CA3DMM: A New Algorithm Based on a Unified View of Parallel Matrix Multiplication

Hua Huang
School of Computational Science and Engineering
Georgia Institute of Technology
Atlanta, Georgia, U.S.A.
huangh223@gatech.edu

Edmond Chow
School of Computational Science and Engineering
Georgia Institute of Technology
Atlanta, Georgia, U.S.A.
echow@cc.gatech.edu

***Abstract*—This paper presents the Communication-Avoiding 3D Matrix Multiplication (CA3DMM) algorithm, a simple and novel algorithm that has optimal or near-optimal communication cost. CA3DMM is based on a unified view of parallel matrix multiplication. Such a view generalizes 1D, 2D, and 3D matrix multiplication algorithms to reduce the data exchange volume for different shapes of input matrices. CA3DMM further minimizes the actual communication costs by carefully organizing its communication patterns. CA3DMM is much simpler than some other generalized 3D algorithms, and CA3DMM does not require low-level optimization. Numerical experiments show that CA3DMM has good parallel scalability and has similar or better performance when compared to state-of-the-art PGEMM implementations for a wide range of matrix dimensions and number of processes.**

***Index Terms*—matrix multiplication, communication optimization, parallel algorithm, high-performance computing**

I. Introduction

Matrix-matrix multiplication (MM) is one of the most fundamental computational kernels in scientific computing. It is used in linear algebra algorithms [1, 2, 3], graph processing [4, 5], computational chemistry [6, 7, 8, 9], and other domains. Accelerating matrix multiplication routines is of great importance and is widely studied.

The calculations in matrix multiplication have plenty of parallelism and highlight the importance of efficiently using parallel resources for obtaining high-performance. On distributed-memory platforms, the cost of transferring data between processing units (*communication*) has for a long time become relatively more expensive than arithmetic operations (*computation*). Therefore, minimizing communication costs in distributed-memory parallel algorithms is in the spotlight. In this work, we focus on minimizing communication costs for distributed-memory parallel dense general matrix-matrix multiplication (PGEMM).

Many algorithms have been proposed for reducing the data transfer size in PGEMM. The PGEMM communication cost lower bound has been discussed in multiple works [10, 11, 12, 13]. The widely used SUMMA algorithm [14] achieves optimal communication complexity for certain matrix dimensions or if no extra memory is present. With extra memory, 3D [15] and 2.5D [16] algorithms can achieve the communication cost lower-bound for matrix dimensions m, n, and k (see (1)) in certain ranges [17, 18]. The CARMA algorithm [17] was the first to generalize 1D, 2D, and 3D algorithms with recursive dimension-splitting to achieve an asymptotic communication lower bound for any matrix dimensions. The COSMA algorithm [18] adopted a new approach for finding a communication-optimal PGEMM parallelization scheme. COSMA achieves the communication cost lower bound (not just asymptotically) for any matrix dimensions and any number of processes.

Unfortunately, state-of-the-art PGEMM algorithms still have their obvious limitations. SUMMA is easy to understand and is widely used in linear algebra libraries, but it cannot utilize extra memory to reduce communication costs. CARMA requires the number of processes to be a power of two and requires special matrix distributions for its distributed-memory version. Such limitations make it extremely hard to use CARMA in real-world applications. As for COSMA, only its high-level principles and ideas are described in the literature, and implementing these ideas is complicated.

The comparison between CARMA and COSMA in [18] also indicates an important issue. When then number of processes is a power of two, CARMA and COSMA use the same 3D process grid and therefore have the same theoretical communication cost for many matrix dimensions and numbers of processes, but COSMA usually performs better than CARMA. Having the optimal process grid and reaching the theoretical communication cost lower bound are necessary but not sufficient conditions for achieving the best performance. Matrix partitioning and the resulting communication patterns have very large impacts on the performance of PGEMM. They should be analyzed and designed carefully to achieve high performance.

In this paper, we present the Communication-Avoiding 3D Matrix Multiplication (CA3DMM) algorithm, a novel PGEMM algorithm that achieves the communication cost lower bound with a simple but efficient approach. CA3DMM first computes an optimal or near-optimal 3D process grid based on the dimensions of the input matrices. Then CA3DMM performs the matrix multiplication using multiple low-rank updates. Both the calculations in each low-rank update and the computations of different low-rank updates are parallelized. In contrast, SUMMA [14] performs different low-rank updates sequentially, although each low-rank update is performed in parallel. CA3DMM is based on a unified view of distributed matrix multiplication that generalizes 1D, 2D, and 3D algorithms, so CA3DMM can be reduced to 1D, 2D, or 3D algorithms in different cases. Numerical experiments show that CA3DMM has good parallel scalability and has similar or better performance when compared to state-of-the-art PGEMM implementations for a wide range of matrix dimensions and number of processes.

II. Background

Consider a general dense matrix-matrix multiplication

$$C = A \times B,\ A \in \mathbb{R}^{m\times k},\ B \in \mathbb{R}^{k\times n},\ C \in \mathbb{R}^{m\times n}. \quad (1)$$

PGEMM algorithms can be categorized into 1D, 2D, and 3D (2.5D) algorithms.

1D algorithms partition only the m-dimension, n-dimension, or the k-dimension of (1). If the m-dimension / n-dimension is partitioned, matrix B / A will be replicated in the algorithm. If the k-dimension is partitioned, a reduction operation is needed to obtain the final C matrix. Matrix multiplications involving tall-and-skinny matrices usually use 1D algorithms.

2D algorithms partition A, B, and C matrices in 2D and organize processes in a 2D grid. The first 2D algorithm was proposed by Cannon [19], which works for square process grids. Later, the PUMMA algorithm [20] was developed and supported rectangular matrices, transposed matrices, non-square 2D process grids, and different matrix distribution layouts. The SUMMA algorithm [14] further reduced communication costs with communication-computation overlap. SUMMA is the most widely-used (2D) algorithm and it is implemented in the ScaLAPACK library [21] and the SLATE library [22].

2D algorithms can be viewed as applying a 2D partitioning to the C matrix and computing each block of the C matrix with only one process. The original 3D algorithm [15] and the 2.5D algorithm [16] further extract parallelism from the "k-dimension of computation" and reduce communication costs. In the original 3D algorithm and the 2.5D algorithm, a 2D partition is still applied to the C matrix, but two or more processes compute the partial results of the same C matrix block and use reduce-sum to obtain the final result. The original 3D algorithm uses a cuboidal process grid and a 3D partitioning of work. It uses more memory than 2D algorithms since A and B are replicated across the m-dimension and the n-dimension of the process grid, respectively, and C has multiple partial results across the k-dimension. Compared to 2D algorithms, the communication cost of the original 3D algorithm is reduced from $\mathcal{O}(N^2/P^{1/2})$ to $\mathcal{O}(N^2/P^{2/3})$, where N is the (square) matrix dimension, and P is the number of processes. As a trade-off, the memory requirement of the original 3D algorithm increases from $\mathcal{O}(N^2/P)$ to $\mathcal{O}(N^2/P^{2/3})$. The 2.5D algorithm bridges the gap between 2D and the original 3D algorithm by introducing a parameter c to control the number of replicated copies of the input matrices for use in the original 3D algorithm.

However, the original 3D algorithm and the 2.5D algorithm are not optimal for all matrix dimensions. Demmel *et al.* showed that these approaches usually perform poorly when one of the matrix dimensions is much larger than the other two dimensions [17]. The authors then proposed CARMA, a recursive algorithm that achieves asymptotic communication cost lower bounds for all dimension and memory size configurations. In each step, CARMA bisects the largest dimension of the current problem and assigns each resulting subproblem to half of the processes. The process is continued recursively until a single process is assigned to each subproblem. Each bisection corresponds to a replication of an A or B matrix block, or an all-reduction of a C matrix block. This recursive bisection approach requires the number of processes to be a power of two and also requires special matrix distributions in any MPI implementation. These factors limit the application of CARMA in practice.

Recently, Kwasniewski *et al.* proposed COSMA [18], a communication optimal PGEMM algorithm for all combinations of parameters. COSMA uses a "bottom-up" approach to minimize the total number of words transferred during the matrix multiplication. It first finds an optimal or near-optimal sequential matrix multiplication strategy "by explicitly modeling data reuse in the red-blue pebble game". Then, an optimal parallel scheme is derived from the sequential scheme by solving an optimization problem. COSMA implements its own binary broadcast tree to take advantage of their special data layout and utilizes one-sided asynchronous communication operations to further reduce its communication latency.

The 2.5D matrix multiplication algorithm [23], using any number of processes, is implemented in the Cyclops Tensor Framework (CTF) [24] for parallel tensor calculations. CTF is optimized for distributed-memory dense and sparse tensor operations.

III. The CA3DMM Algorithm

In this section, we present the Communication-Avoiding 3D Matrix Multiplication (CA3DMM) algorithm, a PGEMM algorithm based on a unified view of parallel matrix multiplication. CA3DMM works for all combinations of matrix dimensions and process numbers. CA3DMM is designed with a top-down approach, so it is easy to understand and implement. Meanwhile, CA3DMM achieves optimal or near-optimal communication costs with the same memory usage as the original 3D algorithm.

A. A Unified View of MM and the Minimal Communication Parallelization

In this work, we do not consider possible special properties (for example, symmetry) of A, B, and C matrices. We only discuss real matrices here for convenience, and the conclusions can be applied to complex matrix multiplication. We do not discuss fast matrix multiplication algorithms, for example, the Strassen algorithm [25].

The number of arithmetic operations (scalar additions and multiplications) and the number of matrix elements to be loaded and updated correspond to the volume and surface area of an $m \times k \times n$ cuboid, respectively. On the cuboid, we call an $m \times k$ face an "A-face", a $k \times n$ face a "B-face", and an $m \times n$ face a "C-face". A subdomain (cuboid block) of the cuboid corresponds to a sub-task in 3D matrix multiplication. The projections of a subdomain on the A-face, B-face, and C-face correspond, respectively, to the A and B matrix blocks required for computing a C matrix block. Figure 1 illustrates the connection between matrix multiplication and a cuboid.

Based on the cuboid view of MM, the parallelization of a MM is equivalent to partitioning the cuboid into multiple subdomains and assigning subdomains to processes. To balance the flops across processes, the total volume of the subdomains on each process should be mnk/P. The total number of matrix elements to be transferred (read and updated) by all processes equals half of the sum of all subdomains' surface area minus the area of A, B, and C. We ignore the subtracted term in our analysis since it is a constant. We assume that A and B are distributed on all P processes at the beginning, and C is distributed on all P processes at the end. In other words, all P processes together have only one copy of A and B at the beginning, and only one copy of C at the end. This assumption holds for all existing 2D, 2.5D, and 3D algorithms.

Denote the size of a 3D process grid as $p_m \times p_k \times p_n$, where positive integers p_m, p_k, p_n denote the number of processes along the m-dimension, k-dimension, and n-dimension, respectively. We further assume each process has only one subdomain. Having two or more subdomains on each process with the same total volume will have larger total surface area. The size of the subdomains along the m-dimension will be either $\lceil m/p_m \rceil$ or $\lfloor m/p_m \rfloor$, and similarly for the n-dimension and k-dimension. To minimize the total number of transferred matrix elements, we need to minimize the total surface area of all the subdomains. Among cuboids that have the same volume, the perfect cube has the smallest surface area. Since the total volume of each subdomain is fixed as mnk/P, we want to make the shape of each subdomain as close to a cube as possible. Denote $d_m = m/p_m$, $d_k = k/p_k$, $d_n = n/p_n$. For convenience of analysis, we assume d_m, d_k, and d_n are integers. When

$$d_m = d_k = d_n = \left(\frac{mnk}{P}\right)^{1/3}, \tag{2}$$

a subdomain has the minimal surface area $6(mnk)^{2/3}P^{-2/3}$, and the sum of all subdomains' surface area is

$$S_{total} = 6(mnk)^{2/3}P^{1/3}. \tag{3}$$

In practice, one can enumerate all possible process grid sizes $p_m \times p_k \times p_n$ and find the optimal solution that minimizes the sum of all subdomains' surface area. Combined with the complexity analysis in Section III-D, (3) matches the I/O complexity lower bound in [10].

For some values of P, for example, prime numbers, it is impossible to find a good 2D or 3D process grid size that achieves near-optimal communication cost. Previous studies have shown that the performance of PGEMM is bound by communication when scaling to a large number of processes, even if communication-optimal algorithms are used. Thus, a PGEMM algorithm can allow some processes to be idle in matrix multiplication, making the communication cost close to optimal with a small extra computation cost. This technique was recently used in the COSMA algorithm [18].

B. The CA3DMM Algorithm: Communication Patterns and Matrix Partitionings

The CA3DMM algorithm is based on the aforementioned unified view of matrix multiplication and 3D process grid selection. In CA3DMM, we enumerate all possible $p_m \times p_k \times p_k$ combinations and find a solution that minimizes

$$S_{total} = 2(p_m kn + p_n mk + p_k mn) \tag{4}$$

with constraint

$$l \cdot P \leq p_m \times p_k \times p_n \leq P, \tag{5}$$

where $l = 0.95$ is a tunable parameter. Using a larger l allows fewer processes to be idle but also makes it harder to find a valid solution under the constraint. A sub-target

$$\max\ p_m \times p_k \times p_n \tag{6}$$

is also used to maximize the utilization of processes but its priority is lower than that of (4).

Having an optimal or near-optimal 3D process grid is just half of building the CA3DMM algorithm. Different communication patterns can be used for the same process grid, and their communication costs can be very different. We interpret the 3D process grid with a unified view of parallel matrix multiplication: *a matrix multiplication is p_k independent rank-(k/p_k) updates to a zero matrix*. More precisely, each set of $p_m \times p_n$ processes forms a *k-task group* and computes a rank-(k/p_k) update using a 2D algorithm. Then, all k-task groups reduce-sum p_k rank-(k/p_k) updates to obtain the final C matrix. This view of parallel matrix multiplication is a unified view since it can fall back to optimal 2D or 1D algorithms if necessary. Even for degenerate problems, for example, rank-1 update ($k = 1$), matrix-vector product ($n = 1$ or $m = 1$), and vector inner product ($m = n = 1$), the obtained algorithms are the same as the optimal algorithms.

We use Cannon's algorithm [19] in CA3DMM to compute rank-(k/p_k) updates. Section III-E further discusses the choice of the 2D algorithm. The original Cannon's algorithm only works with a square process grid, so it is usually not possible to directly use the original Cannon's algorithm in a k-task group. The generalized Cannon's algorithm [26] (GCA) is a possible solution. However, GCA is designed for block-cyclic distributed matrices and it also has some

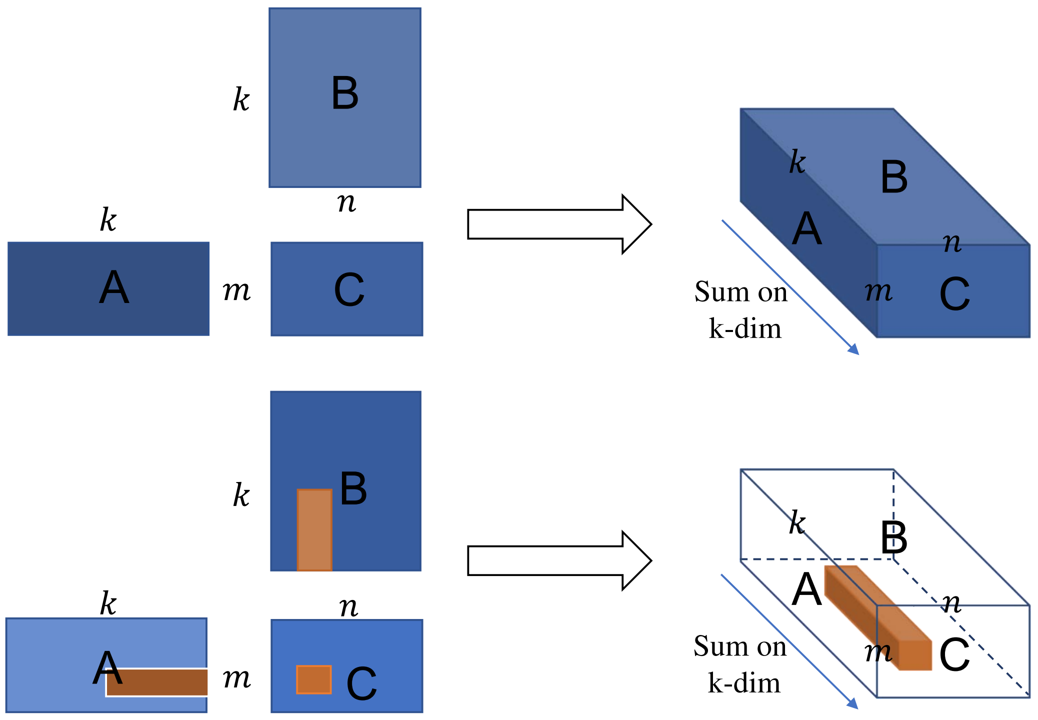


Fig. 1: Illustration of the connection between matrix multiplication and a cuboid. A unit volume in the cuboid corresponds to one scalar multiplication and addition. The surface area of a cuboid subdomain corresponds to the number of A and B matrix elements to be loaded and the number of C matrix elements to be updated.

restrictions on the matrix dimensions. Instead of using GCA, we add an intermediate layer between the k-task group and the original Cannon's algorithm by allowing CA3DMM to use a sub-optimal 3D process grid. We add a constraint to the 3D grid size:

$$\mathrm{mod}(\max(p_m, p_n),\ \min(p_m, p_n)) = 0. \tag{7}$$

Each k-task group is further split into

$$c = \max(p_m, p_n)/\min(p_m, p_n) \tag{8}$$

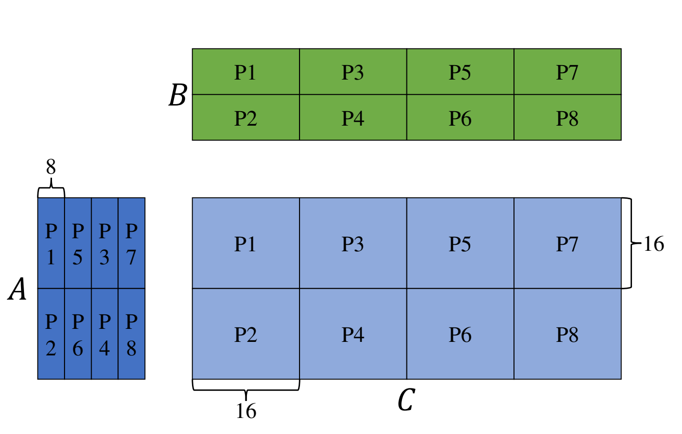
Cannon groups with s^2 processes in each Cannon group, $s = \min(p_m, p_n)$. A block of A or B is replicated c times across Cannon groups in a k-task group. If $c = 1$, the initial distributions of A and B in each k-task group are the distributions of the original Cannon's algorithm. If $c > 1$ and A / B need to be replicated, each A / B matrix block in the original Cannon's algorithm initial distribution for c^2 processes is further row-partitioned or column-partitioned into c sub-blocks. Each process in a k-task group stores a sub-block of A / B and a block of A / B initially. Then A / B is replicated by using an allgather operation before performing Cannon's algorithm. This scheme guarantees that A and B are 2D partitioned among all P processes initially. It also balances the memory usage for storing the initial A and B. In Cannon's algorithm, each process first sends its A and B blocks to two processes in the same process row and column (the "initial skewing"). In the first $s - 1$ steps, each process circularly shifts its current A and B blocks to its left and upper neighbor processes, respectively. Therefore, Cannon's algorithm only requires neighbor communications with fixed patterns.

The reduce-sum of p_k rank-(k/p_k) updates is simple and independent of the choice of 2D algorithm. All p_k processes having the partial results of the same C block reduce-scatter sum (equivalent to first reduce-summing the message, then scattering the results) their partial results, and the final C block is row-partitioned or column-partitioned into p_k sub-blocks. This scheme also guarantees the final C matrix is 2D partitioned among all $p_m \times p_k \times p_n$ active processes.

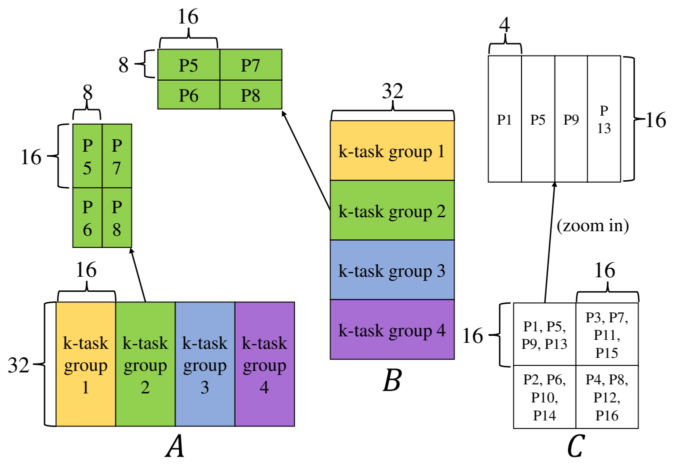
We provide three simple examples to help the reader understand the initial and final partitioning of matrices in CA3DMM. We use MATLAB colon notation to indicate matrix blocks in the examples.

Example 1. $m = 32$, $k = 16$, $n = 64$, $P = 8$. The optimal process grid is $p_m = 2$, $p_k = 1$, $p_n = 4$. Since $p_k = 1$, CA3DMM falls back to 2D Cannon's algorithm. Since $c = p_n/p_m = 2$, matrix A needs to be replicated. Block $A(1:16, 1:16)$ is replicated across processes P_1 and P_5, initially P_1 has $A(1:16, 1:8)$ and P_5 has $A(1:16, 9:16)$. Similarly, block $A(17:32, 1:16)$ is replicated across P_2 and P_6, initially P_2 has $A(17:32, 1:8)$ and P_6 has $A(17:32, 9:16)$. Figure 2a shows the complete partitionings.

Example 2. $m = n = 32$, $k = 64$, $P = 16$. The optimal process grid is $p_m = p_n = 2$, $p_k = 4$. Processes $P_{1\leq i\leq 4}$ form the first k-task group and compute $A(:, 1:16) \times B(1:16, :)$, processes $P_{5\leq i\leq 8}$ form the second k-task group and compute



(a) $m = 32$, $k = 16$, $n = 64$, $P = 8$, $p_m = 2$, $p_k = 1$, $p_n = 4$



(b) $m = n = 32$, $k = 64$, $P = 16$, $p_m = p_n = 2$, $p_k = 4$

Fig. 2: CA3DMM initial and final matrix partitioning examples.

$A(:, 17 : 32) \times B(17 : 32, :)$, and so on. Processes P_1, P_5, P_9, P_{13} have partial results of $C(1 : 16, 1 : 16)$. After reduce-scatter, P_1 has the final $C(1 : 16, 1 : 4)$, P_5 has the final $C(1 : 16, 5 : 8)$, P_9 has the final $C(1 : 16, 9 : 12)$, and P_{13} has the final $C(1 : 16, 13 : 16)$. Figure 2b shows the complete partitionings.

Example 3. $m = n = 32$, $k = 64$, $P = 17$. The optimal process grid is $p_m = p_n = 2$, $p_k = 4$. Processes P_{17} only participates in matrix redistribution. Processes $P_{1 \le i \le 16}$ have the same roles as in *Example 2*.

The initial and final distributions of A, B, and C matrices in CA3DMM are 2D distributions, but these distributions are usually unable to map to a natural row-major or column-major 2D process grid. In any case, the applications using CA3DMM may have different matrix distributions, so the matrices need to be redistributed before and after calling CA3DMM. Such matrix layout conversions are common in 3D and 2.5D algorithms. The original 3D algorithm and the 2.5D algorithm use natural 2D distributions of A, B, and C. However, the matrices are only stored on a subset of processes. CARMA and COSMA also have algorithm-specific initial and final matrix distributions. COSMA supports user-defined input and output matrix partitionings and the 2D block cyclic partitioning used in ScaLAPACK with an internal matrix redistribution library. CA3DMM also adopts a small subroutine to redistribute the input A and B matrices from user-defined distributions to CA3DMM initial distributions and to redistribute the final C matrix to the user-defined distribution. Further, CA3DMM utilizes the redistribution steps of A and B for computing

$$C = op(A) \times op(B), \; op() = \text{transpose or no-transpose}.$$

We note that distributed matrix layout conversion and handling the transpose operation in PGEMM are not the major concerns in this work, so the matrix redistribution subroutine in CA3DMM is not fully optimized. We leave this as a topic for future study.

Algorithm 1 shows the complete CA3DMM algorithm. For simplicity, CA3DMM organizes the $p_m \times p_n \times p_k$ 3D process grid in a "column-major" way, i.e., all MPI processes in the same k-task group and the same Cannon group have contiguous MPI ranks. We note that Figure 2 shows the partitionings of A and B matrices *after* the redistribution (step 2 in Algorithm 1) and the partitioning of C *before* the redistribution (step 8 in Algorithm 1).

Algorithm 1 CA3DMM algorithm

```
Input: 1D or 2D partitioned $A$ and $B$ matrices distributed on $P$ processes
Output: 2D partitioned $C = op(A) \times op(B)$ distributed on $P$ processes
1: Find 3D process grid $p_m \times p_k \times p_n$ by minimizing (4) and maximizing (6) with constraints (5) and (7).
2: Organize the first $p_m \times p_k \times p_n$ processes as $p_k$ k-task group(s), each active process computes its required initial block of $A$ and $B$ matrices and the final $C$ matrix block. The last $P - p_m \times p_k \times p_n$ process(es) remain idle outside the redistribution steps.
3: Each k-task group organizes its $p_m \times p_n$ processes as $c = \max(p_m, p_n)/\min(p_m, p_n)$ Cannon group(s).
4: All $P$ processes participate in the redistribution of $A$ and $B$ matrices.
5: Replicate a block of $A$ or $B$ in each k-task group using allgather if $c > 1$.
6: Each Cannon group performs Cannon's algorithm to compute a partial result of a $C$ block.
7: For each group of $p_k$ process(es) holding partial results of the same $C$ block, form the final $C$ matrix blocks using reduce-scatter if $p_k > 1$.
8: All $P$ processes participate in the redistribution of the $C$ matrix.
```

C. Differences Between COSMA and CA3DMM

Both COSMA and CA3DMM have optimal or near-optimal communication costs for all matrix dimensions and any number of processes. In many cases, COSMA and CA3DMM may use the same optimal 3D process grid, but COSMA and CA3DMM organize the communication and computation in different ways. We discuss the differences between COSMA and CA3DMM in this section.

To compare COSMA with CA3DMM, we first analyze the actual behaviors of the COSMA source code since the COSMA paper only discusses the high-level ideas without presenting detailed operations. The actual behaviors in the COSMA source code are very similar to the CARMA algorithm. In some sense, the COSMA source code can be considered as a generalized CARMA algorithm implementation.

The COSMA source code first finds an optimal or near-optimal 3D process grid $p_m \times p_k \times p_n$ s.t. $m/p_m \approx k/p_k \approx n/p_n$ by enumerating all possible solutions. It does not explicitly solve an optimization problem described in the COSMA paper to find a optimal subdomain of size $a \times b \times a$, where $m/p_m = n/p_n = a$ and $k/p_k = b$. Then, the COSMA source code factorizes p_m, p_n, and p_k to obtain its parallel strategy containing one or multiple steps. Consider *Example 2* in Section III-B. The COSMA source code generates a parallel strategy with three steps: (1) k-dimension splitting of size 4, (2) m-dimension splitting of size 2, and (3) n-dimension splitting of size 2. CARMA only bisects the largest dimension of the current problem and the process group in each step. COSMA generalizes the bisection and partitions the largest dimension of the current problem into multiple parts. Correspondingly, COSMA replaces the point-to-point communications in CARMA with collective operations. Specifically, in each step, if the m / n dimension is partitioned into s parts, COSMA uses an all-gather operation involving s processes to replicate the B / A matrix; if the k dimension is partitioned into s parts, COSMA uses a reduce-scatter operation involving s processes for s partial C matrix results and obtains a final C matrix or another partial C result.

In general, COSMA first replicates A and/or B in one or multiple steps using all-gather operations, then calculates one local matrix multiplication to obtain a partial C result block on each process, and finally reduces the partial C results to get the final C matrix. The original 3D algorithm follows the same procedure, but it uses one broadcast operation to replicate A and one broadcast operation to replicate B. In contrast, CA3DMM does not complete all replications of A and/or B before local computations. CA3DMM organizes a parallel matrix multiplication as multiple independent low-rank updates. The communications and computations in each low-rank update are pipelined and overlapped in the Cannon's algorithm stage. The partial C result reduction in CA3DMM is the same as that in COSMA.

D. Complexity Analysis of CA3DMM

In this section, we analyze the communication size, communication latency, and memory usage of CA3DMM. We assume $p_m \times p_k \times p_n = P$, $\min(p_m, p_n, p_k) > 1$, and (4) equals 1. We further assume butterfly network collectives for communication size and latency analysis [27], which are optimal or near-optimal in the $\alpha - \beta$ model. The cost of collective operations (assuming "large" messages) used in the analysis are listed here, where n is the message size, P is the number of processes, α is network latency, and β is the inverse of network bandwidth:

$$\begin{aligned} T_{allgather}(n, P) &= \alpha \log_2(P) + \beta n \frac{P-1}{P}, \\ T_{broadcast}(n, P) &= \alpha \left(\log_2(P) + P - 1\right) + 2\beta n \frac{P-1}{P}, \\ T_{reduce-scatter}(n, P) &= \alpha(P-1) + \beta n \frac{P-1}{P}. \end{aligned}$$

We also assume that steps 4 and 8 in Algorithm 1 can be skipped to make our cost analysis comparable to those in the literature.

We define the communication size Q as the maximum number of matrix elements transferred by any process in Algorithm 1. Based on (3) and the assumptions in this section, we immediately obtain

$$Q = 3\left(\frac{mnk}{P}\right)^{2/3}. \tag{9}$$

We define the communication latency L as the maximum number of messages sent by any process in Algorithm 1. Define $p_s = \min(p_m, p_n)$. In Algorithm 1, steps 5, 6, and 7 have latency $\log_2(c)$, p_s, and $p_k - 1$, respectively. Thus, the communication latency is

$$L = \log_2(c) + p_s + p_k - 1. \tag{10}$$

Fixing m, n, k and increasing P, (2) shows that the ratios p_m/p_k, p_k/p_n, and c remain unchanged, so $p_s = uP^{1/3}$ where u is a constant, and thus $L = \mathcal{O}\left(P^{1/3}\right)$.

We define the memory usage S as the maximum number of matrix elements stored on any process in Algorithm 1. We first assume $m \le n$. After step 4, each process stores $(mk + kn)/P$ elements of A and B. After step 5, A is replicated c times, each process stores $(cmk+kn)/P$ elements of A and B. CA3DMM uses a dual buffer in Cannon's algorithm to overlap communication with computation, so each process needs another $(cmk + kn)/P$ elements for the second buffer of A and B. After Cannon's algorithm, each k-task group has a partial result of C, so each process stores $k_p mn/P$ elements of the partial C matrix. After reduce-scatter, each process stores mn/P elements of the final C matrix in the partial C matrix block buffer. Therefore, the memory usage is

$$S = 2\frac{cmk + kn}{P} + \frac{k_p mn}{P}. \tag{11}$$

If $m = n = k$, then $c = 1$ and

$$S = 4m^2/P + m^2/P^{2/3} = \mathcal{O}(\frac{m^2}{P^{2/3}}),$$

so S has the same asymptotic complexity as the memory usage of the original 3D algorithm. For $m > n$, the analysis is similar, and the conclusion remains unchanged.

E. Choosing the 2D Algorithm in CA3DMM

The 2D algorithm in CA3DMM determines the initial input matrix distributions and the communication pattern during the calculation. SUMMA is the conventional choice. It can handle all 2D process grid sizes, and it is easy to implement. We choose Cannon's algorithm since we believe it can outperform SUMMA in CA3DMM for most problem settings as we explain now. Denote CA3DMM-C and CA3DMM-S as CA3DMM using Cannon's algorithm and SUMMA, respectively. Assume CA3DMM-C and CA3DMM-S use the same process grid and $p_m \geq p_n$. Both approaches have the same communication size Q. If we use the largest possible panel sizes to reduce the number of communication operations in SUMMA, we still need p_m iterations, and each iteration has a communication latency

$$\max(\log_2(p_m)+p_m-1, \log_2(p_n)+p_n-1) = \log_2(p_m)+p_m-1$$

for panel broadcast. The latency of CA3DMM-S is

$$L_{SUMMA} = p_m\left(\log_2(p_m) + p_m - 1\right) + (p_k - 1),$$

giving

$$\begin{aligned} L_{SUMMA} - L &= p_m\left(\log_2(p_m) + p_m - 1\right) + (p_k - 1) \\ &\quad - \left(\log_2(p_m/p_n) + p_n + (p_k - 1)\right) \\ &\geq (p_m - 1)\log_2(p_m) + p_m^2 - p_m - p_n \\ &\geq (p_m - 1)\log_2(p_m) + p_m^2 - 2p_m. \end{aligned}$$

If $p_m = p_n = 1$, no 2D algorithm is needed. If $p_m \geq 2$, $L_{SUMMA} - L \geq 0$. If $p_m < p_n$, the same conclusion holds. The latency of CA3DMM-C is always not larger than the latency of CA3DMM-S when using the same process grid. On the other hand, CA3DMM-S does not have the constraint in (7). The optimal grid size for CA3DMM-S may give a smaller Q and/or a smaller L, but the new values should not be much better than the optimal or near-optimal Q and L values in CA3DMM-C. Considering the above discussion, we choose CA3DMM-C instead of CA3DMM-S.

F. Implementation of CA3DMM

We implement CA3DMM in C + OpenMP + MPI. We enumerate all possible solutions to find the optimal 3D process grid for CA3DMM. In any practical case, the cost of the enumeration is less than 1% of the actual parallel matrix multiplication time. The matrix redistribution subroutine in Algorithm 1 steps 4 and 8 simply packs and unpacks matrix blocks and exchanges data using `MPI_Neighbor_alltoallv`. This subroutine does not have other optimizations. Algorithm 1 steps 5 and 7 use `MPI_Allgather(v)` and `MPI_Reduce_scatter`. We use a dual-buffer in Cannon's algorithm to overlap communication with computation. To maintain the efficiency of local matrix multiplication, we perform multiple shifts for one local matrix multiplication if A and B blocks in Cannon's algorithm do not have a large enough k-dimension size. These two optimizations are common for Cannon's algorithm. Local (shared-memory) matrix multiplications are handled by an OpenMP-parallelized BLAS library. CA3DMM can also run in pure MPI mode by using only one OpenMP thread per MPI rank.

IV. Numerical Experiments

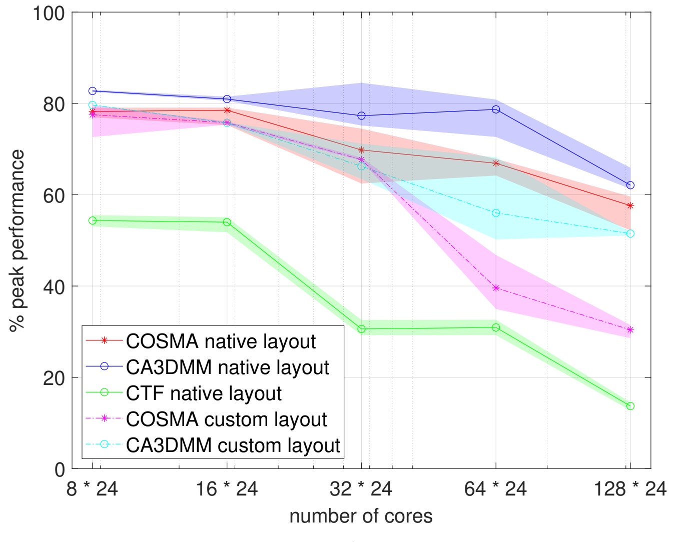
All experiments in this section are performed on the Georgia Tech PACE-Phoenix cluster. Each CPU compute node has two CPU sockets and 192 GB DDR4 memory. Each socket has an Intel Xeon Gold 6226 12-core processor. Each GPU compute node has the same CPU and memory as a CPU compute node but also has two NVIDIA Tesla V100 GPUs. Each Tesla V100 GPU has 16 GB HBM2 memory. Compute nodes are connected with 100 Gbps InfiniBand networking.

A. Scalability of Different PGEMM Algorithms

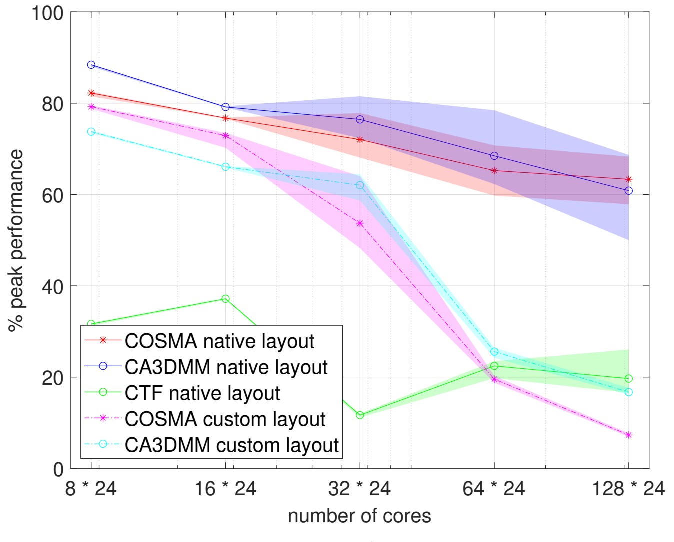
We test and compare three PGEMM libraries that use 3D or 2.5D algorithms and can handle any number of processes: COSMA, CTF, and CA3DMM. The three libraries are compiled using Intel C/C++ compiler v19.0.5 with optimization flags "-xHost -O3", and use Intel MKL v19.0.5 for shared-memory matrix multiplication and MVAPICH2 2.3.2 for the MPI backend.

We test four classes of problem dimensions: (1) *square*, $m = n = k$, (2) *large-K*, $m = n \ll k$, (3) *large-M*, $m \gg n = k$, and (4) *flat*, $m = n \gg k$. Such types of calculations are taken from real-world applications. Some examples are the following. The *square* class is used in density matrix purification and polar decomposition [7, 28]. The *large-K* and *large-M* classes are used in CholeskyQR and Rayleigh-Ritz projection [8, 29, 30]. The *flat* class comes from the trailing matrix update in matrix factorization algorithms, for example, LU, Cholesky, and Householder QR.

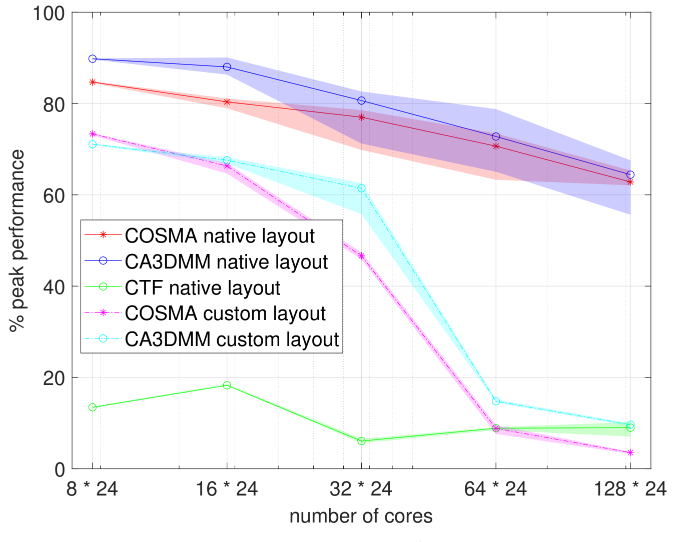
Figure 3 shows the strong scaling test results for different matrix dimensions. All three libraries use one CPU core per MPI rank. COSMA uses communication-computation overlap and can use unlimited extra memory. One-time initialization costs, including finding the optimal 3D process grid in CA3DMM, finding the optimal parallelization strategy in COSMA, initializing MPI communicators, and allocating work buffers, are not counted. Both "library-native" and 1D column matrix partitionings are tested for COSMA and CA3DMM. Since the library-native matrix partitionings of COSMA and CA3DMM are 2D partitions, the 1D column partition aims to show the possible heavy cost of matrix layout conversion. When using library-native matrix partitions, COSMA and CA3DMM have good parallel scalability on all problem classes, showing that both algorithms have optimal or near-optimal communication costs in practice. CTF is not



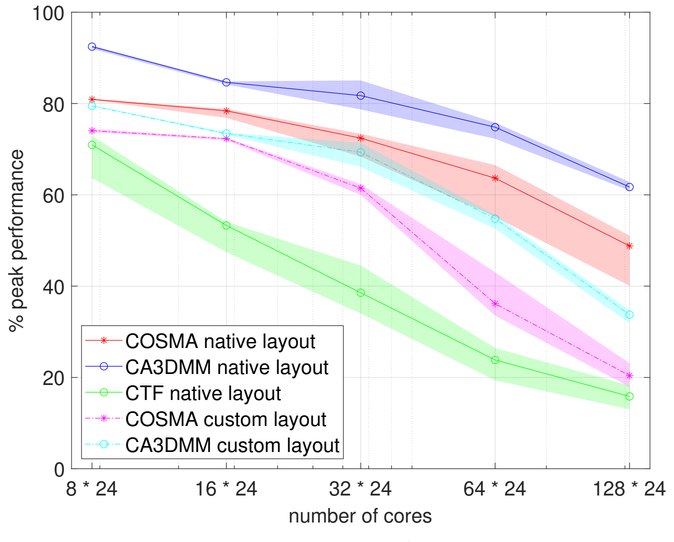
(a) $m = n = k = 50,000$



(b) $m = n = 6,000,\ k = 1,200,000$



(c) $m = 1,200,000,\ n = k = 6,000$

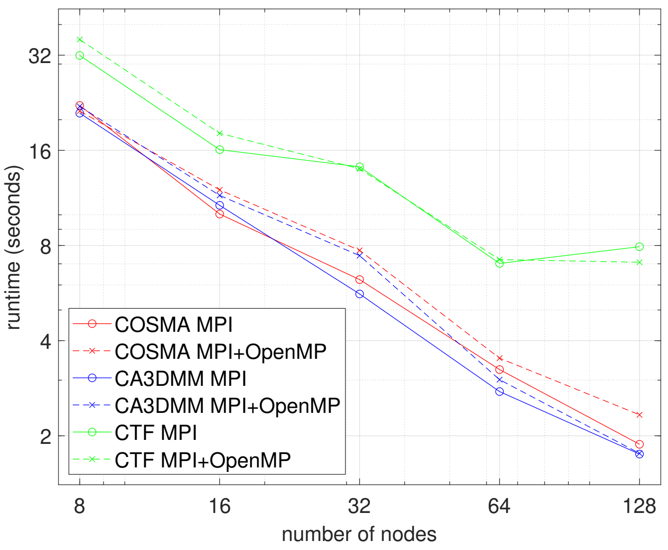


(d) $m = n = 100,000,\ k = 5,000$

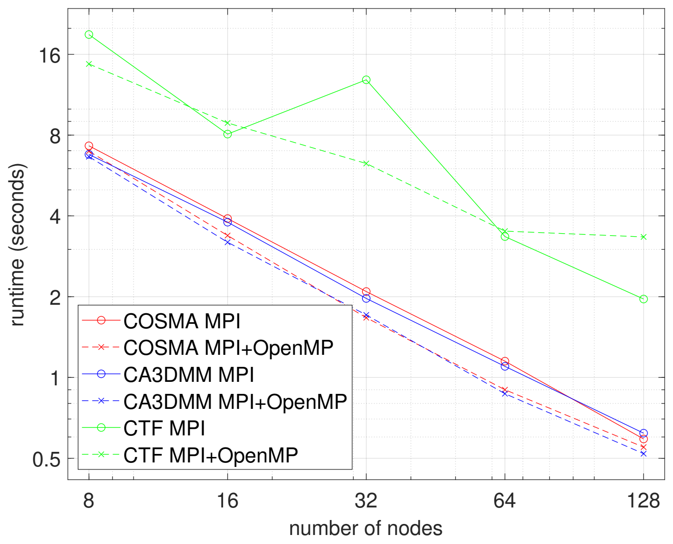
Fig. 3: Strong scaling tests of COSMA, CA3DMM, and CTF for different matrix dimensions. Neither A nor B is transposed. All tested implementations use one core per MPI process. Minimal, mean (marked line), and maximal achieved percentages of peak performance in ten runs are shown. "Native layout" and "Custom layout" refer to the library-native and 1D column partitionings of A, B, and C matrices, respectively.

fine tuned for matrix multiplication, so its parallel efficiency is less satisfying. A previous study suggested that its process grid and matrix decomposition may be far from optimal [18]. For *large-K* and *large-M* problems, COSMA and CA3DMM have very similar performance. The major communication cost in both algorithms is C matrix reduction for *large-K* and B matrix replication for *large-M*, so it is reasonable that both algorithms have similar communication costs. For *square* and *flat* problems, CA3DMM outperforms COSMA. The difference in process grid size may also have an impact, and we will discuss this in Section IV-B. Figure 3b and Figure 3c also show the high matrix layout conversion costs in COSMA and CA3DMM when using unfavorable matrix partitionings for tall-and-skinny matrices. Adopting library-native matrix partitioning to reduce or avoid matrix layout conversion cost in other parallel algorithms is a significant issue to address in the future.

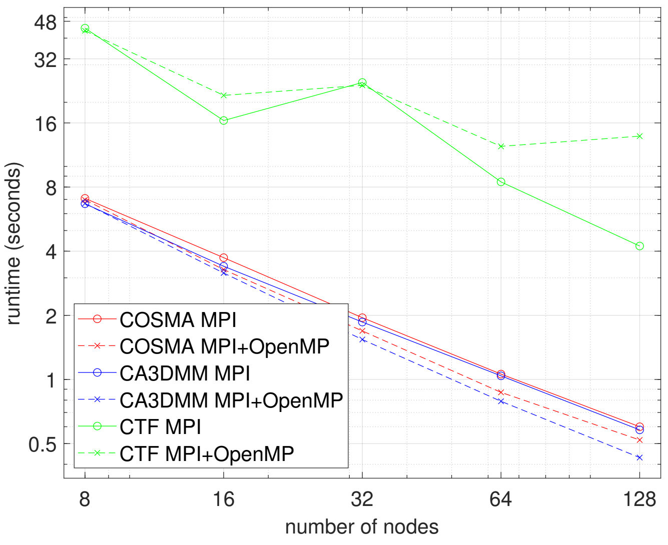
Figure 4 shows the strong scaling test results for different matrix dimensions and parallelization modes. All three libraries use library-native matrix partitionings, and COSMA still uses communication-computation overlap without a limitation on extra memory. (3) and (9) show that switching from pure MPI parallel to MPI + OpenMP hybrid parallel decreases the total number of words transferred between processes but also increases per-process data transfer size. For the *square* problem, both COSMA and CA3DMM have better



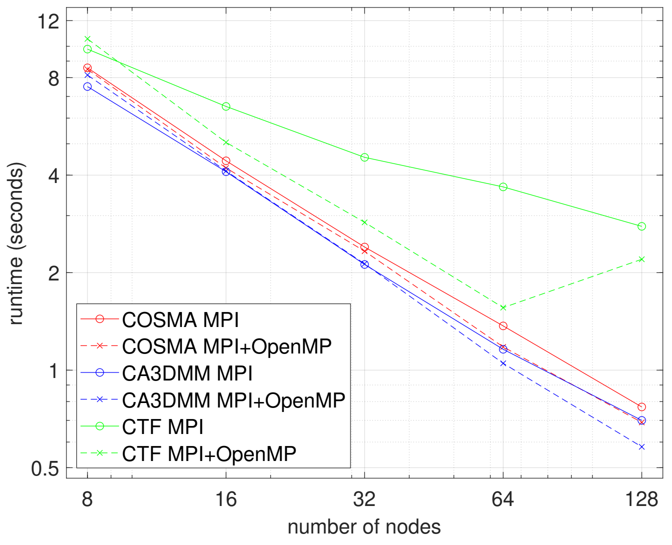
(a) $m = n = k = 50,000$



(b) $m = n = 6,000,\ k = 1,200,000$



(c) $m = 1,200,000,\ n = k = 6,000$



(d) $m = n = 100,000,\ k = 5,000$

Fig. 4: Strong scaling tests of COSMA, CA3DMM, and CTF for different matrix dimensions and parallelization modes. Neither A nor B is transposed. Solid lines with circle markers are pure MPI parallel (one core per MPI process, 24 MPI processes per node) results. Dashed lines with cross markers are MPI + OpenMP parallel (24 cores per MPI process, one MPI process per node) results. Reported values are averaged over ten runs. All libraries use their library-native matrix partitionings.

performance in pure MPI mode than in MPI + OpenMP mode. Runtime breakdowns show that both libraries have larger communication costs in hybrid parallel mode. One possible reason is that the pure MPI parallel mode has a smaller inter-node communication volume. Another possible reason is that communication operations from different MPI processes in the same node can overlap with each other and better utilize inter-node network bandwidth [31]. For the *large-K* and *large-M* problems, COSMA and CA3DMM run faster using MPI + OpenMP parallelization. In these cases, only one type of communication operation is performed in a much smaller process group, leading to a much lower communication cost. For the *flat* problem, COSMA and CA3DMM also have better performance in MPI + OpenMP mode due to a smaller communication cost. CTF has various performance behaviors when using hybrid parallelization, which needs further study for a better understanding.

We test different l values in the range $[0.85, 0.99]$ for (5). Test results show that using other l values give the same 3D process grid as using the value $l = 0.95$ in almost all cases (detailed results omitted).

Table I shows the memory usage per process (in MB) of COSMA and CA3DMM for different problem dimensions. For the *square* class problem, CA3DMM always uses less

	Problem Size m, n, k ($\times 10^3$)	192	384	768	1536	3072
		Number of MPI Processes				
COSMA	50, 50, 50	2086	1242	770	484	292
	6, 6, 1200	848	561	424	283	171
	1200, 6, 6	848	561	424	283	171
	100, 100, 5	993	616	387	293	176
CA3DMM	50, 50, 50	1490	696	398	137	106
	6, 6, 1200	1987	1397	497	284	125
	1200, 6, 6	1428	851	710	213	102
	100, 100, 5	1797	855	433	206	128

TABLE I: COSMA and CA3DMM memory usage per process (in MB) for different problem dimensions. COSMA has no limitation on extra memory. Both libraries use library-native matrix distributions.

memory than COSMA. For the other three problem classes, CA3DMM uses more memory than COSMA when the number of MPI processes is not very large, but the memory usage of CA3DMM decreases more rapidly than COSMA with the increase of number of MPI processes. CA3DMM still uses less memory than COSMA when using more than 1536 MPI processes. Since both COSMA and CA3DMM have the same asymptotic maximum memory usage of $\mathcal{O}(mnk/P^{2/3})$, CA3DMM should still use less memory than COSMA when using more than 3072 MPI processes for these four classes of problem dimensions. We notice that the memory usage in CA3DMM greatly decreases in two cases: (1) *large-K* from 384 processes to 768 processes, and (2) *large-M* from 768 processes to 1536 processes. The reason for the large decreases in these cases is the change of process grid size, with the changes in memory usage matching (11).

B. Process Grid Dimensions and Performance

In this section, we study the impact of process grid dimensions on the performance of COSMA and CA3DMM. Table II shows the COSMA and CA3DMM runtime for various problem dimensions with different process grid dimensions. When using 2048 cores, COSMA chooses its optimal process grid size for each problem dimension and CA3DMM uses the same process grid. When using 3072 cores, a near-optimal process grid is specified for each problem dimension. We also report the performance of both libraries using their optimal process grids.

The timings in Table II show two remarkable points. First, the performance of a PGEMM algorithm relies on both the process grid dimensions and communication patterns and operations. When using the same (optimal) process grid, COSMA and CA3DMM have the same theoretical communication size Q, but CA3DMM is up to 21% faster than COSMA. Considering that COSMA has its optimized collective operation implementation and CA3DMM uses standard MPI functions, such performance differences can only come from different communication patterns and operations. Second, sub-optimal process grids may outperform the optimal grids chosen by theoretical analysis due to the cost of collective operations. For the *large-K* problem size, CA3DMM is slower when using the theoretical optimal process grid $p_m \times p_n \times p_k = 3 \times 3 \times 341$ instead of a sub-optimal grid $p_m \times p_n \times p_k = 4 \times 2 \times 384$. The optimal process grid uses 99.9% of the cores, so the computational resources are well utilized. A runtime breakdown shows that the major difference in timing comes from the cost of the reduce-scatter operation. For collective operations, $p_k = 341$ is unfavorable.

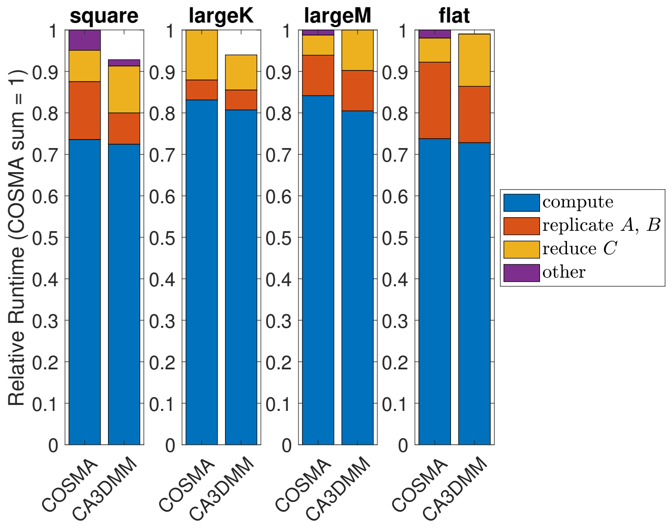


Fig. 5: COSMA and CA3DMM relative runtime breakdowns for 2048-core tests in Table II. For each class of problems, timings are normalized such that the total runtime of COSMA equals 1. For CA3DMM, "replicate A, B" includes step 5 in Algorithm 1 and the cost of shifting A and B blocks in Cannon's algorithm.

Figure 5 further shows the relative runtime breakdowns for 2048-core tests in Table II. COSMA and CA3DMM have similar local computation and communication (sum of "replicate A, B" and "reduce C") costs in all problem classes. Instead of organizing the 3D process grid in a fixed way like CA3DMM, COSMA "crafts the binary reduction tree in all three dimensions" of communications and has different 3D process grid organizations for different problem classes. Therefore, the "reduce C" costs in COSMA are similar to or smaller than that of CA3DMM in different cases, depending on how close the MPI processes that reduce sum the partial C results are placed on the hardware. The same comment applies to the "replicate A, B" costs.

C. GPU Performance

We implement a CA3DMM GPU prototype by simply offloading local matrix multiplications from CPUs to GPUs. Table III compares the GPU performance of COSMA, CTF, and our CA3DMM GPU prototype. The GPU part of these three libraries are compiled using CUDA 10.2 and use cuBLAS for local matrix multiplications. The maximum number of GPUs we can use is 32, so we test the performance using 16 and 32 GPUs. Since the numbers of MPI processes are powers of two and are small, COSMA and CA3DMM have the same or effectively the same 3D process grids

Number of Cores	Problem Size	COSMA		CA3DMM	
	$m, n, k\ (\times 10^3)$	p_m, p_n, p_k	Runtime (s)	p_m, p_n, p_k	Runtime (s)
2048	50, 50, 50	8, 16, 16	2.65	*8, 16, 16*	2.46
	6, 6, 1200	2, 2, 512	0.84	2, 2, 512	0.78
	1200, 6, 6	512, 2, 2	0.82	512, 2, 2	0.82
	100, 100, 5	32, 32, 2	1.03	32, 32, 2	1.02
3072	50, 50, 50	*16, 16, 12*	2.11	16, 16, 12	1.75
		12, 16, 16	1.88		
	6, 6, 1200	*4, 2, 384*	0.61	*4, 2, 384*	0.54
		2, 3, 512	0.59	3, 3, 341	0.62
	1200, 6, 6	*384, 4, 2*	0.62	384, 4, 2	0.58
		512, 2, 3	0.60		
	100, 100, 5	*32, 32, 3*	0.85	*32, 32, 3*	0.82
		32, 48, 2	0.77	39, 39, 2	0.70

TABLE II: COSMA and CA3DMM runtime (seconds) for different problem dimensions with process grid dimensions. Reported runtime values are averaged over ten runs. Both libraries use one CPU core per MPI rank and library-native matrix distributions. Process grid sizes in italics are *not* the default optimal grid sizes chosen by the library.

Number of GPUs	Problem Size	COSMA		CA3DMM		CTF
	$m, n, k\ (\times 10^3)$	p_m, p_n, p_k	Runtime (s)	p_m, p_n, p_k	Runtime (s)	Runtime (s)
16	50, 50, 50	2, 2, 4	5.45	2, 2, 4	6.44	15.46
	10, 10, 300	1, 1, 16	0.91	1, 1, 16	0.94	4.64
	300, 10, 10	16, 1, 1	0.90	16, 1, 1	0.89	13.77
	50, 50, 10	4, 4, 1	1.22	4, 4, 1	1.23	11.61
32	50, 50, 50	2, 4, 4	4.70	4, 2, 4	5.39	15.20
	10, 10, 300	1, 1, 32	0.70	1, 1, 32	0.78	3.70
	300, 10, 10	32, 1, 1	0.64	32, 1, 1	0.65	14.82
	50, 50, 10	4, 8, 1	0.82	8, 4, 1	0.84	12.46

TABLE III: COSMA, CA3DMM, and CTF runtime (seconds) for different problem dimensions on GPUs. Reported runtime values are averaged over ten runs. All libraries use one GPU per MPI rank and library-native matrix distributions.

in all problem settings. COSMA outperforms CA3DMM on *square* and *large-K* problems where the k-dimension reduction is needed. On *square* problems, the partial C result block is larger than a threshold in MVAPICH2, which degrades the performance of reduce-scatter. In the MPI + OpenMP tests (Figure 4a), CA3DMM also has the same performance issue, but it is less obvious since the total runtime is larger. The MVAPICH2 user manual does not list a related runtime environment variable. We leave the optimization of the reduce-scatter step for future study. For *flat* and *large-M* problems, COSMA and CA3DMM have almost the same performance. The GPU acceleration of CTF is still in development.

V. Conclusions and Open Problems

In this work, we present the CA3DMM algorithm, a simple and scalable parallel dense general matrix multiplication algorithm based on a unified view of parallel matrix multiplication. The unified view organizes a PGEMM as multiple low-rank updates and parallelizes both the calculations in each low-rank update and the computations of different low-rank updates. This unified view generalizes 1D, 2D, and 3D algorithms in an intuitive way, which allows one to understand and implement it easily. We prove CA3DMM can achieve optimal or near-optimal communication cost with extra memory for all matrix dimensions and any number of processes. Numerical results show that CA3DMM can scale to a large number of cores efficiently and the performance of CA3DMM is comparable or better than state-of-the-art communication-optimal PGEMM codes for a wide range of problem dimensions and numbers of processes. The theoretical analysis and experimental data also point to some future study directions for CA3DMM.

The first topic for future study is controlling the usage of extra memory in CA3DMM while minimizing communication costs. (11) suggests two possible approaches. The first approach is replacing Cannon's algorithm with the SUMMA algorithm. The SUMMA algorithm uses a tunable broadcast block size b. The extra memory required for dual buffering and overlapping communication with computation is $\mathcal{O}(\max(m/p_m, n/p_n))$. This CA3DMM algorithm would be simpler since neither the A matrix nor the B matrix would need to be replicated before calling SUMMA. As discussed in Section III-E, the communication pattern in SUMMA is less preferable than that in Cannon's algorithm, so the SUMMA version is very likely to be slower in practice. The second approach is reducing the number of k-task groups, i.e., reducing the number of partial C matrix results. This approach makes CA3DMM move toward 2D algorithms and increases the communication size Q. These two approaches can be applied together to further reduce the usage of extra memory.

Another open question for CA3DMM is reducing matrix distribution conversion costs in real-world applications.

Indeed, CARMA, COSMA, and CA3DMM all need to address this issue since they all have library-native matrix partitionings that are not easy to use directly by higher-level driver algorithms. Real-world applications usually use natural 1D or 2D partitionings for matrices and process grids, or block-cyclic 2D matrix partitioning for ScaLAPACK or other distributed-memory linear algebra libraries. Two example driver algorithms are the Rayleigh-Ritz step in Chebyshev-filtered subspace iteration [8] and the repeated matrix multiplications in density matrix purification [9]. As Figure 3 shows, the cost of converting a distributed matrix to a library-native distribution could be very high. Therefore, it is essential to design library-native matrix partitionings or other matrix partitionings that can help reduce the matrix layout conversion cost.

CA3DMM is released in open-source form at https://github.com/scalable-matrix/CA3DMM and is being integrated into the distributed-memory large-scale real-space density functional theory (DFT) program SPARC [32]. The need for a high-performance PGEMM for various matrix dimensions used in SPARC was the original motivation for developing CA3DMM.

Acknowledgments

The authors gratefully acknowledge funding from the U.S. Department of Energy, grant DE-SC0019410. The authors acknowledge the Partnership for an Advanced Computing Environment (PACE) at the Georgia Institute of Technology for providing HPC resources that have contributed to the research results reported within this paper. The authors also offer their thanks to Shikhar Shah for meaningful discussions on the CA3DMM algorithm, Lucas Erlandson for his assistance on testing CA3DMM and adopting CA3DMM in the SPARC library, and Marko Kabić for his helpful suggestions on testing the COSMA library.

References

[1] C. D. Meyer, *Matrix Analysis and Applied Linear Algebra*. USA: Society for Industrial and Applied Mathematics, 2000.

[2] N. J. Higham, *Functions of Matrices: Theory and Computation*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2008.

[3] Y. Saad, *Numerical Methods for Large Eigenvalue Problems*. Society for Industrial and Applied Mathematics, 2011. [Online]. Available: https://epubs.siam.org/doi/abs/10.1137/1.9781611970739

[4] A. Azad, A. Buluç, and J. Gilbert, "Parallel triangle counting and enumeration using matrix algebra," in *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*, 2015, pp. 804–811.

[5] J. Kepner, P. Aaltonen, D. Bader, A. Buluç, F. Franchetti, J. Gilbert, D. Hutchison, M. Kumar, A. Lumsdaine, H. Meyerhenke, S. McMillan, C. Yang, J. D. Owens, M. Zalewski, T. Mattson, and J. Moreira, "Mathematical foundations of the GraphBLAS," in *2016 IEEE High Performance Extreme Computing Conference (HPEC)*, 2016, pp. 1–9.

[6] R. A. Kendall and H. Fruchtl, "The impact of the resolution of the identity approximate integral method on modern *Ab-initio* algorithm development," *Theoretical Chemistry Accounts*, vol. 97, Oct. 1997. [Online]. Available: https://www.osti.gov/biblio/1783269

[7] A. H. R. Palser and D. E. Manolopoulos, "Canonical purification of the density matrix in electronic-structure theory," *Physical Review B*, vol. 58, no. 19, pp. 12 704–12 711, Nov. 1998. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevB.58.12704

[8] Y. Zhou, Y. Saad, M. L. Tiago, and J. R. Chelikowsky, "Self-consistent-field calculations using Chebyshev-filtered subspace iteration," *Journal of Computational Physics*, vol. 219, no. 1, pp. 172–184, Nov. 2006. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S002199910600146X

[9] X. Liu, A. Patel, and E. Chow, "A new scalable parallel algorithm for Fock matrix construction," in *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. Phoenix, AZ, USA: IEEE, May 2014, pp. 902–914. [Online]. Available: http://ieeexplore.ieee.org/document/6877321/

[10] J.-W. Hong and H.-T. Kung, "I/O complexity: The red-blue pebble game," ser. STOC '81. New York, NY, USA: Association for Computing Machinery, 1981, p. 326–333. [Online]. Available: https://doi.org/10.1145/800076.802486

[11] D. Irony, S. Toledo, and A. Tiskin, "Communication lower bounds for distributed-memory matrix multiplication," *Journal of Parallel and Distributed Computing*, vol. 64, no. 9, pp. 1017–1026, 2004. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0743731504000437

[12] G. Ballard, J. Demmel, O. Holtz, and O. Schwartz, "Minimizing communication in numerical linear algebra," *SIAM Journal on Matrix Analysis and Applications*, vol. 32, no. 3, pp. 866–901, 2011. [Online]. Available: https://doi.org/10.1137/090769156

[13] G. Ballard, E. E. Carson, J. Demmel, M. Hoemmen, N. Knight, and O. Schwartz, "Communication lower bounds and optimal algorithms for numerical linear algebra," *Acta Numerica*, vol. 23, p. 1–155, 2014.

[14] R. A. van de Geijn and J. Watts, "SUMMA: scalable universal matrix multiplication algorithm," *Concurrency: Practice and Experience*, vol. 9, pp. 255–274, 1997.

[15] R. C. Agarwal, S. M. Balle, F. G. Gustavson, M. Joshi, and P. Palkar, "A three-dimensional approach to parallel matrix multiplication," *IBM Journal of Research and Development*, vol. 39, no. 5, pp. 575–582, 1995.

[16] E. Solomonik and J. Demmel, "Communication-optimal parallel 2.5D matrix multiplication and LU factorization algorithms," in *Euro-Par 2011 Parallel Processing*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 90–109.

[17] J. Demmel, D. Eliahu, A. Fox, S. Kamil, B. Lipshitz, O. Schwartz, and O. Spillinger, "Communication-optimal parallel recursive rectangular matrix multiplication," in *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. Cambridge, MA, USA: IEEE, May 2013, pp. 261–272. [Online]. Available: http://ieeexplore.ieee.org/document/6569817/

[18] G. Kwasniewski, M. Kabić, M. Besta, J. VandeVondele, R. Solcà, and T. Hoefler, "Red-blue pebbling revisited: near optimal parallel matrix-matrix multiplication," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. Denver, CO, USA: ACM, Nov. 2019, pp. 1–22.

[19] L. E. Cannon, "A cellular computer to implement the Kalman filter algorithm," Ph.D. dissertation, Montana State University, USA, 1969.

[20] J. Choi, D. W. Walker, and J. J. Dongarra, "PUMMA: Parallel universal matrix multiplication algorithms on distributed memory concurrent computers," *Concurrency: Practice and Experience*, vol. 6, no. 7, pp. 543–570, Oct. 1994. [Online]. Available: https://onlinelibrary.wiley.com/doi/10.1002/cpe.4330060702

[21] J. Choi, J. J. Dongarra, R. Pozo, and D. W. Walker, "ScaLAPACK: a scalable linear algebra library for distributed memory concurrent computers," in *[Proceedings 1992] The Fourth Symposium on the Frontiers of Massively Parallel Computation*. McLean, VA, USA: IEEE Comput. Soc. Press, 1992, pp. 120–127. [Online]. Available: http://ieeexplore.ieee.org/document/234898/

[22] M. Gates, J. Kurzak, A. Charara, A. YarKhan, and J. Dongarra, "SLATE: Design of a modern distributed and accelerated linear algebra library," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3295500.3356223

[23] E. Solomonik, E. Carson, N. Knight, and J. Demmel, "Trade-offs between synchronization, communication, and computation in parallel linear algebra computations," *ACM Transactions on Parallel Computing*, vol. 3, no. 1, Jan. 2017. [Online]. Available: https://doi.org/10.1145/2897188

[24] E. Solomonik, D. Matthews, J. Hammond, and J. Demmel, "Cyclops tensor framework: Reducing communication and eliminating load imbalance in massively parallel contractions," in *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, 2013, pp. 813–824.

[25] V. Strassen, "Gaussian elimination is not optimal," *Numerische Mathematik*, vol. 13, no. 4, pp. 354–356, Aug. 1969. [Online]. Available: https://doi.org/10.1007/BF02165411

[26] H.-J. Lee, J. P. Robertson, and J. A. B. Fortes, "Generalized Cannon's algorithm for parallel matrix multiplication," in *Proceedings of the 11th International Conference on Supercomputing*, ser. ICS '97. New York, NY, USA: Association for Computing Machinery, 1997, p. 44–51. [Online]. Available: https://doi.org/10.1145/263580.263591

[27] R. Thakur, R. Rabenseifner, and W. Gropp, "Optimization of collective communication operations in MPICH," *Int. J. High Perform. Comput. Appl.*, vol. 19, no. 1, p. 49–66, Feb. 2005.

[28] Y. Nakatsukasa and N. J. Higham, "Stable and efficient spectral divide and conquer algorithms for the symmetric eigenvalue decomposition and the SVD," *SIAM Journal on Scientific Computing*, vol. 35, no. 3, pp. A1325–A1349, Jan. 2013. [Online]. Available: http://epubs.siam.org/doi/10.1137/120876605

[29] S. Das, P. Motamarri, V. Gavini, B. Turcksin, Y. W. Li, and B. Leback, "Fast, scalable and accurate finite-element based *Ab-initio* calculations using mixed precision computing: 46 PFLOPS simulation of a metallic dislocation system," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. Denver, Colorado: ACM Press, 2019, pp. 1–11. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3295500.3357157

[30] T. Fukaya, R. Kannan, Y. Nakatsukasa, Y. Yamamoto, and Y. Yanagisawa, "Shifted Cholesky QR for computing the QR factorization of ill-conditioned matrices," *SIAM Journal on Scientific Computing*, vol. 42, no. 1, pp. A477–A503, 2020.

[31] H. Huang and E. Chow, "Overlapping communications with other communications and its application to distributed dense matrix computations," in *2019 IEEE 33rd International Parallel and Distributed Processing Symposium*. Rio de Janeiro, Brazil: IEEE, May 2019, pp. 501–510. [Online]. Available: https://ieeexplore.ieee.org/document/8821006/

[32] Q. Xu, A. Sharma, B. Comer, H. Huang, E. Chow, A. J. Medford, J. E. Pask, and P. Suryanarayana, "SPARC: Simulation package for *Ab-initio* real-space calculations," *SoftwareX*, vol. 15, p. 100709, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2352711021000546

Appendix: Artifact Description/Artifact Evaluation

SUMMARY OF THE EXPERIMENTS REPORTED

1. Abstract

This description contains the information needed to compile and launch the computational experiments in the SC22 paper submission "CA3DMM: A New Algorithm Based on A Unified View of Parallel Matrix Multiplication". We describe how to compile the CA3DMM library and example programs and run the experiments in Section VI in the submitted paper.

2. Artifact description

2.1 Artifact meta information

- Algorithm: Communication-Avoiding 3D Matrix Multiplication
- Program: Example parallel matrix multiplication program provided in the CA3DMM library
- Compilation: Intel C/C++ compilers version 19, an MPI-3 supported MPI library and a BLAS library with a C language interface
- Binary: MPI executable
- Data set: Not needed
- Run-time environment: Linux environment with MPI
- Hardware: Intel CPU and NVIDIA GPU
- Output: On-screen output: runtime and correctness check
- Publicly available?: Yes

2.2 How delivered

Intel Parallel Studio XE including Intel C/C++ compile, Intel MPI, and Intel MKL. Licenses can be applied with an education email account or for trial. For MPI-3 supported MPI libraries, the source code of MPICH, MVAPICH2, and OpenMPI can be downloaded from their official websites and compiled using Intel compilers. CA3DMM source code can be cloned from https://github.com/scalable-matrix/CA3DMM.git. MPICH, MVAPICH2, and OpenMPI can also be installed using the Spack package manager: https://github.com/spack/spack.

2.3 Hardware dependencies

For CPU-only results, no hardware dependency. We used Intel Xeon Gold 6226 12-core CPU and Mellanox 100 Gbps IB network for reproducibility. For GPU results, NVIDIA GPU is required.

2.4 Software dependencies

CA3DMM requires a C11 compliant compiler, an MPI-3 supported MPI library and a BLAS library with a C language interface. We tested the Intel C compiler and Intel MKL in Intel Parallel Studio XE 2019 update 4, and MVAPICH2 2.3.2. We suggest using MVAPICH2 for clusters with IB networking and using vendor-optimized MPI libraries on supercomputers.

2.5 Datasets

The experiments in the paper use randomly generated general non-zero matrices.

3. Compilation

Clone the CA3DMM library from GitHub:

```
git clone https://github.com/scalable-matrix/CA3DMM.git
```

Enter directory CA3DMM/src. CA3DMM provides four make files in CA3DMM/src:

- icc-mkl-impi.make: Use Intel C compiler, Intel MKL, and Intel MPI library
- icc-mkl-anympi.make: Use Intel C compiler, Intel MKL, and any MPI library
- icc-mkl-nvcc-impi.make: Use Intel C compiler, Intel MKL, NVCC compiler for NVIDIA GPU support, and Intel MPI library
- icc-mkl-nvcc-anympi.make: Use Intel C compiler, Intel MKL, NVCC compiler for NVIDIA GPU support, and any MPI library

We use the icc-mkl-anympi.make and icc-mkl-nvcc-anympi.make make files for compiling the CPU and GPU versions on the Georgia Tech PACE-Phoenix cluster. Run the following command to compile the CA3DMM library:

```
make -f icc-mkl-anympi.make -j
```

After compilation, the dynamic and static library files are copied to the directory CA3DMM/lib, and the C header files are copied to CA3DMM/include. Enter directory CA3DMM/examples to compile the example program. This directory also contains four make files similar to the make files in CA3DMM/src. Run the following command to compile the CA3DMM library:

```
make -f icc-mkl-anympi.make -j
```

The compiled example program we need is

`CA3DMM/examples/example_AB.exe.`

4. Experiment workflow

For single node execution or launching on clusters without a job scheduling system, the following command should run on most platforms (assuming that you are in directory CA3DMM/examples):

```
mpirun -np <nprocs> ./example_AB.exe <M> <N> <K>
        <transA> <transB> <validation> <ntest> <dtype>
          <mp>  <np>  <kp>
```

Where:

- nprocs: Number of MPI processes
- M, N, K: Sizes of input matrices, A matrix is $M \times K$, B matrix is $K \times N$
- transA, transB: 0 or 1, 0 for no transpose, 1 for transpose
- validation: 0 or 1, 0 for skipping result correctness check, 1 for result correctness check
- ntest: Number of tests to run, should be a non-negative integer
- dtype: Calculation device type, 0 for CPU, 1 for NVIDIA GPU (if the library and the example program are compiled with NVIDIA GPU support)
- mp, np, kp: Optional, the 3D process grid size, mp * np * kp should <= nprocs

To explicitly control MPI + OpenMP hybrid parallelization, you need to specify OpenMP environment variables, and process affinity environment variables for some MPI libraries. In the paper, we use the following environment variables for MPI + OpenMP parallel tests on the Georgia Tech PACE-Phoenix cluster:

```
OMP_PLACES=cores
OMP_NUM_THREADS=24
MV2_CPU_BINDING_POLICY=hybrid
MV2_THREADS_PER_PROCESS=24
```

For clusters and supercomputers with job scheduling systems like slurm, you need to write job scripts for launching the example program on multiple nodes. We provide three PBS job scripts in the directory CA3DMM/examples for running on the PACE-Phoenix cluster:

- *phoenix-64node-mpi.pbs*: MPI only parallelization, 1 CPU core per MPI rank, 64 nodes
- *phoenix-64node-mpiomp.pbs*: MPI + OpenMP parallelization, 24 CPU core per MPI rank, 1 MPI rank per node, 64 nodes
- *phoenix-16node-GPU.pbs*: MPI + GPU parallelization, 1 NVIDIA GPU device per MPI rank, 2 MPI ranks per node, 16 nodes

5. Expected results

The example program prints timing results to the screen output. Here is an example output on a single node using 1 core per MPI process:

```
$ mpirun -np 24 ./example_AB.exe 8000 8000 8000 0 0 1 10 0
Test problem size m * n * k : 8000 * 8000 * 8000
Transpose A / B             : 0 / 0
Number of tests             : 10
Check result correctness    : 1
Device type                 : 0

CA3DMM partition info:
Process grid mp * np * kp  : 4 * 2 * 3
Work cuboid  mb * nb * kb  : 2000 * 4000 * 2667
Process utilization        : 100.00 %
Comm. volume / lower bound : 1.04
Rank 0 work buffer size    : 244.28 MBytes


A, B, C redist   : 80 79 77 79 80 77 82 78 78 77
A / B allgather  : 16 16 16 16 16 16 16 16 16 16
2D Cannon      : 649 649 667 665 664 667 666 666 667 667
C reduce-scatter : 41 42 42 51 43 42 41 42 42 41
matmul only     : 706 708 725 733 724 725 724 724 725 725
total execution  : 786 786 802 812 803 802 806 802 803 801

=============== CA3DMM algorithm engine ===============
* Initialization         : 4.89 ms
* Number of executions   : 10
* Execution time (avg)   : 800.46 ms
  * Redistribute A, B, C : 78.74 ms
  * Allgather A or B     : 16.30 ms
  * 2D Cannon execution  : 662.62 ms
  * Reduce-scatter C     : 42.79 ms
------------- 2D Cannon algorithm engine -------------
* Initialization : 0.04 ms
* Number of executions  : 10
* Execution time (avg)  : 662.62 ms
  * Initial shift       : 33.79 ms
  * Loop shift wait     : 23.21 ms
  * Local DGEMM         : 605.62 ms
* Per-rank performance  : 64.40 GFlops
-----------------------------------------------------
=====================================================
CA3DMM output : 0 error(s)
```

The example program uses a 1D column partition for the input A and B matrices and the output C matrix. The "Process grid" line shows the 3D process grid size. The "matmul only" line gives the runtime (in milliseconds) using library-native matrix partitioning.

6. Reproducing experiment results in the paper

The SC22_AD directory contains all scripts and detailed instructions for reproducing all experiment results in the paper. Please follow the instructions in readme.md in this folder.

AUTHOR-CREATED OR MODIFIED ARTIFACTS:

Artifact 1

Persistent ID: `https://doi.org/10.5281/zenodo.6929079`
Artifact name: Communication-Avoiding 3D Matrix Multiplication

Reproduction of the artifact without container: The program requires an optimized MPI library, which is not suitable for using container images. Meanwhile, compiling the source code is very easy on HPC clusters.