Breaking The Monopoly of Dimensions: Towards I/O-Optimal Matrix Multiplication

Anonymous Author(s)

Contents

List of Figures

PPoPP'19, February 16–20, 2019, Washington DC, USA
2018. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

From left to right, **par in ij**, **par in ijk** and **cubic**. Note the trade-off between I/O (shrinkage of local domain) and communication (dashed red lines showing parallelization and required communication in **k** dimension).

4 Parallel efficiency of different parallelization schemes. Three vertical dashed lines correspond to thresholds $p1 = mn/S$, $p2 = mnk/S^{3/2}$ and $p3 = (\frac{3}{S})^{3/2}mnk$ (Table 3). Note that for cases (b) and (c), increasing memory S six times reduces the number of processes required to fully saturate it drops from $p2 = 741455$ to $p2 = 50449$.

5 Process decomposition for square matrices and 65 processes. To utilize all resources, the local domain is drastically stretched (a). Dropping one process results in a symmetric grid (b) that increases the computation per process only by 1%, but reduces the communication by 36%.

6 An MMM CDAG forming a 3D iteration space $\mathcal{V} \subset \mathbb{Z}^3$. An input matrix A (blue vertices) is represented as its projection $\alpha = \phi_{ik}(\mathcal{V})$ on an **ik** plane - similarly, an input matrix B (red vertices) is a projection $\beta = \phi_{kj}(\mathcal{V})$ on the **kj** plane and an output matrix C (light yellow) is a projection $\gamma = \phi_{ij}(\mathcal{V})$ on the **ij** plane. Each vertex in this iteration space (except of the vertices in the bottom layer) has three parents - blue, red, and yellow and one yellow child (except of vertices in the top layer). $\alpha \cup \beta \cup \gamma$ form the dominator set $Dom(V_i)$. subcomputation $V_i \subset \mathcal{V}$ of an S-partition must satisfy $|Dom(V_i)| = |\alpha| + |\beta| + |\gamma| \leq S$ (number of inputs must be smaller than S. 6b optimal surface to volume subset shape. Note that in a subsequent subset computation only one of the three planes (blue, red or yellow) can be reused. 6c the optimal subset shapes when data reuse is considered. Observe that even though $|V_i| > |V_j|$, but $|V_i|/(|\alpha_i| + |\beta_i|) < |V_j|/(|\alpha_j| + |\beta_j|)$.

List of Tables

1 The formal summary of 2D, 2.5D, CARMA, and COMM schemes for matrix-matrix multiplication (The 3D decomposition is a special case of 2.5D, where $c = p^{1/3}$). The most important symbols used here are described in Table 2. The "easiest case" is when the matrices are square and there is no extra memory for redundant copies of input data. The "hardest case" is when $k \gg m = n$ and there is space for extra $p^{1/3}$ copies. For simplicity, we assume that parameters are chosen such that all divisions have integer results.

2 The most important symbols used in the paper.

3 Total communication volume of MMM and its parallel efficiency of the different parallelization schemes. We assume that all the respective local domain sizes divide the global matrix sizes evenly (e.g., for cubic domain in range $p \leq mn/S$, we assume that $\sqrt{S/3} = a_1M = a_2N = a_3K$ for $a_1, a_2, a_3 \in \mathbb{N}$). The optimal schedule (§ 7.3) switches from **par. in ijk** to **cubic** when $p \geq \frac{mnk}{S^{3/2}}$.

1 Abstract

[Greg: Old sketch of the abstract. To be done at the end] In this paper we address an allegedly well-known problem of distributed Matrix Matrix Multiplication and show new optimizations both from theoretical and implementation sides. We establish a new framework for assessing analytical data movement lower bounds, together with deriving optimal sequential and parallel scheduling. Our "bottom-up" parallelization technique, opposed to state of the art "top-down" approaches is naturally agnostic to problem dimensions and is provably optimal in all scenarios, resulting in up to $\sqrt{3}$ times reduction in communication volume. With our fine-tuned data layout transformations, we are able to achieve X % peak FLOP/s running on up to Y nodes on the Piz Daint supercomputer, beating currently fastest-known algorithms by a factor of Z, using provably I/O optimal, robust schedule and employing RDMA mechanisms for communication-computation overlap.

2 Introduction

Matrix-matrix multiplication (MMM) is one of the most fundamental building blocks in scientific computing. It is widely used not only in virtually all linear algebra algorithms (Cholesky and LU decomposition [42], eigenvalue factorization [14], triangular solvers [16]), but also in numerous graph algorithms such as Breadth-First Search (BFS) [19] or Triangle Counting [6] through efforts such as GraphBLAS [36]. Other use cases include spectral clustering [44] or machine learning [2]. Thus, accelerating MMM routines is of great significance for many domains.

Hong and Kung [34] proved nearly 30 years ago a sequential I/O lower bound for matrix multiplication to be $\Omega\left(\frac{n^3}{\sqrt{S}}\right)$ in their two-level memory model, where S is the size of the fast memory [1] Irony et al. [32] extended those results to the parallel machine with p processes, each of which has a fast private memory of size S, proving the $\Omega\left(\frac{n^3}{p\sqrt{S}}\right)$ bound on communication per process. If the memory size S is bounded by the size the input data ($pS = O(n^2)$), the bound

[1] 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 [4, 5, 32].

$O\left(\frac{n^2}{\sqrt{p}}\right)$ is asymptotically attainable by algorithms such as Cannon's [11] or SUMMA [1]. In the presence of extra memory ($pS = O(p^{1/3}n^2)$), the 3D decomposition [3] yields a better asymptotic bound $O\left(\frac{n^2}{P^{2/3}}\right)$. The 2.5D algorithm by Solomonik and Demmel [52] effectively interpolates between those two results, depending on the available memory.

Another class of optimizations, based on group theory, are Strassen-like routines [54], which asymptotically reduce the number of arithmetic operations. Currently, the algorithm with the lowest asymptotic complexity $O(n^{2.3729})$ is by Le Gall [37]. In our work, however, we focus only on the "classical" MMM algorithms which perform n^3 additions and multiplications, as they are often faster in practice on modern parallel architectures with deep memory hierarchies [20].

While the largest focus has traditionally been placed on accelerating the multiplication of square matrices, more focus has recently been put on non-square matrices as well. Such scenarios are common in a number of relevant areas and problems, for example in machine learning [58] or computational chemistry [46]. Unfortunately, it has been shown that algorithms optimized for squared matrices often perform poorly in cases when matrix sizes vary significantly [23]. This issue is tackled in the work by Demmel et al. [23], who introduced CARMA, an algorithm that achieves asymptotic lower bounds for all configurations of dimensions and memory size.

However, we observe several problems of the current state-of-the art algorithms, such as ScaLAPACK [17] (an implementation of SUMMA), Cyclops Tensor Framework (CTF) [53] (an implementation of the 2.5D MMM algorithm), or CARMA [23]. ScaLAPACK supports only the 2D decomposition and requires a user to fine-tune parameters such as block sizes or process decomposition. CTF yields poor performance on matrices with dimensions which are multiples of large primes [Greg: this is what preliminary results show. Waiting for full results], a common case in, e.g, computational chemistry [22]. Moreover, the recursive implementation of CARMA cannot directly handle scenarios when the number of processes is not the power of two, a common case, as the number of processes is usually determined by the available hardware resources, rarely a power of two. For example Piz Daint, the third fastest supercomputer in 2017, has 1,813 CPU nodes, which are impossible to arrange in a cubic grid. Furthermore,we emphasize that asymptotic complexity is an insufficient measure of speedup. For example, the Coppersmith–Winograd algorithm [18] achieves computational complexity of $O(n^{2.376})$, but its constant terms are prohibitively big and make it slower in practice than the classical $O(n^3)$ algorithms. We later (§ 9) show that CARMA suffers from similar overheads.

In this work, we present COMM (Communication Optimal Matrix Multiplication): an algorithm that takes a new approach to multiplying matrices and alleviates the above issues. COMM is I/O optimal *with regard to all constant factors* for all combinations of parameters. To prove the optimality of our algorithm, we (1) first provide a new constructive proof of sequential I/O lower bound, improving the constant factors of an existing bound by Smith and van de Gein [50], (2) derive the communication cost of parallelizing the sequential schedule, (3) construct a communication optimal parallel schedule. or SUMMA [1]. The detailed communication analysis of COMM, 2D, 3D and recursive decompositions is presented in Table 1. Our algorithm reduces the data movement volume up to $\sqrt{3} \approx 1.73$ times compared to the asymptotically optimal recursive decomposition and up to $\max\{n, m, k\}$ times compared to the 2D algorithms, like Cannon's [38] or SUMMA [1].

Unlike CARMA that is based on the recursive data layout, our implementation enables transparent integration with the ScaLAPACK format [15] and delivers near-optimal computation throughput. We later (§ 8) illustrate that the obtained schedule naturally expresses computation-communication overlap, which can be used for even higher speedups using Remote Direct Memory Access (RDMA) mechanisms. Our implementation also uses various techniques, like buffer and layout optimization to further accelerate the execution time. Finally, our I/O optimal approach is easily generalizable to other linear algebra kernels.

We provide the following contributions:

- We present COMM : the distributed MMM algorithm that is I/O optimal for any combination of parameters.
- Extending the general analysis of I/O complexity, we show a new proof constructive of the sequential MMM I/O lower bound, delivering tight constant factors (§ 6. We extend this proof to both distributed and shared memory machines (§ 7),
- We use block-cyclic data layout(§ 8.1) and a static buffer pre-allocation(§ 8.5), which reduce memory footprint for communication buffers and guarantees minimal reshuffling of input data, providing better compatibility with ScaLAPACK format than CARMA (§ 8),
- We naturally express the computation-communication overlap, which enables even higher speedups by leveraging the advantages of the RDMA mechanism (§ 8),
- We perform extensive evaluation on the Piz Daint supercomputer for an extensive selection of problem dimensions, memory sizes, and numbers of processes, showing the speedup of up to ?? (?? on average), compared to ScaLAPACK, CTF and CARMA (§ 9).

3 General Background

In this section, we discuss necessary concepts and terminology needed for our analysis. The most important symbols used in the paper are gathered in Table 2.

3.1 Computation Model

We now briefly specify a model of a *general* computation; we use this model in both the sequential and parallel setting.

An algorithm is modeled with a computational directed acyclic graph (CDAG) $G = (V, E)$ [12, 27, 49]. First, V is a set of vertices; one vertex represents one elementary operation in the given computation. E is a set of edges; an edge represents a data dependency between operations in V. We call a set of all immediate predecessors (or successors) of a vertex 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 children, respectively). A *schedule* is a dependency-preserving sequence of elementary operations (ordering of vertices of the CDAG). One schedule therefore corresponds to one of topological orders of G. For example, it is an order in which all n^3 multiplications in MMM are performed.

3.2 Machine Model

We use a well-established model for parallel computations with p processes, each with local memory of size S words [23, 52]). A process can hold, send, and receive from any other process up to S words at a time.

3.3 Naming

Throughout this paper we focus on an *input/output (I/O) cost* W and *I/O optimality*. The I/O cost W is a number of elements transfered during the execution of a schedule. In the context of a sequential or shared memory machine equipped with small-and-fast and slow-and-big memories, these transfers refer to load and store operations from and to the slow memory (also called the *vertical I/O*). In the context of distributed machine with a limited memory per node, the transfers indicate a communication 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. An *I/O complexity* Q of a CDAG is a lower bound of the I/O cost of the optimal schedule $\forall_W W \geq Q$.

In addition to the I/O cost, we also model a *latency cost* L, which is a required number of steps in which I/O operations are performed.

[Greg: Not sure should we add this.] The shared and distributed machines may be nested of combined to model, e.g., a distributed machine with a local memory hierarchy per node. We then have two I/O costs, vertical W_v: a total number of elements loaded from/stored to the nodes' memories, and horizontal W_h: a total number of elements communicated between the nodes. of A schedule in this model may be horizontally *and/or* vertically optimal, if it minimizes the communication and/or load and store operations among all schedules.

3.4 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 $A \in \mathbb{R}^{m \times k}, B \in C \in \mathbb{R}^{m \times n}$, where m,, and k are matrix dimensions.

2D decomposition In the SUMMA algorithm, processes are arranged in the $a \times b$ grid, where a and b are user-specified. Computation is performed in k/t steps, for a user specified value t. At each step, processes broadcast their local data among rows and columns, and perform $2MNt/p$ computations. For large k, the broadcast cost is amortized [1]. The I/O cost is $W = (a+b)k$ and the latency cost $C_L = 2K/t$. The memory requirement per process is $S \geq (mn + mk + nk)/p$.

3D decomposition Here, processes are arranged in the $p^{1/3} \times p^{1/3} \times p^{1/3}$ grid. Each process receives $mk/p^{2/3}$ elements of A, $nk/p^{2/3}$ elements of B, computes locally mnk/p partial results and sends $mn/p^{2/3}$ elements of C. The algorithm is performed in a single step. However, the distribution of A and B is performed with a broadcast tree, yielding the I/O cost $W = (mn + nk + mk)/p^{2/3}$ and the latency cost $C_L = \log(p)$. The memory requirement per process is $S \geq (mn + nk + mk)/p^{2/3}$.

2.5 decomposition Processes are arranged in the $\sqrt{(p/c)} \times \sqrt{(p/c)} \times c$ grid, where $c = pS/(mn + mk + nk)$. The computation is performed in $\sqrt{p/c^3} - 1$ steps. At each step, processes perform Cannon's algorithm on a subset of data of size $2MNc/p$. For $I = mn + mk + nk$, The I/O cost is $W = (mk+nk)\sqrt{\frac{I}{S}}/p + MNS/I$ and the latency cost $C_L = \frac{2}{p}\left(\frac{I}{S}\right)^{3/2} + \log(pS/I)$. For $m = n = k$, this simplifies to $W = \frac{2\sqrt{S}n^3}{p\sqrt{S}}$ and $C_L = \frac{6\sqrt{S}n^3}{pS^{3/2}}$. The memory requirement per process is $S \geq (mn + mk + nk)/p$.

Recursive decomposition Processes are decomposed recursively in the $p_1 \times p_2 \times p_3$ grid, where $p_i = 2^{k_i}, p_1 p_2 p_3 = p$ for some $k_1, k_2, k_3 \in \ltimes$. The computation is performed in $\log(\max\{p, \sqrt{I/S}\})$ steps. The I/O cost is $W = O\left(\frac{mnk}{p\sqrt{S}}, \left(\frac{mnk}{P}\right)^{2/3}\right\}\right)$ and the latency cost $C_L = \log(\max\{p, \sqrt{I/S}\})$. The memory requirement per process is $S \geq (mn + mk + nk)/p$.

3.5 MMM and iteration space

In a most straightforward implementation of a classical MMM algorithm, values of matrix C are updated k times. To represent this algorithm as a CDAG, we express it in a Single Static Assignment (SSA) form to remove the self loops corresponding to updating the values of C. Then, the algorithm may be written as a tripe-nested loop:

```
1 for (i1 = 1:n)
2   for (i2 = 1:m)
3     for (i3 = 1:k)
4       V(i1,i2,i3) = V(i1,i2,i3-1) + A(i1,i3)*B(i3,i2)
5     end
6     C(i1,i2) = V(i1,i2,k)
```

```
7   end
8 end
```

Listing 1. A pseudocode of a classical MMM using the Single Static Assignment (SSA) form

assuming that `C(i1,i2,0) = 0` for all `i1,i2`. Due to this structure, the corresponding CDAG can be represented as a 3D *iteration space* $\mathcal{V}$ [57](Figure 6). Let **i**, **j**, **k** be its orthonormal spanning vectors corresponding to the loops at lines 2,3,4. Denote a plane **uv** as a plane spanned by vectors **u** and **v**, for some **u**,**v** $\in$ {**i**, **j**, **k**}. Assume, that $p = (i1, i2, i3) \in \mathcal{V}$ is a point in the iteration space. Then $\phi_{ij}(p) = (i1, i2)$ is a projection of this point to the plane **ij**.

For each evaluation of Line 5, we map elements `V(i1,i2,i3)` to a point $v = (i1, i2, i3) \in \mathcal{V}$, `V(i1,i2,i3-1)` to $\phi_{ij}(v) = (i1, i2)$, `A(i1,i3)` to $\phi_{ik}(v) = (i1, i3)$, and `B(i3,i2)` to $\phi_{kj}(v) = (i3, i2)$.

The iteration space $\mathcal{V}$ contains mnk integer points ($v = (i1, i2, i3) \in \mathcal{V}, i1, i2, i3 \in \mathbb{Z}$) corresponding to mnk elements of `V(i1,i2,i3)`. A whole computation of MMM CDAG (and its corresponding $\mathcal{V}$) may be split into a series of subcomputations (subsets $V_i \subset \mathcal{V}$), called a partitioning or a *schedule* $\mathcal{S} = \{V_i\}$. Later in this paper we will discuss both sequential and parallel schedules, that is, series of subcomputations V_i that are either evaluated sequentially by a single process, or in parallel by p processes. A subcomputation may form a *cuboid*, if its corresponding $V_i \subset \mathcal{V}$ has a geometrical shape of a cuboid in the 3D iteration space.

4 COMM: High-Level Description

We now provide a high-level description of COMM, the fast and elegant algorithm that enables communication-optimal MMM. COMM decomposes processes by parallelizing the optimal sequential schedule under given constraints: equal work distribution and memory size per process. Such a local sequential schedule is independent of matrix dimensions. Thus, intuitively, instead of dividing the global domain among p processes (the *top-down* approach), we start from deriving an I/O optimal *sequential* schedule and then parallelize it, minimizing the I/O and latency cost C_B, C_L (the *bottom-up* approach). This direction enables proving the optimality of the data movement, both horizontally and vertically (i.e., I/O) in our schedule, deriving all constant factors. Our experiments (§ 10) show that COMM is indeed faster than the state-of-the-art algorithms.

The sketch of COMM is presented in Algorithm 1. In lines 1 and 2, we derive a communication optimal local domain (§ 7.3). In line 3, we find a process grid D that evenly distributes this domain by the matrix dimensions m, n, and k. If this distribution is unequal (e.g., matrix dimensions are not divisible by a_{opt} and b_{opt}), this function also evaluates new values of a and b by finding the best matching decomposition (§ 8.2). In line 5, the local distribution of matrices A_l, B_l and C_l is determined, either as a fixed input data layout, or an optimal block-recursive one (§ 8.1). Also, a *Map* structure is determined, which translates global matrix indices to a current owning process and a local offset in its memory. In line 6 we compute the number of sequential steps (lines 7 to 10) in which every process: (1) distributes its local data A_l and B_l among the grid D, based on the *Map* structure (line 8), and (2) multiplies locally A_l and B_l (line 9). Finally, the local partial results C_l are reduced over the grid D (line 11).

Algorithm 1 COMM

Input: matrices $A \in \mathbb{R}^{m\times k}, B \in \mathbb{R}^{k\times n}$,
number of processes: p, memory size: S
Output: matrix $C = AB \in \mathbb{R}^{m\times n}$

1: $a_{opt} \leftarrow \left\lfloor \min\left\{\sqrt{S}, \left(\frac{mnk}{p}\right)^{1/3}\right\}\right\rfloor$ ▷ (Equation 17)
2: $b_{opt} \leftarrow \left\lfloor \max\left\{\frac{mnk}{pS}, \left(\frac{mnk}{p}\right)^{1/3}\right\}\right\rfloor$
3: $(D, a, b) \leftarrow FitRanks(m, n, k, a_{opt}, b_{opt}, p)$
4: **for all** $p_i \in \{1 \dots p\}$ **do in parallel**
5: $(A_l, B_l, C_l, Map) \leftarrow GetDataDecomp(A, B, D, p_i)$
6: $t \leftarrow \left\lceil \frac{2ab}{S-a^2} \right\rceil$ ▷ number of steps
7: **for** $j \in \{1 \dots t\}$ **do**
8: $(A_l, B_l) \leftarrow DistrData(A_l, B_l, D, Map, j, p_i)$
9: $C_l \leftarrow Multiply(A_l, B_l, j)$ ▷ compute locally
10: **end for**
11: $C_l \leftarrow Reduce(C_l, D, Map)$ ▷ Reduce the partial results
12: **end for**

5 I/O Lower Bounds for Arbitrary CDAGs

5.1 Intro

In this section we shortly introduce a general mathematical machinery we use to proof the I/O optimality of COMM. We extend Lemma 1 by Hong and Kung [34], which provides a method to find an I/O lower bound for a given CDAG. This lemma, however, does not give a tight bound, as it overestimates a *reuse set* size (cf. Lemma 4). Our key result here, Lemma 4 allows us to derive a constructive proof of a tighter I/O lower bound for a sequential execution of MMM CDAG (§ 6). We use this result in (§ 7) to prove the parallel I/O optimality of COMM.

Our method heavily relies on a red-blue pebble game (Definition 1) and an S-partition abstractions (Definition 3) introduced by Hong and Kung. Due to the space constraints, here we only bring necessary definitions and lemmas. The extended explanation, intuition and a proof of our Lemma 4 is included in the Appendix [Greg: (separate pdf, whatever we call it)].

5.2 Red-Blue Pebble Game

5.2.1 Short Version

Red-Blue Pebble Game A red-blue pebble game is an abstraction modeling 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 this data is inside a fast (or slow) memory.

Definition 1 (Red-Blue Pebble Game [34]). *Let $G = (V, E)$ be a CDAG. In the initial/terminal configuration, only inputs/outputs of the CDAG have blue pebbles. There can be at most S red pebbles used. A complete CDAG computation is a sequence of moves that lead from the initial to the terminal pebble configuration. The allowed moves are as follows: ① placing a red pebble on any vertex with a blue pebble (load), ② placing a blue pebble on any vertex with a red pebble (store), ③ placing a red pebble on a vertex whose parents have all red pebbles (compute), ④ removing any pebble (red or blue) from any vertex (freeing memory).*

An I/O optimal execution of a CDAG corresponds to a sequence of moves (called *pebbling* of a graph) which minimizes load ① and store ② moves.

5.2.2 Long Version

A red-blue pebble game is an abstraction modeling an execution of an algorithm in a two-level memory structure with a small-and-fast as well as large-and-slow memory. It is a powerful tool, extensible to arbitrarily many memory levels [47], that was used to derive lower bounds for algorithms such as sorting or FFT [34].

Intuition In the red-blue pebble game, a red (or blue) pebble placed on a vertex denotes that its value is inside the fast (or slow) memory. The actual computation (referred to as *pebbling*) is a series of allowed moves (e.g., moving a pebble from one vertex to another) that correspond to load, store, compute, or freeing-memory operations. The *I/O cost of a computation* is the number of pebble moves that correspond to loads and stores between the slow and the fast memory; finding the pebbling that minimizes this cost is PSPACE-complete [27, 39].

Details One first places a blue pebble on each of k input CDAG vertices (initializing the slow memory with the input data of size k). The pebbles are moved (i.e., the computation proceeds) until the final configuration of pebbles is obtained (i.e., the desired output is delivered). In the initial/terminal configuration, only inputs/outputs of the CDAG have blue pebbles. There can be at most S red pebbles used. A complete CDAG computation is a sequence of moves that lead from the initial to the terminal pebble configuration.

The allowed moves are as follows: ① placing a red pebble on any vertex with a blue pebble (load), ② placing a blue pebble on any vertex with a red pebble (store), ③ placing a red pebble on a vertex whose parents have all red pebbles (compute), ④ removing any pebble (red or blue) from any vertex (freeing memory).

5.2.3 Connections to MMM

Connections to MMM For a CDAG of MMM, an example pebbling moves include placing red pebbles on some vertices corresponding to elements of matrices A and B (load operations), placing red pebbles on the corresponding elements of C (compute), removing red pebbles from the loaded inputs (freeing memory) and placing blue pebbles on the computed elements of C (store).

5.3 S-Partitions

5.3.1 Short version

S-Partition Finding an I/O optimal pebbling is PSPACE-complete [39]. An S-partition abstraction allows us to find a lower bound Q by finding a partition of a CDAG with the following properties:

Definition 2 (S-partition of a CDAG [34]). *Let $G = (V, E)$ be a CDAG. An S-partition of G is a collection $\{V_1, ..., V_h\}$ of h subcomputations of V such that: ① $V_i \cap V_j = \emptyset$ and $\bigcup_{i=1}^{h} V_i = V$ for any $1 \leq i, j \leq h$, ② $\forall i \quad |Dom(V_i)| \leq S$, ③ $\forall i \quad |Min(V_i)| \leq S$, and ④ there is no cyclic dependence between subcomputations.*

$Dom(V_i) \not\subset V_i$ is the *dominator set*: a set of vertices such that every path from an input of the CDAG to a vertex in V_i contains some vertex in $Dom(V_i)$. $Min(V_i) \subset V_i$ is the *minimum set* of V_i: it contains vertices that do not have any children in V_i. Finally, $H(S)$ is the cardinality of the smallest valid S-partition of a given CDAG.

5.3.2 Long version

The notion of an *S-partition* facilitates deriving lower bounds on the I/O computation cost [34]. Here, one divides a given CDAG into consecutive *subcomputations*, each of which requires at least S load and store operations. The key element in more straightforward lower bound proofs is to analytically bound the size (vertex count) of the largest subcomputation, given its input and output size (i.e., the number of vertices outside (inside) the subcomputation that have a child inside (outside) of it). Formally:

Definition 3 (S-partition of a CDAG [34]). *Let $G = (V, E)$ be a CDAG. An S-partition of G is a collection $\{V_1, ..., V_h\}$ of h subcomputations of V such that: ① $V_i \cap V_j = \emptyset$ and $\bigcup_{i=1}^{h} V_i = V$ for any $1 \leq i, j \leq h$, ② $\forall i \quad |Dom(V_i)| \leq S$, ③ $\forall i \quad |Min(V_i)| \leq S$, and ④ there is no cyclic dependence between subcomputations.*

$Dom(V_i) \not\subset V_i$ is the *dominator set*: a set of vertices such that every path from an input of the CDAG to a vertex in V_i contains some vertex in in $Dom(V_i)$. $Min(V_i) \subset V_i$ is the

minimum set of V_i: it contains vertices that do not have any children in V_i. Finally, $H(S)$ is the cardinality of the smallest valid S-partition of a given CDAG.

We use a symbol $\mathcal{S}(S) = \{V_1, \ldots, V_h\}$ to denote an S-partition.

5.3.3 Connections to MMM

Connections to MMM In MMM, a subcomputation V_i is a calculation of partial sums of C that can be computed with at most S elements of A and B, and that contributes to at most S outputs. Those elements from A and B, as well as previous values of C being updated, form $Dom(V_i)$. Then, $Min(V_i)$ corresponds to the result of this subcomputation (cf. Figure 6). $H(S)$ denotes the number of such subsets required to calculate the final result. Assuming that each subcomputation computes the same number of partial results $\forall_{i,j}|V_i| = |V_j|$, and observing the total number of partial results $|V| = mnk$, we have $H(S) = \frac{mnk}{V_i}$.

5.4 Existing General I/O Lower Bound

5.4.1 Short version

Lemma 1 (Lower bound on the number of I/Os [34]). *The minimal number Q of I/O operations for any valid execution of a CDAG of any I/O computation is bounded by*

$$Q \geq S \cdot (H(2S) - 1) \tag{1}$$

We now sketch the original proof, as we use it as a basis for our key Lemma 4. Assume that we know the optimal schedule of the CDAG. Divide the computation into h consecutive subcomputations $V_1, V_2, ..., V_h$, such that during the execution of V_i, $i < h$, there are exactly S I/O operations, and in V_h there are at most S operations. Now, for each V_i, we define two subsets of V, $V_{R,i}$ and $V_{BR,i}$. $V_{R,i}$ contains vertices that have at least one child in V_i and have red pebbles placed on them just before V_i begins. $V_{BR,i}$ contains vertices that have blue pebbles placed on them just before V_i begins, and have red pebbles placed on them during V_i. Using these definitions, we have: ❶ $V_{R,i} \cup V_{BR,i} = Dom(V_i)$, ❷ $|V_{R,i}| \leq S$, ❸ $|V_{BR,i}| \leq S$, and ❹ $|V_{R,i} \cup V_{BR,i}| \leq |V_{R,i}| + |V_{BR,i}| \leq 2S$. We define similar subsets $W_{BR,i}$ and $W_{R,i}$ for the minimum set $Min(V_i)$, that is $W_{BR,i}$ contains all vertices in V_i that have a blue pebble placed on them during V_i, and $W_{R,i}$ contains all vertices in V_i that have a red pebble at the end of V_i. By the definition of V_i, $W_{BR,i} \leq S$, by the constraint on the red pebbles, we have $W_{R,i} \leq S$, and by te definition of the minimum set,$Min(V_i) \subset W_{R,i} \cup W_{BR,i}$. Finally, by Definition 3, $V_1, V_2, ..., V_h$ form a valid $2S$-partition of the CDAG.

5.4.2 Long version

We now cite a *general* lower bound on the cost of *any* I/O computation [34] and sketch the proof, which is the basis for our *tighter general* bound on the I/O cost (Lemma 4).

Intuition The key notion in the existing bound is to use $2S$-partitions for a given fast memory size S. For some subcomputation V_i, if $|Dom(V_i)| = 2S$ vertices, then at most S of them could contain a red pebble before V_i begins. Thus, at least S additional pebbles need to be loaded from the memory. The similar argument goes for $Min(V_i)$. Therefore, knowing the lower bound on the number of sets V_i in a valid $2S$*-partition*, together with the observation that each V_i performs at least S I/O operations, we have:

Lemma 2 (Lower bound on the number of I/Os [34]). *The minimal number Q of I/O operations for any valid execution of a CDAG of any I/O computation is bounded by*

$$Q \geq S \cdot (H(2S) - 1) \tag{2}$$

Details Assume that we know the optimal schedule of the CDAG. Divide the computation into h consecutive subcomputations $V_1, V_2, ..., V_h$, such that during the execution of V_i, $i < h$, there are exactly S I/O operations, and in V_h there are at most S operations. Now, for each V_i, we define two subsets of V, $V_{R,i}$ and $V_{BR,i}$. $V_{R,i}$ contains vertices that have red pebbles placed on them just before V_i begins. $V_{BR,i}$ contains vertices that have blue pebbles placed on them just before V_i begins, and have red pebbles placed on them during V_i. Using these definitions, we have: ❶ $V_{R,i} \cup V_{BR,i} = Dom(V_i)$, ❷ $|V_{R,i}| \leq S$, ❸ $|V_{BR,i}| \leq S$, and ❹ $|V_{R,i} \cup V_{BR,i}| \leq |V_{R,i}| + |V_{BR,i}| \leq 2S$. We define similar subsets $W_{BR,i}$ and $W_{R,i}$ for the minimum set $Min(V_i)$, that is $W_{BR,i}$ contains all vertices in V_i that have a blue pebble placed on them during V_i, and $W_{R,i}$ contains all vertices in V_i that have a red pebble at the end of V_i. By the definition of V_i, $W_{BR,i} \leq S$, by the constraint on the red pebbles, we have $W_{R,i} \leq S$, and by te definition of the minimum set,$Min(V_i) \subset W_{R,i} \cup W_{BR,i}$. Finally, by Definition 3, $V_1, V_2, ..., V_h$ form a valid $2S$-partition of the CDAG.

5.4.3 Data Reuse

A more careful look at the sets $V_{R,i}, V_{BR,i}, W_{R,i}$ and $W_{RB,i}$ allows us to refine the bound on the number of I/O operations on a CDAG. By its definition, $V_{BR,i}$ is a set of vertices on which we perform move ① (load). On the other hand, $V_{R,i}$ is the set of vertices that were already computed (red-pebbled) during some $V_j, j < i$, and are reused during V_i (the move ① is not needed). We call $V_{R,i}$ a *reuse set* of V_i. Similarly, by its definition, $W_{BR,i}$ contains all the vertices on which we perform move ② (store) during V_i. Therefore, if Q_i is the number of I/O operations during the subcomputation V_i, then we have $Q_i \geq |V_{BR,i}| + |W_{BR,i}|$. The trivial bounds are $V_{R,i} leS$ and $W_{BR,i} \geq 0$, but if one can show that $\exists_{R(S)\in\mathbb{Z}} : \forall_i : V_{R,i} \leq R(S) < S$ or $\exists_{T(S)\in\mathbb{Z}} : \forall_i : W_{BR,i} \geq T(S)$, we can use $R(S)$ and $T(S)$ to tighten a bound on Q. We call $R(S)$ a *maximum reuse* and $T(S)$ a *minimum store size* of a CDAG.

5.5 Tighter General I/O Lower Bounds

5.5.1 Short version

Key Result: Reuse-based Lemma Observe that $V_{R,i}$ by definition is a set of vertices that have red pebbles placed on them during some $V_j, j < i$ and that are required to pebble V_i. We call $V_{R,i}$ a *reuse set* of V_i, as it reuses precomputed vertices from previous steps without storing and loading them back. Therefore, only set $V_{BR,i}$ contributes to the load instructions. We use this observation in our key lemma:

Lemma 3. *The minimal number Q of I/O operations for any valid execution of a CDAG $G = (V, E)$ is bounded by*

$$Q \geq (S - R(S)) \cdot (H(S) - 1) \tag{3}$$

$R(S)$ is the upper bound on the reuse set size $R(S) = max_i\{|V_{R,i}\}$. Moreover:

$$H(S) \geq \frac{|V|}{|V_{max}|} \tag{4}$$

where $V_{max} = \arg\max_{V_i \in \mathcal{S}} |V_i|$ is the largest subset of vertices in $\mathcal{S}$.

5.5.2 Long version

We now enhance the general I/O lower bound *by tightening the bound on the set $V_{R,i}$*, that is, the *data reuse between the computations* bounds. We later show the schedule attaining this bound (§ 6) and we use this schedule to minimize horizontal communication between processes in a distributed MMM computation (§ 7). Our main result in this section is as follows:

Lemma 4. *The minimal number Q of I/O operations for any valid execution of a CDAG $G = (V, E)$ is bounded by*

$$Q \geq (S - R(S) + T(S)) \cdot (H(S) - 1) \tag{5}$$

$R(S)$ is the maximum reuse and $T(S)$ is the minimum store size (§ 5.4.3). Moreover:

$$H(S) \geq \frac{|V|}{|V_{max}|} \tag{6}$$

where $V_{max} = \arg\max_{V_i \in \mathcal{S}} |V_i|$ is the largest subset of vertices in $\mathcal{S}(S)$.

Proof. To prove Eq. (5), we use analogous reasoning as in Lemma 1. We associate the optimal pebbling with h consecutive subcomputations $V_1, \dots V_h$ with the difference that each subcomputation V_i performs $Q_{i,s}$ store and $Q_{i,l}$ load operations, such that $S - R(S) \leq Q_{i,s} \leq S$ and $T(S) \leq Q_{i,l} \leq S$ for each V_i. We now define sets $V_{R,i}, V_{BR,i}, W_{R,i}$ and $W_{RB,i}$ analogously as in the previous proof. we first observe:

$$\begin{gathered}\forall_i : (S - R(S)) \leq |V_{BR,i}| \leq S \\ \forall_i |V_{R,i}| \leq R(S) \\ R(S) \leq S\end{gathered}$$

Using the fact that $V_{R,i} \cup V_{BR,i} = Dom(V_i)$ (❶ from § 5.4):

$$\begin{gathered}|Dom(V_i)| = |V_{R,i}| + |V_{BR,i}| \\ |Dom(V_i)| \leq R(S) + (S - R(S)) \\ |Dom(V_i)| \leq S\end{gathered} \tag{7}$$

By making analogous construction for the store operations, we show that $V_1 \dots V_h$ form a valid S-partition $\mathcal{S}(S)$. Therefore, a schedule performing $Q \geq (S - R(S) + T(S))h$ operations has an associated $\mathcal{S}(S)$, such that $|\mathcal{S}(S)| = h$. By the definition $H(S) = \min_{\mathcal{S}(S)} |\mathcal{S}(S)|$, we have $Q \geq (S - R(S)) \cdot (H(S) - 1)$.

To prove Eq. (6), observe that V_{max} by definition is the largest subset in the optimal S-partition. As the subsets are disjoint, any other subset covers fewer remaining vertices to be pebbled than P_{max}. Because there are no cyclic dependencies between subsets, we can order them topologically as $V_1, V_2, \dots V_{H(S)}$. To ensure correct indices, we also define $V_0 \equiv \emptyset$. Now, define W_i to be the set of vertices not included in any subset from 1 to i, that is $W_i = V - \bigcup_{j=1}^{i} V_j$. Clearly, $W_0 = V$ and $W_{H(S)} = \emptyset$. Then, we have

$$\begin{gathered}\forall_i \quad |V_i| \leq |V_{max}| \\ |W_i| = |W_{i-1}| - |V_i| \geq |W_{i-1}| - |V_{max}| \geq i|V_{max}| \\ |W_{H(S)}| = 0 \geq H(S) \cdot |V_{max}|\end{gathered}$$

that is, after $H(S)$ steps, we have $H(S)|V_{max}| \geq |V|$. □

5.5.3 Computational Intensity

For graphs of parametric sizes (e.g., MMM graph has $mnk + mk + kn$ vertices), we need a tool to allow us to bound the I/O complexity of the whole graph by bounding the maximum size of a single subcomputation. We provide an observation that connects the minimal number Q of I/O operations (cf. Eq. (5)) and a notion of *computational intensity*. Define computational intensity of the subcomputation V_i as $\rho_i = \frac{|V_i|}{V_{BR,i} + W_{BR,i}}$. Intuitively, computational intensity is the ratio of the number of computed elements ($|V_i|$) and the number of loaded and stored vertices ($V_{BR,i} + W_{BR,i}$). Having the bounds $S - R(S)$ and $T(S)$ on $V_{BR,i}$ and $W_{BR,i}$, respectively, and inserting Inequality 6 to 5, we derive the following corollary:

Corollary (Computational intensity). *Denote $\rho = \max_i(\rho_i) \leq \left(\frac{|V_{max}|}{S - R(S) + T(S)}\right)$. Then, the number of I/O operations Q is bounded by:*

$$Q \geq \frac{|V|}{\rho} \tag{8}$$

5.5.4 $2S$-Partition vs S-Partition: An MMM Example

To show why Lemma 2 gives a tighter bound than Lemma 1, consider a CDAG of MMM. We can draw it in a 3D iteration

space $\mathcal{V}$. (§ 3.5, 6). Then, an S-partition is a decomposition of this space into subcomputations V_i whose number of inputs ($|\alpha_i| + |\beta_i| + |\gamma_i|$) is smaller than S. When a subcomputation is viewed as a subspace in the iteration space $V_i \in \mathcal{V}$, this input constraint may be viewed as a generalization of the Loomis-Whitney inequality [41], which relates a cardinality of a finite set in $\mathbb{Z}^t$ with cardinalities of its projections to all t dimensions. This technique was used by Irony et al. [33] to derive first parallel I/O MMM lower bounds.

Assume that each subcomputation forms a cuboid $[a{\times}b{\times}c]$ in this iteration space (we actually prove it in § 6.1). Its faces form the dominator set. Now based on Lemma 2, we construct a $2S$-partition with a minimal cardinality, deriving subcomputations of a cubic ($a = b = c$) shape (Figure 6 b), with a cube side $a = \sqrt{2S/3}$. Such schedule will perform $2a^2\frac{n^3}{a^3} = \sqrt{\frac{3}{2}}\frac{2N^3}{\sqrt{S}}$ I/O operations.

Now, the question is: what is the size of a maximum reuse $R(S)$? Observe that only one of three faces of this cuboid can be reused (used by a subsequent subcomputation while still being in fast memory). This observation will help us derive in (§ 6.1) a "flat" shape (Figure 6 c) which performs $\frac{2N^3}{\sqrt{S}}$ I/O operations - an $\sqrt{3/2}$ improvement.

6 Optimal Sequential Matrix Multiplication

We now proceed to establish a tight sequential I/O lower bound of MMM. Using the mathematical machinery of Lemma 4, we show a constructive proof of $Q \geq 2MNK/\sqrt{S} + MN$. This result has three main contributions (1) it is an additive improvement over the bound $2mnk/\sqrt{S} - 2S$ by Smith and van de Geijn [50] by an additive factor of $2S + mn$, (2) the proof is constructive - what immediately follows is a corresponding schedule, (3) generality of a used technique is later used to derive I/O optimal parallel schedule (§ 7) and may be used in other linear algebra problems.

Theorem 1 (Sequential Matrix Multiplication I/O lower bound)**.** *Multiplying matrices of sizes $m{\times}k$ and $k{\times}m$ requires a minimum number of $\frac{mnk}{\sqrt{S}} + mn$ I/O operations.*

Proof. We first discuss a short intuition behind the proof. Next, we provide all details.

Proof Intuition Using Lemma 4, we first establish a valid subset V_i of an S-partition that achieves maximum computational intensity ρ and find its size (the number of vertices). Knowing ρ and that there are mnk vertices in the whole MMM CDAG, we find the total I/O cost of the complete execution. Throughout the proof, in addition to full details, we include intuitive interpretations of key steps. To help keep up with the intuition, assume that V_i is a cuboid (§ 3.5), and that we want to find its optimal size under certain conditions.

Full Proof In this proof, we will use the iteration space $\mathcal{V}$ (§ 3.5) to represent the CDAG of MMM. By Definition 3, we divide the whole computation into $H(S)$ subcomputations $V_i, i \in \{1, \dots H(S)\}$, such that $Dom(V_i), Min(V_i) \leq S$. As stated in (§ 3.5), the dominator set of V_i is $Dom(V_i) = \phi_{ik}(V_i) \cup \phi_{kj}(V_i) \cup \phi_{ij}(V_i) = \alpha_i \cup \beta_i \cup \gamma_i$. Because all the read-after-write (RAW) dependencies are parallel to the vector **k** (Line 5 in Listing 1), the projection of the minimum set $Min(V_i)$ onto the plane **ij** is also equal to $\gamma_i : (\phi_{ij}(Min(V_i)) = \gamma_i)$.

Intuition: α_i, β_i and γ_i are the faces of our cuboid V_i. The "bottom" face γ_i (Figure 6) is formed by partial results of C from previous subcomputations, and the "top" face is formed by partial results evaluated by V_i, also forming input for the next subcomputation. Because all the dependencies are facing "upwards", the minimum set (the "top" face) is positioned directly above the required partial results of C, that is $\phi_{ij}(Dom(V_i)) = \phi_{ij}(Min(V_i))$.

By the definition of S-partition, we have:

$$|Dom(V_i)| = |\alpha_i| + |\beta_i| + |\gamma_i| \leq S \qquad (9)$$
$$|Min(V_i)| = |\gamma_i| \leq S$$

On the other hand, the Loomis-Whitney inequality [41] bounds the volume of V_i:

$$V_i \leq \sqrt{|\alpha_i||\beta_i||\gamma_i|} \qquad (10)$$

Now we proceed to find the upper bound on the maximum reuse size $R(S) = \max_{j=1\dots H(S)}(|V_{R,i}|)$. Consider two subsequent computations, V_i and V_{i+1}. Right after V_i, α_i, β_i and γ_i may have red pebbles on them (they are in the fast memory). On the other hand the dominator set of V_{i+1} is $Dom(V_{i+1}) = \alpha_{i+1} \sum \beta_{i+1} \sum \gamma_{i+1}$. Then, the reuse set $V_{R,i+1}$ is an intersection of those two sets:

$$V_{R,i+1} \subset (\alpha_i \cup \beta_i \cup \gamma_i) \cap (\alpha_{i+1} \cup \beta_{i+1} \cup \gamma_{i+1})$$
$$= (\alpha_i \cap \alpha_{i+1}) \cup (\beta_i \cap \beta_{i+1}) \cup (\gamma_i \cap \gamma_{i+1}) \qquad (11)$$

Intuition: Visualizing V_i and V_{i+1} as two cuboids, they can only be positioned in the 3D space so that they share at most one of their sides (either $\alpha_i \cap \alpha_{i+1} \neq \emptyset$ or $\beta_i \cap \beta_{i+1} \neq \emptyset$ or $\gamma_i \cap \gamma_{i+1} \neq \emptyset$). If we allow other shapes than cuboids, their shared surface is a sum of its projections into the three planes, $\alpha_i \cap \alpha_{i+1}$, $\beta_i \cap \beta_{i+1}$, and $\gamma_i \cap \gamma_{i+1}$.

Note that $\alpha_i, \beta_i \subset \mathcal{I}$ are inputs of the computation: therefore, by the Definition 1, they start in the slow memory (they already have blue pebbles). γ_i, on the other hand, is the projection of the minimum set $\phi_{ij}(Min(V_i)) = \gamma_i$. Therefore, at the end of V_i, vertices in $Min(V_i)$ may have only red pebbles placed on them (they may have not been stored back to the slow memory yet). Furthermore, by the Definition 3, these vertices have children that have not been pebbled yet. They either have to be reused forming the reuse set $V_{R,i+1}$, or stored back, forming $W_{BR,i}$ and requiring the placement of the blue pebbles. Because by their definition,

$\gamma_i \cap \alpha_i = \gamma_i \cap \beta_i = \emptyset$, we have $\gamma_i \cap V_{R,i+1} = \gamma_i \cap \gamma_{i+1}$ and $W_{BR,i} = \gamma_i \setminus \gamma_{i+1}$. Then, the number of I/O operations Q_i performed by the subcomputation V_i (cf. Lemma 4):

$$Q_i \geq |Dom(V_i)| - |V_{R,i}| + |W_{BR,i}| \tag{12}$$

Intuition: The dominator set of each subcomputation consists of certain elements from input matrices A and B, as well as the partial results of C. The number of loads required by V_i is the size of the dominator set $Dom(V_i)$ reduced by the elements already loaded (the reuse set $V_{R,i}$). The number of stores is equal to the number of outputs $|\gamma_i|$ that are not reused in subcomputation V_{i+1}, that is, not contained in $|\gamma_{i+1}|$.

We now proceed to find α_i, β_i, and γ_i that maximizes the computational intensity $\rho_i = \rho$ (§ 5.5.3). We can bound ρ_i by inserting Equations 9, 10, 11 and 12:

$$\begin{aligned}\rho_i &= \frac{|V_i|}{|Dom(V_i)| - |V_{R,i}| + |W_{BR,i}|} \\ &\leq \frac{\sqrt{|\alpha_i||\beta_i||\gamma_i|}}{|\alpha_i| + |\beta_i| + |\gamma_i| - |\gamma_i \cap \gamma_{i+1}| + |\gamma_i \setminus \gamma_{i+1}|}\end{aligned}$$

We immediately see that to maximize ρ_i, we have $\gamma_{i+1} = \gamma_i$, as it both maximizes $|\gamma_i \cap \gamma_{i+1}|$ and minimizes $|\gamma_i \setminus \gamma_{i+1}|$.

We then formulate it as an optimization problem:

$$\begin{gathered}\text{maximize } \sqrt{\gamma_i}\frac{\sqrt{|\alpha_i||\beta_i|}}{|\alpha_i| + |\beta_i|} \\ \text{subject to:} \\ |Dom(V_i)| = |\alpha_i| + |\beta_i| + |\gamma_i| \leq S \\ |\alpha_i|, |\beta_i|, |\gamma_i| \in \mathcal{Z}\end{gathered} \tag{13}$$

Observe that to maximize $\frac{\sqrt{|\alpha_i||\beta_i|}}{|\alpha_i|+|\beta_i|}$ we have $|\alpha_i| = |\beta_i|$. Then the solution to this maximization problem gives $|\alpha_i| = |\beta_i| \to 0$, $|\gamma_i| \to S$. But because α_i, β_i and γ_i are sets of vertices, therefore $|\alpha_i|, |\beta_i|, |\gamma_i| \in \mathcal{Z}$. Furthermore, we have $\phi_{kj}(\alpha_i) = \phi_{ik}(\beta_i)$, $\phi_{ij}(\alpha_i) = \phi_{ik}(\gamma_i)$ and $\phi_{ij}(\beta) = \phi_{kj}(\gamma)$. Finally, we get:

$$\begin{gathered}|\alpha_i| = |\beta_i| = \left\lfloor \sqrt{S+1} - 1 \right\rfloor \\ |\phi_{kj}(\alpha)| = |\phi_{ik}(\beta)| = 1 \\ |\gamma| = |\alpha||\beta| = \left\lfloor (\sqrt{S+1} - 1)^2 \right\rfloor\end{gathered} \tag{14}$$

From now on, to keep the calculations simpler, we use the following approximation:

$$\left\lfloor \sqrt{S+1} - 1 \right\rfloor \approx \sqrt{S} \tag{15}$$

All the following results may be easily rewritten using the precise formula and are correct up to this approximation factor.

Using the approximation 15, we have $|V_i| = S, R(S) = V_{R,i} = |\gamma_i| = S, \rho = \sqrt{S}/2$. Finally, by Corollary 5.5.3:

$$Q \geq \frac{|V|}{\rho} = \frac{2mnk}{\sqrt{S}}$$

This is the I/O cost of putting a red pebble at least once on every vertex in $\mathcal{V}$. Note however, that we did not put any blue pebbles on the outputs yet (all vertices in $\mathcal{V}$ had only red pebbles placed on them during the execution). By Definition 1, we need to place blue pebbles on mn output vertices, corresponding to the output matrix C, resulting in additional mn I/O operations, yielding final bound

$$Q \geq \frac{2mnk}{\sqrt{S}} + mn$$

□

6.1 I/O Optimal Schedule

The proof of Theorem 1 is constructive: that is, from it we immediately obtain a schedule achieving it. The optimal subcomputation V_i corresponds to S calculated partial products of C. Its inputs α and β are subsets of a single column of A and a single row of B, both of length $\sqrt{S}$. This may be viewed as an outer product formulation of MMM: each subcomputation is an outer product of two vectors of length $\sqrt{S}$, with subsequent subcomputations updating this result (Listing 2).

```
1  T = √S
2  for i_1 = 1:m/T
3    for j_1 = 1:n/T
4      for r = 1:k    % k is the outer loop
5        %elementary subcomputation V
6        for i_2 = i_1*T : (i_1+1)*T
7          for j_2 = j_1*T : (j_1+1)*T
8            C(i_2,j_2) = C(i_2,j_2) + A(i_2,r)*B(r,j_2)
9          end [Mac: ---> end for ?] [Greg: i use matlab notation
               for (...) end. Ifyou prefer, i can chcange it to
               end for]
10       end
11     end
12   end
13 end
```

Listing 2. Pseudocode of I/O optimal sequential MMM

7 Optimal Parallel Matrix Multiplication

In this section we derive a communication optimal parallel schedule from the results derived in § 6.1. The explicit notion of reuse determines not only the sequential execution, as discussed shown there, but also the parallel scheduling.

7.1 Sequential and Parallel Schedules

Here we briefly describe the differences and connections between these two worlds. In the sequential schedule $\mathcal{S}$, the CDAG $G = (V, E)$ is partitioned into $H(S)$ subcomputations V_i connected (acyclically) by the data dependencies: $G = \bigcup_{i=1}^{H(S)} V_i$, $\mathcal{S} = \{V_1, \dots V_{H(S)}\}$. The I/O optimal *sequential* schedule $\mathcal{S}_{opt}$ maximizes the computational intensity ρ_i of the subcomputations V_i. The parallel schedule $\mathcal{P}$ distributes $\mathcal{S}$ among p processes $\mathcal{P} = \{\mathcal{D}_1, \cdots \mathcal{D}_p\}, \bigcup_{j=1}^{p} \mathcal{D}_j = \mathcal{S}$. The set $\mathcal{D}_j$ of all V_k assigned to process j is called a *local domain* of j. If the distributed schedule $\mathcal{S} = \mathcal{S}_{opt}$, then the parallel schedule $\mathcal{P}_{IOopt}$ is *I/O optimal*.

If two local domains $\mathcal{D}_k, \mathcal{D}_l$ are dependent, that is, $\exists_{u \in \mathcal{D}_k, v \in \mathcal{D}_l} : (u, v) \in E$, then u has to be *communicated* from process k to l. The total number of vertices communicated between all processes is the *total communication cost* of a schedule $W(\mathcal{P})$. Note that the dependencies between domains in parallel schedule $\mathcal{P}$ always increase the communication cost. However, the corresponding sequential schedule $\mathcal{S}$ may not increase the I/O cost, as during the pebbling of vertex v, vertex u may already contain a red pebble due to the data reuse. We say that the parallel schedule $\mathcal{P}_{Copt}$ is *communication optimal* if $W(\mathcal{P})$ is minimized among all possible parallel schedules. Clearly, if $W(\mathcal{P}) = 0$, then $\mathcal{P}$ is communication optimal. A parallel schedule may be I/O *and* communication optimal, which we call *data movement* optimal $\mathcal{P}_{DMopt}$.

For MMM, recall that the CDAG may be represented as the 3D iteration space $\mathcal{V}$ of size $[m \times n \times k]$. We call a schedule $\mathcal{P}$ *parallelized in dimension* $\mathbf{d}$ if each local domain $\mathcal{D}_j \in \mathcal{V}, j = 1 \dots p$ is a cuboid of size $[m/p_m \times n/p_n \times k/p_k]$, where $p_r = p$ if $r = \mathbf{d}$ and $p_r = 1$ otherwise, for $r \in \{m, n, k\}$. The schedule may also be parallelized in two $\mathbf{d}_1\mathbf{d}_2$ or three dimensions $\mathbf{d}_1\mathbf{d}_2\mathbf{d}_3$ for some p_m, p_n, p_k, such that $p_m p_n p_k = p$. We call p_m, p_n, p_k the *process grid* of a schedule.

[Greg: Not sure if this paragraph needed. For MMM it is obvious. This note is important only in general case of parallelizing different DAGs, where we cannot just blindly parallelize among the dependencies.] The important observation is that the parallelization of MMM in $\mathbf{k}$ dimension is possible only because the addition operation is associative, therefore processes may start the execution of local domains simultaneously- before receiving partial result inputs γ. We note that in general, one can only parallelize dependent domains $\mathcal{D}_j$ only if the operation induced by the dependencies is associative. Therefore, the parallelization requires additional knowledge about the CDAG, namely, which vertices correspond to associative operations.

7.2 Parallelization strategies for MMM

Intuitively, the I/O optimal sequential schedule $\mathcal{S}_{opt}$ consists of mnk/S elementary outer product calculations, arranged in $\sqrt{S} \times \sqrt{S} \times k$ "blocks" (Figure ??). The number $p_1 = mn/\sqrt{S}$ of dependency-free subcomputations V_i (that is, having no parents except of input vertices) in $\mathcal{S}_{opt}$ determines the maximum size of data movement optimal parallel schedule $|\mathcal{P}_{DMopt}|$. This is the maximum degree of parallelism up to which no additional data movement (that is, except of one required by the sequential lower bound) is performed. Each process has assigned a local domain $\mathcal{D}$ of size $|\mathcal{D}| = mnk/(pS)$, each of which requires $2mnk/(p\sqrt{S})$ I/O operations. There are no dependencies between local domains, therefore $W = 0$.

Above the threshold p_1, reducing the size of local domains $\mathcal{D}_j$ either enforce data dependencies between them - causing additional communication $W > 0$ (Figure ?? e), or not utilizing the whole available memory, making the schedule not I/O optimal $\mathcal{S} \neq \mathcal{S}_{opt}$ (Figure ?? d). Therefore, by the definition from § 7.1, there is no data movement optimal schedule $\mathcal{P}_{DMopt}$ for $p > p_1$.

We now analyze three possible parallelization strategies that either increase the I/O or the communication cost (or both) (Figure ??):

1. $\mathcal{P}ij$**:** The schedule is parallelized in dimensions $\mathbf{i}$ and $\mathbf{j}$. The process grid is $[m/a, n/a, 1]$, where $a = \sqrt{\frac{mn}{p}}$. Because all dependencies are parallel to dimension $\mathbf{k}$, no communication is required $W(\mathcal{P}ij) = 0$, therefore $\mathcal{P}ij$ is communication optimal. Because $a < \sqrt{S}$, the corresponding sequential schedule $\mathcal{S}ij$ is not I/O optimal.
2. $\mathcal{P}ijk$ The schedule is parallelized in all dimensions. The process grid is $[m/\sqrt{S}, n/\sqrt{S}, mnk/(pS)]$. Each subcomputation's V_i size is determined by Equation 14: the sequential schedule is I/O optimal. The parallelization in $\mathbf{k}$ dimension creates dependencies between local domains, enforcing communication $W(\mathcal{P}ijk) > 0$.
3. $\mathcal{P}cubic$ The schedule is parallelized in all dimensions. The process grid is $[m/a_c, n/a_c, mnk/a_c]$, where $a_c = \min\left\{\left(\frac{mnk}{P}\right)^{1/3}, \sqrt{\frac{S}{3}}\right\}$. Because $a_c < \sqrt{S}$, the corresponding sequential schedule $\mathcal{S}ij$ is not I/O optimal. Also, the parallelization in $\mathbf{k}$ dimension creates dependencies between local domains, enforcing communication $W(\mathcal{P}cubic) > 0$.

Comparison with other algorithms:

- If $m = n$, $\mathcal{P}ij$ scheme is reduced to the classical 2D decomposition (e.g., Cannon's algorithm [11] or SUMMA [1]).
- If $m = n = k$, $\mathcal{P}ijk$ reduces to 2.5D decomposition [52].
- CARMA [23] asymptotically reaches $\mathcal{P}cubic$ scheme, guaranteeing that the longest dimension of a local cuboidal domain is at most two times larger than the smallest one.

7.3 Data Movement Optimal Parallel Schedule

Observe that none of those schedules is optimal in the whole range of parameters. As discussed in § 6.1, 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 local domain is sent. Therefore, the optimal parallel schedule $\mathcal{P}_{Copt}$ minimizes the communication, that is, sum of the inputs' sizes plus the output size, under the sequential I/O constraint on subcomputations $\forall_{V_i \in \mathcal{P}_{Copt}} |\mathcal{I}_i| \leq S \wedge |\mathcal{O}_i| \leq S$.

The local domain $\mathcal{D}_j$ is a cuboid of size $[a \times a \times b]$. The optimization problem of finding $\mathcal{P}_{Copt}$ using the computational intensity (§ 5.5.3) can be formulated as follows:

$$\begin{gathered} \text{maximize } \rho = \frac{a^2 b}{ab + ab + a^2} \\ \text{subject to:} \\ a + a + a^2 \leq S \text{ (I/O constraint)} \\ a^2 b = \frac{mnk}{p} \text{ (load balance constraint)} \end{gathered} \tag{16}$$

where p is the number of processes. The inequality constraint $2a + a^2 \leq S$ is tight (binding) for $p \geq \frac{mnk}{S^{3/2}}$. Therefore, the solution to this problem (using the approximation 15):

$$\begin{aligned} a &= \min\left\{\sqrt{S}, \left(\frac{mnk}{p}\right)^{1/3}\right\} \\ b &= \max\left\{\frac{mnk}{pS}, \left(\frac{mnk}{p}\right)^{1/3}\right\} \end{aligned} \tag{17}$$

This can be intuitively interpreted geometrically as follows: if we imagine the optimal local domain "growing" with decreasing number of processes, then it is cubic until it is still "small enough" (its side is smaller than $\sqrt{S}$). After that point, its face in **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 $(S \geq \left(\frac{mnk}{p}\right)^{2/3})$.

Lemma 5. *Parallel schedule $\mathcal{P}_{Copt} = \{\mathcal{D}_j\}$ composed of p local domains $\mathcal{D}_j$ of size determined by Equation 17 is asymptotically communication optimal, that is, the communication volume is minimal across all parallel schedules $\mathcal{P}$ with p processes.*

Proof. The communication cost W of each local domain $\mathcal{D} \in \mathcal{P}_{Copt}$ is the sum of communications of all subcomputations $V_i \in \mathcal{D}$. Each subcomputation V_i requires $2a$ elements to be communicated (§ 6). The number of subcomputations is $|\mathcal{D}| = b$. Furthermore, the last subcomputation V_b must communicate a^2 outputs. Therefore, the total communication cost W is: t

$$W = \max\left\{\frac{2mnk}{\sqrt{S}} + S, 3\left(\frac{mnk}{p}\right)^{2/3}\right\}$$

This schedule is asymptotically communication optimal, as it reaches both memory-dependent and memory-independent lower bounds [7]. □

7.4 Discussion

7.4.1 I/O-Latency Trade-off

As showed in § 7.3, the local domain $\mathcal{D}$ of the I/O optimal schedule $\mathcal{P}$ is a cuboid of size $[a \times a \times b]$, where a, b are given by Equation 17. The I/O cost of each $\mathcal{D}$ is $2ab$ (size of the inputs) plus a^2 (output). The corresponding sequential schedule $\mathcal{S}$ is a sequence of b outer products of vectors of length a. Denote the size of the required inputs in each step by $I_{step} = 2a$. However, this corresponds to b steps of communication ($C_L = b$).

The number of steps (latency) is equal to the communication volume divided by the volume per step $C_L = W / I_{step}$. To reduce the latency, one either have to decrease W or increase I_{step}, under the memory constraint that $I_{step} + a^2 \leq S$ (otherwise we cannot fit 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 now express the I/O-latency trade-off:

$$\begin{gathered} W = 2ab + a^2 \\ C_L = \frac{b}{h} \\ a^2 + 2ah \leq S \text{(I/O constraint)} \\ a^2 b = \frac{mnk}{p} \text{(load balance constraint)} \end{gathered} \tag{18}$$

Solving this problem, we have $W = \frac{2mnk}{pa} + a^2$ and $C_L = \frac{mnk}{pa(S-a^2)}$, where $a \leq \sqrt{S}$. Increasing a we *reduce* I/O cost W and *increase* latency cost C_L. For minimal value of W (Lemma 5), $C_L = \max\{\frac{2a^2}{S-a^2}\}$, where $a = \min\{\sqrt{S}, (mnk/p)^{1/3}\}$. Based on our experiments, we observe that the I/O cost is vastly greater than the latency cost, therefore our schedule by default minimizes W and uses extra memory (if any) to reduce C_L. However, we leave that as a tunable parameter dependent on the hardware parameters.

7.4.2 Parallel efficiency

We now proceed to analyze the communication's *parallel efficiency* of all parallelization schemes. We define it as a ratio between the total data movement required by the sequential processor (Theorem 1) and sum of data movements performed by all parallel processes. More formally, if $Q(p, S)$ is the data movement cost per process of the algorithm using p processes, each of local fast memory size S, then the parallel efficiency metric for communication is defined as $E(p, S) = \frac{Q(1,S)}{pQ(p,S)}$.

In the proof of Lemma 5 we derive $pQ(p, S)$ for $\mathcal{P}Copt$. Using the same technique, we obtain $pQ(p, S)$ for the remaining schedules $\mathcal{P}ij$, $\mathcal{P}ijk$, and $\mathcal{P}cubic$. The results of our analysis is shown in Table 3 and Figure 4.

8 Implementation

In this Section we discuss various implementation insights that further increase the performance of our code. To leverage the RDMA mechanisms of current high-speed network interfaces, we use MPI one-sided interface(§ 8.4). We also use an block-cyclic data layout(§ 8.1), grid-fitting technique(§ 8.2) and an optimized binary broadcast tree using static information about the communication pattern(§ 8.3) together with the buffer swapping(§ 8.5). For the local matrix operations, we use BLAS routines. Together, those optimizations account for XX total runtime reduction.

8.1 Blocked Data Layout

8.2 Fitting Process Grid to Problem Size

Throughout the paper, we assume all operations required to assess the decomposition (divisions, roots, powers) result in natural numbers. We note that in practice it is rarely the case - e.g., the $[8704 \times 8704 \times 933888]$ domain, emerging from an exemplary water molecule simulation [22], when launched on the whole Piz Daint supercomputer with 1813 nodes, cannot be equally divided. Moreover, the memory size per node (64 GiB DDR, 90 MiB L3 cache) does not give the integer solution for the Equation 14 ($a = \sqrt{S+1} - 1$) for the local domain size. One may use the straightforward approach to take the floor and ceiling functions for non-natural numbers, but this may result in suboptimal results, as we demonstrate in the following example.

Assume that we multiply two square matrices with 65 processes (Figure 5). The only integer factors of 65 are 1, 5 and 13. Therefore, to utilize all the resources, the processes must be arranged in a $[13 \times 5 \times 1]$ grid (up to a permutation). On the other hand, if we decide *not to utilize all resources* and drop one process, then we can arrange remaining 64 in a $[4 \times 4 \times 4]$ grid. Such arrangement *trades communication for computation:* it increases the number of arithmetic operations per process by 1%, while decreasing the communication volume per process by 36%. However, because the dense MMM is compute bound, the decision which arrangement is better is not obvious. In most real life scenarios this trade-off may vary and the optimal decomposition depends on the computation and communication performance of a machine.

The existing algorithms either cannot handle such case (CARMA), require a user to manually specify the decomposition (ScaLAPACK), or simply discard the "non-fitting" processes using floor functions (Cyclops). Our algorithm balances this computation-communication trade-off by "stretching" the local domain size derived in § 7.3 to fit the global domain by adjusting its width, height and length. The range of this tuning (how far is the shape of the local domain from the optimal, increasing communication but utilizing more resources) depends on the hardware specification of the machine (peak FLOP/s, memory and network bandwidth) and it is a tunable parameter. For our experiments on Piz Daint we chose the maximal ratio between the optimal and applied size to be ??, accounting for up to ?? speedup compared to a straightforward (applying floor and ceiling functions) decomposition.

8.3 Communication Pattern

As shown in Algorithm 1, COMM executes $\frac{2ab}{S-a^2}$ steps. In each step, each process receives $\frac{S-a^2}{2a}$ elements of A and B. Therefore, the inputs has are broadcast among the **i** and **j** dimensions of the process grid. After the last step, the partial results of C are reduced among the **k** dimension. The communication pattern is therefore similar to ScaLAPACK or Cyclops.

To accelerate the collective communication, we implement our own binary broadcast tree, taking advantage of a static data layout, process grid and communication pattern. Knowing the initial data layout(§ 8.1) and the process grid(§ 8.2), we craft the binary reduction tree in all three dimensions **i**, **j** and **k** such that a distance in the grid between communicating processes is minimized.

Given a $[g_M, g_N, g_K]$ process grid, in each dimension we organize processes in a balanced binary tree with g_i leaves ($i \in \{m, n, k\}$). The communication (either distributing inputs A and B, or reducing output C) is performed in $\lceil log_2(g_i) \rceil$ steps, sending data of size $2^(s-1)D$, where s is the step number and D is the local size of data to be distributed.

8.4 One-Sided and Two-Sided Communication

To reduce the latency [Greg: cite something?] we implemented the communication using MPI one-sided [31]. This interface utilizes the underlying features of RDMA (Remote Direct Memory Access) mechanism, bypassing the OS on the receiver side and providing zero-copy communication (data sent is not buffered in a temporary address, instead, it is written directly to its destination).

Due to the static nature of the algorithm, all communication windows are pre-allocated using MPI_Win_Allocate with the size of maximum message in the broadcast tree $2^(s-1)D$ (§ 8.3). Communication in each step is performed using MPI_Put with passive synchronization (MPI_Lock / MPI_Unlock).

For compatibility reasons, as well as for the performance comparison, we also implemented a communication backend using MPI two-sided (message passing abstraction). In our tests on Piz Daint, we observed that the two-sided version was XX faster/slower than the one-sided.

8.5 Local buffer reuse

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

To alleviate this problem, we use our static knowledge and pre-allocate a single buffer per matrix 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 COMM (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.

9 Evaluation

We evaluate COMM 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 real-world "tall-and-skinny" cases with uneven number of available processes.

Comparison Targets As a comparison, we use a widely used ScaLAPACK library (commit number 3287fdd on May 17, 2018), as well as Cyclops Tensor Framework (commit number 6aaf037 on February 16, 2018) and the original CARMA implementation (commit number 7589212 on July 19, 2013).

Matrix Dimensions and Number of Processes We use both square ($m = n = k$) and "skinny" ($m = n << k$) matrices. Due to the space constraints, we skip the "flat" (e.g., $m << n = k$) case and limit ourselves to these two extreme cases. The matrix dimensions and number of processes 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 from a random distribution. For our "skinny" matrix, we consider the problem of water molecules interaction in the DFT simulation[22]. There, to simulate w molecules, the sizes of the matrices are $m = n = 136w$ and $k = 228w^2$. We use $w = 64$ as in the original paper, yielding $m = n = 8704, k = 933888$.

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

Programming Models As stated in (§ 8.4), we use both RMA and Message Passing models. CARMA, ScaLAPACK and CTF ?? use MPI two-sided (Message Passing).

Experimentation Methodology For each combination of parameters, we perform 20 runs. As all the algorithms use BLAS routines for local matrix computations, we discard the first run due to the BLAS setup overhead. We report median and 95% confidence intervals of the runtime.

Experimental Setup and Architectures We run our experiments on CSCS Piz Daint - Cray supercomputer equipped with 5320 XC50 heterogeneous nodes with NVIDIA Tesla P100 and 1813 XC40 nodes with dual-socket Intel Xeon E5-2695 v4 processors (3.30 GHz,45 MiB L3 shared cache, 64 GiB DDR3 RAM), interconnected with Cray Aries network.

Infrastructure and Implementation Details All implementations were compiled using the gcc 5.3.0 compiler. We use Cray-MPICH 3.1 implementation of MPI. The intra-node parallelism is handled internally by the MKL BLAS version 2017.4.196.

10 Results

[Greg: No results yet. Waiting for Marko]

11 Related work

Data movement minimization essentially may be divided into two aspects: across memory hierarchy (vertical, also called I/O minimization) and between parallel processes (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 two together, that is, analyzing trade-offs between communication optimal (distributed memory) and I/O optimal schedule (shared memory). We observe that even though algorithms of the second class tend to be problem size independent, distributed memory algorithms fix the domain sizes based on the even process grid (as discussed in Introduction - "top-down" vs "bottom-up" approach).

11.1 General I/O Lower Bounds

I/O optimization techniques date back to work by Sethi [49], where he used one color pebble game to model minimum number of registers required to perform a computation. It has been proven by Gilbert et.al [27] that this problem is P-SPACE complete. Despite the complexity, much work based on pebble game abstraction. Paul and Tarjan [45] studied time-space tradeoffs in pebble games. Dymond and Tompa [24] developed a two-player game to study parallel speedups. Most notably, Hong and Kung [34] introduced red-blue pebble game, on which our work is based. Elango et al. [25]

extended this work to red-blue-white game and Liu and Terman [40] proved that it is also P-SPACE complete. Chan [13] studied different variants of pebble games in context of memory space and parallel time. Aggarwal and Vitter [4] introduced a two-memory machine that models a blocked access and latency in an external storage. Large et al. [5] extended this model to a parallel machine. Solomonik et al. [51] combined the communication, synchronization and computation in their general cost model and applied it to several linear algebra algorithms.

11.2 Shared Memory Optimizations

Memory optimization for linear algebra includes such techniques as loop tiling and skewing [57], interchanging and reversal [56]. 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 [21] identified cases when this problem has polynomial complexity.

Toledo [55] in his survey on Out-Of-Core (OOC) algorithms analyzed various I/O minimizing techniques for dense and sparse matrices, like multiplication, factorization, or iterative methods. Mohanty [43] in his thesis optimized various OOC algorithms for QR, QZ decompositions and eigenvalue computation, expressing them with MMM kernels. Irony et al. [32] proved the I/O lower bound of classical MMM on a parallel machine. Ballard et al. [8] proved analogous results for Strassen's algorithm. This analysis was extended by Scott et al. [48] 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. [9] extended Aggarwal's external memory model [4] and showed I/O complexity of sparse matrix-vector (SpMV) multiplication under various data layouts. Greiner [29] extended those results and provided parallel I/O complexities of other sparse computations, like bilinear forms or sparse-sparse and sparse-dense matrix multiplications.

11.3 Distributed Memory Optimizations

Parallel algorithms for matrix multiplication dates back to the work of Cannon [11], which has been analyzed and extended many times, e.g, [30] [38]. In the presence of extra memory, Aggarwal et al. [3] included parallelization in the third dimension. Solomink and Demmel [52] extended this scheme to arbitrary range of available memory, effectively interpolating between Cannon's 2D and Agarwal's 3D scheme. Recursive, cache-oblivious MM algorithm was introduced by Blumofe et al. [10] and extended to rectangular matrices by Frigo et al. [26]. Demmel el al. [23] showed that their recursive CARMA algorithm achieves asymptotic complexity for all matrix and memory sizes.

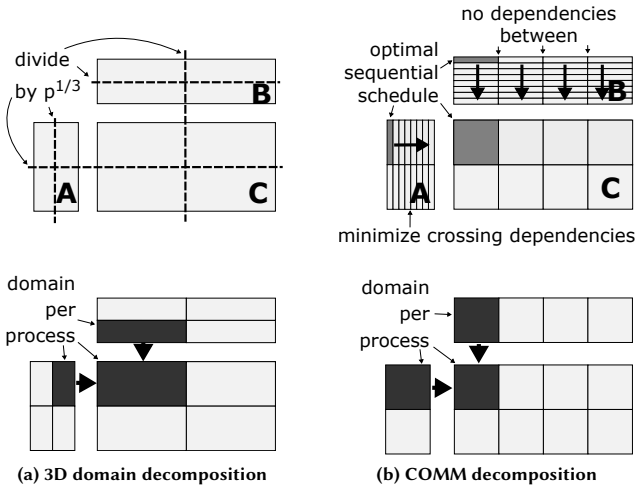


Figure 1. Domain decomposition using $p = 8$ processes. In the scenario (a), a straightforward 3D decomposition divides every dimension in $p^{1/3} = 2$. In the scenario (b), COMM starts by finding an optimal sequential schedule and then parallelizes it minimizing crossing dependencies. The total communication volume is reduced by 17% compared to the former strategy.

12 Conclusions

In our work we show that starting from the sequential I/O optimality we achieve optimal parallel communication optimality for any combination of input parameters. We introduce a new proof of sequential and parallel matrix multiplication data movement complexity, giving tight leading constants. We advocate that asymptotic analysis alone is not enough to draw conclusions on the real-life performance of the algorithms. We compare (name of our algorithm here) with existing state-of-the-art schedules, showing the superiority of the former both in terms of theoretical analysis and achieved performance.

13 Tighter General I/O Lower Bounds

To prove the I/O optimality of COMM, we start with deriving a new constructive proof of the MMM sequential I/O lower bound. We refine the existing asymptotic bound $Q = \Omega(\frac{mnk}{\sqrt{S}})$ by Hong and Kung [34], and $Q \geq \frac{2mnk}{\sqrt{S}} - 2S$ by Smith and van de Gein [50] and derive a new bound $Q \geq \frac{2mnk}{\sqrt{S}}$. Although we note that this is just an additive improvement over the latter result, we use the same mathematical machinery to generate an I/O optimal *parallel* schedule. This machinery extends the methods used by Hong and Kung, heavily relying on the *red-blue pebble game* and the *S-partition abstraction.*

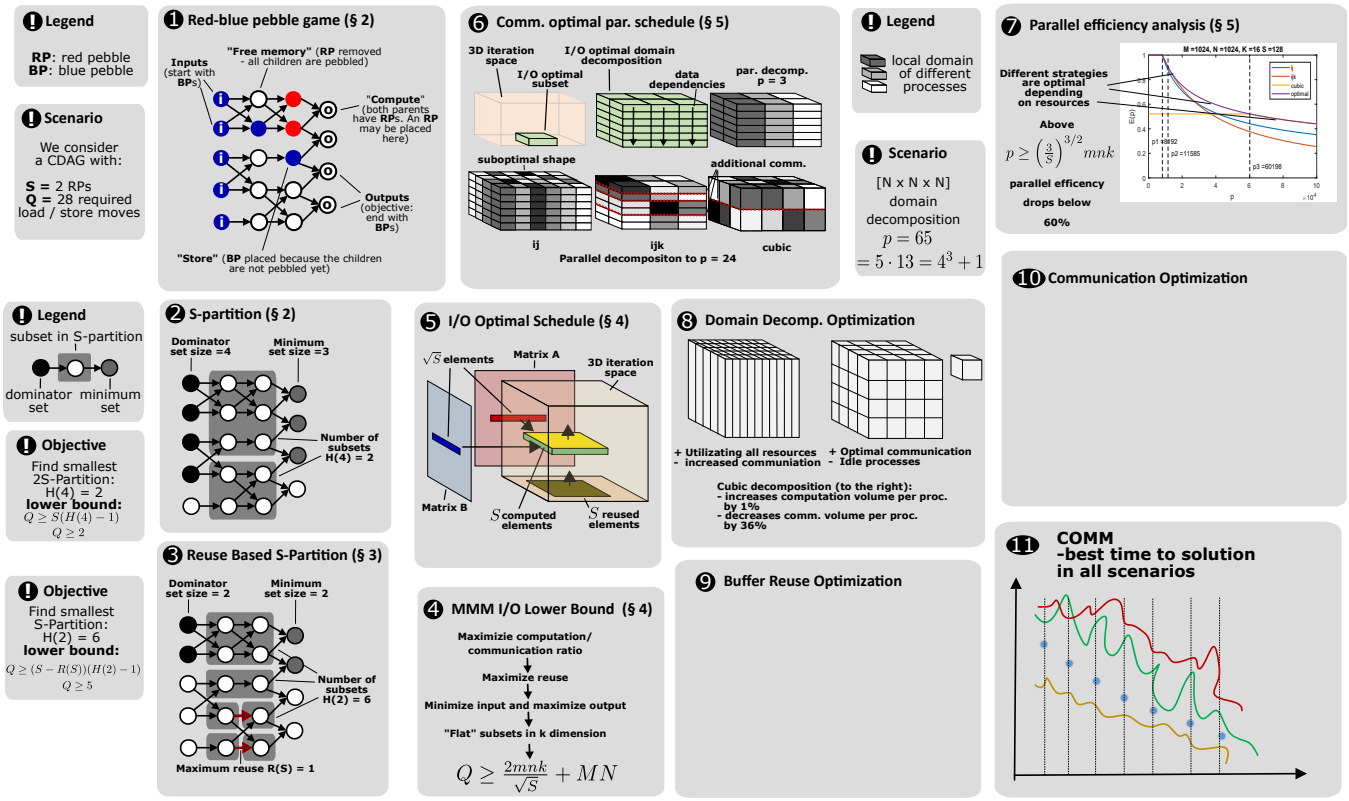


Figure 2. Achieving data movement optimal MMM in eleven simple steps: (I) I/O lower bounds and red-blue pebble game, (II) $2S$-partition lemma, (III) reuse-based lemma, (IV) MMM I/O lower bound, (V) I/O optimal schedule, (VI) communication optimal parallel schedule, (VII) parallel efficiency analysis, (VIII) buffer optimization, (IX) communication optimization, (X) process decomposition optimization, (XI) best time-to-solution result.

	2D [1]	2.5D [52]	CARMA [23]	Our work (§ 6.1)
process decomposition $[p_m \times p_n \times p_k]$	$\left[\sqrt{p} \times \sqrt{p} \times 1\right]$	$\left[\sqrt{p/c} \times \sqrt{p/c} \times c\right]$, $c = \frac{pS}{mk+nk}$	$[2^{a_1} \times 2^{a_2} \times 2^{a_3}]$, $a_1 + a_2 + a_3 = \log_2(p)$	$\left[\frac{m}{\sqrt{S}} \times \frac{n}{\sqrt{S}} \times \frac{k}{d}\right]$, $d = \frac{mnk}{pS}$
domain size	$\left[\frac{m}{\sqrt{p}} \times \frac{n}{\sqrt{p}} \times k\right]$	$\left[\frac{m}{\sqrt{p/c}} \times \frac{n}{\sqrt{p/c}} \times \frac{k}{c}\right]$	$\left[\frac{m}{2^{a_1}} \times \frac{n}{2^{a_1}} \times \frac{k}{2^{a_1}}\right]$	$\left[\sqrt{S} \times \sqrt{S} \times d\right]$
communication volume	$\frac{1}{\sqrt{p}}(mk + nk)$	$(mk + nk)\sqrt{\frac{I}{p^2 S}} + \frac{MNS}{I}$, $I = mn + mk + nk$	$2\min\left\{\sqrt{3}\frac{mnk}{p\sqrt{S}}, \left(\frac{mnk}{P}\right)^{2/3}\right\}$ $+\left(\frac{mnk}{P}\right)^{2/3}$	$\min\left\{S + 2 \cdot \frac{mnk}{p\sqrt{S}}, 3\left(\frac{mnk}{P}\right)^{2/3}\right\}$
"the easiest case": $m = n = k$, $S = 2\frac{n^2}{p}$, $p = 2^{3n}$	$\frac{2N^2}{\sqrt{p}}$	$\frac{2N^2}{\sqrt{p}}$	$2N^2\left(\sqrt{\frac{3}{2p}} + \frac{1}{2p^{2/3}}\right)$	$\frac{2N^2}{\sqrt{p}}$
"the hardest case": $m = n = \sqrt{p}$, $k = \frac{p^{3/2}}{4}$, $S = 2\frac{nk}{p^{2/3}}$, $p = 2^{3n+1}$	$\frac{p^{3/2}}{2}$	$\frac{p^{4/3}}{2} + p^{1/3}$	$\frac{3p}{4}$	$\frac{3-2^{1/3}}{2^{4/3}}p \approx 0.69p$

Table 1. The formal summary of 2D, 2.5D, CARMA, and COMM schemes for matrix-matrix multiplication (The 3D decomposition is a special case of 2.5D, where $c = p^{1/3}$). The most important symbols used here are described in Table 2. The "easiest case" is when the matrices are square and there is no extra memory for redundant copies of input data. The "hardest case" is when $k \gg m = n$ and there is space for extra $p^{1/3}$ copies. For simplicity, we assume that parameters are chosen such that all divisions have integer results.

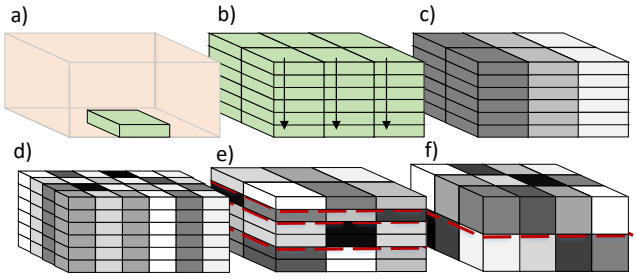


Figure 3. Different parallelization schemes of matrix multiplication arriving from the optimal sequential schedule. Up to six processes may be used optimally - above this limit we either increase I/O or communication. Shades of gray represent local domains of different processes. a) Global iteration domain (pink) and the optimal subset shape (green). b) Global iteration space scheduling (arrows represent data dependencies). c) Optimal parallelization using $p = 3$ processes. d-f) Suboptimal parallelization using $p = 24$ processes. From left to right, par in ij, par in ijk and cubic. Note the trade-off between I/O (shrinkage of local domain) and communication (dashed red lines showing parallelization and required communication in k dimension).

MMM	m, n, k	Matrix dimensions
	A, B	Input matrices, $A \in \mathbb{R}^{m \times k}$, $B \in \mathbb{R}^{k \times n}$
	$C = AB$	Output matrix, $C \in \mathbb{R}^{m \times n}$
	p	Number of processes
	S	Memory size per process
I/O complexity (§ 5.1)	G	A directed acyclic graph $G = (V, E)$
	S	Number of red pebbles (size of the fast memory)
	V_i	An i-th subcomputation of an S-partition
	$Dom(V_i)$, $Min(V_i)$	Dominator and minimum sets of subcomp. V_i
	$V_{R,i}$	Set of vertices containing red pebbles just before V_i starts and used by V_i (reuse set)
	$H(S)$	Cardinality of a smallest S-partition
	$R(S)$	Maximum reuse volume between subcomputations
	Q	Minimal number of I/O operations of any valid execution of G
	ρ_i	Computational intensity of V_i
scheduling (§ 6, § 7)	$\mathcal{V}$	3D iteration space of of MMM [57]
	i,**j**,**k**	orthonormal vectors spanning $\mathcal{V}$
	uv	plane spanned by vectors **u** and **v**
	$\phi_{uv}(V)$	projection of subspace V onto plane **uv**
	$\mathcal{I}_i$	Surface of a subcomputation V_i (size of the input)
	$\mathcal{S} = \{V_i\}$	Sequential schedule (an ordered set of V_i)
	$\mathcal{P} = \{\mathcal{S}_j\}$	Parallel schedule (a set of sequential schedules $\mathcal{S}_i$)
	W	I/O cost of a schedule

Table 2. The most important symbols used in the paper.

14 Figures

15 Tables

References

[1] Van De Geijn R. A. and Watts J. [n. d.]. SUMMA: scalable universal matrix multiplication algorithm. *Concurrency: Practice and Experience* 9, 4 ([n. d.]), 255–274.

[2] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. Tensorflow: a system for large-scale machine learning.. In *OSDI*, Vol. 16. 265–283.

[3] R. C. Agarwal, S. M. Balle, F. G. Gustavson, M. Joshi, and P. Palkar. 1995. A three-dimensional approach to parallel matrix multiplication. *IBM Journal of Research and Development* 39, 5 (Sep 1995), 575–582. https://doi.org/10.1147/rd.395.0575

[4] Alok Aggarwal and S. Vitter, Jeffrey. 1988. The Input/Output Complexity of Sorting and Related Problems. *Commun. ACM* 31, 9 (Sept. 1988).

[5] Lars Arge, Michael T Goodrich, Michael Nelson, and Nodari Sitchinava. 2008. Fundamental parallel algorithms for private-cache chip multiprocessors. In *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*. ACM, 197–206.

[6] Ariful Azad, Aydin Buluç, and John Gilbert. 2015. Parallel triangle counting and enumeration using matrix algebra. In *2015 IEEE International Parallel and Distributed Processing Symposium Workshop (IPDPSW)*. IEEE, 804–811.

[7] Grey Ballard, James Demmel, Olga Holtz, Benjamin Lipshitz, and Oded Schwartz. 2012. Strong scaling of matrix multiplication algorithms and memory-independent communication lower bounds. *arXiv preprint arXiv:1202.3177* (2012).

[8] Grey Ballard, James Demmel, Olga Holtz, and Oded Schwartz. 2012. Graph expansion and communication costs of fast matrix multiplication. *Journal of the ACM (JACM)* 59, 6 (2012), 32.

[9] Michael A Bender, Gerth Stølting Brodal, Rolf Fagerberg, Riko Jacob, and Elias Vicari. 2010. Optimal sparse matrix dense vector multiplication in the I/O-model. *Theory of Computing Systems* 47, 4 (2010), 934–962.

[10] Robert D Blumofe, Matteo Frigo, Christopher F Joerg, Charles E Leiserson, and Keith H Randall. 1996. An analysis of dag-consistent distributed shared-memory algorithms. In *Proceedings of the eighth annual ACM symposium on Parallel algorithms and architectures*. ACM, 297–308.

[11] Lynn Elliot Cannon. 1969. *A Cellular Computer to Implement the Kalman Filter Algorithm*. Ph.D. Dissertation. Bozeman, MT, USA. AAI7010025.

[12] Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. 1981. Register allocation via coloring. *Computer Languages* 6, 1 (1981), 47 – 57.

[13] S. M. Chan. 2013. Just a Pebble Game. In *2013 IEEE Conference on Computational Complexity*. 133–143. https://doi.org/10.1109/CCC.2013.22

[14] Franàoise Chatelin. 2012. *Eigenvalues of Matrices: Revised Edition*. Vol. 71. Siam.

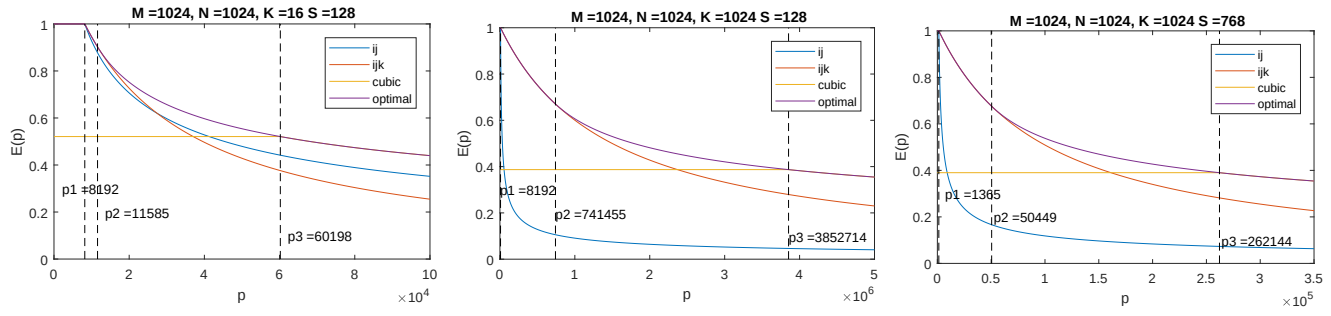
[15] J. Choi, J. Demmel, I. Dhillon, J. Dongarra, S. Ostrouchov, A. Petitet, K. Stanley, D. Walker, and R.C. Whaley. 1996. ScaLAPACK: a portable linear algebra library for distributed memory computers — design issues and performance. *Computer Physics Communications* 97, 1 (1996), 1 – 15. https://doi.org/10.1016/0010-4655(96)00017-3 High-Performance Computing in Science.

[16] Jaeyoung Choi, Jack J. Dongarra, L. Susan Ostrouchov, Antoine P. Petitet, David W. Walker, and R. Clint Whaley. 1996. Design and Implementation of the ScaLAPACK LU, QR, and Cholesky Factorization Routines. *Sci. Program.* 5, 3 (Aug. 1996).

[17] Jaeyoung Choi, Jack J Dongarra, Roldan Pozo, and David W Walker. 1992. ScaLAPACK: A scalable linear algebra library for distributed memory concurrent computers. In *Frontiers of Massively Parallel Computation, 1992., Fourth Symposium on the*. IEEE, 120–127.

[18] Don Coppersmith and Shmuel Winograd. 1990. Matrix multiplication via arithmetic progressions. *Journal of Symbolic Computation* 9, 3 (1990), 251 – 280. https://doi.org/10.1016/S0747-7171(08)80013-2 Computational algebraic complexity editorial.

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



(a) "flat" matrices **(b) square matrices, small S** **(c) square matrices, large S**

Figure 4. Parallel efficiency of different parallelization schemes. Three vertical dashed lines correspond to thresholds $p1 = mn/S$, $p2 = mnk/S^{3/2}$ and $p3 = (\frac{3}{S})^{3/2}mnk$ (Table 3). Note that for cases (b) and (c), increasing memory S six times reduces the number of processes required to fully saturate it drops from $p2 = 741455$ to $p2 = 50449$.

range	metric	par in **ij** dim	par. in **ijk** dim	cubic
$p \le \frac{mn}{S}$	loc. dom.	$[\sqrt{S} \times \sqrt{S} \times k]$	$[\sqrt{S} \times \sqrt{S} \times k]$	$\left[\sqrt{\frac{S}{3}} \times \sqrt{\frac{S}{3}} \times \sqrt{\frac{S}{3}}\right]$
	$pQ(p,S)$	$\frac{2mnk}{\sqrt{S}} + mn$	$\frac{2mnk}{\sqrt{S}} + mn$	$\frac{3\sqrt{3}mnk}{\sqrt{S}}$
	$E(p,S)$	1	1	$\frac{2K+\sqrt{S}}{3\sqrt{3}k}$
$\frac{mn}{S} < p$ $p \le \left(\frac{3}{S}\right)^{3/2} mnk$	loc. dom.	$\left[\sqrt{\frac{mn}{p}} \times \sqrt{\frac{mn}{p}} \times k\right]$	$\left[\sqrt{S} \times \sqrt{S} \times \frac{mnk}{Sp}\right]$	$\left[\sqrt{\frac{S}{3}} \times \sqrt{\frac{S}{3}} \times \sqrt{\frac{S}{3}}\right]$
	$pQ(p,S)$	$2K\sqrt{pmn} + mn$	$\frac{2mnk}{\sqrt{S}} + pS$	$\frac{3\sqrt{3}mnk}{\sqrt{S}}$
	$E(p,S)$	$\frac{mnk(\frac{2}{\sqrt{S}}+\frac{1}{k})}{2K\sqrt{MNp}+mn}$	$\frac{2mnk(1+\frac{\sqrt{S}}{2K})}{2mnk+S^{3/2}p}$	$\frac{2K+\sqrt{S}}{3\sqrt{3}k}$
$p > \left(\frac{3}{S}\right)^{3/2} mnk$	loc. dom.	$\left[\sqrt{\frac{mn}{p}} \times \sqrt{\frac{mn}{p}} \times k\right]$	$\left[\sqrt{S} \times \sqrt{S} \times \frac{mnk}{Sp}\right]$	$\left[a \times a \times a\right]$ $a = \left(\frac{mnk}{p}\right)^{1/3}$
	$pQ(p,S)$	$2K\sqrt{pmn} + mn$	$\frac{2mnk}{\sqrt{S}} + pS$	$3p\left(\frac{mnk}{p}\right)^{2/3}$
	$E(p,S)$	$\frac{mnk(\frac{2}{\sqrt{S}}+\frac{1}{k})}{2K\sqrt{MNp}+mn}$	$\frac{2mnk(1+\frac{\sqrt{S}}{2K})}{2mnk+S^{3/2}p}$	$\frac{2mnk(1+\frac{\sqrt{S}}{2K})}{3p^{1/3}\sqrt{S}(mnk)^{2/3}}$

Table 3. Total communication volume of MMM and its parallel efficiency of the different parallelization schemes. We assume that all the respective local domain sizes divide the global matrix sizes evenly (e.g., for cubic domain in range $p \le mn/S$, we assume that $\sqrt{S/3} = a_1M = a_2N = a_3K$ for $a_1, a_2, a_3 \in \mathbb{N}$). The optimal schedule (§ 7.3) switches from par. in ijk to cubic when $p \ge \frac{mnk}{S^{3/2}}$.

[20] Paolo D'Alberto and Alexandru Nicolau. 2008. Using recursion to boost ATLAS's performance. In *High-Performance Computing*. Springer, 142–151.

[21] Alain Darte. 1999. On the complexity of loop fusion. In *Parallel Architectures and Compilation Techniques, 1999. Proceedings. 1999 International Conference on*. IEEE, 149–157.

[22] Mauro Del Ben, Ole Schütt, Tim Wentz, Peter Messmer, Jürg Hutter, and Joost VandeVondele. 2015. Enabling simulation at the fifth rung of DFT: Large scale RPA calculations with excellent time to solution. *Computer Physics Communications* 187 (2015), 120–129.

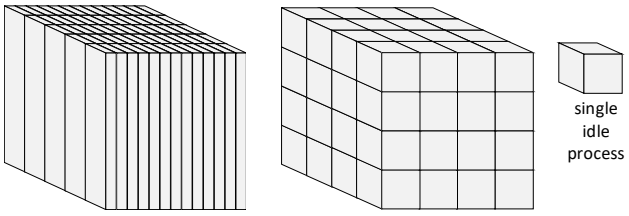
[23] J. Demmel, D. Eliahu, A. Fox, S. Kamil, B. Lipshitz, O. Schwartz, and O. Spillinger. 2013. Communication-Optimal Parallel Recursive Rectangular Matrix Multiplication. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. 261–272. https://doi.org/10.1109/IPDPS.2013.80

[24] Patrick W. Dymond and Martin Tompa. 1985. Speedups of deterministic machines by synchronous parallel machines. *J. Comput. System Sci.* 30, 2 (1985), 149 – 161. https://doi.org/10.1016/0022-0000(85)90011-X

[25] V. Elango, F. Rastello, L.-N. Pouchet, J. Ramanujam, and Sadayappan P. 2013. Data access complexity: The red/blue pebble game revisited. (2013).

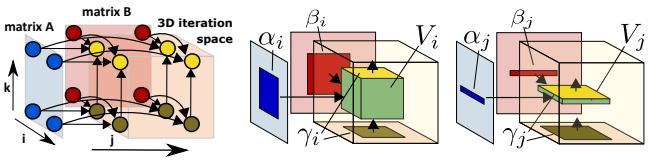
[26] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. 1999. Cache-oblivious algorithms. In *40th Annual Symposium on Foundations of Computer Science (Cat. No.99CB37039)*. 285–297.

[27] John R. Gilbert, Thomas Lengauer, and Robert Endre Tarjan. 1979. The Pebbling Problem is Complete in Polynomial Space. In *Proceedings of the Eleventh Annual ACM Symposium on Theory of Computing (STOC '79)*. ACM.



(a) $1 \times 3 \times 15$ grid (b) $4 \times 4 \times 4$ grid with one idle process

Figure 5. Process decomposition for square matrices and 65 processes. To utilize all resources, the local domain is drastically stretched (a). Dropping one process results in a symmetric grid (b) that increases the computation per process only by 1%, but reduces the communication by 36%.



(a) MMM CDAG (b) cubic decomposition (c) "flat" decomposition

Figure 6. An MMM CDAG forming a 3D iteration space $\mathcal{V} \subset \mathbb{Z}^3$. An input matrix A (blue vertices) is represented as its projection $\alpha = \phi_{ik}(\mathcal{V})$ on an ik plane - similarly, an input matrix B (red vertices) is a projection $\beta = \phi_{kj}(\mathcal{V})$ on the kj plane and an output matrix C (light yellow) is a projection $\gamma = \phi_{ij}(\mathcal{V})$ on the ij plane. Each vertex in this iteration space (except of the vertices in the bottom layer) has three parents - blue, red, and yellow and one yellow child (except of vertices in the top layer). $\alpha \cup \beta \cup \gamma$ form the dominator set $Dom(V_i)$. subcomputation $V_i \subset \mathcal{V}$ of an S-partition must satisfy $|Dom(V_i)| = |\alpha| + |\beta| + |\gamma| \leq S$ (number of inputs must be smaller than S. 6b optimal surface to volume subset shape. Note that in a subsequent subset computation only one of the three planes (blue, red or yellow) can be reused. 6c the optimal subset shapes when data reuse is considered. Observe that even though $|V_i| > |V_j|$, but $|V_i|/(|\alpha_i| + |\beta_i|) < |V_j|/(|\alpha_j| + |\beta_j|)$.

[28] Kazushige Goto and Robert A. van de Geijn. 2008. Anatomy of High-performance Matrix Multiplication. *ACM Trans. Math. Softw.* 34, 3, Article 12 (May 2008), 25 pages. https://doi.org/10.1145/1356052.1356053

[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 *Parallel Processing, 1993. ICPP 1993. International Conference on*, Vol. 3. 115–123. https://doi.org/10.1109/ICPP.1993.160

[31] T. Hoefler, J. Dinan, R. Thakur, B. Barrett, P. Balaji, W. Gropp, and K. Underwood. 2015. Remote Memory Access Programming in MPI-3. *ACM Transactions on Parallel Computing (TOPC)* (Jan. 2015). accepted for publication on Dec. 4th.

[32] Dror Irony, Sivan Toledo, and Alexander Tiskin. 2004. Communication Lower Bounds for Distributed-memory Matrix Multiplication. *J. Parallel Distrib. Comput.* 64, 9 (Sept. 2004), 1017–1026. https://doi.org/10.1016/j.jpdc.2004.03.021

[33] Dror Irony, Sivan Toledo, and Alexander Tiskin. 2004. Communication lower bounds for distributed-memory matrix multiplication. *J. Parallel and Distrib. Comput.* 64, 9 (2004), 1017 – 1026.

[34] Hong Jia-Wei and Hsiang-Tsung Kung. 1981. I/O complexity: The red-blue pebble game. In *Proceedings of the thirteenth annual ACM symposium on Theory of computing*. ACM, 326–333.

[35] Ken Kennedy and Kathryn S McKinley. 1993. Maximizing loop parallelism and improving data locality via loop fusion and distribution. In *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 301–320.

[36] Jeremy Kepner, Peter Aaltonen, David Bader, Aydın Buluç, Franz Franchetti, John Gilbert, Dylan Hutchison, Manoj Kumar, Andrew Lumsdaine, Henning Meyerhenke, et al. 2016. Mathematical foundations of the GraphBLAS. *arXiv preprint arXiv:1606.05790* (2016).

[37] François Le Gall. 2014. Powers of tensors and fast matrix multiplication. In *Proceedings of the 39th international symposium on symbolic and algebraic computation*. ACM, 296–303.

[38] Hyuk-Jae Lee, James P. Robertson, and José A. B. Fortes. 1997. Generalized Cannon's Algorithm for Parallel Matrix Multiplication. In *Proceedings of the 11th International Conference on Supercomputing (ICS '97)*. ACM, New York, NY, USA, 44–51. https://doi.org/10.1145/263580.263591

[39] Quanquan Liu. 2017. Red-Blue and Standard Pebble Games: Complexity and Applications in the Sequential and Parallel Models.

[40] Quanquan Liu and Christopher J. Terman. 2017. Red-Blue and Standard Pebble Games: Complexity and Applications in the Sequential and Parallel Models.

[41] L. H. Loomis and H. Whitney. 1949. An inequality related to the isoperimetric inequality. *Bull. Amer. Math. Soc.* 55, 10 (10 1949), 961–962. https://projecteuclid.org:443/euclid.bams/1183514163

[42] Carl D Meyer. 2000. *Matrix analysis and applied linear algebra*. Vol. 71. Siam.

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

[44] Andrew Y Ng, Michael I Jordan, and Yair Weiss. 2002. On spectral clustering: Analysis and an algorithm. In *Advances in neural information processing systems*. 849–856.

[45] W. J. Paul and R. E. Tarjan. 1978. Time-space trade-offs in a pebble game. *Acta Informatica* 10, 2 (01 Jun 1978), 111–115. https://doi.org/10.1007/BF00289150

[46] Donald W. Rogers. 2003. *Computational Chemistry Using the PC* (3 ed.). John Wiley & Sons, Inc., New York, NY, USA.

[47] John E Savage. 1995. Extending the Hong-Kung model to memory hierarchies. In *International Computing and Combinatorics Conference*. Springer, 270–281.

[48] Jacob Scott, Olga Holtz, and Oded Schwartz. 2015. Matrix multiplication I/O-complexity by path routing. In *Proceedings of the 27th ACM symposium on Parallelism in Algorithms and Architectures*. ACM, 35–45.

[49] Ravi Sethi. 1973. Complete Register Allocation Problems. In *Proceedings of the Fifth Annual ACM Symposium on Theory of Computing (STOC '73)*. ACM, New York, NY, USA.

[50] Tyler Michael Smith and Robert A. van de Geijn. 2017. Pushing the Bounds for Matrix-Matrix Multiplication. *CoRR* abs/1702.02017 (2017). arXiv:1702.02017 http://arxiv.org/abs/1702.02017

[51] Edgar Solomonik, Erin Carson, Nicholas Knight, and James Demmel. 2016. Trade-Offs Between Synchronization, Communication, and Computation in Parallel Linear Algebra Computations. *ACM Transactions on Parallel Computing (TOPC)* 3, 1 (2016), 3.

[52] Edgar Solomonik and James Demmel. 2011. Communication-Optimal Parallel 2.5D Matrix Multiplication and LU Factorization Algorithms. In *Euro-Par 2011 Parallel Processing*, Emmanuel Jeannot, Raymond Namyst, and Jean Roman (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 90–109.

[53] Edgar Solomonik, Devin Matthews, Jeff Hammond, and James Demmel. 2013. Cyclops tensor framework: Reducing communication and eliminating load imbalance in massively parallel contractions. In *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*. IEEE, 813–824.

[54] Volker Strassen. 1969. Gaussian Elimination is Not Optimal. *Numer. Math.* 13, 4 (Aug. 1969), 354–356. https://doi.org/10.1007/BF02165411

[55] Sivan Toledo. 1999. A survey of out-of-core algorithms in numerical linear algebra. *External Memory Algorithms and Visualization* 50 (1999), 161–179.

[56] Michael E. Wolf and Monica S. Lam. 1991. A Data Locality Optimizing Algorithm. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation (PLDI '91)*. ACM, New York, NY, USA, 30–44. https://doi.org/10.1145/113445.113449

[57] M. Wolfe. 1989. More Iteration Space Tiling. In *Proceedings of the 1989 ACM/IEEE Conference on Supercomputing (Supercomputing '89)*. ACM, New York, NY, USA, 655–664. https://doi.org/10.1145/76263.76337

[58] Qinqing Zheng and John D. Lafferty. 2016. Convergence Analysis for Rectangular Matrix Completion Using Burer-Monteiro Factorization and Gradient Descent. *CoRR* abs/1605.07051 (2016).