small memory to the largest and slowest memory [AGW01]. We next consider the lower and upper bounds of Cholesky decomposition assuming such a model of *hierarchical memory machines*. We observe that none of the known algorithms for Cholesky decomposition allows cache oblivious optimal latency (Conclusion 4 of the Introduction), except the Square Recursive Cholesky algorithm (Conclusion 5 of the Introduction).

3.2.1 Lower bound

The hierarchy model has been given explicit lower bounds for various problems, including matrix multiplication (see for example, [Sav95]). Moreover, the known lower bound for the two-level memory model can be applied here, by considering any two consecutive level of the hierarchy as the fast and slow memories, and treating all faster memories as part of the fast memory, and all the slower memories as part of the slow one. Assuming the number of levels is some constant, this may be the correct lower bound, up to a constant factor. For the Cholesky decomposition, this gives the following bandwidth and latency lower bounds:

Corollary 3.2. *Let Alg be a 'classical' Cholesky decomposition algorithm implemented on a machine with d levels of memory, of size $M_1 \leq \cdots \leq M_d$, with inverse bandwidth $\beta_1 \leq \cdots \leq \beta_d$ and with latency $\alpha_1 \leq \cdots \alpha_d$. Then the bandwidth cost of Alg is*

$$B(n) = \Omega\left(\sum_{i \in [d-1]} \left\{\beta_i \cdot \left(\frac{n^3}{\sqrt{M_i}} - M_i\right)\right\}\right) \tag{11}$$

and the latency cost is

$$L(n) = \Omega\left(\sum_{i\in[d-1]}\left\{\alpha_i \cdot \frac{n^3}{M_i^{3/2}}\right\}\right) \tag{12}$$

3.2.2 Upper bounds revisited

An algorithm may perform not optimally on a hierarchical memory model due to various reasons. It may have below optimal bandwidth or latency in the two memory levels model, which implies the same for the hierarchical memory model (the parameter may get worse).

Moreover, an algorithm that performs optimally on a machine with two memory levels may not necessarily perform as well on a machine with deeper memory hierarchy. For example, an algorithm that has a parameter (e.g., block size) which allows tuning for better communication performance, may be harder or impossible to tune optimally in the hierarchical memory model. We next revisit the communication performance of the Cholesky decomposition algorithms above in the context of hierarchical memory model.

The $LAPACK$ Implementation. In the two-level memory model, the $LAPACK$ implementation can achieve optimal bandwidth if column-major data structure is used, and both optimal bandwidth and optimal latency, if block-contiguous data structure is used. To this end one has to tune the block size parameter b of the algorithm (and the block size of the data structure, in case a non recursive data structure is used).

This has a drawback when applying the $LAPACK$ algorithm to a multi-level memory machines: setting b to fit a smaller memory level results in inefficient bandwidth and latency in the higher levels. Setting b according to a larger memory level results in block readings that are too large to fit the smaller levels. Either way, we end up with non-optimal bandwidth and latency.

The Recursive Algorithm. Recall that the recursive Cholesky decomposition algorithm adopted from [Tol97], cannot guarantees optimality of latency when $M > n^{2/3}$. Therefore, if, for example, there is a memory level of size $> n^{2/3}$ (other than the slowest one holding the original input) this algorithm yields suboptimal latency.

An Optimal Algorithm? An algorithm that performs well for *any* fast memory size (with no need for cache-aware tuning) may also perform well on an multi-level memory machine. To see that, split the memory hierarchy at some level i, and consider only communication performed between levels i and $i+1$. That is, we consider all memory faster than the ith level as fast-memory, and all other levels as slow-memory. Now, as the algorithm performs optimally (regardless of the memory size) we are guaranteed that the bandwidth and the latency between levels i and $i+1$ is optimal. As this holds for every i, we are done.

Note that for this argument to hold, the algorithm has to have no explicit or implicit parameter which has to be (automatically) adapted to the fast memory size of the machine. Such is the case of the bandwidth of the above recursive algorithm. It does not hold for algorithms that automatically self-tune a fast-memory size dependent parameter.

3.2.3 Square Recursive Cholesky decomposition algorithm

We are interested in an algorithm that has optimal latency and bandwidth for any fast memory size M that needs no tuning. Such algorithm is also an algorithm that has a tight latency and

bandwidth for hierarchical memory model (with no need for cache-aware tuning).

This is obtained in [AP00] using the next algorithm where the underlying data structure at each memory level is block-contiguous recursive format.

Algorithm 6 $L = SquareRChol(A)$: Square Recursive Cholesky Algorithm

Input: A, an $n \times n$ semidefinite matrix.
Output: L, a lower triangular matrix, so that $A = L \cdot L^T$

1: **if** $n = 1$ **then**
2: $L = \sqrt{A(1,1)}$
3: **else**
4: $L_{11} = SquareRChol(A_{11})$
5: $L_{21} = RTRSM(A_{21}, L_{11}^T)$ \ \ See Section 3.2.5 for $RTRSM$.
6: $A_{22} = A_{22} - L_{21} \cdot L_{21}^T$ \ \ Multiplication done recursively. See Section 3.2.4.
7: $L_{22} = SquareRChol(A_{22})$
8: **end if**
9: **return** L

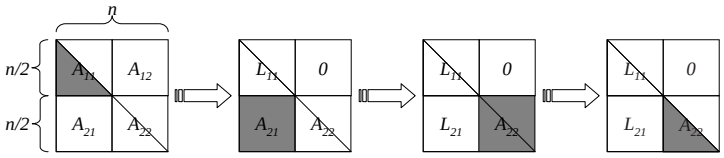

Figure 5: Square Recursive Cholesky Algorithm.

Analysis. We next analyze the bandwidth and latency of this algorithm. The analyses of the recursive algorithms for matrix multiplication and for triangular solving follow.

Ahmed and Pingali [AP00] suggest this algorithm (however with no asymptotic analysis of the bandwidth and latency). In [ST04] this algorithm is also considered, with no specific assumption on the data structure. A detailed probabilistic cache-misses performance is given, however no asymptotic claims on the worst-case bandwidth and latency are stated.

We note that a recursive algorithm similar to Algorithm 6 is given in [AGW01, GJ01], which is bandwidth-optimal, for two- and multi-levels memory machines, but is not latency-optimal (neither the latency nor the bandwidth is explicitly given in that paper). This algorithm uses a recursive packed data structure (saving half the storage space). Since it is designed to utilize BLAS3 GEMM subroutines, elements of recursively sized blocks are stored contiguously in column-major order. That is, one fourth of the original matrix (half of the packed matrix) is a contiguous block whose elements are stored in column-major order. For this reason, the algorithm cannot attain optimal latency, although it does perform as well as full storage LAPACK routines which use column-major ordering.

Bandwidth. As no communication is needed for sufficiently small matrices (other than reading the entire input, and writing the output), the bandwidth of this algorithm is

$$B(n) = \begin{cases} 2 \cdot B\left(\frac{n}{2}\right) + O\left(\frac{n^3}{\sqrt{M}} + n^2\right) & \text{if } n > \sqrt{\frac{M}{3}} \\ 2n^2 & \text{otherwise,} \end{cases} \tag{13}$$

where the second summand stands for the matrix multiplication (including the subtraction) and for the triangular solver. Consider the recursion tree T with $n > \sqrt{\frac{M}{3}}$. The contribution from the internal nodes is

$$\sum_{i=0}^{d} 2^i \cdot O\left(\frac{(\frac{n}{2^i})^3}{\sqrt{M}} + \left(\frac{n}{2^i}\right)^2\right) = O\left(\frac{n^3}{\sqrt{M}}\sum_{i=0}^{d}\frac{1}{4^i} + n^2\sum_{i=0}^{d}\frac{1}{2^i}\right) = O\left(\frac{n^3}{\sqrt{M}} + n^2\right).$$

Since each internal node of T has 2 children and T has depth $d = \log\frac{n}{\sqrt{M}}$, there are $2^d = \frac{n}{\sqrt{M}}$ leaves, each of which contributes $O(M)$ words to the total bandwidth. Thus, the total contribution from the leaves is $O\left(n\sqrt{M}\right)$. In the case that $n > \sqrt{\frac{M}{3}}$, the dominating term is $\frac{n^3}{\sqrt{M}}$. In the case that $n \leq \sqrt{\frac{M}{3}}$, the bandwidth is $O(n^2)$. Thus, the bandwidth for general matrices is given by

$$B(n) = O\left(\frac{n^3}{\sqrt{M}} + n^2\right).$$

Latency. Assuming recursive-block data structure, the latency is

$$L(n) = \begin{cases} 2 \cdot L\left(\frac{n}{2}\right) + O\left(\frac{n^3}{M^{\frac{3}{2}}} + \frac{n^2}{M}\right) & \text{if } n > \sqrt{\frac{M}{3}} \\ 2 & \text{otherwise,} \end{cases} \tag{14}$$

and by similar analysis we find the leaves of the recursion tree contribute $O\left(\frac{n}{\sqrt{M}}\right)$ and the internal nodes contribute $O\left(\frac{n^3}{M^{3/2}} + \frac{n^2}{M}\right)$, so we have

$$L(n) = O\left(\frac{n^3}{M^{3/2}}\right).$$

3.2.4 Upper Bound for Matrix Multiplication

In this section we recall the matrix multiplication algorithm of Frigo, Leiserson, Prokop and Ramachandran [FLPR99]. Their algorithm works by straightforward divide-and-conquer approach, where at each step the algorithm splits the largest of three dimensions.

Theorem 3 ([FLPR99]). *The bandwidth $B_{MM}(n, m, r)$ of multiplying two matrices of dimensions $n \times m$ and $m \times r$ is*

$$B_{MM}(n, m, r) = \Theta\left(\frac{nmr}{\sqrt{M}} + nm + mr + nr\right)$$

This holds for the multilevel model as well, with M_i replacing M (see Section 3.2 for definitions).

The proof follows the one in [FLPR99], differing slightly[7].

Proof. Without loss of generality, we assume that each dimension is a power of 2. We define α as the largest positive constant such that $\max\{m', n', r'\} \leq \alpha\sqrt{M}$ implies that three matrices of size $m' \times n'$, $n' \times r'$, and $m' \times r'$ can fit in fast memory simultaneously. We then distinguish four cases based on the relative sizes of the original matrices to the size of the fast memory.

[7]Formally, there is a slight variation in the way communication is measured in [FLPR99] and here.

Algorithm 7 $C = RMatMul(A, B)$: A Recursive Matrix Multiplication Algorithm

Input: A, B, two matrices of dimensions $m \times n$ and $m \times r$.

Output: C, a matrix, so that $C = A \cdot B$

\\ Here $A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}$ where we divide each dimension in half, and similarly for B and C.

if $n = m = r = 1$ **then**
  $C_{11} = A_{11} \cdot B_{11}$
else if $n = \max\{n, m, r\}$ **then**
  $\begin{pmatrix} C_{11} & C_{12} \end{pmatrix} = RMatMul(\begin{pmatrix} A_{11} & A_{12} \end{pmatrix}, B)$
  $\begin{pmatrix} C_{21} & C_{22} \end{pmatrix} = RMatMul(\begin{pmatrix} A_{21} & A_{22} \end{pmatrix}, B)$
else if $m = \max\{n, m, r\}$ **then**
  $C = RMatMul\left(\begin{pmatrix} A_{11} \\ A_{21} \end{pmatrix}, \begin{pmatrix} B_{11} & B_{12} \end{pmatrix}\right)$
  $C = C + RMatMul(\begin{pmatrix} A_{12} \\ A_{22} \end{pmatrix}, \begin{pmatrix} B_{21} & B_{22} \end{pmatrix})$
else
  $\begin{pmatrix} C_{11} \\ C_{21} \end{pmatrix} = RMatMul\left(A, \begin{pmatrix} B_{11} \\ B_{21} \end{pmatrix}\right)$
  $\begin{pmatrix} C_{12} \\ C_{22} \end{pmatrix} = RMatMul\left(A, \begin{pmatrix} B_{12} \\ B_{22} \end{pmatrix}\right)$
end if
return C

Case I: $m, n, r > \alpha\sqrt{M}$. In this case, all dimensions are large, and at each level of recursion, the largest dimension is halved. Thus the recurrence is

$$B(m,n,r) = \begin{cases} \Theta(mn + nr + mr) & \text{if } m, n, r \leq \alpha\sqrt{M}, \\ 2B(m/2, n, r) & \text{else if } m \geq n, r \\ 2B(m, n/2, r) & \text{else if } n \geq r \\ 2B(m, n, r/2) & \text{else.} \end{cases}$$

The base case arises when all three matrices fit into fast memory. Let m', n', r' be the dimensions at the base case, then

$$\frac{\alpha}{2}\sqrt{M} \leq m', n', r' \leq \alpha\sqrt{M}$$

and thus $m', n', r' = \Theta(\sqrt{M})$. The depth of the binary recursion tree is given by $d = \log \frac{m}{m'} + \log \frac{n}{n'} + \log \frac{r}{r'}$, so there are $2^d = \frac{mnr}{m'n'r'}$ leaves. Each leaf requires a bandwidth of $m'n' + n'r' + m'r'$, to the total bandwidth is

$$\Theta\left(\frac{mnr}{r'} + \frac{mnr}{m'} + \frac{mnr}{n'}\right) = \Theta\left(\frac{mnr}{\sqrt{M}}\right).$$

Case II: $m, n > \alpha\sqrt{M}$ and $r \leq \alpha\sqrt{M}$. In this case, one dimension is so small that it will never be halved in the recursion. We consider the case where r is small; the other cases are proved in the same way. Here, the recursion is

$$B(m,n,r) = \begin{cases} \Theta(mn + nr + mr) & \text{if } m, n \leq \alpha\sqrt{M}, \\ 2B(m/2, n, r) & \text{else if } m \geq n \\ 2B(m, n/2, r) & \text{else.} \end{cases}$$

Let m', n', r be the dimensions when the recursion reaches the base case, then $m', n' = \Theta(\sqrt{M})$ as before. The depth of this recursion tree is $d = \log \frac{m}{m'} + \log \frac{n}{n'}$, so there are $2^d = \frac{mn}{m'n'}$ leaves. Each leaf contributes a bandwidth of $\Theta(m'n' + n'r + m'r)$, so the total bandwidth is

$$\Theta\left(mn + \frac{mnr}{m'} + \frac{mnr}{n'}\right) = \Theta\left(mn + \frac{mnr}{\sqrt{M}}\right) = \Theta(mn).$$

Case III: $m > \alpha\sqrt{M}$ and $n, r \leq \alpha\sqrt{M}$. In this case, two dimensions are so small that neither will never be halved in the recursion. We consider the case were n and r are small; the other cases are proved in the same way. Here the recursion is

$$B(m, n, r) = \begin{cases} \Theta(mn + nr + mr) & \text{if } m \leq \alpha\sqrt{M}, \\ 2B(m/2, n, r) & \text{else.} \end{cases}$$

Let m' be the dimension when the recursion reaches the base case, then $m' = \Theta(\sqrt{M})$. The depth of this recursion tree is $d = \log \frac{m}{m'}$, so there are $2^d = \frac{m}{m'}$ leaves. Each leaf contributes a bandwidth of $\Theta(m'n + nr + m'r)$, so the total bandwidth is

$$\Theta\left(mn + \frac{mnr}{m'} + mr\right) = \Theta\left(mn + \frac{mnr}{\sqrt{M}} + mr\right) = \Theta(mn + mr).$$

Case IV: $m, n, r \leq \alpha\sqrt{M}$. In this case, the original matrices are small enough to all fit in fast memory, so the bandwidth is $\Theta(mn + nr + mr)$. □

The latency performance of this algorithm varies according to the data structure used and the dimensions of the input and output matrices. We only consider the case of square matrices.

Claim 3.3. *If all three matrices are square, then the latency is*

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

when using recursive contiguous-block data structure, and

$$L_{MM}(n) = L_{MM}(n, n, n) = \Theta\left(\frac{n^3}{M}\right)$$

when using column-major or row-major layout.

Proof. Let m', n', r' be the dimensions of the problem the first time all three matrices fit into the fast memory. m', n', r' are at most a factor 2 one away from the other and are all $\Theta(\sqrt{M})$. Therefore, each of the three matrices reside in $O(1)$ square blocks, and therefore their reading and writing incur a latency of $\Theta(1)$, assuming the recursive data structure. Thus, the total latency is $L_{MM}(n) = \Theta(B_{MM}(n)/M)$.

If the column-major or row-major data structures are used, then each such block of size $\Theta(\sqrt{M}) \times \Theta(\sqrt{M})$ is read or written using $\Theta(\sqrt{M})$ messages (one for each of its rows or columns), and therefore the total latency is $L_{MM}(n) = \Theta(B_{MM}(n)/\sqrt{M})$. □

3.2.5 Recursive TRSM

Algorithm 8 $X = RTRSM(A, U)$: Recursive Triangular Solver

Input: A, U two $n \times n$ matrices, U is upper triangular.

Output: X, so that $X = A \cdot U^{-1}$

```
if n = 1 then
    X = A/U
else
    X_11 = RTRSM(A_11, U_11)
    X_12 = RTRSM(A_12 − X_11 · U_12, U_22)
    X_21 = RTRSM(A_21, U_11)
    X_22 = RTRSM(A_22 − X_21 · U_12, U_22)
end if
return X
```

Analysis. We next analyze the bandwidth and latency of this algorithm.

Bandwidth. As no communication is needed for sufficiently small matrices (other than reading the entire input, and writing the output), the bandwidth of this algorithm is

$$B(n) = \begin{cases} 4 \cdot B\left(\frac{n}{2}\right) + 2 \cdot B_{MM}(\frac{n}{2}) & \text{if } n > \sqrt{\frac{M}{3}} \\ 3n^2 & \text{otherwise,} \end{cases} \tag{15}$$

where $B_{MM}(n) = B_{MM}(n, n, n)$ is the bandwidth complexity of two n-by-n matrices multiplication. If the matrix-multiplication is done communication efficiently (e.g., by the recursive algorithm), then $B_{MM}(n) = O\left(\frac{n^3}{\sqrt{M}} + n^2\right)$. Consider the recursion tree T. Each internal node has 4 children and the tree has depth $d = \log \frac{n}{\sqrt{M}}$, so there are $4^d = \frac{n^2}{M}$ leaves. Each leaf contributes $3\left(\frac{n}{2^d}\right)^2 = 3M$ words to the total bandwidth. Thus the contribution of all the leaves is $O(n^2)$. The contribution of the internal nodes is

$$\begin{aligned} &\sum_{i=0}^{d} 4^i \cdot O\left(\frac{(\frac{n}{2^i})^3}{\sqrt{M}} + \left(\frac{n}{2^i}\right)^2\right) \\ &= O\left(\frac{n^3}{\sqrt{M}} \sum_{i=0}^{d} \frac{1}{2^i} + n^2 d\right) \\ &= O\left(\frac{n^3}{\sqrt{M}} + n^2 \log \frac{n}{\sqrt{M}}\right) \\ &= O\left(\frac{n^3}{\sqrt{M}}\right) \end{aligned}$$

so the the total bandwidth is

$$B(n) = O\left(\frac{n^3}{\sqrt{M}} + n^2\right).$$

Latency. Assuming contiguous-block data structure (recursive, or with the correct block size picked), the latency is

$$L(n) \leq \begin{cases} 4 \cdot L\left(\frac{n}{2}\right) + 2 \cdot L_{MM}(\frac{n}{2}) & \text{if } n > \sqrt{\frac{M}{3}} \\ 3 & \text{otherwise,} \end{cases} \tag{16}$$

where $L_{MM}(n) = O\left(\frac{n^3}{M^{3/2}}\right)$ is the bandwidth complexity of matrix multiplication performed efficiently (e.g., by the recursive algorithm). Thus, by similar analysis to the bandwidth, the leaf nodes contribute $O\left(\frac{n^2}{M}\right)$ messages and the internal nodes contribute $O\left(\frac{n^3}{M^{3/2}}\right)$. Thus, we have

$$L(n) = O\left(\frac{n^3}{M^{3/2}}\right).$$

3.3 2D Parallel Algorithms

Let us now consider the so-called $2D$ parallel algorithms, namely those that assume $M = \frac{N^2}{P}$ local memory size and start with the n^2 matrix elements spread across the processors (i.e, no repetition of elements). We show a matching (up to a logarithmic factor) lower and upper bounds (as stated in Conclusion 6 of the Introduction).

3.3.1 The *ScaLAPACK* Implementation

The ScaLAPACK [BJCD+97] routine `PxPOTRF` computes the Cholesky decomposition of a symmetric positive definite distributed matrix A of order n over a grid of P processors. The matrix is distributed block-cyclically, but only half of the matrix is referenced or overwritten (we assume the lower half here).

Algorithm 9 ScaLAPACK routine `PxPOTRF`

```
1: for j = 1 to n/b do
2:    processor owning block (j, j) computes Cholesky decomposition
3:    broadcast result to processors down processor column
4:    parallel for each processor owning blocks in column panel j do
5:       update blocks in panel with triangular solve
6:       broadcast results across processor row
7:    end for
8:    parallel for each processor owning diagonal block (i, i) (i > j) do
9:       re-broadcast results down processor column
10:   end for
11:   parallel for each processor owning blocks in trailing matrix do
12:      update blocks with symmetric rank-b update
13:   end for
14: end for
```

Analysis. We assume that the block dimension is $b \times b$ where n/b is an integer and the processors are organized in a square grid ($P_r = P_c = \sqrt{P}$) where $n/\sqrt{P}$ is also an integer. After computing the Cholesky decomposition of a diagonal block (j, j) locally, the result must be communicated to all processors which own blocks in the column panel below the diagonal block (i.e., blocks (i, j) for $j < i \leq n/b$) in order to update those blocks. Since the matrix is distributed block-cyclically, there can be at most $\sqrt{P}$ such processors. After the column panel is updated using a triangular solve with multiple RHS, those results must be communicated to other processors in order to perform the rank-b updates to blocks in the trailing matrix. For a given block (k, l) in the trailing matrix A_{22}, the update depends on the k^{th} block of the column panel (block (k, j)) and the transpose of the l^{th} block of the column panel (block (l, j)). Thus, after a processor computes an update to

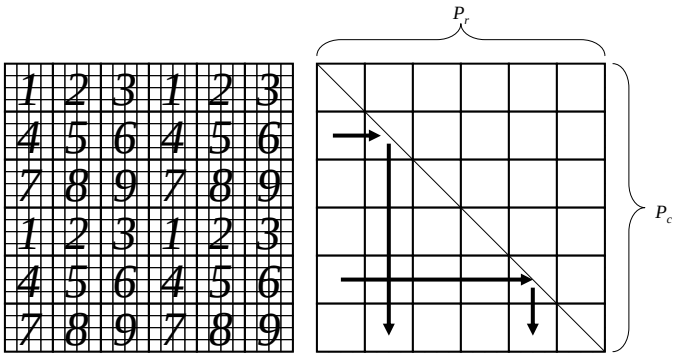

Figure 6: *ScaLAPACK*.
Left: Block-cyclical distribution of the matrix to the processors.
Here $n = 24, b = 4, P = 9, P_c = P_r = 3$.
Right: Information flow.

block (i, j) in the column panel, it must broadcast the result to processors which own blocks in the i^{th} row panel ($P_r = \sqrt{P}$ different processors). Then after the processor owning the diagonal block (i, i) receives that update, it re-broadcasts to processors which own blocks in the i^{th} column panel ($P_c = \sqrt{P}$ different processors).

The result of the Cholesky decomposition of the diagonal block is a lower triangular matrix, so the number of words in each message of the broadcast down the column is only $b(b+1)/2$. A broadcast to P_c processors requires $\log P_c$ messages. Any processor which owns a block in the column panel below the diagonal block (j, j) will own $\frac{n/b}{P_r} = \frac{n}{P_r b}$ blocks. Such a processor computes the updates for all the blocks it owns, and then broadcasts all the results together (total of $\frac{n}{P_r b}$ words) to P_r processors (which is done using $\log P_r$ messages). Once a processor owning the corresponding diagonal blocks receives this message, it re-broadcasts the message down the column, requiring another $\log P_c$ messages. Thus, the total number of messages along the critical path is

$$\begin{aligned}
&\sum_{j=1}^{n/b} \log P_r + (\log P_r + \log P_c) \\
&= \frac{n}{b}(2 \log P_r + \log P_c) \\
&= \frac{3}{2}\frac{n}{b} \log P
\end{aligned}$$

and the number of words along the critical path is

$$\begin{aligned}
&\sum_{j=1}^{n/b} \frac{b^2}{2} \log P_r + \frac{nb}{P_r}(\log P_r + \log P_c) \\
&= \frac{nb}{2} \log P_r + \frac{n^2}{P_r}(\log P_r + \log P_c) \\
&= \left(\frac{nb}{4} + \frac{n^2}{\sqrt{P}}\right) \log P
\end{aligned}$$

Thus, setting the block size at $b = \frac{n}{\sqrt{P}}$, the total number of messages required is

$$\frac{3}{2}\sqrt{P} \log P$$

and the total number of words is

$$\frac{5}{4}\frac{n^2}{\sqrt{P}}\log P.$$

We note that by setting $b = \frac{n}{\sqrt{P}}$, the matrix is no longer stored block-cyclically. Instead, each processor owns one large block of the matrix, and since the full matrix is stored, nearly half of the processors own blocks which are never referenced or updated. Further, parallelism is lost as the algorithm progresses across the column panels. However, this increases the computation time of the parallel algorithm only by a constant factor. For each column panel, there are three phases of computation: Cholesky decomposition of the diagonal block, triangular solve to update the column panel, and matrix multiply to update the trailing matrix. Since each of these computations requires $O(n^3)$ flops for a problem of order n, the total number of flops required along the critical path in executing Algorithm 9 is

$$\begin{aligned}
&\sum_{j=1}^{n/b}\left[O(b^3)+\frac{(n/b-j)}{\sqrt{P}}O(b^3)+\sum_{k=j+1}^{n/b}\frac{(n/b-k)}{P}O(b^3)\right]\\
&\leq \sum_{j=1}^{n/b}\left[O(b^3)+\frac{n/b}{\sqrt{P}}O(b^3)+\sum_{k=j+1}^{n/b}\frac{n/b}{P}O(b^3)\right]\\
&\leq \frac{n}{b}\left[O(b^3)+\frac{n/b}{\sqrt{P}}O(b^3)+\frac{n}{b}\frac{n/b}{P}O(b^3)\right]\\
&\leq O(nb^2)+O\left(\frac{n^2b}{\sqrt{P}}\right)+O\left(\frac{n^3}{P}\right)
\end{aligned}$$

and in the case $b = \frac{n}{\sqrt{P}}$, each of these terms is $O\left(\frac{n^3}{P}\right)$. Thus, in choosing a large block size to attain the latency lower bound, we do not sacrifice the algorithm's ability to meet the computational asymptotic lower bound.

References

[ABB+92] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK's user's guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1992. Also available from http://www.netlib.org/lapack/.

[AGW01] B. S. Andersen, F. G. Gustavson, and J. Wasniewski. A recursive formulation of Cholesky factorization of a matrix in packed storage format. *ACM Transactions on Mathematical Software*, 27(2):214–244, jun 2001.

[AP00] N. Ahmed and K. Pingali. Automatic generation of block-recursive codes. In *Euro-Par '00: Proceedings from the 6th International Euro-Par Conference on Parallel Processing*, pages 368–378, London, UK, 2000. Springer-Verlag.

[BJCD+97] L. S. Blackford, A. Cleary J. Choi, E. DAzevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide*. SIAM, Philadelphia, PA, USA, May 1997. Also available from http://www.netlib.org/scalapack/.

[DGHL08a] J. Demmel, L. Grigori, M. Hoemmen, and J. Langou. Communication-optimal parallel and sequential QR and LU factorizations. UC Berkeley Technical Report EECS-2008-89, Aug 1, 2008; Submitted to SIAM. J. Sci. Comp., 2008.

[DGHL08b] J. Demmel, L. Grigori, M. Hoemmen, and J. Langou. Implementing communication-optimal parallel and sequential QR and LU factorizations. submitted to SIAM. J. Sci. Comp., 2008.

[DGX08] J. Demmel, L. Grigori, and H. Xiang. Communication-avoiding Gaussian elimination. Supercomputing 08, 2008.

[EGJK04] E. Elmroth, F. G. Gustavson, I. Jonsson, and B. Kågström. Recursive blocked algorithms and hybrid data structures for dense matrix library software. 46(1):3–45, March 2004.

[FLPR99] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *FOCS '99: Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, page 285, Washington, DC, USA, 1999. IEEE Computer Society.

[GJ01] F. G. Gustavson and I. Jonsson. High performance Cholesky factorization via blocking and recursion that uses minimal storage. In *PARA '00: Proceedings of the 5th International Workshop on Applied Parallel Computing, New Paradigms for HPC in Industry and Academia*, pages 82–91, London, UK, 2001. Springer-Verlag.

[Hig02] N. J. Higham. *Accuracy and Stability of Numerical Algorithms*. SIAM, Philadelphia, PA, 2nd edition, 2002.

[HK81] J. W. Hong and H. T. Kung. I/O complexity: The red-blue pebble game. In *STOC '81: Proceedings of the thirteenth annual ACM symposium on Theory of computing*, pages 326–333, New York, NY, USA, 1981. ACM.

[IEE08] IEEE standard for floating-point arithmetic. *IEEE Std. 754-2008*, pages 1–58, 29 2008.

[ITT04] D. Irony, S. Toledo, and A. Tiskin. Communication lower bounds for distributed-memory matrix multiplication. *J. Parallel Distrib. Comput.*, 64(9):1017–1026, 2004.

[Sav95] J. E. Savage. Extending the Hong-Kung model to memory hierarchies. In *COCOON*, pages 270–281, 1995.

[ST04] I. Simecek and P. Tvrdik. Analytical model for analysis of cache behavior during Cholesky factorization and its variants. In *ICPPW '04: Proceedings of the 2004 International Conference on Parallel Processing Workshops*, pages 190–197, Washington, DC, USA, 2004. IEEE Computer Society.

[Tol97] S. Toledo. Locality of reference in LU decomposition with partial pivoting. *SIAM J. Matrix Anal. Appl.*, 18(4):1065–1081, 1997.