Red-Blue Pebbling Revisited: Near Optimal Parallel Matrix-Matrix Multiplication

Grzegorz Kwasniewski[1], Marko Kabić[2,3], Maciej Besta[1],
Joost VandeVondele[2,3], Raffaele Solcà[2,3], Torsten Hoefler[1]
[1]Department of Computer Science, ETH Zurich, [2]ETH Zurich, [3]Swiss National Supercomputing Centre (CSCS)

ABSTRACT

We propose COSMA: a parallel matrix-matrix multiplication algorithm that is near communication-optimal for all combinations of matrix dimensions, processor counts, and memory sizes. The key idea behind COSMA is to derive an optimal (up to a factor of 0.03% for 10MB of fast memory) sequential schedule and then parallelize it, preserving I/O optimality. To achieve this, we use the red-blue pebble game to precisely model MMM dependencies and derive a constructive and tight sequential and parallel I/O lower bound proofs. Compared to 2D or 3D algorithms, which fix processor decomposition upfront and then map it to the matrix dimensions, it reduces communication volume by up to $\sqrt{3}$ times. COSMA outperforms the established ScaLAPACK, CARMA, and CTF algorithms in all scenarios up to 12.8x (2.2x on average), achieving up to 88% of Piz Daint's peak performance. Our work does not require any hand tuning and is maintained as an open source implementation.

1 INTRODUCTION

Matrix-matrix multiplication (MMM) is one of the most fundamental building blocks in scientific computing, used in linear algebra algorithms [13, 15, 41], machine learning [6], graph processing [4, 8, 18, 36, 43, 51], computational chemistry [21], and others. Thus, accelerating MMM routines is of great significance for many domains. In this work, we focus on minimizing the amount of transferred data in MMM, both across the memory hierarchy (*vertical I/O*) and between processors (*horizontal I/O*, aka "communication")[1].

The path to I/O optimality of MMM algorithms is at least 50 years old. The first parallel MMM algorithm is by Cannon [10], which works for square matrices and square processor decompositions. Subsequent works [24, 25] generalized the MMM algorithm to rectangular matrices, different processor decompositions, and communication patterns. PUMMA [17] package generalized previous works to transposed matrices and different data layouts. SUMMA algorithm [55] further extended it by optimizing the communication, introducing pipelining and communication-computation overlap. This is now a state-of-the-art so-called 2D algorithm (it decomposes processors in a 2D grid) used e.g., in ScaLAPACK library [14].

[1]We also focus only on "classical" MMM algorithms which perform n^3 multiplications and additions. We do not analyze Strassen-like routines [53], as in practice they are often slower [19].

SC'19, November 17–22, 2019, Denver, CO, USA
© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

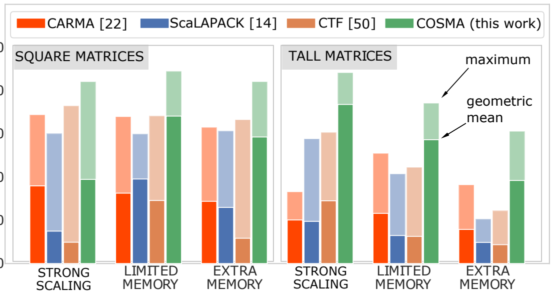


Figure 1: Percentage of peak flop/s across the experiments ranging from 109 to 18,432 cores achieved by COSMA and the state-of-the-art libraries (Sec. 9).

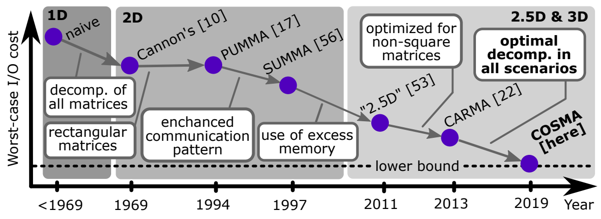


Figure 2: Illustratory evolution of MMM algorithms reaching the I/O lower bound.

Agarwal et al. [1] showed that in a presence of extra memory, one can do better and introduces a 3D processor decomposition. The 2.5D algorithm by Solomonik and Demmel [52] effectively interpolates between those two results, depending on the available memory. However, Demmel et al. showed that algorithms optimized for square matrices often perform poorly when matrix dimensions vary significantly [22]. Such matrices are common in many relevant areas, for example in machine learning [59, 60] or computational chemistry [44, 48]. They introduced CARMA [22], a recursive algorithm that achieves asymptotic lower bounds for all configurations of dimensions and memory sizes. This evolution for chosen steps is depicted symbolically in Figure 2.

Unfortunately, we observe several limitations with state-of-the art algorithms. ScaLAPACK [14] (an implementation of SUMMA) supports only the 2D decomposition, which is communication–inefficient in the presence of extra memory. Also, it requires a user to fine-tune parameters such as block sizes or a processor grid size. CARMA supports only scenarios when the number of processors is a power of two [22], a serious limitation, as the number of processors is usually determined by the available hardware resources. Cyclops Tensor Framework (CTF) [49] (an implementation of the 2.5D decomposition) can utilize any number of processors, but its decompositions may be far from optimal (§ 9). We also emphasize that *asymptotic complexity is an insufficient measure of practical performance.* We later (§ 6.2) identify that CARMA performs up to $\sqrt{3}$ more communication. Our observations are summarized in Table 1.

	2D [55]	2.5D [52]	recursive [22]	COSMA (this work)
Input:	User–specified grid	Available memory	Available memory, matrix dimensions	Available memory, matrix dimensions
Step 1	Split m and n	Split m, n, k	Split recursively the largest dimension	Find the optimal sequential schedule
Step 2	Map matrices to processor grid	Map matrices to processor grid	Map matrices to recursion tree	Map sequential domain to matrices
	👎 Requires manual tuning 👎 Asymptotically more comm.	👍 Optimal for $m = n$ 👎 Inefficient for $m \ll n$ or $n \ll m$ 👎 Inefficient for some p	👍 Asymptotically optimal for all m, n, k, p 👎 Up to $\sqrt{3}$ times higher comm. cost 👎 p must be a power of 2	👍 Optimal for all m, n, k 👍 Optimal for all p 👍👍 Best time-to-solution

Table 1: Intuitive comparison between the COSMA algorithm and the state-of-the-art 2D, 2.5D, and recursive decompositions. $C = AB$, $A \in \mathbb{R}^{m \times k}$, $B \in \mathbb{R}^{k \times n}$

Their practical implications are shown in Figure 1, where we see that all existing algorithms perform poorly for some configurations.

In this work, we present COSMA (Communication Optimal S-partition-based Matrix multiplication Algorithm): an algorithm that takes a new approach to multiplying matrices and alleviates the issues above. COSMA is I/O optimal for *all combinations of parameters* (up to the factor of $\sqrt{S}/(\sqrt{S+1}-1)$, where S is the size of the fast memory[2]). The driving idea is to develop a general method of deriving I/O optimal schedules by explicitly modeling data reuse in the red-blue pebble game. We then parallelize the sequential schedule, minimizing the I/O between processors, and derive an optimal domain decomposition. This is in contrast with the other discussed algorithms, which fix the processor grid upfront and then map it to a sequential schedule for each processor. We outline the algorithm in § 3. To prove its optimality, we first provide a new constructive proof of a sequential I/O lower bound (§ 5.1), then we derive the communication cost of parallelizing the sequential schedule (§ 6.2), and finally we construct an I/O optimal parallel schedule (§ 6.3). The detailed communication analysis of COSMA, 2D, 2.5D, and recursive decompositions is presented in Table 2. Our algorithm reduces the data movement volume by a factor of up to $\sqrt{3} \approx 1.73$x compared to the asymptotically optimal recursive decomposition and up to $\max\{m, n, k\}$ times compared to the 2D algorithms, like Cannon's [39] or SUMMA [55].

Our implementation enables transparent integration with the ScaLAPACK data format [16] and delivers near-optimal computation throughput. We later (§ 7) show that the schedule naturally expresses communication–computation overlap, enabling even higher speedups using Remote Direct Memory Access (RDMA). Finally, our I/O-optimal approach is generalizable to other linear algebra kernels. We provide the following contributions:

- We propose COSMA: a distributed MMM algorithm that is nearly-optimal (up to the factor of $\sqrt{S}/(\sqrt{S+1}-1)$) for *any combination of input parameters* (§ 3).
- Based on the red-blue pebble game abstraction [34], we provide a new method of deriving I/O lower bounds (Lemma 2), which may be used to generate optimal schedules (§ 4).
- Using Lemma 2, we provide a new constructive proof of the sequential MMM I/O lower bound. The proof delivers constant factors tight up to $\sqrt{S}/(\sqrt{S+}-1)$(§ 5).
- We extend the sequential proof to parallel machines and provide I/O optimal parallel MMM schedule (§ 6.3).
- We reduce memory footprint for communication buffers and guarantee minimal local data reshuffling by using a blocked data layout (§ 7.6) and a static buffer pre-allocation (§ 7.5), providing compatibility with the ScaLAPACK format.
- We evaluate the performance of COSMA, ScaLAPACK, CARMA, and CTF on the CSCS Piz Daint supercomputer for an extensive selection of problem dimensions, memory sizes, and numbers of processors, showing significant I/O reduction and the speedup of up to 8.3 times over the second-fastest algorithm (§ 9).

2 BACKGROUND

We first describe our machine model (§ 2.1) and computation model (§ 2.2). We then define our optimization goal: *the I/O cost* (§ 2.3).

2.1 Machine Model

We model a parallel machine with p processors, each with local memory of size S words. A processor can send and receive from any other processor up to S words at a time. To perform any computation, all operands must reside in processor' local memory. If shared memory is present, then it is assumed that it has infinite capacity. A cost of transferring a word from the shared to the local memory is equal to the cost of transfer between two local memories.

2.2 Computation Model

We now briefly specify a model of *general* computation; we use this model to derive the theoretical I/O cost in both the sequential and parallel setting. An execution of an algorithm is modeled with the *computational directed acyclic graph* (CDAG) $G = (V, E)$ [11, 28, 46]. A vertex $v \in V$ represents one elementary operation in the given computation. An edge $(u, v) \in E$ indicates that an operation v depends on the result of u. A set of all immediate predecessors (or successors) of a vertex are its *parents* (or *children*). Two selected subsets $I, O \subset V$ are *inputs* and *outputs*, that is, sets of vertices that have no parents (or no children, respectively).

Red-Blue Pebble Game

Hong and Kung's red-blue pebble game [34] models an execution of an algorithm in a two-level memory structure with a small-and-fast as well as large-and-slow memory. A red (or a blue) pebble placed on a vertex of a CDAG denotes that the result of the corresponding elementary computation is inside the fast (or slow) memory. In the initial (or terminal) configuration, only inputs (or outputs) of the CDAG have blue pebbles. There can be at most S red pebbles used at any given time. A *complete CDAG calculation* is a sequence of moves that lead from the initial to the terminal configuration. One is allowed to: place a red pebble on any vertex with a blue pebble (load), place a blue pebble on any vertex with a red pebble (store), place a red pebble on a vertex whose parents all have red pebbles (compute), remove any pebble, red or blue, from any vertex (free

[2]Throughout this paper we use the original notation from Hong and Kung to denote the memory size S. In literature, it is also common to use the symbol M [2, 3, 33].

memory). An *I/O optimal* complete CDAG calculation corresponds to a sequence of moves (called *pebbling* of a graph) which minimizes loads and stores. In the MMM context, it is an order in which all n^3 multiplications are performed.

2.3 Optimization Goals

Throughout this paper we focus on the *input/output (I/O) cost* of an algorithm. The I/O cost Q is the total number of words transferred during the execution of a schedule. On a sequential or shared memory machine equipped with small-and-fast and slow-and-big memories, these transfers are load and store operations from and to the slow memory (also called the *vertical I/O*). For a distributed machine with a limited memory per node, the transfers are communication operations between the nodes (also called the *horizontal I/O*). A schedule is *I/O optimal* if it minimizes the I/O cost among all schedules of a given CDAG. We also model a *latency cost* L, which is a maximum number of messages sent by any processor.

2.4 State-of-the-Art MMM Algorithms

Here we briefly describe strategies of the existing MMM algorithms. Throughout the whole paper, we consider matrix multiplication $C = AB$, where $A \in \mathbb{R}^{m\times k}, B \in \mathbb{R}^{k\times n}, C \in \mathbb{R}^{m\times n}$, where m, n, and k are matrix dimensions. Furthermore, we assume that the size of each matrix element is one word, and that $S < \min\{mn, mk, nk\}$, that is, none of the matrices fits into single processor's fast memory.

We compare our algorithm with the 2D, 2.5D, and recursive decompositions (we select parameters for 2.5D to also cover 3D). We assume a square processor grid $[\sqrt{p}, \sqrt{p}, 1]$ for the 2D variant, analogously to Cannon's algorithm [10], and a cubic grid $[\sqrt{p/c}, \sqrt{p/c}, c]$ for the 2.5D variant [52], where c is the amount of the "extra" memory $c = pS/(mk + nk)$. For the recursive decomposition, we assume that in each recursion level we split the largest dimension m, n, or k in half, until the domain per processor fits into memory. The detailed complexity analysis of these decompositions is in Table 2. We note that ScaLAPACK or CTF can handle non-square decompositions, however they create different problems, as discussed in § 1. Moreover, in § 9 we compare their performance with COSMA and measure significant speedup in *all* scenarios.

3 COSMA: HIGH-LEVEL DESCRIPTION

COSMA decomposes processors by parallelizing the near optimal sequential schedule under constraints: (1) equal work distribution and (2) equal memory size per processor. Such a local sequential schedule is independent of matrix dimensions. Thus, intuitively, instead of dividing a global domain among p processors (the *top-down* approach), we start from deriving a near I/O optimal *sequential* schedule. We then parallelize it, minimizing the I/O and latency costs Q, L (the *bottom-up* approach); Figure 3 presents more details.

COSMA is sketched in Algorithm 1. In Line 1 we derive a sequential schedule, which consists of series of $a \times a$ outer products. (Figure 4 b). In Line 2, each processor is assigned to compute b of these products, forming a *local domain* $\mathcal{D}$ (Figure 4 c), that is each $\mathcal{D}$ contains $a \times a \times b$ vertices (multiplications to be performed - the derivation of a and b is presented in § 6.3). In Line 3, we find a processor grid $\mathcal{G}$ that evenly distributes this domain by the matrix dimensions m, n, and k. If the dimensions are not divisible by a or b, this function also evaluates new values of a_{opt} and b_{opt}

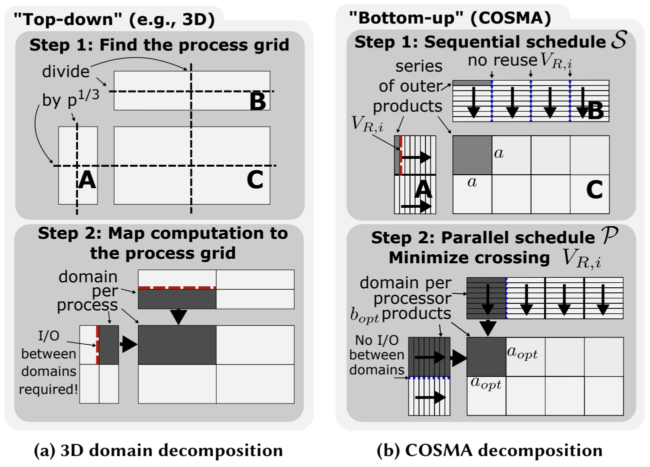

"Top-down" (e.g., 3D)
Step 1: Find the process grid
divide
by $p^{1/3}$
A
B
C
Step 2: Map computation to the process grid
domain per process
I/O between domains required!
"Bottom-up" (COSMA)
Step 1: Sequential schedule $\mathcal{S}$
series of outer products
no reuse $V_{R,i}$
$V_{R,i}$
a
Step 2: Parallel schedule $\mathcal{P}$ Minimize crossing $V_{R,i}$
domain per processor b_{opt} products
No I/O between domains
a_{opt}
(a) 3D domain decomposition
(b) COSMA decomposition

Figure 3: Domain decomposition using $p = 8$ processors. In scenario (a), a straightforward 3D decomposition divides every dimension in $p^{1/3} = 2$. In scenario (b), COSMA starts by finding a near optimal sequential schedule and then parallelizes it minimizing crossing data reuse $V_{R,i}$ (§ 5). The total communication volume is reduced by 17% compared to the former strategy.

by fitting the best matching decomposition, possibly not utilizing some processors (§ 7.1, Figure 4 d-f). The maximal number of idle processors is a tunable parameter δ. In Line 5, we determine the initial decomposition of matrices A, B, and C to the submatrices A_l, B_l, C_l that are local for each processor. COSMA may handle any initial data layout, however, an optimal block-recursive one (§ 7.6) may be achieved in a preprocessing phase. In Line 6, we compute the size of the communication step, that is, how many of b_{opt} outer products assigned to each processor are computed in a single round, minimizing the latency (§ 6.3). In Line 7 we compute the number of sequential steps (Lines 8–11) in which every processor: (1) distributes and updates its local data A_l and B_l among the grid $\mathcal{G}$ (Line 9), and (2) multiplies A_l and B_l (Line 10). Finally, the partial results C_l are reduced over $\mathcal{G}$ (Line 12).

I/O Complexity of COSMA

Lines 2–7 require no communication (assuming that the parameters m, n, k, p, S are already distributed). The loop in Lines 8-11 executes $\lceil 2ab/(S - a^2)\rceil$ times. In Line 9, each processor receives $|A_l| + |B_l|$ elements. Sending the partial results in Line 12 adds a^2 communicated elements. In § 6.3 we derive the optimal values for a and b, which yield a total of $\min\left\{S + 2 \cdot \frac{mnk}{p\sqrt{S}}, 3\left(\frac{mnk}{P}\right)^{2/3}\right\}$ elements communicated.

4 ARBITRARY CDAGS: LOWER BOUNDS

We sketch key concepts and methods for deriving I/O lower bounds for general CDAGs. We extend the S-partition method and the related main lemma by Hong and Kung [34]. That lemma, however, fails to provide a tight I/O bound. In our generalization, the **X-*partition***, we let the constraint X to take any value (not only S, the fast memory size). Our key result here, Lemma 2, allows us to derive a constructive proof of a tight (for large S) I/O lower bound for a sequential execution of the MMM CDAG (§ 5). This method is extended in Section 6 to establish a tight parallel I/O lower bound.

Algorithm 1 COSMA

```
Input: matrices $A \in \mathbb{R}^{m\times k}$, $B \in \mathbb{R}^{k\times n}$,
    number of processors: $p$, memory size: $S$, computation-I/O tradeoff ratio $\rho$
Output: matrix $C = AB \in \mathbb{R}^{m\times n}$
1: $a \leftarrow FindSeqSchedule(S, m, n, k, p)$        ▷ sequential I/O optimality (§ 5)
2: $b \leftarrow ParallelizeSched(a, m, n, k, p)$       ▷ parallel I/O optimality (§ 6)
3: $(\mathcal{G}, a_{opt}, b_{opt}) \leftarrow FitRanks(m, n, k, a, b, p, \delta)$
4: for all $p_i \in \{1 \ldots p\}$ do in parallel
5:     $(A_l, B_l, C_l) \leftarrow GetDataDecomp(A, B, \mathcal{G}, p_i)$
6:     $s \leftarrow \left\lfloor \frac{S-a_{opt}^2}{2a_{opt}} \right\rfloor$        ▷ latency-minimizing size of a step (6.3)
7:     $t \leftarrow \left\lceil \frac{b_{opt}}{s} \right\rceil$        ▷ number of steps
8:     for $j \in \{1 \ldots t\}$ do
9:         $(A_l, B_l) \leftarrow DistrData(A_l, B_l, \mathcal{G}, j, p_i)$
10:        $C_l \leftarrow Multiply(A_l, B_l, j)$        ▷ compute locally
11:    end for
12:    $C \leftarrow Reduce(C_l, \mathcal{G})$        ▷ reduce the partial results
13: end for
```

Due to space constraints, a formal definition of X-partition, proofs of all our lemmas, and a detailed discussion on their derivation and implications, are in an extended technical report [3]. Readers interested in practical concerns may proceed directly to Section 6.

Reuse-based Lemma

A proof of the original red-blue pebble lemma [34] is based on the fact that an optimal complete calculation (pebbling) which performs q I/O operations, $hS \leq q < (h+1)S$ for some h, can be associated with an X-partition for $X = 2S$. Subsets (also called *subcomputations*) $V_1, \ldots V_h \in V$ contain vertices that are red-pebbled *between every S I/O operations of this calculation* (i.e., between every (tS)-th and $[(t+1)S]$-th I/O operation, $t \in \{1, ..., h-1\}$). We refer the reader to the original paper for details of construction of sets V_i, $i = 1 \ldots h$. A sketch of this construction is also in the technical report.

Before we present the lemmas, we need to introduce four more subsets of V, defined for each V_i. $V_{R,i}$ (the *reuse set*) contains vertices that have red pebbles placed on them just before V_i begins (these vertices stay in the fast memory and are *reused* during V_i). $V_{B,i}$ (the *load set*) contains vertices that have blue pebbles placed on them just before V_i begins, and have red pebbles placed on them during V_i (for these vertices, we need to load data from the slow to the fast memory). By definition[4], $Dom(V_i) = V_{R,i} \cup V_{B,i}$. We define similar subsets $W_{B,i}$ and $W_{R,i}$ for the minimum set[5] $Min(V_i)$. $W_{B,i}$ (the *store set*) contains all vertices in V_i that have a blue pebble placed on them during V_i (i.e., the corresponding data must be stored in the slow memory). $W_{R,i}$ (called the *cache set*) contains all vertices in V_i that have a red pebble at the end of V_i (the data resides in the fast memory after V_i).

Denote an upper bound on $|V_{R,i}|$ and $|W_{B,i}|$ as $R(S)$ ($\forall_i \max\{|V_{R,i}|, |W_{B,i}|\} \leq R(S) \leq S$). Further, denote a lower bound on $|V_{B,i}|$ and $|W_{B,i}|$ as $T(S)$ ($\forall_i 0 \leq T(S) \leq \min\{|V_{B,i}|, |W_{B,i}|\}$). We now use $R(S)$ and $T(S)$ to tighten the bound on Q.

We now use the above definitions and observations to ***generalize the result of Hong and Kung [34].***

Lemma 1. *Denote $H(X)$ as the minimum number of subcomputations in any valid X-partition of a CDAG $G = (V, E)$, for any $X \geq S$. The minimal number Q of I/O operations for any valid execution of a CDAG $G = (V, E)$ is bounded by*

$$Q \geq (X - R(S) + T(S)) \cdot (H(X) - 1) \tag{1}$$

where $R(S)$ is the maximum reuse set size and $T(S)$ is the minimum store set size. Moreover, we have

$$H(X) \geq \frac{|V|}{|V_{max}|} \tag{2}$$

where $V_{max} = \arg\max_{V_i \in \mathcal{S}(X)} |V_i|$ is the largest subset of vertices in the CDAG schedule $\mathcal{S}(X) = \{V_1, \ldots, V_h\}$.

From this lemma, we derive the following lemma that we use to prove a tight I/O lower bound for MMM (Theorem 1):

Lemma 2. *Define the number of computations performed by V_i for one loaded element as the* computational intensity $\rho_i = \frac{|V_i|}{X-|V_{R,i}|+|W_{B,i}|}$ *of the subcomputation V_i. Denote $\rho = \max_i(\rho_i) \leq \frac{|V_{max}|}{X-R(S)+T(S)}$ to be the* maximal computational intensity. *Then, the number of I/O operations Q is bounded by $Q \geq |V|/\rho$.*

5 NEAR-OPTIMAL SEQUENTIAL MMM

In this section, we present our main theoretical contribution: a constructive proof of a tight I/O lower bound for classical matrix-matrix multiplication. In § 6, we extend it to the parallel setup (Theorem 2). This result is tight (up to diminishing factor $\sqrt{S}/(\sqrt{S+1}-1)$), and therefore may be seen as the last step in the long sequence of improved bounds. Hong and Kung [34] derived an asymptotic bound $\Omega\left(n^3/\sqrt{S}\right)$ for the sequential case. Irony et al. [33] extended the lower bound result to a parallel machine with p processes, each having a fast private memory of size S, proving the $\frac{n^3}{4\sqrt{2}p\sqrt{S}} - S$ lower bound on the communication volume per process. Recently, Smith and van de Gein [47] proved a tight sequential lower bound (up to an additive term) of $2mnk/\sqrt{S} - 2S$. Our proof improves the additive term and extends it to a parallel schedule.

Theorem 1 (Sequential Matrix Multiplication I/O lower bound). *Any pebbling of MMM CDAG which multiplies matrices of sizes $m \times k$ and $k \times n$ by performing mnk multiplications requires a minimum number of $\frac{2mnk}{\sqrt{S}} + mn$ I/O operations.*

The proof of Theorem 1 requires Lemmas 3 and 4, which in turn, require a following definition:

Greedy schedule

A schedule $\mathcal{S} = \{V_1, \ldots, V_h\}$ is *greedy* if during every subcomputation V_r every vertex u that will hold a red pebble either has a child in V_r or belongs to V_r. Then, our first result is an I/O lower bound for such schedules:

Lemma 3. *Any greedy schedule that multiplies matrices of sizes $m \times k$ and $k \times n$ using mnk multiplications requires a minimum number of $\frac{2mnk}{\sqrt{S}} + mn$ I/O operations.*

Intuition: Restricting the analysis to greedy schedules provides explicit information of a state of memory (sets V_r, $V_{R,r}$, $W_{B,r}$), and to a corresponding CDAG pebbling. Additional constraints (§ 5.1) guarantee feasibility of a derived schedule (and therefore, lower bound tightness).

[3]Available at https://arxiv.org/abs/1908.09606
[4] A dominator set $Dom(V_i)$ is a set of vertices in V, such that every path from any input of a CDAG to any vertex in V_i must contain at least one vertex in $Dom(V_i)$.
[5] The minimum set $Min(V_i)$ is a set of all vertices in V_i with no children in V_i.

5.1 Near-Optimal Greedy Schedule

In the technical report, it is proven that an optimal greedy schedule is composed of $\frac{mnk}{R(S)}$ outer product calculations, while loading $\sqrt{R(S)}$ elements of each of matrices A and B. While the lower bound is achieved for $R(S) = S$, such a schedule is infeasible, as at least some additional red pebbles, except the ones placed on the reuse set $V_{R,r}$, have to be placed on $2\sqrt{R(S)}$ vertices of A and B.

A direct way to obtain a feasible greedy schedule is to set $X = S$, ensuring that the dominator set can fit into the memory. Then each subcomputation is an outer-product of column-vector of matrix A and row-vector of B, both holding $\sqrt{S+1}-1$ values. Such a schedule performs $\frac{2mnk}{\sqrt{S+1}-1} + mn$ I/O operations, a factor of $\frac{\sqrt{S}}{\sqrt{S+1}-1}$ more than a lower bound, which quickly approach 1 for large S. Listing 1 provides a pseudocode of this algorithm, which is a well-known rank-1 update formulation of MMM.

```
for $i_1 = 1 : \lceil m/a \rceil$
  for $j_1 = 1 : \lceil n/a \rceil$
    for $r = 1 : k$
      for $i_2 = i_1 \cdot T : \min((i_1 + 1) \cdot a, m)$
        for $j_2 = j_1 \cdot T : \min((j_1 + 1) \cdot a, n)$
          $C(i_2, j_2) = C(i_2, j_2) + A(i_2, r) \cdot B(r, j_2)$
```

Listing 1: Pseudocode of near optimal sequential MMM, $a = \sqrt{S+1} - 1$

5.2 Greedy vs Non-greedy Schedules

Greedy schedules provide means to directly derive near-optimal schedules from the corresponding lower bounds, as shown in § 5.1. However, restricting analysis just to them may not provide correct lower bounds for general CDAGs. In this section, a different approach is used to prove the lower bound for non-greedy schedules - however, without a direct way to retrieve the corresponding schedule. These two bounds match, proving that the derived schedule (Listing 1) is near-optimal.

LEMMA 4. *Any non-greedy schedule computing classical matrix multiplication performs at least $\frac{2mnk}{\sqrt{S}} + mn$ I/O operations.*

Proof of Theorem 1:
Lemma 3 establishes that the I/O lower bound for any greedy schedule is $Q = 2mnk/\sqrt{S} + mn$. Lemma 4 establishes that no other schedule can perform less I/O operations.

□

6 OPTIMAL PARALLEL MMM

We now derive the schedule of COSMA from the results from § 5.1. The key notion is the data reuse, that determines not only the sequential execution, as discussed in § 4 , but also the parallel scheduling. Specifically, if the data reuse set spans across multiple local domains, then this set has to be communicated between these domains, increasing the I/O cost (Figure 3). We first introduce a formalism required to parallelize the sequential schedule (§ 6.1). In § 6.2, we generalize parallelization strategies used by the 2D, 2.5D, and recursive decompositions, deriving their communication cost and showing that none of them is optimal in the whole range of parameters. We finally derive the optimal decomposition (*FindOptimalDomain* function in Algorithm 1) by expressing it as an optimization problem (§ 6.3), and analyzing its I/O and latency cost. The remaining steps in Algorithm 1: *FitRanks*, *GetDataDecomp*, as well as *DistrData* and *Reduce* are discussed in § 7.1, § 7.6, and § 7.2, respectively. For a distributed machine, we assume that all matrices fit into collective memories of all processors: $pS \geq mn + mk + nk$. For a shared memory setting, we assume that all inputs start in a common slow memory.

6.1 Sequential and Parallel Schedules

We now describe how a parallel schedule is formed from a sequential one. The sequential schedule $\mathcal{S}$ partitions the CDAG $G = (V, E)$ into $H(S)$ subcomputations V_i. The parallel schedule $\mathcal{P}$ divides $\mathcal{S}$ among p processors: $\mathcal{P} = \{\mathcal{D}_1, \dots \mathcal{D}_p\}$, $\bigcup_{j=1}^{p} \mathcal{D}_j = \mathcal{S}$. The set $\mathcal{D}_j$ of all V_k assigned to processor j forms a *local domain* of j (Fig. 4c).

If two local domains $\mathcal{D}_k$ and $\mathcal{D}_l$ are dependent, that is, $\exists u, \exists v : u \in \mathcal{D}_k \land v \in \mathcal{D}_l \land (u, v) \in E$, then u has to be *communicated* from processor k to l. The total number of vertices communicated between all processors is the *I/O cost Q* of schedule $\mathcal{P}$. We say that the parallel schedule $\mathcal{P}_{opt}$ is *communication–optimal* if $Q(\mathcal{P}_{opt})$ is minimal among all possible parallel schedules.

The vertices of MMM CDAG may be arranged in an $[m \times n \times k]$ 3D grid called an *iteration space* [58]. The orthonormal vectors **i**, **j**, **k** correspond to the loops in Lines 1-3 in Listing 1 (Figure 3a). We call a schedule $\mathcal{P}$ *parallelized in dimension* **d** if we "cut" the CDAG along dimension **d**. More formally, each local domain $\mathcal{D}_j, j = 1 \dots p$ is a grid of size either $[m/p, n, k]$, $[m, n/p, k]$, or $[m, n, k/p]$. The schedule may also be parallelized in two dimensions ($\mathbf{d}_1\mathbf{d}_2$) or three dimensions ($\mathbf{d}_1\mathbf{d}_2\mathbf{d}_3$) with a local domain size $[m/p_m, n/p_n, k/p_k]$ for some p_m, p_n, p_k, such that $p_m p_n p_k = p$. We call $\mathcal{G} = [p_m, p_n, p_k]$ the *processor grid* of a schedule. E.g., Cannon's algorithm is parallelized in dimensions **ij** , with the processor grid $[\sqrt{p}, \sqrt{p}, 1]$. COSMA, on the other hand, may use any of the possible parallelizations, depending on the problem parameters.

6.2 Parallelization Strategies for MMM

The sequential schedule $\mathcal{S}$ (§ 5) consists of mnk/S elementary outer product calculations, arranged in $\sqrt{S} \times \sqrt{S} \times k$ "blocks" (Figure 4). The number $p_1 = mn/S$ of dependency-free subcomputations V_i (i.e., having no parents except for input vertices) in $\mathcal{S}$ determines the maximum degree of parallelism of $\mathcal{P}_{opt}$ for which no reuse set $V_{R,i}$ crosses two local domains $\mathcal{D}_j, \mathcal{D}_k$. The optimal schedule is parallelized in dimensions **ij**. There is no communication between the domains (except for inputs and outputs), and all I/O operations are performed inside each $\mathcal{D}_j$ following the sequential schedule. Each processor is assigned to p_1/p local domains $\mathcal{D}_j$ of size $\left[\sqrt{S}, \sqrt{S}, k\right]$, each of which requires $2\sqrt{S}k + S$ I/O operations (Theorem 1), giving a total of $Q = 2mnk/(p\sqrt{S}) + mn/p$ I/O operations per processor.

When $p > p_1$, the size of local domains $|\mathcal{D}_j|$ is smaller than $\sqrt{S} \times \sqrt{S} \times k$. Then, the schedule has to either be parallelized in dimension **k**, or has to reduce the size of the domain in **ij** plane. The former option creates dependencies between the local domains, which results in additional communication (Figure 4e). The latter does not utilize the whole available memory, making the sequential schedule not I/O optimal and decreasing the computational intensity ρ (Figure 4d). We now analyze three possible parallelization strategies (Figure 4) which generalize 2D, 2.5D, and recursive decomposition strategies; see Table 2 for details.

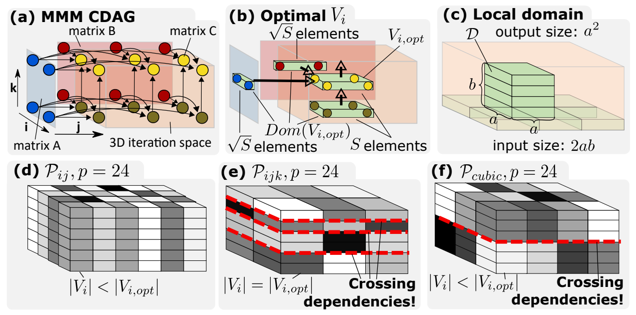


Figure 4: (a) An MMM CDAG as a 3D grid (iteration space). Each vertex in it (except for the vertices in the bottom layer) has three parents - blue (matrix A), red (matrix B), and yellow (partial result of matrix C) and one yellow child (except for vertices in the top layer). (b) A union of inputs of all vertices in V_i form the dominator set $Dom(V_i)$ (two blue, two red and four dark yellow). Using approximation $\sqrt{S+1}-1 \approx \sqrt{S}$, we have $|Dom(V_{i,opt})| = S$. (c) A local domain $\mathcal{D}$ consists of b subcomputations V_i, each of a dominator size $|Dom(V_i)| = a^2 + 2a$. (d-f) Different parallelization schemes of near optimal sequential MMM for $p = 24 > p_1 = 6$.

Schedule $\mathcal{P}_{ij}$ The schedule is parallelized in dimensions **i** and **j**. The processor grid is $\mathcal{G}_{ij} = \left[\frac{m}{a}, \frac{n}{a}, 1\right]$, where $a = \sqrt{\frac{mn}{p}}$. Because all dependencies are parallel to dimension **k**, there are no dependencies between $\mathcal{D}_j$ except for the inputs and the outputs. Because $a < \sqrt{S}$, the corresponding sequential schedule has a reduced computational intensity $\rho_{ij} < \sqrt{S}/2$.

Schedule $\mathcal{P}_{ijk}$ The schedule is parallelized in all dimensions. The processor grid is $\mathcal{G}_{ijk} = \left[\frac{m}{\sqrt{S}}, \frac{n}{\sqrt{S}}, \frac{k}{pS}\right]$. The computational intensity $\rho_{ijk} = \sqrt{S}/2$ is optimal. The parallelization in **k** dimension creates dependencies between local domains, requiring communication and increasing the I/O cost.

Schedule $\mathcal{P}_{cubic}$ The schedule is parallelized in all dimensions. The grid is $\left[\frac{m}{a_c}, \frac{n}{a_c}, \frac{k}{a_c}\right]$, where $a_c = \min\left\{\left(\frac{mnk}{p}\right)^{1/3}, \sqrt{\frac{S}{3}}\right\}$. Because $a_c < \sqrt{S}$, the corresponding computational intensity $\rho_{cubic} < \sqrt{S}/2$ is not optimal. The parallelization in **k** dimension creates dependencies between local domains, increasing communication.

Schedules of the State-of-the-Art Decompositions
If $m = n$, the $\mathcal{P}_{ij}$ scheme is reduced to the classical 2D decomposition (e.g., Cannon's algorithm [10]), and $\mathcal{P}_{ijk}$ is reduced to the 2.5D decomposition [52]. CARMA [22] asymptotically reaches the $\mathcal{P}_{cubic}$ scheme, guaranteeing that the longest dimension of a local cuboidal domain is at most two times larger than the smallest one. We present a detailed complexity analysis comparison for all algorithms in Table 2.

6.3 I/O Optimal Parallel Schedule

Observe that none of those schedules is optimal in the whole range of parameters. As discussed in § 5, in sequential scheduling, intermediate results of C are not stored to the memory: they are consumed (reused) immediately by the next sequential step. Only the final result of C in the local domain is sent. Therefore, the optimal parallel schedule $\mathcal{P}_{opt}$ minimizes the communication, that is, sum of the inputs' sizes plus the output size, under the sequential I/O constraint on subcomputations $\forall_{V_i \in \mathcal{D}_j \in \mathcal{P}_{opt}} |Dom(V_i)| \le S \land |Min(V_i)| \le S$.

The local domain $\mathcal{D}_j$ is a grid of size $[a \times a \times b]$, containing b outer products of vectors of length a. The optimization problem of finding $\mathcal{P}_{opt}$ using the computational intensity (Lemma 2) is formulated as follows:

$$\text{maximize } \rho = \frac{a^2 b}{ab + ab + a^2} \tag{3}$$
$$\text{subject to:}$$
$$a^2 \le S \text{ (the I/O constraint)}$$
$$a^2 b = \frac{mnk}{p} \text{ (the load balance constraint)}$$
$$pS \ge mn + mk + nk \text{ (matrices must fit into memory)}$$

The I/O constraint $a^2 \le S$ is binding (changes to equality) for $p \le \frac{mnk}{S^{3/2}}$. Therefore, the solution to this problem is:

$$a = \min\left\{\sqrt{S}, \left(\frac{mnk}{p}\right)^{1/3}\right\},\ b = \max\left\{\frac{mnk}{pS}, \left(\frac{mnk}{p}\right)^{1/3}\right\} \tag{4}$$

The I/O complexity of this schedule is:

$$Q \ge \frac{a^2 b}{\rho} = \min\left\{\frac{2mnk}{p\sqrt{S}} + S, 3\left(\frac{mnk}{p}\right)^{\frac{2}{3}}\right\} \tag{5}$$

This can be intuitively interpreted geometrically as follows: if we imagine the optimal local domain "growing" with the decreasing number of processors, then it stays cubic as long as it is still "small enough" (its side is smaller than $\sqrt{S}$). After that point, its face in the **ij** plane stays constant $\sqrt{S} \times \sqrt{S}$ and it "grows" only in the **k** dimension. This schedule effectively switches from $\mathcal{P}_{ijk}$ to $\mathcal{P}_{cubic}$ once there is enough memory ($S \ge (mnk/p)^{2/3}$).

Theorem 2. *The I/O complexity of a classic Matrix Multiplication algorithm executed on p processors, each of local memory size $S \ge \frac{mn+mk+nk}{p}$ is*

$$Q \ge \min\left\{\frac{2mnk}{p\sqrt{S}} + S, 3\left(\frac{mnk}{p}\right)^{\frac{2}{3}}\right\}$$

Proof. The theorem is a direct consequence of Lemma 1 and the computational intensity (Lemma 2). The load balance constraint enforces a size of each local domain $|\mathcal{D}_j| = mnk/p$. The I/O cost is then bounded by $|\mathcal{D}_j|/\rho$. Schedule $\mathcal{P}_{opt}$ maximizes ρ by the formulation of the optimization problem (Equation 3). □

I/O-Latency Trade-off
As showed in this section, the local domain $\mathcal{D}$ of the near optimal schedule $\mathcal{P}$ is a grid of size $[a \times a \times b]$, where a, b are given by Equation (4). The corresponding sequential schedule $\mathcal{S}$ is a sequence of b outer products of vectors of length a. Denote the size of the communicated inputs in each step by $I_{step} = 2a$. This corresponds to b steps of communication (the latency cost is $L = b$).

The number of steps (latency) is equal to the total communication volume of $\mathcal{D}$ divided by the volume per step $L = Q/I_{step}$. To reduce the latency, one either has to decrease Q or increase I_{step}, under the memory constraint that $I_{step} + a^2 \le S$ (otherwise we cannot fit both the inputs and the outputs in the memory). Express $I_{step} = a \cdot h$, where h is the number of sequential subcomputations V_i we merge in one communication. We can express the I/O-latency trade-off:

Decomposition **Parallel schedule** $\mathcal{P}$	**2D [55]** $\mathcal{P}_{ij}$ for $m = n$	**2.5D [52]** $\mathcal{P}_{ijk}$ for $m = n$	**recursive [22]** $\mathcal{P}_{cubic}$	**COSMA (this paper)** $\mathcal{P}_{opt}$
grid $[p_m \times p_n \times p_k]$	$[\sqrt{p} \times \sqrt{p} \times 1]$	$\left[\sqrt{p/c} \times \sqrt{p/c} \times c\right]$; $c = \frac{pS}{mk+nk}$	$[2^{a_1} \times 2^{a_2} \times 2^{a_3}]$; $a_1 + a_2 + a_3 = \log_2(p)$	$\left[\frac{m}{a} \times \frac{n}{a} \times \frac{k}{b}\right]$; a, b : Equation 4
domain size	$\left[\frac{m}{\sqrt{p}} \times \frac{n}{\sqrt{p}} \times k\right]$	$\left[\frac{m}{\sqrt{p/c}} \times \frac{n}{\sqrt{p/c}} \times \frac{k}{c}\right]$	$\left[\frac{m}{2^{a_1}} \times \frac{n}{2^{a_1}} \times \frac{k}{2^{a_1}}\right]$	$[a \times a \times b]$
		"General case":		
I/O cost Q	$\frac{k}{\sqrt{p}}(m+n) + \frac{mn}{p}$	$\frac{(k(m+n))^{3/2}}{p\sqrt{S}} + \frac{mnS}{k(m+n)}$	$2\min\left\{\sqrt{3}\frac{mnk}{p\sqrt{S}}, \left(\frac{mnk}{p}\right)^{2/3}\right\} + \left(\frac{mnk}{p}\right)^{2/3}$	$\min\left\{\frac{2mnk}{p\sqrt{S}} + S, 3\left(\frac{mnk}{p}\right)^{2/3}\right\}$
latency cost L	$2k\log_2(\sqrt{p})$	$\frac{(k(m+n))^{5/2}}{pS^{3/2}(km+kn-mn)} + 3\log_2\left(\frac{pS}{mk+nk}\right)$	$\left(3^{3/2}mnk\right)/\left(pS^{3/2}\right) + 3\log_2(p)$	$\frac{2ab}{S-a^2}\log_2\left(\frac{mn}{a^2}\right)$
		Square matrices, "limited memory": $m = n = k$, $S = 2n^2/p$, $p = 2^{3n}$		
I/O cost Q	$2n^2(\sqrt{p}+1)/p$	$2n^2(\sqrt{p}+1)/p$	$2n^2\left(\sqrt{3/2p} + 1/2p^{2/3}\right)$	$2n^2(\sqrt{p}+1)/p$
latency cost L	$2k\log_2(\sqrt{p})$	$\sqrt{p}$	$\left(\frac{3}{2}\right)^{3/2}\sqrt{p}\log_2(p)$	$\sqrt{p}\log_2(p)$
		"Tall" matrices, "extra" memory available: $m = n = \sqrt{p}$, $k = p^{3/2}/4$, $S = 2nk/p^{2/3}$, $p = 2^{3n+1}$		
I/O cost	$p^{3/2}/2$	$p^{4/3}/2 + p^{1/3}$	$3p/4$	$p\left(3 - 2^{1/3}\right)/2^{4/3} \approx 0.69p$
latency cost L	$p^{3/2}\log_2(\sqrt{p})/4$	1	1	1

Table 2: The comparison of complexities of 2D, 2.5D, recursive, and COSMA algorithms. The 3D decomposition is a special case of 2.5D, and can be obtained by instantiating $c = p^{1/3}$ in the 2.5D case. In addition to the general analysis, we show two special cases. If the matrices are square and there is no extra memory available, 2D, 2.5D and COSMA achieves tight communication lower bound $2n^2/\sqrt{p}$, whereas CARMA performs $\sqrt{3}$ times more communication. If one dimension is much larger than the others and there is extra memory available, 2D, 2.5D and CARMA decompositions perform $O(p^{1/2})$, $O(p^{1/3})$, and 8% more communication than COSMA, respectively. For simplicity, we assume that parameters are chosen such that all divisions have integer results.

$$\min(Q, L)$$
$$\text{subject to:}$$
$$Q = 2ab + a^2, L = \frac{b}{h}$$
$$a^2 + 2ah \le S \text{ (I/O constraint)}$$
$$a^2 b = \frac{mnk}{p} \text{ (load balance constraint)}$$

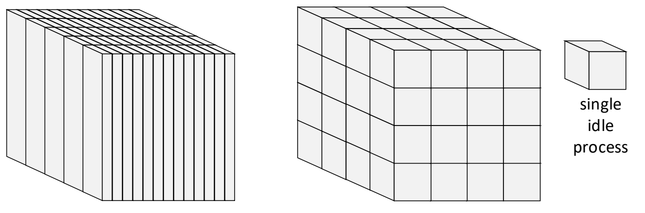
Solving this problem, we have $Q = \frac{2mnk}{pa} + a^2$ and $L = \frac{2mnk}{pa(S-a^2)}$, where $a \le \sqrt{S}$. Increasing a we *reduce* the I/O cost Q and *increase* the latency cost L. For minimal value of Q (Theorem 2), $L = \left\lceil \frac{2ab}{S-a^2} \right\rceil$, where $a = \min\{\sqrt{S}, (mnk/p)^{1/3}\}$ and $b = \max\{\frac{mnk}{pS}, (mnk/p)^{1/3}\}$. Based on our experiments, we observe that the I/O cost is vastly greater than the latency cost, therefore our schedule by default minimizes Q and uses extra memory (if any) to reduce L.

7 IMPLEMENTATION

We now present implementation optimizations that further increase the performance of COSMA on top of the speedup due to our near I/O optimal schedule. The algorithm is designed to facilitate the overlap of communication and computation § 7.3. For this, to leverage the RDMA mechanisms of current high-speed network interfaces, we use the MPI one-sided interface § 7.4. In addition, our implementation also offers alternative efficient two-sided communication back end that uses MPI collectives. We also use a blocked data layout § 7.6, a grid-fitting technique § 7.1, and an optimized binary broadcast tree using static information about the communication pattern (§ 7.2) together with the buffer swapping (§ 7.5). For the local matrix operations, we use BLAS routines for highest performance. Our code is publicly available at https://github.com/eth-cscs/COSMA.

7.1 Processor Grid Optimization

Throughout the paper, we assume all operations required to assess the decomposition (divisions, roots) result in natural numbers. We note that in practice it is rarely the case, as the parameters usually emerge from external constraints, like a specification of a performed calculation or hardware resources (§ 8). If matrix dimensions are not divisible by the local domain sizes a, b (Equation 4), then a straightforward option is to use the floor function, not utilizing the "boundary" processors whose local domains do not fit entirely in the iteration space, which result in more computation per processor. The other option is to find factors of p and then construct the processor grid by matching the largest factors with largest matrix dimensions. However, if the factors of p do not match m, n, and k, this may result in a suboptimal decomposition. Our algorithm allows to not utilize some processors (increasing the computation volume per processor) to optimize the grid, which reduces the communication volume. Figure 5 illustrates the comparison between these options. We balance this communication–computation trade-off by "stretching" the local domain size derived in § 6.3 to fit the global domain by adjusting its width, height, and length. The range of this tuning (how many processors we drop to reduce communication) depends on the hardware specification of the machine (peak flop/s, memory and network bandwidth). For our experiments on the Piz Daint machine, we chose the maximal number of unutilized cores to be 3%, accounting for up to 2.4 times speedup for the square matrices using 2,198 cores (§ 9).



(a) $1 \times 5 \times 13$ **grid** **(b)** $4 \times 4 \times 4$ **grid with one idle processor**

Figure 5: Processor decomposition for square matrices and 65 processors. (a) To utilize all resources, the local domain is drastically stretched. (b) Dropping one processor results in a symmetric grid which increases the computation per processor by 1.5%, but reduces the communication by 36%.

7.2 Enhanced Communication Pattern

As shown in Algorithm 1, COSMA by default executes in $t = \frac{2ab}{S-a^2}$ rounds. In each round, each processor receives $s = ab/t = (S-a^2)/2$ elements of A and B. Thus, the input matrices are broadcast among the **i** and **j** dimensions of the processor grid. After the last round, the partial results of C are reduced among the **k** dimension. The communication pattern is therefore similar to ScaLAPACK or CTF.

To accelerate the collective communication, we implement our own binary broadcast tree, taking advantage of the known data layout, processor grid, and communication pattern. Knowing the initial data layout § 7.6 and the processor grid § 7.1, we craft the binary reduction tree in all three dimensions **i**, **j**, and **k** such that the distance in the grid between communicating processors is minimized. Our implementation outperforms the standard MPI broadcast from the Cray-MPICH 3.1 library by approximately 10%.

7.3 Communication–Computation Overlap

The sequential rounds of the algorithm $t_i = 1, \ldots, t$, naturally express communication–computation overlap. Using double buffering, at each round t_i we issue an asynchronous communication (using either MPI_Get or MPI_Isend / MPI_Irecv § 7.4) of the data required at round t_{i+1}, while locally processing the data received in a previous round. We note that, by the construction of the local domains $\mathcal{D}_j$ § 6.3, the extra memory required for double buffering is rarely an issue. If we are constrained by the available memory, then the space required to hold the partial results of C, which is a^2, is much larger than the size of the receive buffers $s = (S - a^2)/2$. If not, then there is extra memory available for the buffering.

Number of rounds

The minimum number of rounds, and therefore latency, is $t = \frac{2ab}{S-a^2}$ (§ 6.3) . However, to exploit more overlap, we can increase the number of rounds $t_2 > t$. In this way, in one round we communicate less data $s_2 = ab/t_2 < s$, allowing the first round of computation to start earlier.

7.4 One-Sided vs Two-Sided Communication

To reduce the latency [27] we implemented communication using MPI RMA [32]. This interface utilizes the underlying features of Remote Direct Memory Access (RDMA) mechanism, bypassing the OS on the sender side and providing zero-copy communication: data sent is not buffered in a temporary address, instead, it is written directly to its location.

All communication windows are pre-allocated using MPI_Win_allocate with the size of maximum message in the broadcast tree $2^{s-1}D$ (§ 7.2). Communication in each step is performed using the MPI_Get and MPI_Accumulate routines.

For compatibility reasons, as well as for the performance comparison, we also implemented a communication back-end using MPI two-sided (the message passing abstraction).

7.5 Communication Buffer Optimization

The binary broadcast tree pattern is a generalization of the recursive structure of CARMA. However, CARMA in each recursive step dynamically allocates new buffers of the increasing size to match the message sizes $2^{s-1}D$, causing an additional runtime overhead.

To alleviate this problem, we pre-allocate initial, send, and receive buffers for each of matrices A, B, and C of the maximum size of the message ab/t, where $t = \frac{2ab}{S-a^2}$ is the number of steps in COSMA (Algorithm 1). Then, in each level s of the communication tree, we move the pointer in the receive buffer by $2^{s-1}D$ elements.

7.6 Blocked Data Layout

COSMA's schedule induces the optimal initial data layout, since for each $\mathcal{D}_j$ it determines its dominator set $Dom(\mathcal{D}_j)$, that is, elements accessed by processor j. Denote $A_{l,j}$ and $B_{l,j}$ subsets of elements of matrices A and B that initially reside in the local memory of processor j. The optimal data layout therefore requires that $A_{l,j}, B_{l,j} \subset Dom(\mathcal{D}_j)$. However, the schedule does not specify exactly which elements of $Dom(\mathcal{D}_j)$ should be in $A_{l,j}$ and $B_{l,j}$. As a consequence of the communication pattern § 7.2, each element of $A_{l,j}$ and $B_{l,j}$ is communicated to g_m, g_n processors, respectively. To prevent data reshuffling, we therefore split each of $Dom(\mathcal{D}_j)$ into g_m and g_n smaller blocks, enforcing that consecutive blocks are assigned to processors that communicate first. This is unlike the distributed CARMA implementation [22], which uses the cyclic distribution among processors in the recursion base case and requires local data reshuffling after each communication round. Another advantage of our blocked data layout is a full compatibility with the block-cyclic one, which is used in other linear-algebra libraries.

8 EVALUATION

We evaluate COSMA's communication volume and performance against other state-of-the-art implementations with various combinations of matrix dimensions and memory requirements. These scenarios include both synthetic square matrices, in which all algorithms achieve their peak performance, as well as "flat" (two large dimensions) and real-world "tall-and-skinny" (one large dimension) cases with uneven number of processors.

Comparison Targets

As a comparison, we use the widely used ScaLAPACK library as provided by Intel MKL (version: 18.0.2.199)[6], as well as Cyclops Tensor Framework[7], and the original CARMA implementation[8]. *We manually tune ScaLAPACK parameters to achieve its maximum performance.* Our experiments showed that on Piz Daint it achieves the highest performance when run with 4 MPI ranks per compute node, 9 cores per rank. Therefore, for each matrix sizes/node count configuration, we recompute the optimal rank decomposition for ScaLAPACK. Remaining implementations use default decomposition strategy and perform best utilizing 36 ranks per node, 1 core per rank.

Infrastructure and Implementation Details

All implementations were compiled using the GCC 6.2.0 compiler. We use Cray-MPICH 3.1 implementation of MPI. The parallelism within a rank of ScaLAPACK[9] is handled internally by the MKL BLAS (with GNU OpenMP threading) version 2017.4.196. To profile MPI communication volume, we use the mpiP version 3.4.1 [56].

[6]the latest version available on Piz Daint when benchmarks were performed (August 2018). No improvements of P[S,D,C,Z]GEMM have been reported in the MKL release notes since then.
[7]https://github.com/cyclops-community/ctf, commit ID 244561c on May 15, 2018
[8]https://github.com/lipshitz/CAPS, commit ID 7589212 on July 19, 2013
[9]only ScaLAPACK uses multiple cores per ranks

Experimental Setup and Architectures
We run our experiments on the CPU partition of CSCS Piz Daint, which has 1,813 XC40 nodes with dual-socket Intel Xeon E5-2695 v4 processors ($2 \cdot 18$ cores, 3.30 GHz, 45 MiB L3 shared cache, 64 GiB DDR3 RAM), interconnected by the Cray Aries network with a dragonfly network topology. We set p to a number of available cores[10] and S to the main memory size per core (§ 2.1). To additionally capture cache size per core, the model can be extended to a three-level memory hierarchy. However, cache-size tiling is already handled internally by the MKL.

Matrix Dimensions and Number of Cores
We use square ($m = n = k$), "largeK" ($m = n \ll k$), "largeM" ($m \gg n = k$), and "flat" ($m = n \gg k$) matrices. The matrix dimensions and number of cores are (1) powers of two $m = 2^{r_1}, n = 2^{r_2}, m = 2^{r_3}$, (2) determined by the real-life simulations or hardware architecture (available nodes on a computer), (3) chosen adversarially, e.g, $n^3 + 1$. Tall matrix dimensions are taken from an application benchmark, namely the calculation of the random phase approximation (RPA) energy of water molecules [21]. There, to simulate w molecules, the sizes of the matrices are $m = n = 136w$ and $k = 228w^2$. In the strong scaling scenario, we use $w = 128$ as in the original paper, yielding $m = n = 17{,}408$, $k = 3{,}735{,}552$. For performance runs, we scale up to 512 nodes (18,432 cores).

Selection of Benchmarks
We perform both strong scaling and *memory scaling* experiments. The memory scaling scenario fixes the input size per core ($\frac{pS}{I}, I = mn + mk + nk$), as opposed to the work per core ($\frac{mnk}{p} \neq const$). We evaluate two cases: (1) "limited memory" ($\frac{pS}{I} = const$), and (2) "extra memory" ($\frac{p^{2/3}S}{I} = const$).

To provide more information about the impact of communication optimizations on the total runtime, for each of the matrix shapes we also separately measure time spent by COSMA on different parts of the code. for each matrix shape we present two extreme cases of strong scaling - with smallest number of processors (most compute-intense) and with the largest (most communication-intense). To additionally increase information provided, we perform these measurements with and without communication–computation overlap.

Programming Models
We use either the RMA or the Message Passing models. CTF also uses both models, whereas CARMA and ScaLAPACK use MPI two-sided (Message Passing).

Experimentation Methodology
For each combination of parameters, we perform 5 runs, each with different node allocation. As all the algorithms use BLAS routines for local matrix computations, for each run we execute the kernels three times and take the minimum to compensate for the BLAS setup overhead. We report median and 95% confidence intervals of the runtimes.

9 RESULTS
We now present the experimental results comparing COSMA with the existing algorithms. For both strong and memory scaling, we measure total communication volume and runtime on both square and tall matrices. Our experiments show that COSMA always communicates least data and is the fastest in *all* scenarios.

Summary and Overall Speedups
As discussed in § 8, we evaluate three benchmarks – strong scaling, "limited memory" (no redundant copies of the input are possible), and "extra memory" ($p^{1/3}$ extra copies of the input can fit into combined memory of all cores). Each of them we test for square, "largeK", "largeM", and , "flat" matrices, giving twelve cases in total. In Table 3, we present arithmetic mean of total communication volume per MPI rank across all core counts. We also report the summary of minimum, geometric mean, and maximum speedups vs the second best-performing algorithm.

Communication Volume
As analyzed in § 5 and § 6, COSMA reaches I/O lower bound (up to the factor of $\sqrt{S}/(\sqrt{S+1} - 1)$). Moreover, optimizations presented in § 7 secure further improvements compared to other state-of-the-art algorithms. In all cases, COSMA performs least communication. Total communication volume for square and "largeK" scenarios is shown in Figures 6 and 9.

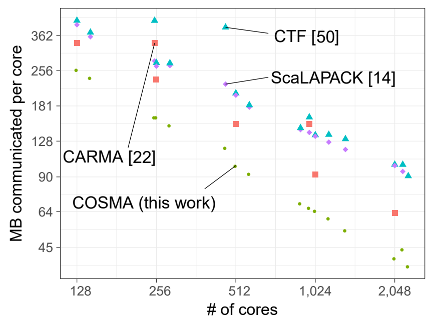
Square Matrices
Figure 8 presents the % of achieved peak hardware performance for square matrices in all three scenarios. As COSMA is based on the near optimal schedule, it achieves the highest performance *in all cases*. Moreover, its performance pattern is the most stable: when the number of cores is not a power of two, the performance does not vary much compared to all remaining three implementations. We note that matrix dimensions in the strong scaling scenarios ($m = n = k = 2^{14}$) are very small for distributed setting. Yet even in this case COSMA maintains relatively high performance for large numbers of cores: using 4k cores it achieves 35% of peak performance, compared to <5% of CTF and ScaLAPACK, showing excellent strong scaling characteristics.

Tall and Skinny Matrices
Figure 9 presents the results for "largeK" matrices - due to space constraints, the symmetric "largeM" case is For strong scaling, the minimum number of cores is 2048 (otherwise, the matrices of size $m = n = 17{,}408$, $k = 3{,}735{,}552$ do not fit into memory). Again, COSMA shows the most stable performance with a varying number of cores.

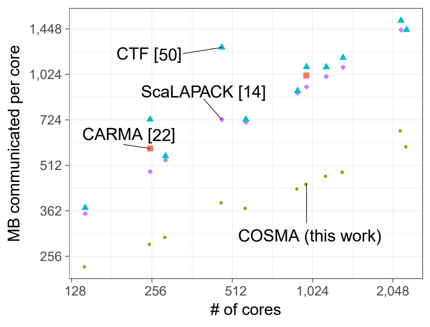
"Flat" Matrices
Matrix dimensions for strong scaling are set to $m = n = 2^{17} = 131{,}072$ and $k = 2^9 = 512$. Our weak scaling scenario models the rank-k update kernel, with fixed $k = 256$, and $m = n$ scaling accordingly for the "limited" and "extra" memory cases. Such kernels take most of the execution time in, e.g., matrix factorization algorithms, where updating Schur complements is performed as a rank-k gemm operation [31].

Unfavorable Number of Processors
Due to the processor grid optimization (§ 7.1), the performance is stable and does not suffer from unfavorable combinations of parameters. E.g., the runtime of COSMA for square matrices $m = n = k = 16{,}384$ on $p_1 = 9{,}216 = 2^{10} \cdot 3^2$ cores is 142 ms. Adding an extra core ($p_2 = 9{,}217 = 13 \cdot 709$), does not change COSMA's runtime, as the optimal decomposition does not utilize it. On the other hand, CTF for p_1 runs in 600 ms, while for p_2 the runtime *increases* to 1613 ms due to a non-optimal processor decomposition.

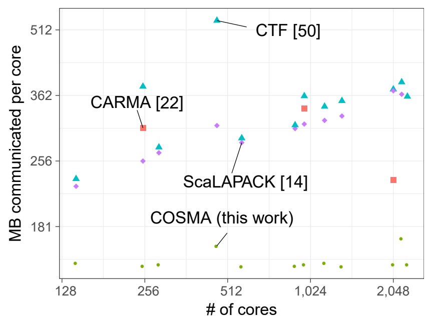
[10]for ScaLAPACK, actual number of MPI ranks is $p/9$



(a) Strong scaling, $n = m = k = 16{,}384$

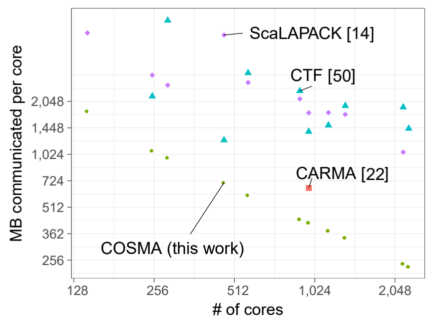


(b) Limited memory, $n = m = k = \sqrt{\frac{pS}{3}}$

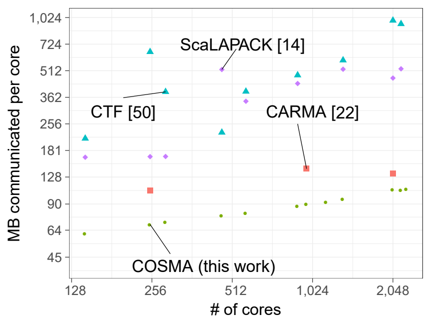


(c) Extra memory,$n = m = k = \sqrt{\frac{p^{2/3}S}{3}}$

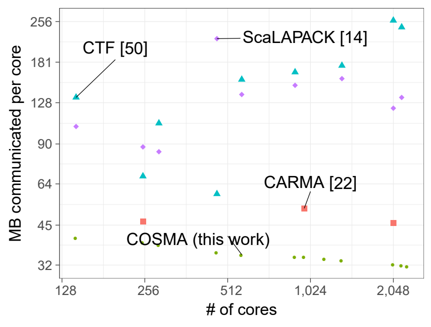
Figure 6: Total communication volume per core carried out by COSMA, CTF, ScaLAPACK and CARMA for square matrices, as measured by the mpiP profiler.



(a) Strong scaling, $n = m = 17{,}408$, $k = 3{,}735{,}552$



(b) Limited memory,$m = n = 979p^{\frac{1}{3}}$, $k = 1.184p^{\frac{2}{3}}$



(c) Extra memory,,$m = n = 979p^{\frac{2}{9}}$, $k = 1.184p^{\frac{4}{9}}$

Figure 7: Total communication volume per core carried out by COSMA, CTF, ScaLAPACK and CARMA for "largeK" matrices, as measured by the mpiP profiler.

Communication-Computation Breakdown

In Figure 10 we present the total runtime breakdown of COSMA into communication and computation routines. Combined with the comparison of communication volumes (Figures 6 and 7, Table 3) we see the importance of our I/O optimizations for distributed setting even for traditionally compute-bound MMM. E.g., for square or "flat" matrix and 16k cores, COSMA communicates more than two times less than the second-best (CARMA). Assuming constant time-per-MB, COSMA would be 40% slower if it communicated that much, being slower than CARMA by 30%. For "largeK", the situation is even more extreme, with COSMA suffering 2.3 times slowdown if communicating as much as the second-best algorithm, CTF, which communicates 10 times more.

Detailed Statistical Analysis

Figure 11 provides a distribution of the achieved peak performance across all numbers of cores for all six scenarios. It can be seen that, for example, in the strong scaling scenario and square matrices, COSMA is comparable to the other implementations (especially CARMA). However, for tall-and-skinny matrices with limited memory available, *COSMA lowest achieved performance is higher than the best performance of CTF and ScaLAPACK.*

10 RELATED WORK

Works on data movement minimization may be divided into two categories: applicable across memory hierarchy (vertical, also called I/O minimization), or between parallel processors (horizontal, also called communication minimization). Even though they are "two sides of the same coin", in literature they are often treated as separate topics. In our work we combine them: analyze trade–offs between communication optimal (distributed memory) and I/O optimal schedule (shared memory).

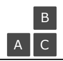
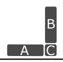
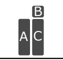
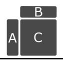
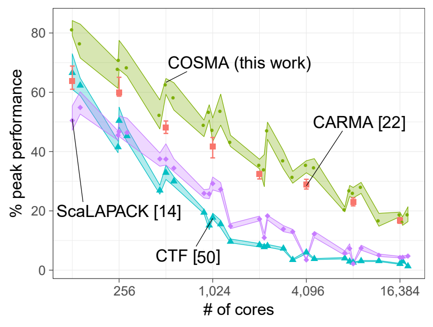
		total comm. volume per rank [MB]				speedup		
shape	**benchmark**	**ScaLAPACK**	**CTF**	**CARMA**	**COSMA**	**min**	**mean**	**max**
	strong scaling limited memory extra memory	203 816 303	222 986 350	195 799 291	107 424 151	**1.07** 1.23 1.14	1.94 1.71 2.03	4.81 2.99 4.73
	strong scaling limited memory extra memory	2636 368 133	2278 541 152	659 128 48	545 88 35	1.24 1.30 1.31	2.00 2.61 2.55	6.55 8.26 6.70
	strong scaling limited memory extra memory	3507 989 122	2024 672 77	541 399 77	410 194 29	1.31 1.42 1.35	2.22 1.7 1.76	3.22 2.27 2.8
	strong scaling limited memory extra memory	134 47 15	68 101 15	10 26 10	7 8 3	1.21 1.31 1.5	4.02 2.07 2.29	**12.81** 3.41 3.59
	overall					**1.07**	**2.17**	**12.81**

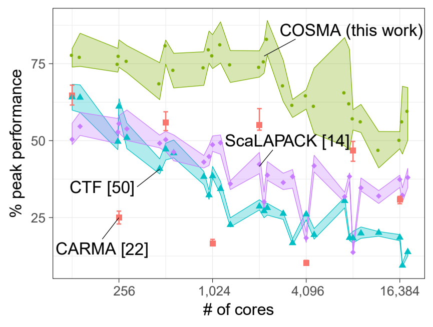
Table 3: Average communication volume per MPI rank and measured speedup of COSMA vs the second-best algorithm across all core counts for each of the scenarios.

10.1 General I/O Lower Bounds

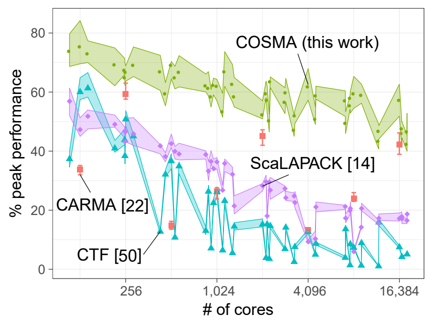
Hong and Kung [34] analyzed the I/O complexity for general CDAGs in their the red-blue pebble game, on which we base our work. As a special case, they derived an asymptotic bound $\Omega\left(n^3/\sqrt{S}\right)$ for MMM. Elango et al. [23] extended this work to the red-blue-white game and Liu and Terman [40] proved that it is also P-SPACE complete. Irony et al. [33] extended the MMM lower bound result to a parallel machine with p processors, each having a fast private



(a) Strong scaling, $n = m = k = 16{,}384$

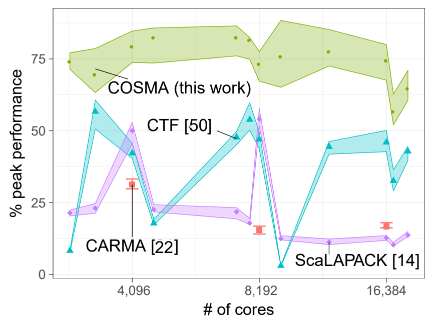


(b) Limited memory, $n = m = k = \sqrt{\frac{pS}{3}}$

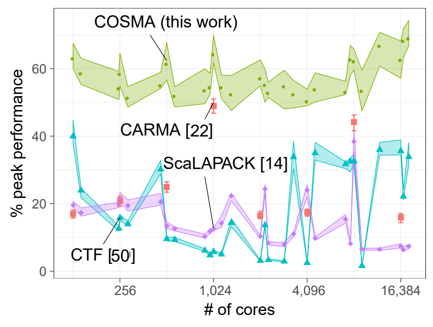


(c) Extra memory, $m = n = k = \sqrt{\frac{p^{2/3}S}{3}}$

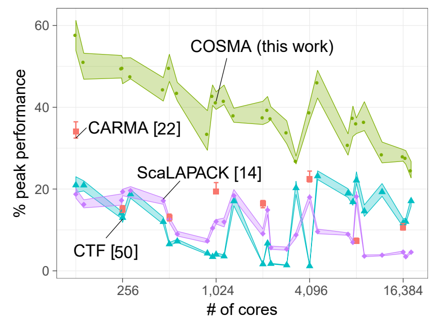
Figure 8: Achieved % of peak performance by COSMA, CTF, ScaLAPACK and CARMA for square matrices, strong and weak scaling. We show median and 95% confidence intervals.



(a) Strong scaling, $n = m = 17{,}408$, $k = 3{,}735{,}552$



(b) Limited memory, $m = n = 979p^{\frac{1}{3}}$, $k = 1.184p^{\frac{2}{3}}$



(c) Extra memory, $m = n = 979p^{\frac{2}{9}}$, $k = 1.184p^{\frac{4}{9}}$

Figure 9: Achieved % of peak performance by COSMA, CTF, ScaLAPACK and CARMA for "largeK" matrices, strong and weak scaling. We show median and 95% confidence intervals.

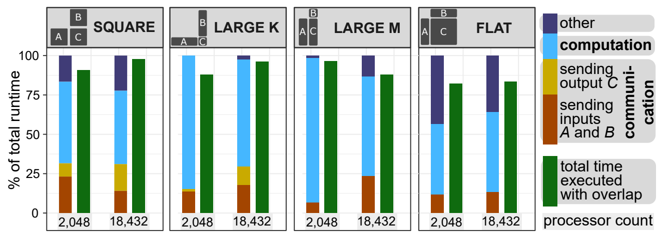


Figure 10: Time distribution of COSMA communication and computation kernels for strong scaling executed on the smallest and the largest core counts for each of the matrix shapes. Left bar: no communication–computation overlap. Right bar: overlap enabled.

memory of size S, proving the $\frac{n^3}{2\sqrt{2p}\sqrt{S}} - S$ lower bound on the communication volume per processor. Chan [12] studied different variants of pebble games in the context of memory space and parallel time. Aggarwal and Vitter [2] introduced a two-memory machine that models a blocked access and latency in an external storage. Arge et al. [3] extended this model to a parallel machine. Solomonik et al. [50] combined the communication, synchronization, and computation in their general cost model and applied it to several linear algebra algorithms. Smith and van de Geijn [47] derived a sequential lower bound $2mnk/\sqrt{S} - 2S$ for MMM. They showed that the leading factor $2mnk/\sqrt{S}$ is tight. We improve this result by 1) improving an additive factor of $2S$, but more importantly 2) generalizing the bound to a parallel machine. Our work uses a simplified model, not taking into account the memory block size, as in the external memory model, nor the cost of computation. We motivate it by assuming that the block size is significantly smaller than the input size, the data is layout contiguously in the memory, and that the computation is evenly distributed among processors.

10.2 Shared Memory Optimizations

I/O optimization for linear algebra includes such techniques as loop tiling and skewing [58], interchanging and reversal [57]. For programs with multiple loop nests, Kennedy and McKinley [35] showed various techniques for loop fusion and proved that in general this problem is NP-hard. Later, Darte [20] identified cases when this problem has polynomial complexity.

Toledo [54] in his survey on Out-Of-Core (OOC) algorithms analyzed various I/O minimizing techniques for dense and sparse matrices. Mohanty [42] in his thesis optimized several OOC algorithms. Irony et al. [33] proved the I/O lower bound of classical MMM on a parallel machine. Ballard et al. [5] proved analogous results for Strassen's algorithm. This analysis was extended by Scott et al. [45] to a general class of Strassen-like algorithms.

Although we consider only dense matrices, there is an extensive literature on sparse matrix I/O optimizations. Bender et al. [7] extended Aggarwal's external memory model [2] and showed I/O complexity of the sparse matrix-vector (SpMV) multiplication. Greiner [29] extended those results and provided I/O complexities of other sparse computations.

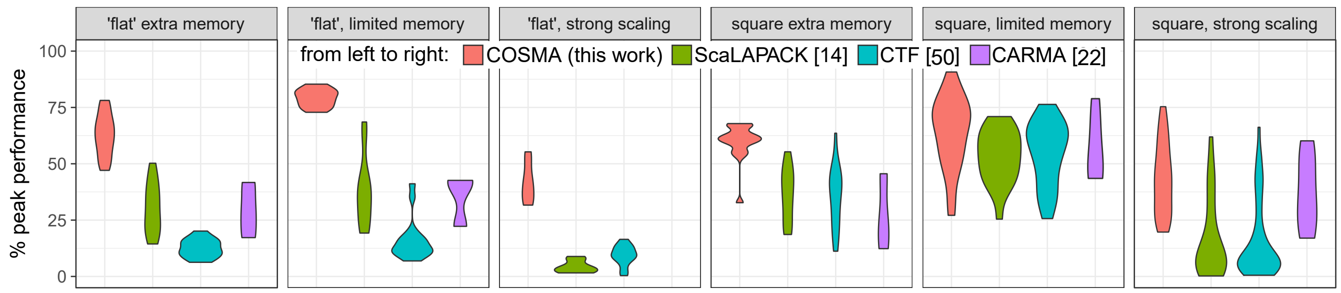


Figure 11: Distribution of achieved % of peak performance of the algorithms across all number of cores for "flat" and square matrices.

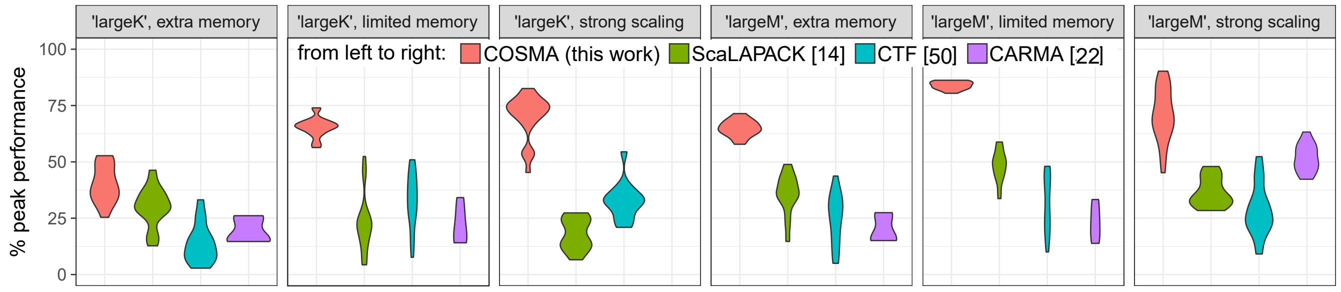


Figure 12: Distribution of achieved % of peak performance of the algorithms across all number of cores for tall-and-skinny matrices.

10.3 Distributed Memory Optimizations

Distributed algorithms for dense matrix multiplication date back to the work of Cannon [10], which has been analyzed and extended many times [30] [39]. In the presence of extra memory, Aggarwal et al. [1] included parallelization in the third dimension. Solomonik and Demmel [52] extended this scheme with their 2.5D decomposition to arbitrary range of the available memory, effectively interpolating between Cannon's 2D and Aggarwal's 3D scheme. A recursive, memory-oblivious MMM algorithm was introduced by Blumofe et al. [9] and extended to rectangular matrices by Frigo et al. [26]. Demmel el al. [22] introduced CARMA algorithm which achieves the asymptotic complexity for all matrix and memory sizes. We compare COSMA with these algorithms, showing that we achieve better results both in terms of communication complexity and the actual runtime performance. Lazzaro et al. [38] used the 2.5D technique for sparse matrices, both for square and rectangular grids. Koanantakool et al. [37] observed that for sparse-dense MMM, 1.5D decomposition performs less communication than 2D and 2.5D schemes, as it distributes only the sparse matrix.

11 CONCLUSIONS

In this work we present a new method (Lemma 1) for assessing tight I/O lower bounds of algorithms using their CDAG representation and the red-blue pebble game abstraction. As a use case, we prove a tight bound for MMM, both for a sequential (Theorem 1) and parallel (Theorem 2) execution. Furthermore, our proofs are constructive: our COSMA algorithm is near I/O optimal (up to the factor of $\frac{\sqrt{S}}{\sqrt{S+1}-1}$, which is less than 0.04% from the lower bound for 10MB of fast memory) for any combination of matrix dimensions, number of processors and memory sizes. This is in contrast with the current state-of-the-art algorithms, which are communication-inefficient in some scenarios.

To further increase the performance, we introduce a series of optimizations, both on an algorithmic level (processor grid optimization (§ 7.1) and blocked data layout (§ 7.6)) and hardware-related (enhanced communication pattern (§ 7.2), communication–computation overlap (§ 7.3), one-sided (§ 7.4) communication). The experiments confirm the superiority of COSMA over the other analyzed algorithms - our algorithm significantly reduces communication in all tested scenarios, supporting our theoretical analysis. Most importantly, our work is of practical importance, being maintained as an open-source implementation and achieving a time-to-solution speedup of up to 12.8x times compared to highly optimized state-of-the-art libraries.

The important feature of our method is that it does not require any manual parameter tuning and is generalizable to other machine models (e.g., multiple levels of memory) and linear algebra kernels (e.g., LU or Cholesky decompositions), both for dense and sparse matrices. We believe that the "bottom-up" approach will lead to developing more efficient distributed algorithms in the future.

Acknowledgements

We thank Yishai Oltchik and Niels Gleinig for invaluable help with the theoretical part of the paper, and Simon Pintarelli for advice and support with the implementation. We also thank CSCS for the compute hours needed to conduct all experiments. This project has received funding from the European Research Council (ERC) under the European Union's Horizon2020 programme (grant agreement DAPP, No.678880), and additional funding from the Platform for Advanced Scientific Computing (PASC).

REFERENCES

[1] R. C. Agarwal et al. 1995. A three-dimensional approach to parallel matrix multiplication. *IBM J. Res. Dev.* (1995).

[2] Alok Aggarwal and S. Vitter, Jeffrey. [n. d.]. The Input/Output Complexity of Sorting and Related Problems. *CACM* (Sept. [n. d.]).

[3] Lars Arge et al. 2008. In *SPAA*.

[4] Ariful Azad et al. 2015. Parallel triangle counting and enumeration using matrix algebra. In *IPDPSW*.

[5] Grey Ballard et al. 2012. Graph expansion and communication costs of fast matrix multiplication. *JACM* (2012).

[6] Tal Ben-Nun and Torsten Hoefler. 2018. Demystifying Parallel and Distributed Deep Learning: An In-Depth Concurrency Analysis. *CoRR* abs/1802.09941 (2018).

[7] Michael A Bender et al. 2010. Optimal sparse matrix dense vector multiplication in the I/O-model. *TOCS* (2010).

[8] Maciej Besta et al. 2017. SlimSell: A Vectorizable Graph Representation for Breadth-First Search. In *IPDPS*.

[9] Robert D Blumofe et al. 1996. An analysis of dag-consistent distributed shared-memory algorithms. In *SPAA*.

[10] Lynn Elliot Cannon. 1969. *A Cellular Computer to Implement the Kalman Filter Algorithm.* Ph.D. Dissertation.

[11] Gregory J. Chaitin et al. 1981. Register allocation via coloring. *Computer Languages* (1981).

[12] S. M. Chan. 2013. Just a Pebble Game. In *CCC*.

[13] Franàoise Chatelin. 2012. *Eigenvalues of Matrices: Revised Edition.* Siam.

[14] Jaeyoung Choi et al. 1992. ScaLAPACK: A scalable linear algebra library for distributed memory concurrent computers. In *FRONTIERS*.

[15] Jaeyoung Choi et al. 1996. Design and Implementation of the ScaLAPACK LU, QR, and Cholesky Factorization Routines. *Sci. Program.* (1996).

[16] J. Choi et al. 1996. ScaLAPACK: a portable linear algebra library for distributed memory computers — design issues and performance. *Comp. Phys. Comm.* (1996).

[17] Jaeyoung Choi, David W Walker, and Jack J Dongarra. 1994. PUMMA: Parallel universal matrix multiplication algorithms on distributed memory concurrent computers. *Concurrency: Practice and Experience* 6, 7 (1994), 543–570.

[18] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. 2009. *Introduction to algorithms.* MIT press.

[19] Paolo D'Alberto and Alexandru Nicolau. 2008. Using recursion to boost ATLAS's performance. In *High-Performance Computing*. Springer.

[20] Alain Darte. 1999. On the complexity of loop fusion. In *PACT*.

[21] Mauro Del Ben et al. 2015. Enabling simulation at the fifth rung of DFT: Large scale RPA calculations with excellent time to solution. *Comp. Phys. Comm.* (2015).

[22] J. Demmel et al. 2013. Communication-Optimal Parallel Recursive Rectangular Matrix Multiplication. In *IPDPS*.

[23] V. Elango et al. 2013. *Data access complexity: The red/blue pebble game revisited.* Technical Report.

[24] Geoffrey C Fox. 1988. Solving problems on concurrent processors. (1988).

[25] Geoffrey C Fox, Steve W Otto, and Anthony JG Hey. 1987. Matrix algorithms on a hypercube I: Matrix multiplication. *Parallel computing* 4, 1 (1987), 17–31.

[26] M. Frigo et al. 1999. Cache-oblivious algorithms. In *FOCS*.

[27] R. Gerstenberger et al. 2013. Enabling Highly-Scalable Remote Memory Access Programming with MPI-3 One Sided. In *SC*.

[28] John R. Gilbert et al. 1979. The Pebbling Problem is Complete in Polynomial Space. In *STOC*.

[29] Gero Greiner. 2012. *Sparse matrix computations and their I/O complexity.* Ph.D. Dissertation. Technische Universität München.

[30] A. Gupta and V. Kumar. 1993. Scalability of Parallel Algorithms for Matrix Multiplication. In *ICPP*.

[31] Azzam Haidar, Stanimire Tomov, Jack Dongarra, and Nicholas J Higham. 2018. Harnessing GPU tensor cores for fast FP16 arithmetic to speed up mixed-precision iterative refinement solvers. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis.* IEEE Press, 47.

[32] T. Hoefler et al. 2015. Remote Memory Access Programming in MPI-3. *TOPC* (2015).

[33] Dror Irony et al. 2004. Communication Lower Bounds for Distributed-memory Matrix Multiplication. *JPDC* (2004).

[34] Hong Jia-Wei and Hsiang-Tsung Kung. 1981. I/O complexity: The red-blue pebble game. In *STOC*.

[35] Ken Kennedy and Kathryn S McKinley. 1993. Maximizing loop parallelism and improving data locality via loop fusion and distribution. In *LCPC*.

[36] Jeremy Kepner et al. 2016. Mathematical foundations of the GraphBLAS. *arXiv:1606.05790* (2016).

[37] Penporn Koanantakool et al. 2016. Communication-avoiding parallel sparse-dense matrix-matrix multiplication. In *IPDPS*.

[38] Alfio Lazzaro et al. 2017. Increasing the efficiency of sparse matrix-matrix multiplication with a 2.5 D algorithm and one-sided MPI. In *PASC*.

[39] Hyuk-Jae Lee et al. 1997. Generalized Cannon's Algorithm for Parallel Matrix Multiplication. In *ICS*.

[40] Quanquan Liu. 2018. Red-Blue and Standard Pebble Games : Complexity and Applications in the Sequential and Parallel Models.

[41] Carl D Meyer. 2000. *Matrix analysis and applied linear algebra.* SIAM.

[42] Sraban Kumar Mohanty. 2010. I/O Efficient Algorithms for Matrix Computations. *CoRR* abs/1006.1307 (2010).

[43] Andrew Y Ng et al. 2002. On spectral clustering: Analysis and an algorithm. In *NIPS*.

[44] Donald W. Rogers. 2003. *Computational Chemistry Using the PC.* John Wiley & Sons, Inc.

[45] Jacob Scott et al. 2015. Matrix multiplication I/O-complexity by path routing. In *SPAA*.

[46] Ravi Sethi. 1973. Complete Register Allocation Problems. In *STOC*.

[47] Tyler Michael Smith and Robert A. van de Geijn. 2017. Pushing the Bounds for Matrix-Matrix Multiplication. *CoRR* abs/1702.02017 (2017).

[48] Raffaele Solcà, Anton Kozhevnikov, Azzam Haidar, Stanimire Tomov, Jack Dongarra, and Thomas C. Schulthess. 2015. Efficient Implementation of Quantum Materials Simulations on Distributed CPU-GPU Systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '15).* ACM, New York, NY, USA, Article 10, 12 pages. https://doi.org/10.1145/2807591.2807654

[49] Edgar Solomonik et al. 2013. Cyclops tensor framework: Reducing communication and eliminating load imbalance in massively parallel contractions. In *IPDPS*.

[50] Edgar Solomonik et al. 2016. Trade-Offs Between Synchronization, Communication, and Computation in Parallel Linear Algebra omputations. *TOPC* (2016).

[51] E. Solomonik et al. 2017. Scaling Betweenness Centrality using Communication-Efficient Sparse Matrix Multiplication. In *SC*.

[52] Edgar Solomonik and James Demmel. 2011. Communication-Optimal Parallel 2.5D Matrix Multiplication and LU Factorization Algorithms. In *EuroPar*.

[53] Volker Strassen. 1969. Gaussian Elimination is Not Optimal. *Numer. Math.* (1969).

[54] Sivan Toledo. 1999. A survey of out-of-core algorithms in numerical linear algebra. *External Memory Algorithms and Visualization* (1999).

[55] Robert A Van De Geijn and Jerrell Watts. 1997. SUMMA: Scalable universal matrix multiplication algorithm. *Concurrency: Practice and Experience* 9, 4 (1997), 255–274.

[56] Jeffrey S Vetter and Michael O McCracken. 2001. Statistical scalability analysis of communication operations in distributed applications. *ACM SIGPLAN Notices* 36, 7 (2001), 123–132.

[57] Michael E. Wolf and Monica S. Lam. 1991. A Data Locality Optimizing Algorithm. In *PLDI*.

[58] M. Wolfe. 1989. More Iteration Space Tiling. In *SC*.

[59] Nan Xiong. 2018. *Optimizing Tall-and-Skinny Matrix-Matrix Multiplication on GPUs.* Ph.D. Dissertation. UC Riverside.

[60] Qinqing Zheng and John D. Lafferty. 2016. Convergence Analysis for Rectangular Matrix Completion Using Burer-Monteiro Factorization and Gradient Descent. *CoRR* (2016).