On the Parallel I/O Optimality of Linear Algebra Kernels: 
Near-Optimal Matrix Factorizations

Grzegorz Kwasniewski\textsuperscript{1}, Marko Kabic\textsuperscript{12}, Tal Ben-Nun\textsuperscript{1}, Alexandros Nikolaos Ziogas\textsuperscript{1}, Jens Eirik Saethre\textsuperscript{1}, André Gaillard\textsuperscript{1}, Timo Schneider\textsuperscript{1}, Maciej Besta\textsuperscript{1}, Anton Kozhevnikov\textsuperscript{12}, Joost VandeVondele\textsuperscript{12}, Torsten Hoefler\textsuperscript{1}

\textsuperscript{1}Department of Computer Science, ETH Zurich, \textsuperscript{2}Swiss National Computing Center

\textbf{ABSTRACT}

Matrix factorizations are among the most important building blocks of scientific computing. However, state-of-the-art libraries are not communication-optimal, underutilizing current parallel architectures. We present novel algorithms for Cholesky and LU factorizations that utilize an asymptotically communication-optimal 2.5D decomposition. We first establish a theoretical framework for deriving parallel I/O lower bounds for linear algebra kernels, and then utilize its insights to derive Cholesky and LU schedules, both communicating \( N^3/(P \sqrt{M}) \) elements per processor, where \( M \) is the local memory size. The empirical results match our theoretical analysis: our implementations communicate significantly less than Intel MKL, SLATE, and the asymptotically communication-optimal CANDMC and CAPITAL libraries. Our code outperforms these state-of-the-art libraries in almost all tested scenarios, with matrix sizes ranging from 2,048 to 524,288 on up to 512 CPU nodes of the Piz Daint supercomputer, decreasing the time-to-solution by up to three times. Our code is ScaLAPACK-compatible and available as an open-source library.

\section{1 INTRODUCTION}

Matrix factorizations, such as LU and Cholesky decompositions, play a crucial role in many scientific computations \citep{24, 32, 55}, and their performance can dominate the overall runtime of entire applications \citep{19}. Therefore, accelerating these routines is of great significance for numerous domains \citep{8, 43}. The ubiquity and importance of LU factorization is even reflected by the fact that it is considered compute-bound. However, the end of Dennard scaling \citep{22} puts increasing pressure on data movement minimization, as the cost of moving data far exceeds its computation cost, both in terms of time and access cost \citep{60, 62}. Thus, deriving algorithmic I/O lower bounds is a subject of both theoretical analysis \citep{15, 56, 57} and practical value for developing I/O-efficient schedules \citep{33, 59, 60}.

While asymptotically optimal matrix factorizations were proposed, among others, by Ballard et al. \citep{7} and Solomonik et al. \citep{33, 61}, we observe two major challenges with the existing approaches: First, the presented algorithms are only asymptotically optimal: the I/O cost of these proposed parallel algorithms can be as high as 7 times the lower bound for LU \citep{61} and up to 16 times for Cholesky \citep{33}. This means that they communicate less than “standard” 2D algorithms like ScaLAPACK \citep{14} only for almost prohibitively large numbers of processors — e.g., according to the LU cost model \citep{61}, it requires more than 15,000 processors to communicate less than an optimized 2D algorithm. Second, their time-to-solution performance can be worse than highly-optimized, existing 2D-parallel libraries \citep{33}.

To tackle these challenges, we first provide a general method for deriving precise I/O lower bounds of Disjoint Array Access Programs (DAAP) — a broad range of programs composed of a sequence of statements enclosed in an arbitrary number of nested loops. We then illustrate the applicability of our framework to derive parallel I/O lower bounds of Cholesky and LU factorizations: \( \frac{2 \sqrt{N^3}}{P \sqrt{M}} \) and \( \frac{2 \sqrt{N^3}}{P \sqrt{M}} \) elements, respectively, where \( N \) is the matrix size, \( P \) is the number of processors, and \( M \) is the local memory size.

Moreover, we use the insights from deriving the above lower bounds to develop \textsc{Co}n\textsc{L}UX and \textsc{Co}n\textsc{f}\textsc{Chox}, near communication-optimal parallel LU and Cholesky factorization algorithms that minimize data movement across the 2.5D processor decomposition. For LU factorization, to further reduce the latency and bandwidth cost, we use a row-masking tournament pivoting strategy resulting in a communication requirement of \( \frac{N^3}{P \sqrt{M}} + O(N^2 P/M) \) elements per processor, where the leading order term is only 1.5 times the lower bound. Furthermore, to secure high performance, we carefully tune block sizes and communication routines to maximize the efficiency of local computations such as \textsc{trsm} (triangular solve) and \textsc{gemm} (matrix multiplication).

We measure both communication volume and achieved performance of \textsc{Co}n\textsc{L}UX and \textsc{Co}n\textsc{f}\textsc{Chox} and compare them to state-of-the-art libraries: a vendor-optimized Intel MKL \citep{34}, SLATE \citep{28} (a...
Figure 2: From the input program through the I/O lower bounds to communication-minimizing parallel schedules and high performing implementations. In this paper, we mainly focus on the Cholesky and LU factorizations. The proofs of the lemmas presented in this work can be found in the AD/AE appendix.

recent library targeting exascale systems), as well as CANDMC [57, 58] and CAPITAL [32, 33] (codes based on the asymptotically optimal 2.5D decomposition). In our experiments on the Piz Daint supercomputer, we measure up to 1.6x communication reduction compared to the second-best implementation. Furthermore, our 2.5D decomposition communicates asymptotically less than SLATE and MKL, with even greater expected benefits on exascale machines. Compared to the communication-avoiding CANDMC library with I/O cost of \( 5N^2/(P\sqrt{M}) \) elements [61], CON/LUX communicates five times less. Most importantly, our implementations outperform all compared libraries in almost all scenarios, both for strong and weak scaling, reducing the time-to-solution by up to three times compared to the second best performing library (Figure 1).

In this work, we make the following contributions:

- A general method for deriving parallel I/O lower bounds of a broad range of linear algebra kernels.
- CON/LUX and CON/CHOX, provably near-I/O-optimal parallel algorithms for LU and Cholesky factorizations, with their full communication volume analysis.
- Open-source and fully ScaLAPACK-compatible implementations of our algorithms that outperform existing state-of-the-art libraries in almost all scenarios.

A bird’s eye view of our work is presented in Figure 2.

2 BACKGROUND

We now establish the background for our theoretical model (Sections 3-5). We use it to derive parallel I/O lower bounds for Cholesky and LU factorizations (Section 6) that will guide the design of our communication-minimizing implementations (Section 7).

2.1 Machine Model

To model algorithmic I/O complexity, we start with a model of a sequential machine equipped with a two-level deep memory hierarchy. We then outline the parallel machine model.

Sequential machine. A computation is performed on a sequential machine with a fast memory of limited size and unlimited slow memory. The fast memory can hold up to \( M \) elements at any given time. To perform any computation, all input elements must reside in fast memory, and the result is stored in fast memory.

Parallel machine. The sequential model is extended to a machine with \( P \) processors, each equipped with a private fast memory of size \( M \). There is no global memory of unlimited size — instead, elements are transferred between processors’ fast memories.

2.2 Input Programs

We consider a general class of programs that operate on multidimensional arrays. Array elements can be loaded from slow to fast memory, stored from fast to slow memory, and computed inside fast memory. These elements have versions that are incremented every time they are updated. We model the program execution as a computational directed acyclic graph (cDAG, details in Section 2.3), where each vertex corresponds to a different version of an array element. Thus, for a statement \( A[i,j] \leftarrow f(A[i,j]) \), a vertex corresponding to \( A[i,j] \) after applying \( f \) is different from a vertex corresponding to \( A[i,j] \) before applying \( f \). In a cDAG, this is expressed as an edge from vertex \( A[i,j] \) before \( f \) to vertex \( A[i,j] \) after \( f \). Initial versions of each element do not have any incoming edges and thus form the cDAG inputs. The distinction between elements and vertices is important for our I/O lower bounds analysis, as we will investigate how many vertices are computed for a given number of loaded vertices.

An input program is a collection of statements \( S \) enclosed in loop nests, each of the following form (we use the loop nest notation introduced by Dinh and Demmel [23]):

\[
\psi_i^1 \in D^1, \psi^2 \in D^2(\psi^1), \ldots, \psi^l \in D^l(\psi^1, \ldots, \psi^{l-1}) : S : A_0[\phi_0(\psi)] \leftarrow f(A_1[\phi_1(\psi)], A_2[\phi_2(\psi)], \ldots, A_m[\phi_m(\psi)]).
\]
which we provide key components of our program representation

\[ \text{I/O Complexity and Pebble Games} \]

directed acyclic graph

and optimal schedules of cDAG evaluation.

pebble game - a powerful abstraction for deriving lower bounds

\[ k \]

\[ \psi \]

Access function vector

\[ \psi \]

loop nest depth is

\[ l \]

different iteration variables present in

Access dimension

given vertex can be referenced by only one access function vector

the newest element versions can be referenced. Furthermore, a

unique

\[ \psi \]

\[ \rho \]

\[ f \]

of every input is an element of array

\[ A \]

\[ S \]

of the cDAG

Access

function

vectors

Data reuse between

statements

\[ \text{STATEMENT } S1 \]

\[ \text{STATEMENT } S2 \]

\[ \text{Iteration variables} \]

\[ \phi \]

\[ j \]

\[ i \]

\[ k \]

\[ \psi \]

\[ \rho \]

\[ f \]

\[ A \]

\[ S \]

\[ k, i \]

\[ j, k \]

\[ \text{Output vertex of the } cDAG \]

\[ \text{Input vertex of the } cDAG \]

\[ \text{Intermediate computation (element)} \]

\[ \text{Multiple vertices - bottom element A0} \]

\[ \text{Dom set} \]

\[ \text{X-partition (S2)} \]

\[ \text{Subcomputation} \]

\[ \text{Input matrices A} \]

\[ V \]

\[ E \]

\[ M \]

\[ X \]

\[ \rho(X) \]

\[ V_H \]

\[ \{H_1, \ldots, H_\ell\} \]

\[ | \]

\[ M \]

\[ \|P(X)\| \leq X \] and \[ |Min(H)| \leq X \]

\[ |P_{opt}(X)| \leq \frac{Q \cdot X - M}{X - M} \] for any \[ X > M \] ([45], Lemma 2).

\[ \text{2.3.3 X-Partitioning} \]

Intuition.

One can think of H’s dominator set as a set of inputs

required to execute subcomputation H, and of H’s minimum set as the

output of H. We use the notions of Dom_{min}(H) and Min(H) when

proving I/O lower bounds. Intuitively, we bound computation “volume”

(number of vertices in H) by its communication “surface”, comprised of

its inputs - vertices in Dom_{min}(H) - and outputs - vertices in

Min(H).

\[ \text{2.3.2 Dominator and Minimum Sets} \]

For any subset of vertices \[ H \subset V \], a dominator set \[ Dom(H) \] is a set such that every path in the

cDAG from an input vertex to any vertex in H must contain at least

one vertex in Dom(H). In general, for a given H, its Dom(H) is not

uniquely defined. The minimum set \[ Min(H) \] is the set of all vertices

in H that do not have any immediate successors in H. In this work,

avoiding the ambiguity of non-uniqueness of dominator set size

(in principle, for any subset, its valid dominator set is always the

whole V), we will refer to Dom_{min}(H) as a minimum dominator

set, i.e. a dominator set with the smallest size.

\[ \text{2.3.4 Deriving lower bounds.} \]

To bound the I/O cost, we further

need to introduce the computational intensity \( \rho \). For each subcomputation \( H_i \), \( \rho_i \) is defined as a ratio of the number of computations (vertices) in \( H_i \) to the number of I/O operations required to pebble \( H_i \), where the latter is bounded by the size of the dominator set \( Dom(H_i) \) [45]. Then, the following lemma bounds the number of I/O operations required to pebble a given cDAG:

**Lemma 1.** (Lemma 4 in [45]) For any constant \( X_c \), the number of I/O operations \( Q \) required to pebble a cDAG \( G = (V, E) \) with \( |V| = n \) vertices using \( M \) red pebbles is bounded by \( Q \geq n / \rho \), where \( \rho = \frac{H_{\rho_{max}}}{X_{c} - M} \) is the maximal computational intensity and \( H_{\rho_{max}} = \arg \max_{H \in P(X_c)} |H| \) is the largest subcomputation among all valid \( X_c \)-partitions.
3 GENERAL SEQUENTIAL I/O LOWER BOUNDS

We now present our method for deriving the I/O lower bounds of a sequential execution of programs in the form defined in Section 2.2. Specifically, in Section 3.2 we derive I/O bounds for programs that contain only a single statement. In Section 4 we extend our analysis to capture interactions and reuse between multiple statements.

In this paper, we present only the key lemmas required to establish the lower bounds of parallel Cholesky and LU factorizations. However, the method covers a much wider spectrum of algorithms. For curious readers, we present all proofs of provided lemmas in the appendix.

We start by stating our key lemma:

**Lemma 2.** If \( |H_{\text{max}}| \) can be expressed as a closed-form function of \( X \), that is if there exists some function \( \chi \) such that \( |H_{\text{max}}| = \chi(X) \), then the lower bound on \( Q \) can be expressed as

\[
Q \geq n(X_0 - M)\chi(X_0),
\]

where \( X_0 = \arg \min_X \rho = \arg \min_X \frac{\chi(X)}{X-M} \).

**Intuition.** \( \chi(X) \) expresses computation "volume", while \( X \) is its input "surface". The term \( X-M \) bounds the required communication and it comes from the fact that not all inputs have to be loaded (at most \( M \) of them can be reused). \( X_0 \) corresponds to the situations where the ratio of this "volume" to the required communication is minimized (corresponding to a highest lower bound).

**Proof.** Note that Lemma 1 is valid for any \( X \) (i.e., for any \( X \), it gives a valid lower bound). Yet, these bounds are not necessarily tight. As we want to find tight I/O lower bounds, we need to maximize the lower bound. \( X_0 \) by definition minimizes \( \rho \); thus, it maximizes the bound. Lemma 2 then follows directly from Lemma 1 by substituting \( \rho = \frac{\chi(X_0)}{X_0-M} \).

Note. If function \( \chi(X) \) is differentiable and has a global minimum, we can find \( X_0 \) by, e.g., solving the equation \( \frac{d\chi(X)}{dX} = 0 \). The key limitation is that it is not always possible to find \( X \), that is, to express \( |H_{\text{max}}| \) solely as a function of \( X \). However, for many linear algebra kernels \( \chi(X) \) exists. Furthermore, one can relax this problem preserving the correctness of the lower bound, that is, by finding a function \( \tilde{\chi} : \forall_X \tilde{\chi}(X) \geq \chi(X) \).

To find \( \chi(X) \), we take advantage of the DAAP structure. Observe that every computation (and therefore, every compute vertex \( v \in V \) in the cDAG \( G = (V, E) \)) is executed in a different iteration of the loop nest, and thus, there is a one-to-one mapping from a value of the iteration vector \( \phi \) to the compute vertex \( v \). Moreover, each vertex accessed from any of the input arrays \( A_j \) is also associated with some iteration vector value - however, if \( \dim(A_j) < l \), this is a one-to-many relation, as the same input vertex may be used to evaluate multiple compute vertices \( v \). This is, in fact, the source of the data reuse, and exploiting this relation is a key to minimizing the I/O cost. If for all input arrays \( A_j \) we have that \( \dim(A_j) = l \), then for each compute vertex \( v \), \( m \) different, unique input vertices are required, there is no data reuse and it implies a trivial computational intensity \( \rho = \frac{1}{m} \).

The high-level idea of our method is to count how many different iteration vector values \( \phi \) can be formed if we know how many different values each iteration variable \( \phi^1, \ldots, \phi^l \) takes. We now formalize this in Lemmas 3-8.

3.1 Iteration vector, iteration domain, access set

Each execution of statement \( S \) is associated with the iteration vector value \( \psi = [\psi^1, \ldots, \psi^l] \in \mathbb{N}^l \) representing the current iteration, that is, the values of iteration variables \( \psi^1, \ldots, \psi^l \). Each subcomputation \( H \) is uniquely defined by all iteration vectors' values associated with vertices pebbled in \( H = \{\psi_1, \ldots, \psi_l \} \). For each iteration variable \( \psi^t \), \( t = 1, \ldots, l \), denote the set of all values that \( \psi^t \) takes during \( H \) as \( D^t \). We define \( D = [D^1, \ldots, D^l] \subseteq \mathcal{D} \) as the iteration domain of subcomputation \( H \).

Furthermore, recall that each input access \( A_j(\phi(\psi)) \) is uniquely defined by \( \dim(\phi_j) \) iteration variables \( \psi^1, \ldots, \psi^{\dim(\phi_j)} \). Denote the set of all values each of \( \psi^t \) takes during \( H \) as \( D^t \). Given \( D^t \), we also denote the number of different vertices that are accessed from each input array \( A_j \) as \( |A_j(D^t)| \).

We now state the lemma which bounds \( |H| \) by the iteration sets' sizes \( |D^t| \) (the intuition behind the lemma is depicted in Figure 4):

**Lemma 3.** Given the ranges of all iteration variables \( D^t, t = 1, \ldots, l \) during subcomputation \( H \), if \( |H| = \prod_{t=1}^{l} |D^t| \), then \( \forall j = 1, \ldots, m \) : \( |A_j(D)| = \prod_{t=1}^{\dim(\phi_j)} |D^t| \) and \( |H| \) is maximized among all valid subcomputations that iterate over \( D = [D^1, \ldots, D^l] \).

**Intuition.** Lemma 3 states that if each iteration variable \( \psi^t \), \( t = 1, \ldots, l \), takes \( R^t_k \) different values, then there are at most \( \prod_{t=1}^{l} |D^t| \) different iteration vectors \( \phi \) which can be formed in \( H \). So, intuitively, we can maximize \( |H| \), all combinations of values \( \psi^t \) should be evaluated. On the other hand, this also implies maximization of all access sizes \( |A_j(D)| = \prod_{t=1}^{\dim(\phi_j)} |D^t| \).

To prove it, we now introduce two auxiliary lemmas:

**Lemma 4.** For statement \( S \), the size \( |H| \) of subcomputation \( H \) (number of vertices of \( S \) computed during \( H \)) is bounded by the sizes of the iteration variables' sets \( R^t_k, t = 1, \ldots, l \):

\[
|H| \leq \prod_{t=1}^{l} |D^t|.
\]

**Proof.** Inequality 1 follows from a combinatorial argument: each computation in \( H \) is uniquely defined by its iteration vector \( [\psi^1, \ldots, \psi^l] \). As each iteration variable \( \psi^t \) takes \( R^t_k \) different values during \( H \), we have \( |R^1_k| \cdot |R^2_k| \cdots |R^l_k| = \prod_{t=1}^{l} |D^t| \) ways how to uniquely choose the iteration vector in \( H \). □
Now, given \( D \), we want to assess how many different vertices are accessed for each input array \( A_j \). Recall that this number is denoted as access size \( |A_j(D)| \).

We will apply the same combinatorial reasoning to \( A_j(D) \). For each access \( A_j(\phi_j(\psi)) \), each one of \( \psi_j \), \( k = 1, \ldots, \text{dim}(\phi_j) \) iteration variables loops over set \( R_{h_j}^k \), during subcomputation \( H \). We can thus bound size of \( A_j(D) \) similarly to Lemma 4:

**Lemma 5.** The access size \( |A_j(D)| \) of subcomputation \( H \) (the number of vertices from the array \( A_j \) required to compute \( H \)) is bounded by the sizes of \( \text{dim}(\phi_j) \) iteration variables’ sets \( R_{h_j}^k \), \( k = 1, \ldots, \text{dim}(\phi_j) \):

\[
\forall j=1,\ldots,m : |A_j(D)| \leq \prod_{k=1}^{\text{dim}(\phi_j)} |D_j^k| \tag{2}
\]

where \( D_j^k \ni \psi_j \) is the set over which iteration variable \( \psi_j \) iterates during \( H \).

**Proof.** We use the same combinatorial argument as in Lemma 4. Each vertex in \( A_j(D) \) is uniquely defined by \( \{\psi_j^1, \ldots, \psi_j^{\text{dim}(\phi_j)}\} \).

Knowing the number of different values each \( \psi_j \) takes, we bound the number of different access vectors \( \phi_j(\psi_h) \). \( \square \)

**Example:** Consider once more statement 51 from LU factorization in Figure 3. We have \( \phi_0 = [i, k], \phi_1 = [i, k], \) and \( \phi_2 = [k, k] \). Denote the iteration subdomain for subcomputation \( H \) as \( D = \{ [k, k], \ldots, [k, k] \} \), \( H \}, \) where each variable \( k \) and \( i \) iterates over its set \( k \in \{k_1, \ldots, k_m\} = D^k \) and \( i \in \{i_1, \ldots, i_m\} = D^i \), for \( g = 1, \ldots, |H| \). Denote the sizes of these sets as \( |D^k| = K \) and \( |D^i| = I \). Hence, \( \psi_j \) takes \( K \) different values and \( i \) takes \( I \) different values. For \( \phi_1 \), both iteration variables used are different: \( k \) and \( i \). Therefore, we have (Equation 2) \( |A_1(D)| \leq K_i \cdot I_i \). On the other hand, for \( \phi_2 \), the iteration variable \( k \) is used twice. Recall that the access dimension is the minimum number of different iteration variables that uniquely address it (Section 2.2), so its dimension is \( \text{dim}(A_2) = 1 \) and the only iteration variable needed to uniquely determine \( \phi_2 \) is \( k \). Therefore, \( |A_2(D)| \leq K_k \).

**Dominator set.** Input vertices \( A_1, \ldots, A_m \) form a dominator set of vertices \( A_0 \), because any path from graph inputs to any vertex in \( A_0 \) must include at least one vertex from \( A_1, \ldots, A_m \). This is also the minimum dominator set, because of the disjoint access property (Section 2.2): any path from graph inputs to any vertex in \( A_0 \) can include at most one vertex from \( A_1, \ldots, A_m \).

**Proof of Lemma 3.** For subcomputation \( H \), we have \( \bigcup_{j=1}^m A_j(D) \leq X \) (by the definition of an X-partition). Again, by the disjoint access property, we have \( \forall j_1 \neq j_2 : A_{j_1}(D) \cap A_{j_2}(D) = \emptyset \). Therefore, we also have \( \bigcup_{j=1}^m A_j(D) = \sum_{j=1}^m |A_j(D)| \). We now want to maximize \( |H| \), that is to find \( H_{\text{max}} \) to obtain computational intensity \( \rho \) (Lemma 2).

Now we prove that to maximize \( |H| \), inequalities 1 and 2 must be tight (become equalities).

From proof of Lemma 4 it follows that \( |H| \) is maximized when iteration vector \( \psi \) takes all possible combinations of iteration variables \( \psi' \in D' \) during \( H \). But, as we visit each combination of all \( I \) iteration variables, for each access \( A_j \) every combination of its \( \{\psi_j^1, \ldots, \psi_j^{\text{dim}(\phi_j)}\} \) iteration variables is also visited. Therefore, for every \( j = 1, \ldots, m \), each access size \( |A_j(D)| \) is maximized (Lemma 5), as access functions are injective, which implies that for each combination of \( \{\psi_j^1, \ldots, \psi_j^{\text{dim}(\phi_j)}\} \), there is one access \( A_j \).

\( \prod_{j=1}^m |D_j^k| \) is then the upper bound on \( |H| \), and its tightness implies that all bounds on access sizes \( |A_j(D)| \leq \prod_{k=1}^{\text{dim}(\phi_j)} |D_j^k| \) are also tight. \( \square \)

**Intuition.** Lemma 3 states that if \( \forall t \geq 1 \), \( \exists \psi_t \) different values, then there are at most \( \prod_{t=1}^m |D_t| \) different iteration vector values \( \psi \) that can be formed in \( H \). Thus, to maximize \( |H| \) all combinations of values of \( \psi_t \) should be evaluated. On the other hand, this also implies the maximization of all access sizes \( |A_j(D)| \leq \prod_{k=1}^{\text{dim}(\phi_j)} |D_j^k| \). This result is more general than, e.g., polyhedral techniques [11, 15, 51] as it does not require loop nests to be affine. Instead, it solely relies on set algebra and combinatorial methods.

### 3.2 Finding the I/O Lower Bound

Denoting \( H_{\text{max}} = \arg \max_{H \in \mathcal{X}(X)} |H| \) as the largest subcomputation among all valid X-partitions, we use Lemma 3 and combine it with the dominator set constraint from Section 2.3.3. Note that all access set sizes are strictly positive integers \( |D_t| \in \mathbb{N}_+ \). In other words, if any of the sets is empty, no computation can be performed. However, as we only want to find the bound on the number of I/O operations, we relax the integer constraints and replace them with \( |D_t| \geq 1 \). Then, we formulate finding \( \chi(X) \) (Lemma 2) as the following optimization problem:

\[
\max \prod_{t=1}^m |D_t| \quad \text{s.t.} \quad \sum_{k=1}^{\text{dim}(\phi_j)} |D_j^k| \leq X \nonumber
\]

\[
\forall t \geq 1 \quad |D_t| \leq 1 \quad |D_t| \geq 1 \quad (3)
\]

We then find \( H_{\text{max}} = \chi(X) \) as a function of \( X \) using Karush–Kuhn–Tucker (KKT) conditions [43]. Next, we solve

\[
\frac{d\chi(X)}{dX} = 0. \tag{4}
\]

Denoting \( X_0 \) as the solution to Equation (4), we finally obtain

\[
\frac{Q}{\chi(X_0)} \geq V (X_0 - M). \tag{5}
\]

**Computational intensity and out-degree-one vertices.** There exist cDAGs where every non-input vertex has at least \( u \geq 0 \) direct predecessors that are input vertices with out-degree 1. We can use this fact to add an additional bound on the computational intensity.

**Lemma 6.** If in a cDAG \( G = (V, E) \) every non-input vertex has at least \( u \) direct predecessors with out-degree one that are graph inputs, then the maximum computational intensity \( \rho \) of this cDAG is bounded by \( \rho \leq \frac{u}{2} \).

**Proof.** By the definition of the red-blue pebble game, all inputs start in slow memory, and therefore, have to be loaded. By the assumption on the cDAG, to compute any non-input vertex \( v \in V \), at least \( u \) input vertices need to have red pebbles placed on them using a load operation. Because these vertices do not have any other direct
successors (their out-degree is 1), they cannot be used to compute any other non-input vertex \( w \). Therefore, each computation of a non-input vertex requires at least \( u \) unique input vertices to be loaded.

Example: Consider Figure 5. In a), each compute vertex \( C[i, j] \) has two input vertices: \( A[i, j] \) with out-degree 1, and \( b[j] \) with out-degree \( n \), thus \( u = 1 \). As both array \( A \) and vector \( b \) start in the slow memory (having blue pebbles on each vertex), for each computed vertex from \( C \), at least one vertex from \( A \) has to be loaded, therefore \( \rho \leq 1 \). In b), each computation needs two out-degree 1 vertices, one from vector \( a \) and one from vector \( b \), resulting in \( u = 2 \). Thus, \( \rho \leq \frac{1}{2} \).

4 DATA REUSE ACROSS MULTIPLE STATEMENTS

Until now, we have analyzed each statement separately. However, almost all computational kernels contain multiple statements connected by data dependencies — e.g., a column update (S1) and a trailing matrix update (S2) in LU factorization (Figure 3). The challenge here is that, in general, I/O cost \( Q \) is not composable: due to the data reuse, the total I/O cost of the program may be smaller than the sum of I/O costs of its constituent kernels. In this section we examine how these dependencies influence the total I/O cost of a program.

We derive I/O lower bounds for programs with \( w \) statements \( S_1, \ldots, S_w \) in two steps. First, we analyze each statement \( S_i \) separately, as in Section 3. Then, we derive how many loads could be avoided at most during one statement if another statement owned shared data. There are two cases where data reuse can occur: I) input overlap, where shared arrays are inputs for multiple statements, and II) output overlap, where the output array of one statement is the input array of another.

Case I). Assume there are \( w \) statements in the program, and there are \( k \) arrays \( A_j, j = 1, \ldots, k \) that are shared between at least two statements. We still evaluate each statement separately, but we will subtract the upper bound on shared loads \( Q_{tot} \geq \sum_{j=1}^{w} Q_j - \sum_{j=1}^{k} \text{Reuse}(A_j) \), where \( \text{Reuse}(A_j) \) is the reuse bound on array \( A_j \) (Section 4.1). Case II). Consider each pair of "producer-consumer" statements \( S \) and \( T \), that is, the output of \( S \) is the input of \( T \). The I/O lower bound \( Q_S \) of statement \( S \) does not change due to the reuse, as the same number of loads has to be performed to evaluate \( S \). On the other hand, it may invalidate \( Q_T \), as the dominator set of \( T \) formulated in Section 3.1 may not be minimal — inputs of a statement may not be graph inputs anymore. For each "consumer" statement \( T \), we reevaluate \( Q_T' \leq Q_T \) using Lemma 8. Finally, for a program consisting of \( w \) statements in total, connected by the output overlap, we have \( Q_{tot} \geq \sum_{j=1}^{w} Q_j' \). Note that for each "producer" statement \( i \), \( Q_i' = Q_i \), i.e. output overlap does not change their I/O lower bound.

4.1 Case I: Input Reuse and Reuse Size

Consider two statements \( S \) and \( T \) that share one input array \( A \). Let \( |A_i(R)] \) denote the total number of accesses to \( A \) during the I/O optimal execution of a program that contains only statement \( S \). Naturally, \( |A_i(R)] \) denotes the same for a program containing only \( T \). Define \( \text{Reuse}(A_i) := \min(|A_i(R)] - |A_i(R)]\). We then have:

Lemma 7. The I/O cost of a program containing statements \( S \) and \( T \) that share the input array \( A_i \) is bounded by

\[ Q_{tot} \geq Q_S + Q_T - \text{Reuse}(A_i), \]

where \( Q_S, Q_T \) are the I/O costs of a program containing only statement \( S \) or \( T \), respectively.

Proof. Consider an optimal sequential schedule of a cDAG \( G_S \) containing statement \( S \) only. For any subcomputation \( H_S \) and its associated iteration domain \( R_S \), its minimum dominator set is \( \text{Dom}(H_S) = \bigcup_{j=1}^{m} A_j(R_S) \). To compute \( H_S \), at least \( \sum_{j=1}^{m} |A_j(R_S)| = M \) vertices have to be loaded, as only \( M \) vertices can be reused from previous subcomputations. We seek if any loads can be avoided in the common schedule if we add statement \( T \), denoting its cDAG \( G_{S+T} \). Consider a subset \( A_j(R_S) \) of vertices in \( A_j \).

Consider some subset of vertices in \( A_j \) which potentially could be reused and denote it \( \Theta_j \). Now denote all vertices in \( A_j \) (statement \( S \)) which depend on any vertex from \( \Theta_j \) as \( \Theta_S \), and, analogously, set \( \Theta_T \) for statement \( T \). Now consider these two subsets \( \Theta_S \) and \( \Theta_T \) separately. If \( \Theta_S \) is computed before \( \Theta_T \), then it had to load all vertices from \( \Theta_S \); avoiding no loads compared to the schedule of \( G_S \) only. Now, computation of \( \Theta_T \) may take benefit of some vertices from \( \Theta_S \), which can still reside in fast memory, avoiding up to \( |\Theta_S| \) loads. The total number of avoided loads is bounded by the number of loads from \( A_j \) which are shared by both \( S \) and \( T \). Because statement \( S \) loads at most \( |A_j(R_S)| \) vertices from \( A_j \) during optimal schedule of \( G_S \), and \( T \) loads at most \( |A_j(R_T)| \) of them for \( G_T \), the upper bound of shared, and possibly avoided loads is \( \text{Reuse}(A_i) = \min(|A_i(R_S)|, |A_i(R_T)|) \).

The reuse size is defined as \( \text{Reuse}(A_i) = \min(|A_i(R_S)|, |A_i(R_T)|) \).

Now, how to find \( |A_i(R_S)| \) and \( |A_i(R_T)| \)?

Observe that \( |A_i(R_S)| \) is a property of \( G_S \), that is, the cDAG containing statement \( S \) only. Denote the I/O optimal schedule parameters of \( G_S \): \( V_{max}^S, X_0^S \) and \( |A_i(R_{max}(X_0^S))| \) (Section 3.2). Similarly, for \( G_T \): \( V_{max}^T, X_0^T, \) and \( |A_i(R_{max}(X_0^T))| \). We now derive: 1) at
least how many subcomputations does the optimal schedule have: 

\[ s \geq \frac{|V|}{|R_{max}|}. \]  

2) at least how many accesses to \( A_j \) are performed per 

optimal subcomputation \( |A_j(R_{max}(X_0))| \).

Then:

\[
\text{Reuse}(A_j) = \min \{ |A_j(R^S_{max}(X^S_j))| \} \cdot \frac{|V^S|}{|V_{max}|} 
+ |A_j(R^T_{max}(X^T_j))| \cdot \frac{|V^T|}{|V_{max}|}.
\]  

Corollary 1. 

Consider the case where the \( A_0 \) of statement \( S \) is also the 

input \( B_j \) of statement \( T \). Furthermore, consider subcomputation \( H \) 

of statement \( T \) (and its associated iteration domain \( D \)). Any path 

from the graph inputs to vertices in \( B_0(D) \) must pass through 

vertices in \( B_j(D) \). The following question arises: Is there a smaller 

set of vertices \( B'_j(D) \), \( |B'_j(D)| < |B_j(D)| \) that every path from 

graph inputs to \( B_j(D) \) must pass through?

Let \( p_S \) denote computational intensity of statement \( S \). With that, 

we can state the following lemma:

Lemma 8. Any dominator set of set \( B_j(D) \) must be of size at least 

\[ |\text{Dom}(B_j(D))| \geq \frac{|B_j(D)|}{p_S}. \]

Proof. By Lemma 1, for one loaded vertex, we may compute 

at most \( p_S \) vertices of \( A_0 \). These are also vertices of \( B_j \). Thus, 

to compute \( |B_j(D)| \) vertices of \( B_j \), at least \( \frac{|B_j(D)|}{p_S} \) loads must be 

performed. We just need to show that at least many vertices 

have to be in any dominator set \( \text{Dom}(B_j(D)) \). Now, consider the 

converse: There is a vertex set \( D = \text{Dom}(B_j(D)) \) such that \( |D| < 

\frac{|B_j(D)|}{p_S} \). But that would mean, that we could potentially 

compute all \( |B_j(D)| \) vertices by only loading \( |D| \) vertices, violating Lemma 1.

Corollary 1. Combining Lemmas 8 and 3, the data access size of 

\( |B_j(D)| \) during subcomputation \( H \) is 

\[
|\text{Dom}(B_j(D))| \geq \prod_{k=1}^{\text{dim}(\phi)} \frac{|D|^k}{p_S}. \quad (7)
\]

Similarly to case I, this result also generalizes to multiple “con- 

sumer” statements that reuse the same output array of a “producer” 

statement, as well as any combination of input and output reuse 

for multiple arrays and statements. Since the actual I/O optimal 

schedule is unknown, the general strategy to ensure correctness 

of our lower bound is to consider each pair of interacting statements 

separately as one of these two cases. Since both Lemma 7 and 8 

overapproximate the reuse, the final bound may not be tight - the 

more inter-statement reuse, the more overapproximation is needed.

Still, this method can be successfully applied to derive tight I/O 

lower bounds for many linear algebra kernels, such as matrix fac-

torizations, tensor products, or solvers.

5 GENERAL PARALLEL I/O LOWER BOUNDS 

We now establish how our method applies to a parallel machine 

with \( P \) processors (Section 2.1). Since we target large-scale dis-

tributed systems, our parallel pebbling model differs from the one 

introduced e.g. by Alwen and Serbinenko [5], which is inspired 

by shared-memory models like PRAM [39]. Instead, we disallow 

sharing memory (pebbles) between the processors, and enforce 

explicit communication — analogous to the load/store operations 

— using red and blue pebbles. This allows us to better match the 

behavior of real, distributed applications that use, e.g., MPI.

Each processor \( p_i \) owns its private fast memory that can hold 

up to \( M \) words, represented in the cDAG as \( M \) vertices of color \( p_i \). 

Vertices with different colors (belonging to different processors) 

cannot be shared between these processors, but any number of 
different pebbles may be placed on one vertex.

All the standard red-blue pebble game rules apply with the fol-

lowering modifications:

1) \textbf{compute}. If all direct predecessors of vertex \( v \) have pebbles 

of \( p_i \)’s color placed on them, one can place a pebble of color 

\( p_i \) on \( v \) (no sharing of pebbles between processors).

2) \textbf{communication}. If a vertex \( v \) has \( a \) pebble placed on it, 
a pebble of any other color may be placed on this vertex.

From this game definition it follows that from a perspective of 
a single processor \( p_i \), any data is either local (the corresponding 
vertex has \( p_i \’s \) pebble placed on it) or remote, without a distinction 
on the remote location (remote access cost is uniform).

Lemma 9. The minimum number of I/O operations in a parallel 

type pebble communication, played on a cDAG with \( |V| \) 

vertices with \( P \) processors each equipped with \( M \) pebbles, is 

\[ Q \geq \frac{|V|}{P^\rho}, \] 

where \( \rho \) is the maximum 

computational intensity, which is independent of \( P \) (Lemma 1).

Proof. Following the analysis of Section 3 and the parallel ma-

chine model (Section 5), the computational intensity \( \rho \) is indepen-
dent of a number of parallel processors - it is solely a property 
of a cDAG and private fast memory size \( M \). Therefore, following 

Lemma 1, what changes with \( P \) is the volume of computation \( |V| \), 
as now at least one processor will compute at least \( |V| \) \( P \) 

vertices. By the definition of the computational intensity, the minimum 

number of I/O operations required to pebble these \( |V| \) vertices is 

\[ |V| P. \]

6 I/O LOWER BOUNDS OF PARALLEL 

FACTORIZATION ALGORITHMS

We gather all the insights from Sections 2 to 5 and use them to obtain 
the parallel I/O lower bounds of LU and Cholesky factorization 
algorithms, which we use to develop our communication-avoiding 

implementations.

Memory size. Clearly, \( M \geq N^2/P, \) as otherwise the input can-

not fit into the collective memory of all processors. Furthermore, 
in the following sections, we analyze the \textbf{memory-dependent} communi-

cation cost [15]. That is, following Solomonik et al. [61], we assume 

\( M \leq N^2/P^{2/3}. \) This is an upper bound on the amount of memory 
per processor that can be efficiently utilized under the assumptions 
that 1) initially, the input is not replicated (every element of input 

matrix \( A \) resides in exactly one location of one of the processors); 
2) every processor performs \( \Theta(N^3/P) \) elementary computations.
For larger \( M \), the communication cost transitions to the memory-independent regime [15]. All presented memory-dependent lower bounds and algorithmic costs can be easily translated to memory-independent version by plugging in the upper bound on the size of the usable memory.

### 6.1 LU Factorization

In our I/O lower bound analysis we omit the row pivoting, since swapping rows can increase the I/O cost by at most \( N^2 \), which is the cost of permuting the entire matrix. However, the total I/O cost of the LU factorization is \( O(N^3/\sqrt{M}) \) [61].

LU factorization (without pivoting) contains two statements (Figure 3). Observe that we can use Lemma 6 (out-degree one vertices) for statement \( S_1: A[i,j] = A[i,k] \rightarrow A[k,j] \). The loop nest depth is \( h_{S_1} = 2 \), with iteration variables \( \psi_1 = k \) and \( \psi_2 = 1 \). The dimension of the access function vector \( (k, j) \) is 1, revealing potential for data reuse: every input vertex \( A[k, j] \) is accessed \( N - k \) times and used to compute vertices \( A[i, k], k + 1 \leq i < N \). However, access function vector \( (i, k) \) has dimension 2; therefore, every compute vertex has one direct predecessor with out-degree one, which is the previous version of element \( A[i, k] \). Using Lemma 6, we therefore have \( \rho_{S_1} \leq 1 \).

We now proceed to statement \( S_2: A[i, j] = A[i, k] \rightarrow A[k, j] \). Let \( |D_1| = K, |D_2| = 1, |D_3| = J \). Observe that there is an output reuse (Section 4.2 and Figure 3, red arrow) of \( A[i, k] \) between statements \( S_1 \) and \( S_2 \). We therefore have the following access size in statement \( S_2: |A_2(D_{S2})| = \frac{IK}{\rho_{S_2}} \geq 1K \) (Equation 7). Note that in this case where the computational intensity is \( \rho_{S_2} \leq 1 \), the output reuse does not change the access size \( |A_2(D_{S2})| \) of statement \( S_2 \). This follows the intuition that it is not beneficial to recomputes vertices if the recomputation cost is higher than loading it from the memory. Denoting \( H_{S2} \) as the maximal subcomputation for statement \( S_2 \) over the subcomputation domain \( D \), we have the following (Lemma 3):

- \( |H_{S2}| = KIJ \)
- \( |A_1(D)| = |A[i, j]| = IJ \)
- \( |A_2(D)| = |A[i, k]| = IK \)
- \( |A_3(D)| = |A[k, j]| = KJ \)
- \( |Dom(H_{S2})| = |A_1(D)| + |A_2(D)| + |A_3(D)| = IJ + IK + KJ \)

We then solve the optimization problem from Section 3.2:

\[
\max \ KIJ,
\quad \text{s.t.} \quad IJ + IK + KJ \leq X
\]

\( I \geq 1, \quad J \geq 1, \quad K \geq 1 \)

Which gives \( H_{S2} = \chi(X) = \left( \frac{X}{2} \right)^2 \) for \( K = I = J = \left( \frac{X}{3} \right)^2 \).

Then, we find \( X_0 \) that minimizes the expression \( \rho_{S2}(X) = \frac{|H_{S2}|}{\chi(X)} \) (Equation 4), yielding \( X_0 = 3M \). Plugging it into \( \rho_{S2}(X) \), we conclude that the maximum computational intensity of \( S_2 \) is bounded by \( \rho_{S2} \leq 3\sqrt{M/2} \).

We bounded the maximum computational intensities \( \rho_{S_1} \) and \( \rho_{S_2} \), that is, the minimum number of I/O operations to compute vertices belonging to statements \( S_1 \) and \( S_2 \). As the last step, we find the total number of compute vertices for each statement: \( |V_1| = \sum_{k=1}^{N} (N-k-1) = N(N-1)/2 \), and \( |V_2| = \sum_{k=1}^{N} \sum_{i=k+1}^{N} (N-k-1) = N(N-1)(N-2)/3 \). Using Lemmas 1 (bounding I/O cost with the computational intensity) and 9 (I/O cost of the parallel machine), the parallel I/O lower bound for LU factorization is therefore

\[
Q_{PLU} \geq \frac{2N^3 - 6N^2 + 4N}{3P^2M} + \frac{N(N-1)}{2P} = \frac{2N^3}{3P^2M} + \frac{N^2}{2P} + \frac{N}{P}.
\]

Previously, Solomonik et al. [61] established the asymptotic I/O bound for sequential execution \( Q = O(N^3/\sqrt{M}) \). Recently, Olivry et al. [51] derived a tight leading term constant \( Q \geq 2N^3/(3\sqrt{M}) \). To the best of our knowledge, our result is the first non-asymptotic bound for parallel execution. The generalization from the sequential to the parallel bound is straightforward. Note, however, that this is only the case due to our pebble-based execution model, and it may thus not apply to other parallel machine models.

### 6.2 Cholesky Factorization

We proceed analogously to our derivation of the LU I/O bound — here we just briefly outline the steps. The algorithm contains three statements (Listing 1). For statements \( S_1 \) and \( S_2 \), we can again use Lemma 6 (out-degree one vertices). For \( S_1: L(k,k) = \sqrt{L(k,k)} \), the loop nest depth is \( h_{S1} = 1 \), we have a single iteration variable \( \psi_1 = k \), and a single input array \( A_1 = L \) with the access function \( \phi_1(\psi_1) = (k,k) \). Since there is only one iteration variable present in \( \phi_1 \), we have \( \text{dim}(\phi_1) = 1 \). Therefore, for every compute vertex \( v \) we have one direct predecessor, which is the previous version of element \( L(k,k) \). We conclude that \( \rho_{S_1} \leq 1 \) and \( |V_{S1}| = N \).

```
1 for k = 1:N
2 S1: L(k,k) = sqrt(L(k,k));
3 for i = 1:k+1:N
4 S2: L(i,k) = (L(i,k)) / L(k,k);
5 for j = k+1:N
6 S3: L(i,j) = L(i,k) * L(j,k);
7 end; end; end;
```

Listing 1: Cholesky Factorization

For statement \( S_2: L(i,k) = (L(i,k)) / L(k,k) \), we also have output reuse of \( L(k,k) \) between statements \( S_2 \) and \( S_1 \). However, as with the output reuse considered in the LU analysis, the computational intensity is \( \rho_{S_2} \leq 1 \). Therefore, it does not change the dominator set size of \( S_2 \). We then use the same reasoning as for the corresponding statement \( S_1 \) in LU factorization, yielding \( \rho_{S_2} \leq 1 \).

For statement \( S_3 \), we derive its bound similarly to \( S_2 \) of LU, with \( \rho_{S_3} = \sqrt{M}/2 \) and \( |V_{S3}| = \sum_{k=1}^{N} \sum_{i=k+1}^{N} (i-k-1) = N(N-1)/2 \). Note that compared to LU, the only significant difference is the iteration domain \( |V_{S3}| \). Even though Cholesky has one statement more — the diagonal element update \( L(k,k) \) — its impact on the final I/O bound is negligible for large \( N \).

Again, using Lemmas 1 and 9 we establish the Cholesky factorization’s parallel I/O lower bound:

\[
Q_{Chol} \geq Q_1 + Q_2 + Q_3 = \frac{|V_1|}{P\rho_1} + \frac{|V_2|}{P\rho_2} + \frac{|V_3|}{P\rho_3} \approx \frac{N^3}{3P^2M} + \frac{N^2}{2P} + \frac{N}{P}.
\]

The derived I/O lower bound for a sequential machine (\( P = 1 \)) improves the previous bound \( Q_{Chol} \geq N^3/(6\sqrt{M}) \) derived by Olivry
et al. [51]. Furthermore, to the best of our knowledge, this is the first parallel bound for this kernel.

7 NEAR-I/O OPTIMAL PARALLEL MATRIX FACTORIZATION ALGORITHMS

We now present our parallel LU and Cholesky factorization algorithms. We start with the former, more complex algorithm, i.e., LU factorization. Pivoting in LU poses several performance challenges. First, since pivots are not known upfront, additional communication and synchronization is required to determine them in each step. Second, the nondeterministic pivot distribution between the ranks may introduce load imbalance of computation routines. Third, to minimize the communication a 2.5D parallel decomposition must be used, i.e., parallelization along the reduction dimension. We address all these challenges with CONfLUX — a near Communication Optimal LU factorization using X-Partitioning.

7.1 LU Dependencies and Parallelization

Due to the dependency structure of LU, the input matrix is often divided recursively into four submatrices \(A_{00}, A_{10}, A_{01},\) and \(A_{11}\) [24, 61]. Arithmetic operations performed in LU create non-commutative dependencies (Figure 6) between vertices in \(A_{00}\) (LU factorization of the top-left corner of the matrix), \(A_{10},\) and \(A_{01}\) (triangular solve of left and top panels of the matrix). Only \(A_{11}\) (Schur complement update) has no such dependencies, and may therefore be efficiently parallelized in the reduction dimension. A high-level summary is presented in Algorithm 1.

7.2 LU Computation Routines

The computation is performed in \(\frac{N}{s}\) steps, where \(s\) is a tunable block size. In each step, only submatrix \(A_r\) of input matrix \(A\) is updated. Initially, \(A_r\) is set to \(A.\) \(A_r\) can be further viewed as composed of four submatrices \(A_{00}, A_{10}, A_{01},\) and \(A_{11}\) (see Figure 7). These submatrices are distributed and updated by routines \(\text{TournPivot, Factorize}_{A_{10}}, \text{Factorize}_{A_{01}},\) and \(\text{Factorize}_{A_{11}}:\)

- \(A_{00}\): This \(\nu \times \nu\) submatrix contains the first \(\nu\) elements of the current \(\nu\) pivot rows. It is computed during \(\text{TournPivot}\) and, as it is required to compute \(A_{10}\) and \(A_{01}\), it is redundantly copied to all processors.
- \(A_{10}\) and \(A_{01}\): Submatrices \(A_{10}\) and \(A_{01}\) of sizes \(N - t \cdot \nu \times \nu\) and \(\nu \times (N - t \cdot \nu)\) are distributed using a 1D decomposition among all processors. They are updated using a triangular solve. 1D decomposition guarantees that there are no dependencies between processors, so no communication or synchronization is performed during computation, as \(A_{00}\) is already owned by every processor.
- \(A_{11}\): This \((N - t \cdot \nu) \times (N - t \cdot \nu)\) submatrix is distributed using a 2.5D block-cyclic distribution (Figure 7). First, the updated submatrices \(A_{10}\) and \(A_{01}\) are broadcast among the processors. Then, \(A_{11}\) (Schur complement) is updated. Finally, the first block column and \(\nu\) chosen pivot rows are reduced, which will form \(A_{10}\) and \(A_{01}\) in the next iteration.

Block size \(\nu\): The minimum size of each block is the number of processor layers in the reduction dimension \(\nu \geq c = \frac{PM}{N^2}\). However, to ensure high performance, this value should be adjusted to hardware parameters of a given machine (e.g., vector length, prefetch distance of a CPU, or warp size of a GPU). Throughout the analysis, we assume \(\nu = a \cdot \frac{PM}{N^2}\) for some small constant \(a\).
7.3 Pivoting

Our pivoting strategy differs from state-of-the-art block [6], tile [4], or recursive [24] pivoting approaches in two aspects:

- To minimize I/O cost, we do not swap pivot rows. Instead, we keep track of which rows were chosen as pivots and we use masks to update the remaining rows.
- To reduce latency, we take advantage of our derived block decomposition and use tournament pivoting [29].

Tournament Pivoting. This procedure finds $v$ pivot rows in each step that are then used to mask which rows will form the new $A_{01}$ and then filter the non-processed rows in the next step. It is shown to be as stable as partial pivoting [29], which might be an issue for, e.g., incremental pivoting [54]. On the other hand, it reduces the $O(N)$ latency cost of partial pivoting, which requires step-by-step column reduction to find consecutive pivots, to $O\left(\frac{N^2}{P}\right)$, where $v$ is the tunable block size parameter.

Row Swapping vs. Row Masking. To achieve close to optimal I/O cost, we use a 2.5D decomposition. This, however, implies that in the presence of extra memory, the data matrix is replicated $\frac{PM}{N^2}$ times. This increases the row swapping cost from $O\left(\frac{N^2}{P}\right)$ to $O\left(\frac{N^3}{PVM}\right)$, which asymptotically matches the I/O lower bound of the entire factorization. Performing row swapping would then increase the constant term of the leading factor of the algorithm from $\frac{N^3}{PVM}$ to $2\frac{N^3}{PVM}$. To keep the I/O cost of our algorithm as low as possible, instead of performing row-swapping, we only propagate pivot row indices. When the tournament pivoting finds the $v$ pivot rows, they are broadcast to all processors with only $v$ cost per step.

Pivoting in CO_{nf}/LUX. In each step $t$ of the outer loop (line 1 in Algorithm 1), $\frac{N}{\sqrt{M}}$ processors perform a tournament pivoting routine using a butterfly communication pattern [55]. Each processor owns $\sqrt{M}\frac{N^{st}}{N}$ rows, among which it chooses $v$ local candidate pivots. Then, final pivots are chosen in $\log(\frac{N}{\sqrt{M}})$ “playoff-like” tournament rounds, after which all $\frac{N}{\sqrt{M}}$ processors own both $v$ pivot row indices and the already factored, new $A_{00}$. This result is distributed to all remaining processors (line 2). Pivot row indices are then used to determine which processors participate in the reduction of the current $A_{01}$ (line 4). Then, the new $A_t$ is formed by masking currently chosen rows $A_t ← A_t[\text{rows, } v : \text{end}]$ (Line 12).

7.4 I/O cost of CO_{nf}/LUX

We now prove the I/O cost of CO_{nf}/LUX, which is only a factor of $\frac{1}{2}$ higher than the obtained lower bound for large $N$.

Lemma 10. The total I/O cost of CO_{nf}/LUX, presented in Algorithm 1, is $Q_{CO_{nf}/LUX} = \frac{N^3}{PVM} + O(M)$.

Proof. We assume that the input matrix $A$ is already distributed in the block cyclic layout imposed by the algorithm. Otherwise, data reshuffling imposes only $O\left(\frac{N^2}{P}\right)$ cost, which does not contribute to the leading order term. We first derive the cost of a single iteration $t$ of the main loop of the algorithm, proving that its cost is $Q_{step}(t) = \frac{2N\epsilon(N-t)\log_t}{PVM} + O\left(\frac{M\epsilon}{N}\right)$. The total cost after $\frac{N}{\tau}$ iterations is:

$$Q_{CO_{nf}/LUX} = \sum_{t=1}^{\frac{N}{\tau}} Q_{step}(t) = \frac{N^3}{PVM} + O(M).$$

We define $P1 = \frac{N^2}{}\sqrt{M}$ and $c = \frac{PM}{N\epsilon}$. $P$ processors are decomposed into the 3D grid $[\sqrt{P1}, \sqrt{P1}, c]$. We refer to all processors that share the same second and third coordinate as $[i, j, k]$. We now examine each of 11 steps in Algorithm 1.

Step 2. Processors with coordinates $[i, t \mod \sqrt{P1}, t \mod c]$ perform the tournament pivoting. Every processor owns the first $v$ elements of $N - (t-1)v$ rows, among which they choose the next $v$ pivots. First, they locally perform the LUF decomposition to choose the local $v$ candidate rows. Then, in $\log_2(\sqrt{P1})$ rounds they exchange $v \times v$ blocks to decide on the final pivots. After the exchange, these processors also hold the factorized submatrix $A_{00}$. I/O cost per processor: $v^2\log_2(\sqrt{P1})$.

Steps 3, 4, 6. Factorized $A_{00}$ and $v$ pivot row indices are broadcast. First $v$ columns and $v$ pivot rows are scattered to all $P$. I/O cost per processor: $v^2 + O\left(\frac{N-v}{P}\right)$.

Steps 1 and 5. $v$ columns and $v$ pivot rows are reduced. With high probability, pivots are evenly distributed among all processors. There are $c$ layers to reduce, each of size $(N - t)v$. I/O cost per processor: $\frac{2(N-t)\epsilon}{P} = \frac{2(N-t)\epsilon}{P}$. Steps 7, 9, 11. The updates Factorize$A_{10}$, Factorize$A_{01}$, and Factorize$A_{11}$ are local and incur no additional I/O cost.

Steps 8 and 10. Factorized $A_{10}$ and $A_{01}$ are scattered among all processors. Each processor requires $\frac{(N-t)}{v\sqrt{P1}}$ elements from $A_{10}$ and $A_{01}$. I/O cost per processor: $\frac{2(N-t)\epsilon}{PVM}$.

Summing steps 1 – 11: $Q_{step}(t) = \frac{2N\epsilon(N-t)\sqrt{P1}}{PVM} + O\left(\frac{M\epsilon}{N}\right).$ 

Note that this cost is a factor $1/3$ over the lower bound established in Section 6.1. This is due to the fact that any processor can only maximally utilize its local memory in the first iteration of the outer loop. In this first iteration, a processor updates a total of $\sqrt{P1} \times \sqrt{P1}$ elements of $A$. In subsequent iterations, however, the local domain shrinks as less rows and columns are updated, which leads to an underutilization of the resources. Since the shape of the iteration space is determined by the algorithm, this behavior is unavoidable for $P \geq N^2/M$. Note that the bound is attainable by a sequential machine, however.

7.5 Cholesky Factorization

From a data flow perspective, Cholesky factorization can be viewed as a special case of LU factorization without pivoting for symmetric, positive definite matrices. Therefore, our Cholesky algorithm — CO_{nf}/CHOX — heavily bases on CO_{nf}/LUX, using the same 2.5D parallel decomposition, block-cyclic data distribution, and analogous computation routines.

For both algorithms, the dominant cost, both in terms of computation and communication, is the $A_{11}$ update. Due to the Cholesky factorization’s iteration domain, which exploits the symmetry of the input matrix, the compute cost is twice as low, as only one half of the matrix needs to be updated. However, the input size required to perform this update is the same — therefore, the communication cost imposed by $A_{11}$ is similar. We list the key differences between the two factorization algorithms in Table 1.
We compare CO with MPI one-sided [31]. For intra-node tasks, we use OpenMP calls are used for time measurements and the maximum execution time among all ranks is reported. We provide distributed runs is counted using the Score-P profiler [41]. We provide default values. We use 1) Intel MKL (v19.1.1.217). While the library is proprietary, our measurements reaffirm that, like ScALAPACK, the implementation uses the suboptimal 2D processor decomposition; 2) SLATE [28] — a state-of-the-art distributed linear algebra framework targeted at exascale supercomputers; 3) the latest version of the CANDMC and CAPITAL libraries [32, 58], which use an asymptotically-optimal 2.5D decomposition. The implementations and their characteristics are listed in Table 2. Problem Sizes. We evaluate the algorithms starting from 2 compute nodes (4 MPI ranks) up to 512 nodes (1,024 ranks). For each node count, matrix sizes range from \( N = 2,048 \) to \( N = 2^{19} = 524,288 \), provided they fit into the allocated memory (e.g., LU or Cholesky factorization on a double-precision input matrix of dimension \( 262,144 \times 262,144 \) cannot be run on less than 32 nodes). Runs in which none of the libraries achieved more than 3% of the hardware peak are discarded since by adding more nodes the performance starts to deteriorate.

Our benchmarks reflect real-world problems in scientific computing. The High-Performance Linpack benchmark uses a maximal size of \( N = 16,473,600 \) [62]. In quantum physics, matrix size scales with \( 2^{\text{qubits}} \). In physical chemistry or density functional theory (DFT), simulations require factorizing matrices of atom interactions, yielding sizes ranging from \( N = 1,024 \) up to \( N = 131,072 \) [18, 66]. In machine learning, matrix factorizations are used for inverting Kronecker factors [52] whose sizes are usually around \( N = 4,096 \). This motivates us to focus not only on exascale problems, but also improve performance for relatively small matrices (\( N \leq 100,000 \)).

Comparison Targets. We use 1) Intel MKL (v19.1.1.217). While the library is proprietary, our measurements reaffirm that, like ScALAPACK, the implementation uses the suboptimal 2D processor decomposition; 2) SLATE [28] — a state-of-the-art distributed linear algebra framework targeted at exascale supercomputers; 3) the latest version of the CANDMC and CAPITAL libraries [32, 58], which use an asymptotically-optimal 2.5D decomposition. The implementations and their characteristics are listed in Table 2.

Comparison Targets. We use 1) Intel MKL (v19.1.1.217). While the library is proprietary, our measurements reaffirm that, like ScALAPACK, the implementation uses the suboptimal 2D processor decomposition; 2) SLATE [28] — a state-of-the-art distributed linear algebra framework targeted at exascale supercomputers; 3) the latest version of the CANDMC and CAPITAL libraries [32, 58], which use an asymptotically-optimal 2.5D decomposition. The implementations and their characteristics are listed in Table 2.

Problem Sizes. We evaluate the algorithms starting from 2 compute nodes (4 MPI ranks) up to 512 nodes (1,024 ranks). For each node count, matrix sizes range from \( N = 2,048 \) to \( N = 2^{19} = 524,288 \), provided they fit into the allocated memory (e.g., LU or Cholesky factorization on a double-precision input matrix of dimension \( 262,144 \times 262,144 \) cannot be run on less than 32 nodes). Runs in which none of the libraries achieved more than 3% of the hardware peak are discarded since by adding more nodes the performance starts to deteriorate.

Our benchmarks reflect real-world problems in scientific computing. The High-Performance Linpack benchmark uses a maximal size of \( N = 16,473,600 \) [62]. In quantum physics, matrix size scales with \( 2^{\text{qubits}} \). In physical chemistry or density functional theory (DFT), simulations require factorizing matrices of atom interactions, yielding sizes ranging from \( N = 1,024 \) up to \( N = 131,072 \) [18, 66]. In machine learning, matrix factorizations are used for inverting Kronecker factors [52] whose sizes are usually around \( N = 4,096 \). This motivates us to focus not only on exascale problems, but also improve performance for relatively small matrices (\( N \leq 100,000 \)).

Comparison Targets. We use 1) Intel MKL (v19.1.1.217). While the library is proprietary, our measurements reaffirm that, like ScALAPACK, the implementation uses the suboptimal 2D processor decomposition; 2) SLATE [28] — a state-of-the-art distributed linear algebra framework targeted at exascale supercomputers; 3) the latest version of the CANDMC and CAPITAL libraries [32, 58], which use an asymptotically-optimal 2.5D decomposition. The implementations and their characteristics are listed in Table 2.

**Problem Sizes.** We evaluate the algorithms starting from 2 compute nodes (4 MPI ranks) up to 512 nodes (1,024 ranks). For each node count, matrix sizes range from \( N = 2,048 \) to \( N = 2^{19} = 524,288 \), provided they fit into the allocated memory (e.g., LU or Cholesky factorization on a double-precision input matrix of dimension \( 262,144 \times 262,144 \) cannot be run on less than 32 nodes). Runs in which none of the libraries achieved more than 3% of the hardware peak are discarded since by adding more nodes the performance starts to deteriorate.

Our benchmarks reflect real-world problems in scientific computing. The High-Performance Linpack benchmark uses a maximal size of \( N = 16,473,600 \) [62]. In quantum physics, matrix size scales with \( 2^{\text{qubits}} \). In physical chemistry or density functional theory (DFT), simulations require factorizing matrices of atom interactions, yielding sizes ranging from \( N = 1,024 \) up to \( N = 131,072 \) [18, 66]. In machine learning, matrix factorizations are used for inverting Kronecker factors [52] whose sizes are usually around \( N = 4,096 \). This motivates us to focus not only on exascale problems, but also improve performance for relatively small matrices (\( N \leq 100,000 \)).

Comparison Targets. We use 1) Intel MKL (v19.1.1.217). While the library is proprietary, our measurements reaffirm that, like ScALAPACK, the implementation uses the suboptimal 2D processor decomposition; 2) SLATE [28] — a state-of-the-art distributed linear algebra framework targeted at exascale supercomputers; 3) the latest version of the CANDMC and CAPITAL libraries [32, 58], which use an asymptotically-optimal 2.5D decomposition. The implementations and their characteristics are listed in Table 2.

| Description | CO/LUX (LU) | CO/CHOX (Cholesky) |
|-------------|-------------|---------------------|
| pivoting    | TournPivot  | (no pivoting)       |
| A_{00}      | local getrf | –                   |
| A_{10} and A_{01} | reduction, local trsm | (similar to LU) |
| A_{11}      | scatter, local gemm | scatter, local gemm |

Table 1: Comparison of the implemented LU and Cholesky factorizations. Even though Cholesky performs half as many computations (the use of gemm instead of gem in A_{11}), it communicates the same amount of data, since the number of elements needed to perform gemm and gemt is the same.

8 IMPLEMENTATION

Our algorithms are implemented in C++, using MPI for inter-node communication. For static communication patterns (e.g., column reductions) we use dedicated, asynchronous MPI collectives. For runtime-dependent communication (e.g., pivot index distribution) we use MPI one-sided [31]. For intra-node tasks, we use OpenMP and local BLAS calls (provided by Intel MKL [34]) for computations. Our code is available as an open-source git repository.

Parallel decomposition. Our experiments show that the parallelization in the reduction dimension, while reducing communication volume, does incur performance overheads. This is mainly due to the increased communication latency, as well as smaller buffer sizes used for local BLAS calls. Since formal modeling of the tradeoff between communication volume and performance is outside of the scope of the paper, we keep the depth of parallelization in the third dimension as a tunable parameter, while providing heuristics-based default values.

Data distribution. CO/LUX and CO/CHOX provide ScALAPACK wrappers by using the highly-optimized COSTA algorithm [38] to transform the matrices between different layouts. In addition, they support the COSTA API for matrix descriptors, which is more general than ScALAPACK’s layout, as it supports matrices distributed in arbitrary grid-like layouts, processor assignments, and local blocks orderings.

9 EXPERIMENTAL EVALUATION

We compare CO/LUX and CO/CHOX with state-of-the-art implementations of corresponding distributed matrix factorizations. Measured values. We measure both the I/O cost and total time-to-solution. For I/O, the aggregate communication volume in distributed runs is counted using the Score-P profiler [41]. We provide both measured values and theoretical cost models. Local std::chrono calls are used for time measurements and the maximum execution time among all ranks is reported.

Infrastructure and Measurement. We run our experiments on the XC40 partition of the CSCS Piz Daint supercomputer which comprises 1,813 CPU nodes equipped with Intel Xeon E5-2695 v4 processors (2x18 cores, 64 GiB DDR3 RAM), interconnected by the Cray Aries network with a Dragonfly network topology. Since the CPUs are dual-socket, two MPI ranks are allocated per compute node.

Comparison Targets. We use 1) Intel MKL (v19.1.1.217). While the library is proprietary, our measurements reaffirm that, like ScALAPACK, the implementation uses the suboptimal 2D processor decomposition; 2) SLATE [28] — a state-of-the-art distributed linear algebra framework targeted at exascale supercomputers; 3) the latest version of the CANDMC and CAPITAL libraries [32, 58], which use an asymptotically-optimal 2.5D decomposition. The implementations and their characteristics are listed in Table 2.

10 RESULTS

Our experiments confirm advantages of CO/LUX and CO/CHOX in terms of both communication volume and time-to-solution over...
Table 2: Parallelization strategies and I/O cost models of the considered matrix factorization implementations. MKL and SLATE require a user to specify the processor decomposition and the block size. CANDMC provides default values, but our experiments show that the performance was significantly improved when we tuned the parameters. CO\textsubscript{nf}LUX and CO\textsubscript{nf}CHOX outperform all state-of-the-art libraries with out-of-the-box parameters. We validated our parallel I/O cost models: for MKL, SLATE, CO\textsubscript{nf}LUX, and CO\textsubscript{nf}CHOX, the error was within +/-3%. For CANDMC and CAPITAL, we used the models derived by the authors [33, 61], which overapproximated the measured values by approx. 30-40%.

Figure 8: Communication volume measurements across different scenarios for MKL, SLATE, CANDMC, and CO\textsubscript{nf}LUX. In all considered scenarios, enough memory $M \geq N^2/\frac{P^2}{3}$ was present to allow for the maximum number of replications $c = P^{1/3}$.

Figure 9: Achieved % of peak performance for LU factorization. We show median and 95% confidence intervals.

all other implementations tested. A significant communication reduction can be observed (up to 1.42 times for CO\textsubscript{nf}LUX compared with the second-best implementation for $P = 1,024$). Moreover, the performance models predict even greater benefits for larger runs (expected 2.1 times communication reduction for a full-machine run on the Summit supercomputer – Figure 8c). Most importantly, our implementations consistently outperform existing implementations (up to three times – Figures 1 and 9).

Communication volume. Fig. 8a presents the measured communication volume per node, as well as our derived cost models (Table 2) presented with solid lines, for $N = 16,384$. Observe that CO\textsubscript{nf}LUX communicates the least for all values of $P$. Note that
Near-I/O-Optimal Matrix Factorizations Technical Report, 2021,

11 RELATED WORK

Previous work on I/O analysis can be categorized into three classes (see Table 3): work based on direct pebbling or variants of it, such as Vitter’s block-based model [64]; works using geometric arguments of projections based on the Loomis-Whitney inequality [47]; and works applying optimizations limited to specific structural properties such as affine loops [27], and more generally, the polyhedral model program representation [9, 48, 51]. Although the scopes of those approaches significantly overlap — for example, kernels like matrix multiplication can be captured by most of the models — there are important differences both in methodology and the end-results they provide, as summarized in Table 3.

Dense linear algebra operators are among the standard core kernels in scientific applications. Ballard et al. [8] present a comprehensive overview of their asymptotic I/O lower bounds and I/O minimizing schedules, both for sparse and dense matrices. Recently, Olivry et al. introduced IOLB [51] — a framework for assessing

Since both MKL and SLATE use similar 2D decompositions, their communication volumes are mostly equal, with a slight advantage for SLATE. In Fig. 8b, we show the weak scaling characteristics of the analyzed implementations. Observe that for a fixed amount of work per node, the 2D algorithms - MKL and SLATE - scale sub-optimally. Figure 8c summarizes the communication volume reduction of CO\textsubscript{nf}/LUX compared with the second-best implementation, both for measurements and theoretical predictions. It can be seen that for all combinations of \( P \) and \( N \), CO\textsubscript{nf}/LUX always communicates the least. For all measured data points, the asymptotically optimal CANDMC performed worse than MKL or SLATE. The figure also presents the predicted communication cost of all considered implementations for up to \( P = 262,144 \) based on our theoretical models.

Performance. Our measurements show that both CO\textsubscript{nf}/LUX and CO\textsubscript{nf}/CHOX outperform all considered state-of-the-art libraries in almost all scenarios (Figures 1 and 11). Thanks to the optimized block decomposition and efficient overlap of computation and communication, our implementations achieve high performance already on relatively small matrices (approx. 40% of hardware peak for cases where \( N^2/P > 2^{27} \)). In cases where the local domain per processor becomes very small (\( N^2/P < 2^{27} \)) our block decomposition does not add that much benefit, since the performance is mostly latency-bound, and not bandwidth-bound. This is visible not only in strong scaling (Figures 9 and 10, a and b), but also in weak scaling (c), where the input size per processor \( N^2/P \) is constant. This is again caused by latency overheads of scattering data between 1D and 2.5D layouts.

However, as the local domains become larger and may be more efficiently pipelined and overlapped using asynchronous MPI routines and intra-node OpenMP parallelism, the advantage becomes significant (Figures 9 and 10). CO\textsubscript{nf}/LUX outperforms existing libraries up to three times (for \( P = 4, N = 4096 \), second-best library is SLATE – Figure 1) and CO\textsubscript{nf}/CHOX achieves up to 1.8 times speedup (e.g., \( P = 4, N = 4,096 \), second-best is again SLATE).

Implications for Exascale. Both the communication models’ predictions (Figure 8c) and measured speedups (Figures 1 and 11) allow us to predict that when running our implementations on exascale machines, we can expect to see further performance improvements over state-of-the-art libraries. Furthermore, throughput-oriented hardware, such as GPUs and FPGAs, may benefit even more from the communication reduction of our schedules. Thus, CO\textsubscript{nf}/LUX and CO\textsubscript{nf}/CHOX not only outperform the state-of-the-art libraries at relatively small scales — which are most common use cases in practice [18, 52, 66] — but also promise speedups on full-scale performance runs on modern supercomputers.

Figure 10: Achieved % of peak performance for Cholesky factorization. We show median and 95% confidence intervals.

Figure 11: Left: measured runtime speedup of CO\textsubscript{nf}/CHOX vs. fastest state-of-the-art library (S=SLATE [28], C=CAPITAL [33], M=MKL [34]). Right: CO\textsubscript{nf}/CHOX’s achieved % of machine peak performance.

\( N = 2^{17} = 131,072 \)

\( N = 2^{14} = 16,384 \)

\( N = 8,192 \sqrt{P} \)
sequential lower bounds for polyhedral programs. However, their computational model disallows recomputation (cf. Section 4.2).

Matrix factorizations are included in most of linear solvers’ libraries. With regard to the parallelization strategy, these libraries may be categorized into three groups: task-based: SLATE [28] (OpenMP tasks), DLAF [35] (HPX tasks), DPLASMA [12] (DaGuE scheduler), or CHAMELEON [3] (StarPU tasks); static 2D parallel: MKL [34], Elemental [53], or Cray LibSci [16]; communication-minimizing 2.5D parallel: CANDMC [57] and CAPITAL [33]. In the last decade, heavy focus was placed on heterogeneous architectures. Most GPU vendors offer hardware-customized BLAS solvers [50]. Agullo et al. [2] accelerated LU factorization using up to 4 GPUs. Azzam et al. [30] utilize NVIDIA’s GPU tensor cores to compute low-precision LU factorization and then iteratively refine the linear problem’s solution. Moreover, some of the distributed memory libraries support GPU offloading for local computations [28].

12 CONCLUSIONS

In this work, we present a method of analyzing I/O cost of DAAP—a general class of programs that covers many fundamental computational motifs. We show, both theoretically and in practice, that our pebbling-based approach for deriving the I/O lower bounds is more general: programs with disjoint array accesses cover a wide variety of applications, more powerful: it can explicitly capture inter-statement dependencies, more precise: it derives tighter I/O bounds, and more constructive: X-partition provides powerful hints for obtaining parallel schedules.

When applying the approach to LU and Cholesky factorizations, we are able to derive new lower bounds, as well as new, communication-avoiding schedules. Not only do they communicate less than state-of-the-art 2D and 3D decompositions — by a factor of up to 1.6x—but most importantly, they outperform existing commercial libraries in a wide range of problem parameters (up to 3x for LU, up to 1.8x for Cholesky). Finally, our code is openly available, offering full ScalAPACK layout compatibility.

13 ACKNOWLEDGEMENTS

This project received funding from the European Research Council (ERC) under the European Union’s Horizon 2020 programme (grant agreement DAPP, no. 678880), EPiGRAM-HS project (grant agreement no. 801039). Tal Ben-Nun is supported by the Swiss National Science Foundation (Ambizione Project #185778). The authors wish to acknowledge the support from the PASC program (Platform for Advanced Scientific Computing), as well as the Swiss National Supercomputing Center (CSCS) for providing computing infrastructure.

REFERENCES

[1] Alok Aggarwal and S Vitter. Jeffrey. 1988. The input/output complexity of sorting and related problems. Commun. ACM 31, 9 (1988), 1116–1127.
[2] Emmanuel Agullo, Cédric Augonnet, Jack Dongarra, Mathieu Faivre, Julien Langou, Hatem Ltaief, and Stanimire Tomov. 2011. LU factorization for accelerator-based systems. In 2011 9th IEEE/ACIS International Conference on Computer Systems and Applications (AICCSA). IEEE, 217–224.
[3] Emmanuel Agullo, Cédric Augonnet, Jack Dongarra, Hatem Ltaief, Raymond Namyst, Samuel Thibault, and Stanimire Tomov. 2010. Faster, Cheaper, Better—a Hybridization Methodology to Develop Linear Algebra Software for GPUs. In GPU Computing Gems. Wu mei W. Huw (Ed.). Vol. 2. Morgan Kaufmann. https://hal.inria.fr/inria-00547847
[4] Emmanuel Agullo, Jack Dongarra, Bilel Hadri, Jakub Kurzak, Julie Langou, Julien Langou, Hatem Ltaief, Piotr Luszczek, and Assim Yarkhan. 2011. PLASMA Users’ Guide. Parallel Linear Algebra Software for Multicore Architectures. Report technique, Innovative Computing Laboratory, University of Tennessee (2011).
[5] Joel Alwen and Vladimir Serbinenko. 2015. High parallel complexity graphs and memory-hard functions. In Proceedings of the forty-seventh annual ACM symposium on Theory of computing. 595–603.
[6] Edward Anderson, Zhaosun Bai, Christian Bischof, Susan Blackford, Jack Dongarra, Jeremy Du Croz, Anne Greenbaum, Sven Hammarling, Alan McKenney, and Danny Sorensen. 1999. LAPACK Users’ guide. Vol. 9. SIAM.
[7] Grey Ballard, James Demmel, Olga Holtz, and Oded Schwartz. 2010. Communication-optimal parallel and sequential Cholesky decomposition. SIAM Journal on Scientific Computing 32, 6 (2010), 3459–3523.
[8] Grey Ballard, James Demmel, Olga Holtz, and Oded Schwartz. 2011. Minimizing communication in numerical linear algebra. SIAM J. Matrix Anal. Appl. 32, 3 (2011), 866–901.
[9] Mohamed-Walid Benabderrahmane, Louis-Noël Pouchet, Albert Cohen, and Cédric Bastoul. 2010. The polyhedral model is more widely applicable than you think. In International Conference on Compiler Construction. Springer, 283–303.
[10] L. S. Blackford, J. Choi, A. Cleary, E. D’Azevedo, J. Demmel, I. Dhillon, J. Dongarra, Atanas Rountev, and P. Sadayappan. 2008. Automatic Transformations for Communication-Optimized Parallel and Related Problems. In Innovative Computing Laboratory, University of Tennessee (2011).
[11] Uday Bondhugula, Muthu Baskaran, Srinath Krishnamoorthy, J. Ramanujam, Atanas Rountev, and P. Sadayappan. 2008. Automatic Transformations for Communication-Optimized Parallel and Related Problems. In Innovative Computing Laboratory, University of Tennessee (2011).
[12] John Bruno and Ravi Sethi. 1976. Code generation for a one-register machine. Journal of the ACM (JACM) 23, 3 (1976), 502–510.
[13] J Choi et al. 1996. ScalAPACK: A portable linear algebra library for distributed memory computers—design issues and performance. Comp. Phys. Comm. (1996).
[14] Michael Christ, James Demmel, Nicholas Knight, Thomas Scannlon, and Katherine Yelick. 2013. Communication lower bounds and optimal algorithms for programs that reference arrays. Part I. arXiv preprint arXiv:1308.0068 (2013).

Table 3: Overview of different approaches to modeling data movement.

| Scope | Features | Problem specific |
|-------|----------|-----------------|
| General cDAGs | Expresses complex data dependencies, Directly exposes schedules, Intuitive | Takes advantage of problem-specific features |
| Well-developed theory and tools, Guaranteed to find solution for given class of programs, Fails to capture dependencies between statements, Limited scope | Tends to provide best practical results, Requires large manual effort for each algorithm separately, Difficult to generalize, Often based on heuristics with no guarantees on optimality |

Matrix of approaches to modeling data movement.

| Pebbling [13, 26, 37, 45, 56] | Projection-based [8, 15, 20, 21, 23, 51] | Problem specific [1, 9, 17, 48, 66] |
|-------------------------------|---------------------------------|-----------------------------------|
| Scope | Features | |
| General | Scope | Expresses complex data dependencies, Directly exposes schedules, Intuitive | Well-developed theory and tools, Guaranteed to find solution for given class of programs, Fails to capture dependencies between statements, Limited scope | Takes advantage of problem-specific features |
| Takes advantage of problem-specific features, Tends to provide best practical results, Requires large manual effort for each algorithm separately, Difficult to generalize, Often based on heuristics with no guarantees on optimality | |
