NEW PARALLEL ALGORITHMS FOR LARGE-SCALE MATRIX COMPUTATIONS

A Dissertation
Presented to
The Academic Faculty

By

Hua Huang

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computational Science and Engineering
College of Computing

Georgia Institute of Technology

August 2024

NEW PARALLEL ALGORITHMS FOR LARGE-SCALE MATRIX COMPUTATIONS

Dissertation committee:

Dr. Edmond Chow
School of Computational Science and Engineering
Georgia Institute of Technology

Dr. Ümit V. Çatalyürek
School of Computational Science and Engineering
Georgia Institute of Technology

Dr. Yuanzhe Xi
Department of Mathematics
Emory University

Dr. Helen Xu
School of Computational Science and Engineering
Georgia Institute of Technology

Dr. Phanish Suryanarayana
School of Civil and Environmental Engineering and School of Computational Science and Engineering
Georgia Institute of Technology

Date approved: July 10, 2024

There is always a longer way to go.

Rubia, Honkai Impact 3rd

For my father Deqiang Huang and my mother Yongqiu Huang

ACKNOWLEDGMENTS

I would like to first express my sincere gratitude to my advisor, Professor Edmond Chow, for his guidance and support in my Ph.D. study and research. His wisdom, professionalism, and patience have guided me through the long and challenging journey of my Ph.D. His guidance in not only the arts of research but also in the soft skills necessary for a successful researcher has been nothing short of transformative for me. It has been a great privilege for me to be mentored and work with him for such a long time.

I would like to express my gratitude to all other members of my committee: Professor Yuanzhe Xi, Professor Phanish Suryanarayana, Professor Ümit V. Çatalyürek, and Professor Helen Xu. I learned many valuable lessons on the kernel matrix and its applications from Professor Xi. He also provided me with many useful suggestions regarding my research and career path. The research works of Professors Xu and Çatalyürek are closely related to this work. They offered significant insights and feedback on my work. My collaboration with Professor Suryanarayana on the SPARC project sparked ideas that eventually contributed to multiple parts of this work.

This work was partially supported by the NSF under grants ACI-1609842 and OAC-2003683, DoE under grants DE-SC0019410, and the Partnership for an Advanced Computing Environment (PACE) at the Georgia Institute of Technology for providing HPC resources. I gratefully acknowledge the support from the aforementioned organizations and projects.

I would like to thank Dr. Xing Liu and Dr. Jeff Hammond for being my internship mentors at Meta and Intel. The hands-on guidance from Dr. Liu helped me quickly understand the mixture-of-experts architecture and apply it in my internship work. Dr. Hammond shared much of his insight on parallel programming frameworks with me and provided valuable assistance in developing my career path.

I would like to thank my labmates: Xin Xing, Jordi Wolfson-Pou, Lucas Erlandson, Shikhar Shah, and Grant Bruer. I learned a lot about hierarchical matrices, low-rank approximation, and iterative solvers from your work. Besides academics, I will never forget some of the interesting conversations and experiences we shared. I would also like to thank my collaborator Dr. Tianshi Xu. It has been a pleasure working with you on multiple projects since 2023.

I would like to thank Dr. Weicai Ye, my undergraduate advisor at Sun Yat-sen University. Thank you for changing my life by guiding me to the world of high-performance computing and pushing me beyond the Great Wall to explore a larger world. I would also like to thank Dr. Yunfei Du and Dr. Yutong Lu, the former chief engineer and the director of the National Supercomputer Center in Guangzhou, for sharing their insights into high-performance computing with me.

I would like to thank all my friends for all good days we had together, especially Yifeng Jiang, Junxuan Wang, Shixi Zhong, Xiaohui Huang, Zenan Chen, Lu Xu, Yanjin (Frank) He, Xin Han, Fan Jiang, Siyuan Liu, Guangnan Feng, Wenkai Shao, and Jiahui Lu. Thank you for sharing your time, happiness, and ideas with me. I also appreciate your support in helping me maintain my mental health, especially during the COVID-19 pandemic.

My deepest gratitude goes to my parents, Deqiang Huang and Yongqiu Huang. They not only brought me into this wonderful world but also laid the foundation of love and support that has allowed me to soar. I wouldn't be here without their love, understanding, companionship, and support.

I would also like to thank Anno Hideaki for finally releasing the movie *EVANGELION:3.0+1.0* in 2021. Thank you for creating the true and final chapter of the *EVANGELION* series. Additionally, I would like to thank HoYoverse and HoYo-MiX for their amazing productions, which made my life during the pandemic a little more interesting.

In closing, this dissertation is not just a product of my own work; many people have contributed to it in different ways. I sincerely thank all of you who have been part of my journey. The omissions and errors in this dissertation, of course, are my sole responsibility.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Matrix computation is a critical part of scientific simulations and data analysis in fields such as density functional theory, recommendation systems, and neural networks. With the exponential growth in dataset sizes and computational problems, it is imperative to develop scalable parallel algorithms for efficient dense and sparse matrix operations. This dissertation presents innovative scale-out algorithms that leverage multiple processors to simultaneously and addressing data movement bottlenecks, and scale-up algorithms focus on single-node performance by exploiting matrix properties like sparsity and low-rank structures.

In this dissertation, we first propose the communication-avoiding 3D matrix multiplication algorithm (CA3DMM). CA3DMM is an approach to parallelize general dense matrix multiplication that minimizes communication sizes by optimizing matrix partitioning and data transfer patterns. CA3DMM achieves the theoretical communication cost lower bound with a simple formation and shows superior parallel performance compared to existing methods.

We then propose a hybrid approach for efficient distributed-memory polar decomposition (PD) calculation, which can be used for computing eigenvalue decomposition and singular value decomposition. The proposed hybrid PD approach combines multiple iterative methods for computing PD and utilizes CA3DMM and a new parallel orthonormalization algorithm for better performance.

For large sparse matrices, we propose the communication-reduced parallel sparse-dense matrix multiplication (CRP-SpMM) algorithm. This algorithm can benefit from existing sparse matrix partitioning methods for sparse matrix-dense vector multiplication (SpMV), and further explores more effective 2D process grid sizes to reduce communication costs by parallelizing the computation of different columns of the dense input matrix. The parallel implementation of CRP-SpMM significantly outperforms existing distributed-memory parallel SpMM codes.

Lastly, we present the design and parallel implementation of H2Pack, a high-performance multi-purpose library for kernel matrices. Combining multiple state-of-the-art mathematical methods, H2Pack can compress kernel matrices using the $\mathcal{H}^2$ matrix format, resulting in $\mathcal{O}(N)$ storage and matrix-vector multiplication costs. As a result, H2Pack can easily and efficiently handle million-by-million kernel matrices on personal computers, while outperforming the widely used fast multipole method (FMM) on multiple tasks.

CHAPTER 1
INTRODUCTION

1.1 Overview and Motivation

Dense and sparse matrix operations appear in many computations, ranging from traditional scientific simulations to ever-evolving social network recommendations. With the rapid increase in problem and dataset sizes comes much larger matrices. For example, in real-space methods for density functional theory calculation and non-negative matrix factorization for recommendation systems, matrices can have millions or even tens of millions of rows and columns. These large matrices and their associated operations need to be handled by scalable parallel algorithms.

We consider two classes of scalability: scale-out and scale-up. Scale-out means an algorithm can be accelerated by leveraging multiple processor cores in a chip and processor chips to perform simultaneous operations. This allows programs to handle larger datasets without significantly increasing execution time as more processors are utilized. The major challenge of designing scale-out parallel algorithms is optimizing data access to mitigate the "access wall", since transferring data between processor cache and system memory and between computing nodes instead of performing arithmetic operations has become the bottleneck of parallel algorithms on modern high-performance computing (HPC) systems. Previous studies have established the theoretical lower bounds of data movement (communication) costs for parallelizing major linear algebra operations, including matrix multiplication, LU and Cholesky factorization, and QR decomposition. However, the gap between theory and practical implementation remains, necessitating further endeavors in algorithm design and performance optimization.

The ability to scale-up means an algorithm can exploit special properties of problems to reduce computational complexity and resource usage, thus accelerate the execution and/or handle larger problems using the same computational resources. For matrix computations, sparsity, symmetry, and low-rank structures are commonly used properties. Some of these properties may also been used to scale-out the algorithm. Taking advantage of these properties comes at additional costs: the algorithm becomes more complicated, and the computation becomes irregular. The same parallel algorithm might have good performance for some sparse matrices but have poor performance if the locations of the non-zeros in the sparse matrices are changed. As a result, designing and implementing high-performance parallel algorithms for using special properties of matrices can be much harder than designing and implementing high-performance parallel algorithms for dense general matrices.

In this dissertation, we focus on designing and implementing parallel algorithms for matrix multiplication, which is the primary operation in matrix computations. Formally, we consider

$$C = A \times B,\ A \in \mathbb{R}^{m\times k},\ B \in \mathbb{R}^{k\times n},\ C \in \mathbb{R}^{m\times n}, \tag{1.1}$$

where A is a general dense matrix (GEMM calculation), a general sparse matrix (SpMM calculation), or a structured dense matrix; B and C are general dense matrices. For GEMM and SpMM calculations, we focus on the design of distributed-memory parallel algorithms that minimize inter-processor data transfer costs. Additionally, we demonstrate the application of our new parallel GEMM algorithm in a new parallel polar decomposition algorithm. Furthermore, for dense rank-structured kernel matrices, we investigate the design and implementation of shared-memory parallel implementations for compressing such matrices using the $\mathcal{H}^2$ format and multiplying an $\mathcal{H}^2$ matrix with a dense matrix or vector.

1.2 Outline and Contributions

In Chapter 2, we propose the Communication-Avoiding 3D Matrix Multiplication (CA3DMM) algorithm, a novel parallel GEMM algorithm that achieves the communication cost lower bound with a simple but efficient approach. Based on a unified view of parallel matrix multiplication, CA3DMM allows for computing an optimal or near-optimal process grid based on the dimensions of input matrices and can be reduced to existing algorithms if necessary. Then, CA3DMM performs the original GEMM calculation using multiple low-rank updates. The computations of different low-rank updates are parallelized, and each low-rank update is computed using existing parallel GEMM algorithms. Compared to other state-of-the-art parallel GEMM implementations, CA3DMM is much easier to understand and implement, with similar or even better performance for a wide range of matrix dimensions and the number of processes.

In Chapter 3, we present multiple hybrid approaches for scaling up polar decomposition calculations on large parallel computers. Polar decomposition (PD) is closely related to the matrix sign function and has recently been used for developing new parallel eigenvalue decomposition and singular value decomposition algorithms. We analyze the convergence properties of different iterative methods for PD and the parallel scalability of different basic linear algebra operations used in these iterative methods, including matrix multiplication, matrix inversion, and column orthonormalization. By introducing CA3DMM for parallel GEMM and a new parallel algorithm for column orthonormalization, we propose multiple hybrid PD (HPD) approaches that combines existing iterative methods. Experiment results show that HPD approaches exhibit better parallel performance and scalability than existing ScaLAPACK-based parallel PD implementations.

In Chapter 4, we analyze the vast design space of parallel SpMM algorithms and introduce the Communication-Reduced Parallel SpMM (CRP-SpMM) algorithm for optimizing process grid size and reducing communication costs. Existing distributed-memory parallel SpMM algorithms either parallelize SpMM computation only by partitioning the sparse matrix or use parallel algorithms designed for GEMM without considering sparsity. We first discuss different parallelization schemes for SpMM and formulate communication cost models for these schemes. Guided by the cost models, we propose CRP-SpMM to optimize the process grid geometry. CRP-SpMM starts with any given 1D row partitioning and explores more effective 2D process grid sizes to reduce communication costs by parallelizing the computation of different columns of the dense input matrix B. This approach allows our algorithm to benefit from existing research on partitioning sparse matrices. Experimental results show that CRP-SpMM can find better process grids that reduce the total communication size even when high-quality 1D row partitionings are used as baselines. Additionally, the parallel implementation of CRP-SpMM significantly outperforms existing distributed-memory parallel SpMM codes.

In Chapter 5, we present the design and parallel implementation of H2Pack, a high-performance multi-purpose library for kernel matrices defined by translationally invariant kernels and low-dimensional (2D/3D) problems. Previous studies have proposed various algorithms for efficiently constructing and using rank-structured matrix representations. However, existing rank-structured matrix packages are either designed for solving problems in a specific domain or lack a high-performance implementation of new algorithms. H2Pack incorporates multiple new mathematical methods to significantly reduce the number of arithmetic operations needed for compressing a dense kernel matrix into an $\mathcal{H}^2$ matrix. We introduce multiple optimization techniques for efficient $\mathcal{H}^2$ matrix construction and multiplication of an $\mathcal{H}^2$ matrix with a dense vector or matrix ($\mathcal{H}^2$-matvec), including the use of a dynamic task scheduler for computational tasks with dependencies and two flexible programming interfaces for $\mathcal{H}^2$-matvec. With its highly optimized algorithms and codes, H2Pack incurs linear storage and computation costs relative to dataset size, making it well-suited for large-scale data-driven calculations. Compared to the fast multiple method (FMM) and other domain-specific libraries, H2Pack can handle a broader range of matrix classes and achieve performance improvements of up to an order of magnitude.

CHAPTER 2
CA3DMM: A NEW ALGORITHM BASED ON A UNIFIED VIEW OF PARALLEL MATRIX MULTIPLICATION

2.1 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, 4], graph processing [5, 6], computational chemistry [7, 8, 9, 10], 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 (processors, memory, and network) 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 [11, 12, 13, 14]. The widely used SUMMA algorithm [15] achieves optimal communication complexity for certain matrix dimensions or if no extra memory is present. With extra memory, 3D [16] and 2.5D [17] algorithms can achieve the communication cost lower-bound for matrix dimensions m, n, and k (see (Equation 1.1)) in certain ranges [18, 19]. The CARMA algorithm [18] was the first to generalize 1D, 2D, and 3D algorithms (the definitions of 1D, 2D, and 3D will be discussed in Section 2.2 soon) with recursive dimension-splitting to achieve an asymptotic communication lower bound for any matrix dimensions. The COSMA algorithm [19] 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 [19] 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 work, 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 [15] 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.

2.2 Background

Consider the matrix multiplication operation in Equation (1.1) where A, B, and C are all general dense matrices. The iteration space for (Equation 1.1) can be viewed as an $m \times k \times n$ cuboid. The computation on all three dimensions of the cuboid is independent. A parallel MM algorithm can be categorized into 1D, 2D, or 3D (2.5D) algorithms if it parallelizes over 1, 2, or 3 dimensions of the iteration space.

1D algorithms partition only the m-dimension, n-dimension, or the k-dimension of (Equation 1.1). 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 [20], which works for square process grids. Later, the PUMMA algorithm [21] was developed and supported rectangular matrices, transposed matrices, non-square 2D process grids, and different matrix distribution layouts. The SUMMA algorithm [15] further reduced communication costs with communication-computation overlap. SUMMA is the most widely-used (2D) algorithm and it is implemented in the ScaLAPACK library [22] and the SLATE library [23].

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 [16] and the 2.5D algorithm [17] 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 [18]. 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 [19], 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 [24], using any number of processes, is implemented in the Cyclops Tensor Framework (CTF) [25] for parallel tensor calculations. CTF is optimized for distributed-memory dense and sparse tensor operations.

2.3 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.

2.3.1 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 [26].

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 2.1 illustrates the connection between matrix multiplication and a cuboid.

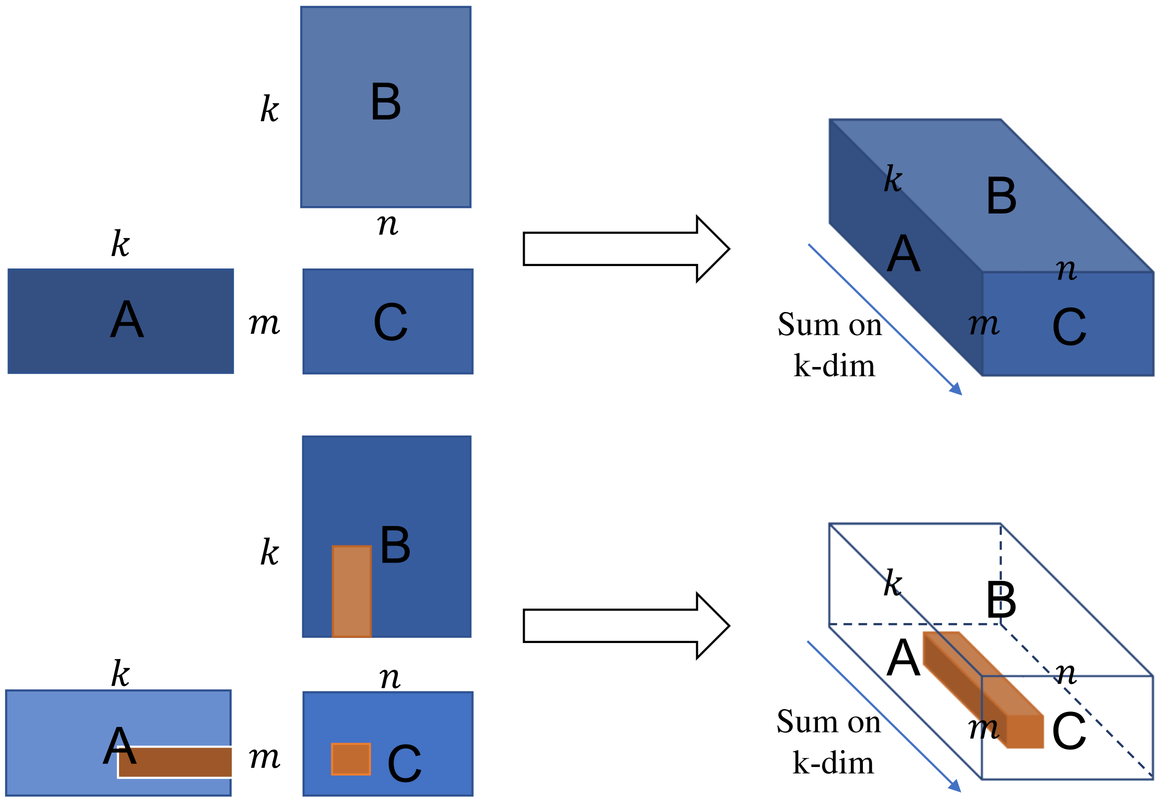


Figure 2.1: Illustration of the connection between matrix multiplication and a cuboid. A unit volume in the cuboid corresponds to one scalar multiplication and addition. The surface area of a cuboid subdomain corresponds to the number of A and B matrix elements to be loaded and the number of C matrix elements to be updated.

Based on the cuboid view of MM, the parallelization of an 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 a 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

$$d_m = d_k = d_n = \left(\frac{mnk}{P}\right)^{1/3}, \tag{2.1}$$

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

$$S_{total} = 6(mnk)^{2/3}P^{1/3}. \tag{2.2}$$

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 2.3.4, (Equation 2.2) matches the I/O complexity lower bound in [11].

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 [19].

2.3.2 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

$$S_{total} = 2(p_m kn + p_n mk + p_k mn) \tag{2.3}$$

with constraint

$$l \cdot P \leq p_m \times p_k \times p_n \leq P, \tag{2.4}$$

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

$$\max\ p_m \times p_k \times p_n \tag{2.5}$$

is also used to maximize the utilization of processes but its priority is lower than that of (Equation 2.3).

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 [20] in CA3DMM to compute rank-(k/p_k) updates. Section 2.3.5 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 [27] (GCA) is a possible solution. However, GCA is designed for block-cyclic distributed matrices and it also has some 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:

$$\mathrm{mod}(\max(p_m, p_n),\ \min(p_m, p_n)) = 0. \tag{2.6}$$

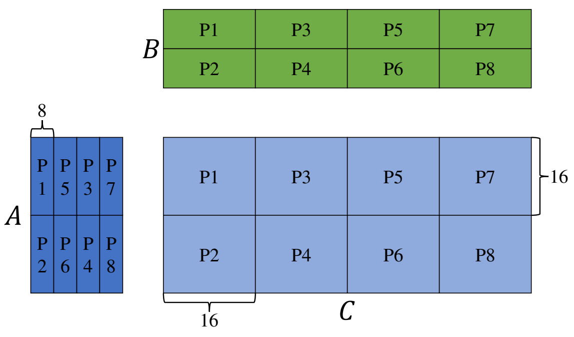
Each k-task group is further split into

$$c = \max(p_m, p_n)/\min(p_m, p_n) \tag{2.7}$$

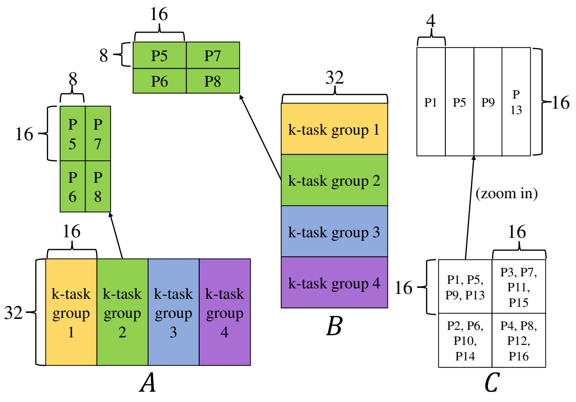
Cannon groups with s^2 processes in each Cannon group, $s = \min(p_m, p_n)$. A block of A or B is replicated c times across Cannon groups in a k-task group. If $c = 1$, the initial distributions of A and B in each k-task group are the distributions of the original Cannon's algorithm. If $c > 1$ and A / 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.



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



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

Figure 2.2: CA3DMM initial and final matrix partitioning 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 2.2a 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 $P_{1\le i\le 4}$ form the first k-task group and compute $A(:, 1 : 16) \times B(1 : 16, :)$, processes $P_{5\le i\le 8}$ form the second k-task group and compute $A(:, 17 : 32) \times B(17 : 32, :)$, 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 2.2b 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 2.2 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 (Equation 2.3) and maximizing
   (Equation 2.5) with constraints (Equation 2.4) and (Equation 2.6).
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.
```

2.3.3 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 an 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 2.3.2. 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 Cannon's algorithm stage. The partial C result reduction in CA3DMM is the same as that in COSMA.

2.3.4 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 (Equation 2.3) equals 1. We further assume butterfly network collectives for communication size and latency analysis [28], 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 (Equation 2.2) and the assumptions in this section, we immediately obtain

$$Q = 3\left(\frac{mnk}{P}\right)^{2/3}. \tag{2.8}$$

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

$$L = \log_2(c) + p_s + p_k - 1. \tag{2.9}$$

Fixing m, n, k and increasing P, (Equation 2.1) 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 $m \leq n$. 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

$$S = 2\frac{cmk + kn}{P} + \frac{k_p mn}{P}. \tag{2.10}$$

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.

2.3.5 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 (Equation 2.6). 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.

2.3.6 Implementation of CA3DMM

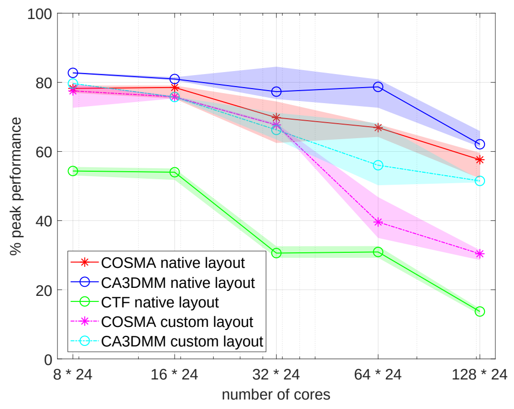
We implement CA3DMM in C + OpenMP + MPI. We enumerate all possible solutions to find the optimal 3D process grid for CA3DMM. In any practical case, the cost of the enumeration is less than 1% of the actual parallel matrix multiplication time. The matrix redistribution subroutine in Algorithm 1 steps 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.

2.4 Numerical Experiments

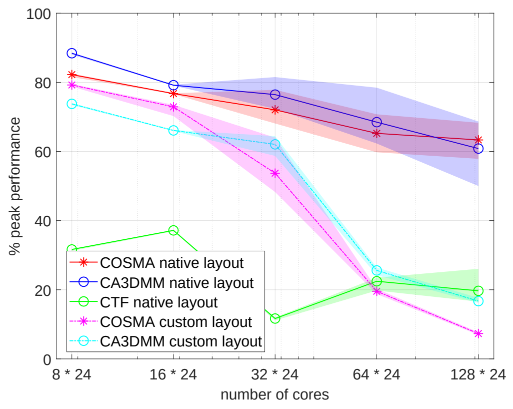
All experiments in this section are performed on the Georgia Tech PACE-Phoenix cluster. Each CPU compute node has two CPU sockets and 192 GB DDR4 memory. Each socket has an Intel Xeon Gold 6226 12-core processor. Each GPU compute node has the same

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.

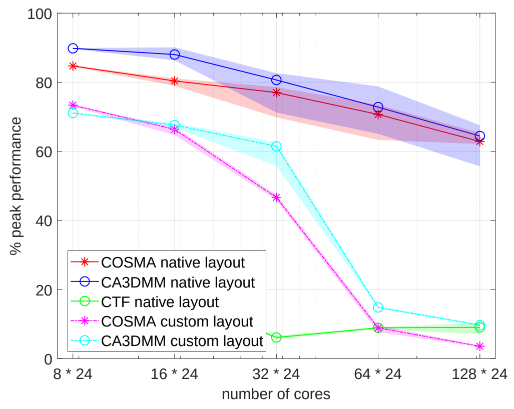
2.4.1 Scalability of Different PGEMM Algorithms



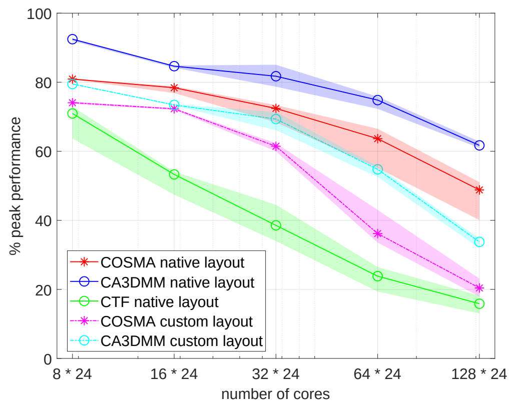
(a) $m = n = k = 50,000$



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



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



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

Figure 2.3: 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.

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 [8, 29]. The *large-K* and *large-M* classes are used in CholeskyQR and Rayleigh-Ritz projection [9, 30, 31]. The *flat* class comes from the trailing matrix update in matrix factorization algorithms, for example, LU, Cholesky, and Householder QR.

Figure 2.3 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 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 [19]. 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 2.4.2. Figure 2.3b and Figure 2.3c 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 2.4 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. (Equation 2.2) and (Equation 2.8) 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 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 [32]. 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 (processor core utilization ratio) in the range $[0.85, 0.99]$ for (Equation 2.4). 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 2.1 shows the memory usage per process (in MB) of COSMA and CA3DMM for

Table 2.1: 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.

	Problem Size	Number of MPI Processes				
	m, n, k ($\times 10^3$)	192	384	768	1536	3072
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

different problem dimensions. For the *square* class problem, CA3DMM always uses less 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 (Equation 2.10).

2.4.2 Process Grid Dimensions and Performance

In this section, we study the impact of process grid dimensions on the performance of COSMA and CA3DMM. Table 2.2 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

Table 2.2: 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 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

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 2.2 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.

Figure 2.5 further shows the relative runtime breakdowns for 2048-core tests in Table 2.2. 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.

2.4.3 GPU Performance

Table 2.3: 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.

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

We implement a CA3DMM GPU prototype by simply offloading local matrix multiplications from CPUs to GPUs. Table 2.3 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 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 2.4a), 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.

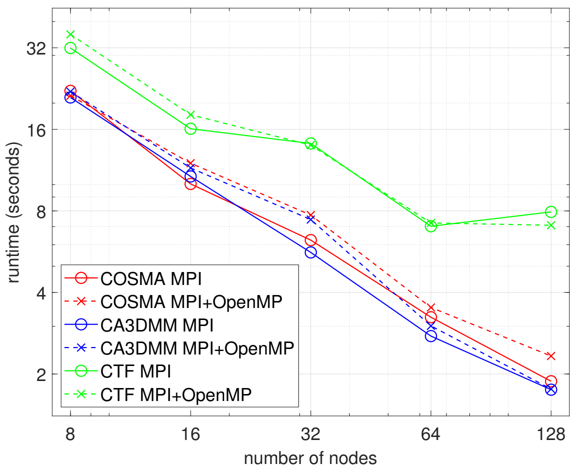
2.5 Conclusions and Open Problems

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 that 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.

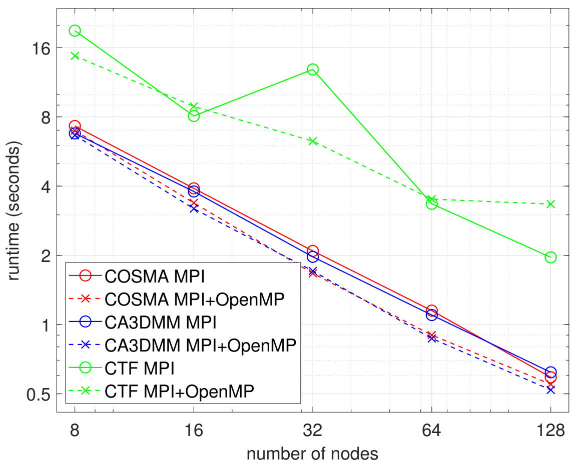
The first topic for future study is controlling the usage of extra memory in CA3DMM while minimizing communication costs. (Equation 2.10) suggests two possible approaches. The first approach is replacing Cannon's algorithm with the SUMMA algorithm. The SUMMA algorithm uses a tunable broadcast block size 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 2.3.5, 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 [9] and the repeated matrix multiplications in density matrix purification [10]. As Figure 2.3 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.

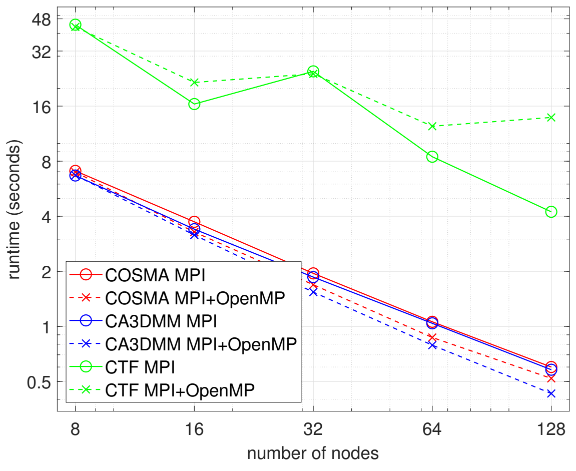
CA3DMM is released in open-source form at https://github.com/scalable-matrix/CA3DMM and is being integrated into the distributed-memory large-scale real-space density functional theory (DFT) program SPARC [33]. The need for a high-performance PGEMM for various matrix dimensions used in SPARC was the original motivation for developing CA3DMM.



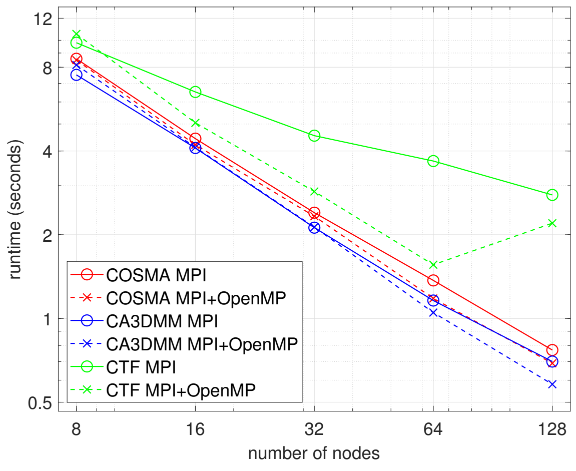
(a) $m = n = k = 50,000$



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



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



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

Figure 2.4: 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.

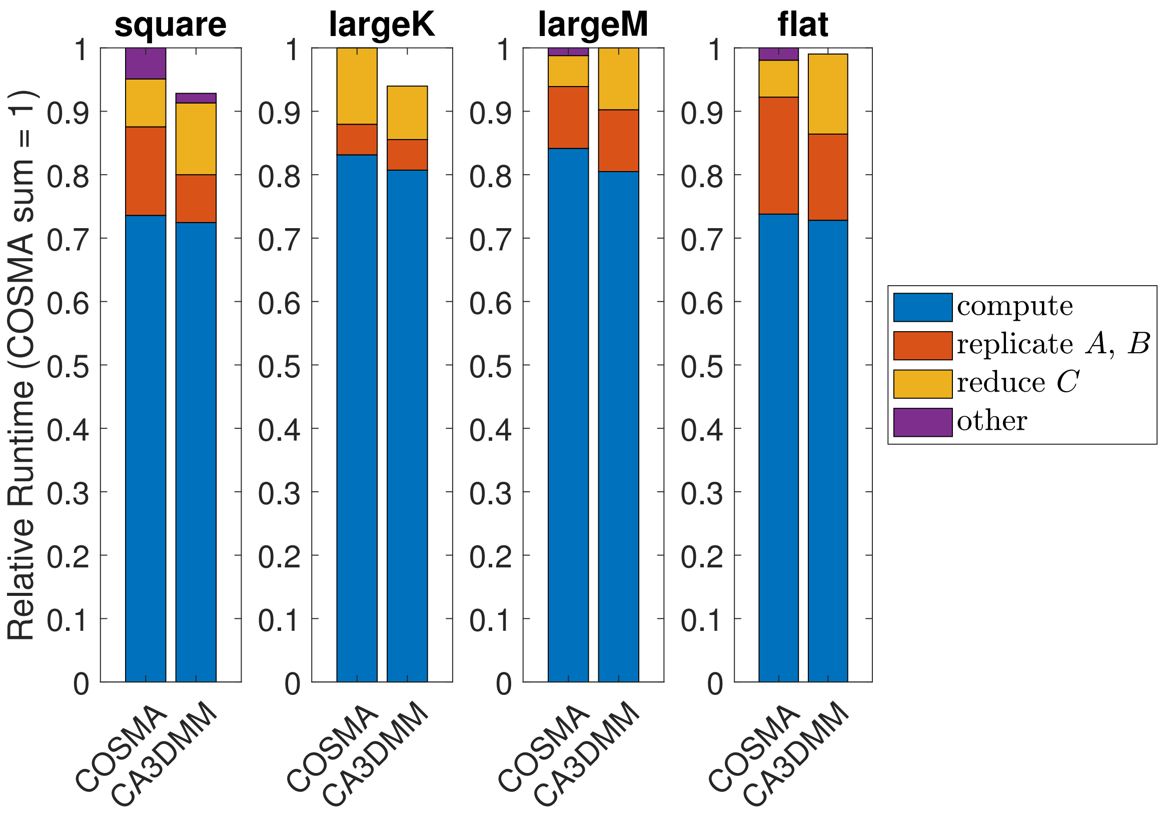


Figure 2.5: COSMA and CA3DMM relative runtime breakdowns for 2048-core tests in Table 2.2. For each class of problems, timings are normalized such that the total runtime of COSMA equals 1. For CA3DMM, "replicate A, B" includes step 5 in Algorithm 1 and the cost of shifting A and B blocks in Cannon's algorithm.

CHAPTER 3

SCALING UP POLAR DECOMPOSITION ON DISTRIBUTED-MEMORY COMPUTERS

3.1 Introduction

The polar decomposition (PD) decomposes a matrix $A \in \mathbb{C}^{m\times n}$ as

$$A = QH, \quad Q \in \mathbb{C}^{m\times n}, H \in \mathbb{C}^{n\times n},$$

where Q has orthonormal columns and H is Hermitian positive semi-definite. Polar decomposition has many applications [3]. In recent years, the use of polar decomposition to compute eigenvalue decomposition (EVD) and singular value decomposition (SVD) in parallel [29, 34, 35, 36] has garnered much attention from researchers. In this work, we focus on scaling up polar decomposition on large distributed-memory computer clusters.

The most straightforward approach to computing a PD is by using the SVD of the target matrix. Since we are interested in using PD to compute EVD or SVD, computing an SVD first is counterproductive. Instead, we focus on using iterative methods to compute PDs. A matrix iteration for PD has the form

$$X_{k+1} = f(X_k), \quad X_0 = A, \tag{3.1}$$

where f is a mapping function, and the output X_s is an approximation to the unitary polar factor, Q. We note that some matrix iterations require the iterates to be square or invertible. In this work, we assume A is square and invertible; the general case can be handled by transforming the original matrix into an invertible one using pivoted QR factorization.

One major attraction of matrix iterations for computing PD is that $f(X_k)$ only requires elemental matrix operations such as matrix multiplication, matrix inversion, and QR decomposition. Optimized distributed-memory implementations of these operations are available on many computing platforms. Recently, many implementations of iterative polar decomposition for distributed-memory platforms and hardware accelerators have been developed [34, 36, 37, 35]. These implementations are built on top of ScaLAPACK [22], the most widely used distributed-memory linear algebra library.

However, the rapid development of new high-performance computers brings new challenges to existing distributed-memory parallel iterative PD implementations. One new challenge involves the communication complexity of linear algebra algorithms. In the past decade, the communication cost lower bounds of many linear algebra operations have been proved, and parallel algorithms that achieve the communication cost lower bounds have been proposed [38, 13, 17, 39, 18]. Some of these communication-optimal algorithms require algorithm-specific matrix layouts, but ScaLAPACK requires the 2D block-cyclic (2DBC) matrix partitioning to balance arithmetic workload and matrix storage. Another challenge is exploiting hierarchical parallelism. ScaLAPACK was originally designed for MPI-only parallelization. The 2DBC partitioning may be unfavorable for MPI + OpenMP and MPI + GPU hybrid parallelization. Using a large block size in 2DBC makes process-local computation more efficient since more arithmetic operations are needed to saturate powerful new processors. On the other hand, using a small block size better balances the workload across MPI processes. In practice, many ScaLAPACK operations have better performance in pure MPI mode than in MPI + OpenMP mode. The new SLATE library [23] addresses hierarchical parallelization, but it still uses 2DBC partitioning and does not provide some communication-optimal algorithms.

In this work, we design multiple new hybrid iterative schemes and adopt new parallel linear algebra algorithms to scale up PD on distributed-memory cluster computers. The hybrid iterative schemes combine existing iterative PD methods to trade floating-point operations (*flops*) for parallel scalability. To better exploit hierarchical parallelism, we design and adopt new parallel algorithms that do not use 2DBC for matrix multiplication, matrix inversion, and column orthonormalization. Test results show that our implementations and the ScaLAPACK-based polar decomposition have a similar runtime on a small number of nodes, and our implementations outperform the ScaLAPACK-based implementation by achieving up to $1.8\times$ speedup on 128 nodes, demonstrating the potential of the new algorithms.

3.2 Iterative Polar Decomposition Methods and Their Matrix Operations

3.2.1 Iterative Polar Decomposition Methods

Four iterative methods for computing polar decomposition relevant to this work are Newton [3, 40], Newton-Schulz [3], Halley [41], and Zolotarev's functions [42] (for convenience, we call this method "ZOLO" in the rest of this Section). The mapping functions of Newton, Newton-Schulz, and Halley are

$$\begin{aligned}
\text{Newton}: f(X) &= \frac{1}{2}(X + X^{-H}),\\
\text{Newton-Schulz}: f(X) &= \frac{1}{2}X(3I - X^H X),\\
\text{Halley}: f(X) &= X(3I + X^H X)(I + 3X^H X)^{-1}.
\end{aligned}$$

The Zolotarev mapping functions are complicated and we will discuss them later in this section. Newton and Halley's methods are globally convergent. Newton-Schulz is convergent when $\|I - A^2\| < 1$, so in practical use A is usually first scaled by $\rho(A)$. Indeed, the spectral information of A is used to accelerate the convergence in the ZOLO method and the scaled versions of Newton, Newton-Schulz, and Halley. We focus on the scaled methods and ZOLO in this work.

Several scaling approaches have been proposed for the Newton method. An inexpensive approach proposed by Byers and Xu [40] is

$$X_{k+1} = \frac{1}{2}(\zeta_k X_k + \zeta_k^{-1} X^{-H}),\ X_0 = A. \tag{3.2}$$

ζ_k is the scaling factor in the k-th iteration:

$$\zeta_0 = (ab)^{-1}, \quad \zeta_1 = \sqrt{2\sqrt{ab}/(a+b)}, \quad \zeta_k = 1/\sqrt{\rho(\zeta_{k-1})} \text{ for } k \geq 2, \tag{3.3}$$

where $0 < a \leq \|A^{-1}\|_2^{-1} \leq \|A\|_2 \leq b$, $\rho(x) = (x + x^{-1})/2$. The scaled Newton (SN) in this work refers to the Newton method with Byers-Xu scaling.

For the Newton-Schulz method, Chen and Chow [43] suggested a stable scaled version (SSNS):

$$X_{k+1} = \frac{1}{2} a_k X_k (3I - a_k^2 X_k^2), \quad X_0 = A/\alpha, \tag{3.4}$$

where $\alpha \geq \|A\|_2$, l_0 is not larger than the smallest singular value of X_0, and

$$\frac{1}{2}\hat{a}(3 - \hat{a}^2) = t, \tag{3.5}$$

$$a_k = \min\left(\sqrt{\frac{3}{1 + l_k + l_k^2}}, \hat{a}\right), \tag{3.6}$$

$$l_{k+1} = \frac{1}{2} a_k l_k (3 - a_k^2 l_k^2), \tag{3.7}$$

where $t = 0.1$ is a prescribed parameter necessary for stability.

For the Halley method, Nakatsukasa *et al.* [41] proposed a scaling

$$X_{k+1} = X_k(a_k I + b_k X_k^H X)(I + c_k X_k^H X_k)^{-1}, \quad X_0 = A/\alpha, \tag{3.8}$$

where $\alpha \geq \|A\|_2$, l_0 is not larger than the smallest singular value of X_0, and

$$a_k = h(l_k), \quad b_k = (a_k - 1)^2/4, \quad c_k = a_k + b_k - 1, \tag{3.9}$$

$$l_k = l_{k-1}(a_{k-1} + b_{k-1}l_{k-1}^2)/(1 + c_{k-1}l_{k-1}^2), \tag{3.10}$$

$$h(l) = \sqrt{1+\gamma} + \frac{1}{2}\sqrt{8 - 4\gamma + \frac{8(2-l^2)}{l^2\sqrt{1+\gamma}}}, \quad \gamma = \sqrt[3]{\frac{4(1-l^2)}{l^4}}. \tag{3.11}$$

This scaling approach is called dynamically weighted Halley (DWH). In DWH, the mapping functions for the singular values are the best type-$(3,2)$ rational functions $R(x) = x\frac{a+bx^2}{1+cx^2}$ that approximate the scalar sign function. DWH has two mathematically equivalent versions. The QR-based DWH (QDWH) is a numerically stable version, it reads

$$X_{k+1} = \frac{b_k}{c_k}X_k + \frac{1}{\sqrt{c_k}}(a_k - b_k/c_k)Q_1Q_2^H, \tag{3.12}$$

where

$$\begin{bmatrix} \sqrt{c_k}X_k \\ I \end{bmatrix} = \begin{bmatrix} Q_1 \\ Q_2 \end{bmatrix} R \tag{3.13}$$

is a thin QR factorization. The Cholesky-based DWH (CDWH) is a faster but less stable version, it reads

$$Z_k = I + c_k X_k^H X_k, \quad W_k = \text{chol}(Z_k), \tag{3.14}$$

$$X_{k+1} = \frac{b_k}{c_k}X_k + \left(a_k - \frac{b_k}{c_k}\right)(X_k W_k^{-1})W_k^{-H}, \tag{3.15}$$

where chol(Z_k) is the Cholesky factor of Z_k.

The ZOLO method generalizes DWH by using type-$(2r+1, 2r)$ best rational approximation of the scalar sign function with $r \geq 1$ to accelerate the convergence. The scaled

Zolotarev function reads

$$\hat{Z}_{2r+1}(x;l) = \frac{Z_{2r+1}(x;l)}{Z_{2r+1}(1;l)} = \hat{M}x \prod_{j=1}^{r} \frac{x^2 + c_{2j}}{x^2 + x_{2j-1}}, \tag{3.16}$$

where

$$l' = \sqrt{1-l}, \quad K' = \int_0^{\pi/2} \left(1 - (l')^2 \sin^2\theta\right)^{-0.5} d\theta, \tag{3.17}$$

$$c_i = l^2 \frac{\text{sn}^2(u;l')}{\text{cn}^2(u;l')}, \quad u = \frac{iK'}{2r+1}, \quad i = 1, 2, \ldots, 2r, \tag{3.18}$$

$$\hat{M} = \prod_{j=1}^{r} \frac{1 + c_{2j-1}}{1 + c_{2j}}, \tag{3.19}$$

$\text{sn}^2(u;l')$ and $\text{cn}^2(u;l')$ are the Jacobi elliptic functions. The ZOLO method reads

$$X_{k+1} = \hat{M}\left(X_k + \sum_{j=1}^{r} a_j X_k \left(X_k^H X_k + c_{2j-1} I\right)^{-1}\right), \tag{3.20}$$

where

$$a_j = -\left(\prod_{p=1}^{r}(c_{2j-1} - c_{2p})\right) \cdot \left(\prod_{p=1, p\neq j}^{r} (c_{2j-1} - c_{2p-1})\right). \tag{3.21}$$

Equation (3.20) can be computed using QR decompositions:

$$\begin{bmatrix} X_k \\ \sqrt{c_{2j-1}} I \end{bmatrix} = \begin{bmatrix} Q_{j1} \\ Q_{j2} \end{bmatrix} R_j, \quad j = 1, 2, \ldots, r, \tag{3.22}$$

$$X_{k+1} = \hat{M}\left(X_k + \sum_{j=1}^{r} \frac{a_j}{\sqrt{c_{2j-1}}} Q_{j1} Q_{j2}^H\right). \tag{3.23}$$

In the rest of this work, we call the ZOLO method with type-$(2r+1, 2r)$ best rational approximation as ZOLO-r for convenience, for example, ZOLO-4 is the ZOLO method with $r = 4$.

3.2.2 QR Factorization

The QDWH method requires computing a QR factorization. Given $A \in \mathbb{R}^{m\times n}$ with $m \geq n$, the reduced QR factorization computes $A = QR$, where $Q \in \mathbb{R}^{m\times n}$ is a matrix with orthonormal columns and $R \in \mathbb{R}^{n\times n}$ is an upper triangular matrix with positive diagonal elements.

Many algorithms have addressed communication-efficient algorithms for the very tall-and-skinny case, $m \gg n$, of reduced QR factorization. The well-known CholeskyQR algorithm [1] computes R as the upper triangular Cholesky factor of the $n \times n$ Gram matrix $A^H A$ and obtains Q by solving $Q = AR^{-1}$. CholeskyQR is communication efficient for very tall-and-skinny matrices. Due to its numerical instability, however, CholeskyQR is rarely used in practice; the Q matrix computed by CholeskyQR has poor orthogonality depending on the condition number of A. To address this issue, CholeskyQR2 [44] applies CholeskyQR again to the Q matrix obtained by CholeskyQR. However, if the condition number of A is larger than $u^{-1/2}$, where u is the floating-point unit round-off, loss of orthogonality may still be an issue. For these cases, shifted CholeskyQR3 [31] can help, by first running CholeskyQR with a diagonal shift on the Gram matrix and then running CholeskyQR2. The tall-and-skinny QR (TSQR) [39] is another algorithm designed for minimizing the communication in QR factorization of tall-and-skinny matrices. It uses a tree reduction to compute the R matrix recursively. TSQR is unconditionally stable. Compared to CholeskyQR2, TSQR has fewer messages but requires more flops.

If a matrix is not tall-and-skinny, the canonical approach is the Householder QR algorithm [1], which is implemented in ScaLAPACK [22]. The blocked version of Householder QR in ScaLAPACK uses the standard Householder QR that builds one reflector vector at a time for panel factorization. Such vector-wise operations have large communication costs. The communication-avoiding QR (CAQR) [39] replaces vector-wise reflector construction in panel factorization with TSQR and modifies the rest of the blocked version of Householder QR to work with TSQR's output. Another communication-avoiding QR algorithm is the communication-avoiding CholeskyQR (CA-CQR) [45]. CA-CQR uses a 3D parallel matrix multiplication algorithm and a 3D parallel Cholesky factorization algorithm to minimize communication costs. CA-CQR also needs to be applied two or three times to get the numerically stable CA-CQR2 and shifted CA-CQR3.

3.2.3 Matrix Multiplication

Parallel GEMM operation is used in all iterative PD methods discussed in Section 3.2.1. We refer the reader to Section 2.2 for detailed discussions of parallel GEMM algorithms.

3.2.4 Matrix Inversion

The matrix inverse is explicitly used in the Newton method, the original Halley method, and the scaling versions of these two methods. The canonical approach for finding the inverse of A is performing a pivoted LU or Cholesky factorization first, then calculating A^{-1} using triangular solves. Previous studies showed this approach usually works well for the SN method in practice [46, 47]. A less common approach is computing the inverse matrix using Gauss-Jordan elimination. When solving linear systems, Gauss-Jordan with row pivoting "yields numerical solutions as good as those obtained by Gaussian elimination." [48] Gauss-Jordan also has a blocked version [49] that utilizes BLAS-3 operations for better performance. The numerical stability of blocked Gauss-Jordan, to the best of our knowledge, has not yet been studied. For better stability on ill-conditioned matrices, one can use methods based on orthogonal transformations. One of these methods is the bidiagonal reduction-based matrix inversion proposed in [40]. It requires two-sided orthogonal transformations to factorize $A = UBV^H$ with U, V unitary and B upper bidiagonal, making this method much more expensive than elimination-based methods.

3.3 Hybrid Polar Decomposition

A hybrid polar decomposition (HPD) method computes the Q matrix in two stages:

$$X_{k+1} = \begin{cases} f_1(X_k), & 1 \le k \le K_p \\ f_2(X_k), & k > K_p \end{cases} \tag{3.24}$$

where K_p is a small integer, both f_1 and f_2 are iterative polar decomposition methods. The HPD method was first introduced by Higham and Schreiber [50], where the HPD algorithm switches from a scaled version of the Newton method (f_1) to the original Newton-Schulz method (f_2) once the estimated $\|I - X_k^H X_k\|_1$ in each iteration is smaller than a prescribed threshold. Motivated by two observations, we extend [50] and consider other polar decomposition methods for f_1 and f_2.

The first observation is the convergence histories of different polar decomposition methods. This observation was not discussed in [50]. All iterative polar decomposition methods map the singular values of X_k from the interval $[l_k, 1]$ to $[l_{k+1}, 1]$, where $l_{k+1} = f(l_k)$ and $0 < l_k < l_{k+1} \le 1$. Therefore, with a proper $l_0 \le \sigma_{\min}$ in the first iteration, we can track the singular value ranges of X_k easily and switch from f_1 to f_2 with no extra cost. Figure 3.1 shows the convergence histories of SN, DWH, SSNS, ZOLO-2, and ZOLO-4 for a matrix with condition number 10^{16} (the largest condition number that can be handled by double-precision data type). The theoretical upper bound of $\kappa_2(X_k)$ decreases rapidly in the first several steps in SN, DWH, and ZOLO methods since they use rational polynomials for approximating the step function. SSNS uses only low-order polynomials, so it is unable to amplify the smallest singular value significantly in one iteration and requires more iterations to converge. SN, DWH, and SSNS have similar iteration counts if $\kappa_2(A) \le 2$, while SN, DWH, and ZOLO methods use no more than three iterations to reduce $\kappa_2(A)$ from 10^{16} to less than 100. Therefore, HPD uses SN or DWH or ZOLO methods as f_1 and uses SN or SSNS as f_2. We can easily read some HPD configurations from Figure 3.1, for example, 2 QDWH + 8 SSNS iterations. Figure 3.2 shows the convergence histories of some HPD configurations. We note that the first iteration of SN always transforms the singular values of the input matrix to $[\kappa_2(A)^{-1/2}, \kappa_2(A)^{1/2}]$, so the condition number of X_k does not change.

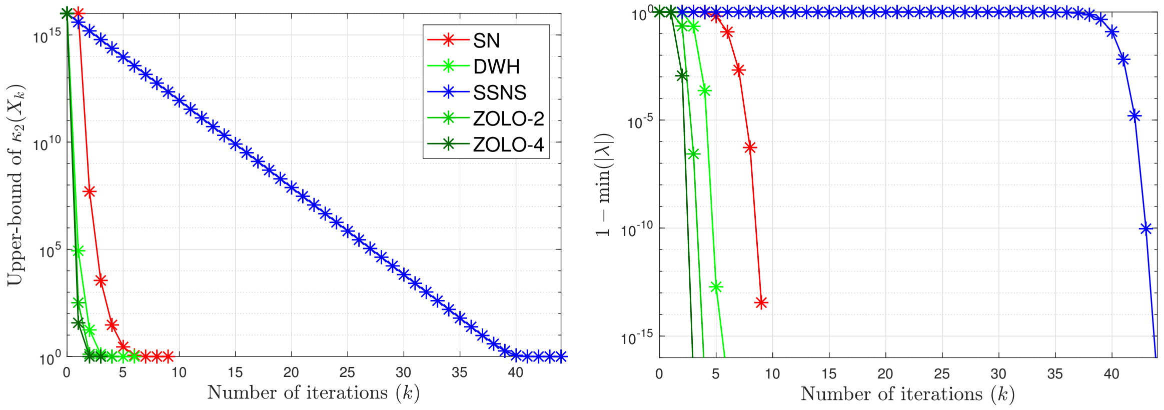


Figure 3.1: The change of $\kappa_2(X_k)$ (left) and $1 - \min(|\lambda|)$ (right) with respect to the number of iterations of different iteration polar decomposition methods for a matrix A with a condition number $\kappa_2(A) = 10^{16}$ and the maximum absolute eigenvalue $\max(|\lambda|) = 1$.

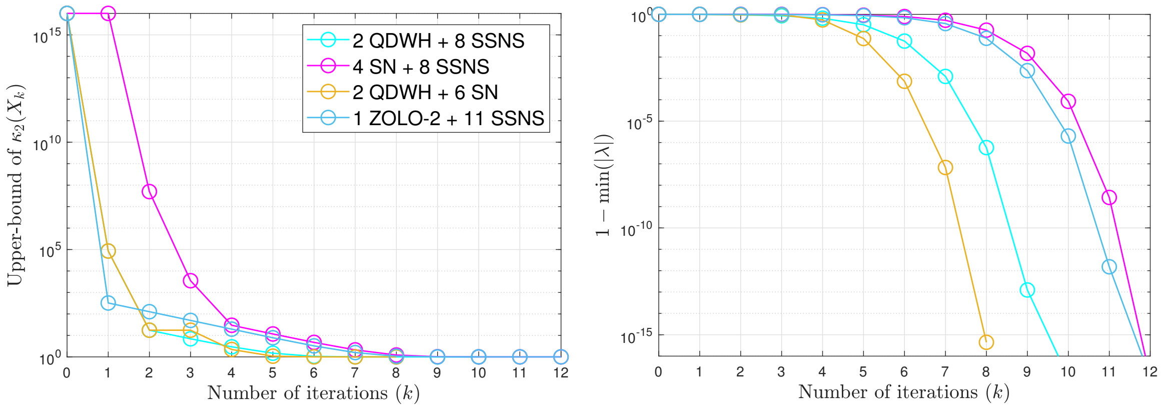


Figure 3.2: The change of $\kappa_2(X_k)$ (left) and $1-\min(|\lambda|)$ (right) with respect to the number of iterations of different HPD algorithms for a matrix A with a condition number $\kappa_2(A) = 10^{16}$ and the maximum absolute eigenvalue $\max(|\lambda|) = 1$.

The second observation is that different polar decomposition methods have different costs for each iteration. This observation is the main motivation for using HPD in [50], although it is unlikely that "a matrix multiplication can be done at twice the rate of matrix inversion" on a single node. We first list the flop counts of each iteration in different methods (highest order term only):

- SN: $22n^3/3$ or $2n^3$ flops using a bidiagonal reduction-based method [40] or a LU factorization,
- QDWH: $26n^3/3$ flops using Householder QR and treat all matrices as dense [29],
- CDWH: $10n^3/3$ flops [29],
- ZOLO-r: $26rn^3/3$ flops using Householder QR and treat all matrices as dense,
- SSNS: $4n^3$ flops.

On the other hand, HPD requires more iterations and may require more total flop counts than using SN or DWH only. Table 3.1 compares the flop counts of some HPD configurations with direct methods for achieving machine precision. Using 9 LU SN iterations has the smallest flop count, but it also has a numerical stability issue (to be discussed in Section 3.5.1). Two HPD configurations, 2 QDWH + 8 SSNS and 2 QDWH + 6 LU SN, require fewer flops than using QDWH only. The 2 QDWH + 4 CDWH configuration also requires fewer flops than six QDWH iterations, but we do not consider it a HPD configuration since it only uses the DWH formula.

To obtain good performance on distributed-memory platforms, the parallel scalability of linear algebra operations is usually more important than the flop counts. On modern supercomputers, inter-node communication is the performance bottleneck of many linear algebra algorithms. Using communication-reducing or communication-optimal algorithms is essential for obtaining good performance on a large number of computing nodes. The parallel implementations of linear algebra operations used in polar decomposition have different parallel scalability. HPD benefits from such differences, which we will discuss in Section 3.5.2 and Section 3.5.3.

Table 3.1: Flop counts of different approaches for calculating polar decomposition. "Stable SN" and "LU SN" refer to SN iteration using a bidiagonal reduction-based method and the LU decomposition.

Iteration Number and Type	*Total Flops*
9 LU SN	$18n^3$
3 stable SN + 6 LU SN	$32n^3$
6 QDWH	$52n^3$
2 QDWH + 4 CDWH	$30.67n^3$
4 ZOLO-2	$58.67n^3$
3 ZOLO-4	$104n^3$
44 SSNS	$176n^3$
2 QDWH + 8 SSNS	$49.33n^3$
2 QDWH + 6 LU SN	$29.33n^3$
1 ZOLO-2 + 11 SSNS	$61.33n^3$
4 stable SN + 8 SSNS	$56n^3$

3.4 Parallel Algorithms

In this section, we describe the new parallel algorithms we used in HPD and analyze their computation and communication costs. We count the total number of flops on the critical path as the computation cost. We assume butterfly network collectives that are optimal or near-optimal in the $\alpha - \beta$ model for communication cost analysis [28], 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, β is the inverse of network bandwidth, and the relatively small computation cost in all-reduce is ignored:

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

For convenience, we assume all divisions in the algorithms have no remainder.

3.4.1 Column Orthonormalization

In the QDWH algorithm, the Q matrix in QR factorization needs to be constructed explicitly. In fact, following the proofs in [41], QR factorization is unnecessary for QDWH. QDWH only needs an orthonormal column basis instead of a strict Q component in the QR factorization. From the parallel implementation point of view, it is very hard to balance the flops in Householder QR (or CAQR) and Cholesky factorization if a 2D matrix partitioning is used due to the shrinking of the trailing matrices in matrix factorization algorithms [1]. Since only an orthonormal column basis is needed, we have more algorithm options. An important observation is that permuting the columns does not invalidate an orthonormal column basis. This observation inspires us to use a simplified and blocked Modified Gram-Schmidt (MGS) with a 2D block matrix partition and a 2D process grid. Algorithm 2 shows the parallel simplified and blocked MGS (PSBMGS) algorithm with MATLAB colon notation, where C_i refers to the column panel $C \in \mathbb{R}^{m\times b}$ used in an iteration step.

Algorithm 2 Parallel Simplified and Blocked Modified Gram-Schmidt for Orthonormalization

```
Input: $p \times q$ process grid, matrix $A \in \mathbb{R}^{m\times n}, m \geq n$ is 2D-partitioned into $p \times q$ blocks,
    each process has a block $A_{ij}$ of size $(m/p)$-by-$(n/q)$, panel size $b$.
Output: Orthogonal matrix $Q \in \mathbb{R}^{m\times n}$ distributed in the same way as $A$.
 1: Process $P_{i,j}$ computes $cs_j = (j-1)b_q + 1$, $ce_j = jb_q$, $b_q = b/q$
 2: for $k = 1 : (n/b)$ do
 3:     $k_s = (k-1)b_q + 1$, $k_e = kb_q$
 4:     Processes $P_{i,j}, j \in [1, q]$ all-gather its $A_{ij}(:, k_s : k_e)$ as $C_i$ ($C_i$ has $m/p$ rows)
 5:     Processes $P_{i,j}, i \in [1, p]$ orthonormalize the columns of $C = [C_1^T, \cdots, C_p^T]^T$ using
        shifted CholeskyQR3 or CholeskyQR
 6:     Process $P_{i,j}$ locally stores $C_i(:, cs_j : ce_j)$ to $Q_{ij}(:, k_s : k_e)$
 7:     Process $P_{i,j}$ locally computes $G_{ij} = C_i^T \times A_{ij}(:, k_e + 1 : n/q)$
 8:     Processes $P_{i,j}, i \in [1, p]$ all-reduce sum $G_{ij}$ as $G_j$
 9:     Process $P_{i,j}$ updates $A_{ij}(:, k_e + 1 : n/q) = A_{ij}(:, k_e + 1 : n/q) - C_i \times G_j$
10: end for
```

To achieve machine precision, Algorithm 2 needs to be applied twice like the MGS2 algorithm. In the first run, the column panels C may be ill-conditioned, so a shifted CholeskyQR3 should be used in line 5. In the second run, the input matrix is wellconditioned. We use the CholeskyQR to reduce computation and communication costs.

Now we analyze the asymptotic complexity of Algorithm 2. The communication cost of line 4 in each iteration is $\alpha \log_2(q) + \beta\frac{mb(q-1)}{pq}$, so its total communication cost is

$$T_{panel} = 2\frac{n}{b}\left(\alpha \log_2(q) + \beta\frac{mb(q-1)}{P}\right).$$

In each CholeskyQR, each process needs to perform $2mb^2/p$, $b^3/3$, and mb^2/p flops in the Gram matrix computation, Cholesky factorization, and back substitution. The communication cost of each CholeskyQR is $2\alpha\log_2(p) + 2\beta b^2\frac{p-1}{p}$. There are $4n/b$ CholeskyQR runs in PSBMGS2, so its computation communication costs are

$$C_{chol} = \frac{12mnb}{p} + \frac{4nb^2}{3},$$
$$T_{chol} = \frac{8n}{b}\left(\alpha\log_2(p) + \beta b^2\frac{p-1}{p}\right).$$

Both lines 7 and 9 require $2\frac{m}{p}b\frac{n-kb}{q}$ flops per iteration, so it adds up to

$$C_{proj} = 4\frac{mn^2}{P} - 4\frac{mnb}{P}$$

flops. In each iteration, line 8 has communication cost $2\alpha\log_2(p) + 2\beta\left(\frac{m}{p}\frac{n-kb}{q}\right)\frac{p-1}{p}$. The total communication cost of line 8 is

$$T_{proj} = \frac{2n}{b}\alpha\log_2(p) + 2\beta\left(\frac{mn^2}{Pb} - \frac{mn}{P}\right)\frac{p-1}{p}.$$

The total computation and communication costs of PSBMGS2 are

$$C_{PSBMGS2} = C_{chol} + C_{proj}, \tag{3.25}$$

$$T_{PSBMGS2} = T_{panel} + T_{chol} + T_{proj}. \tag{3.26}$$

3.4.2 Blocked Gauss-Jordan

We use the blocked Gauss-Jordan to directly compute matrix inversions since it is simple and can work well with 2D matrix partitioning. Algorithm 3 describes the parallel blocked Gauss-Jordan (PBGJ) algorithm with MATLAB colon notation. For simplicity, once the diagonal block $D := A(d_s : d_e, d_s : d_e)$ is determined in an iteration, we partition A and B as

$$\begin{bmatrix} B_{11} & A_{1c}(B_{1c}) & A_{12} \\ B_{r1} & D & A_{r2} \\ B_{21} & A_{2c}(B_{2c}) & A_{22} \end{bmatrix}, \tag{3.27}$$

where B_{11} has size $(d_s - 1) \times (d_s - 1)$, D has size $(d_e - d_s + 1) \times (d_e - d_s + 1)$, and A_{22} has size $(m - d_e) \times (m - d_e)$.

Algorithm 3 Parallel Blocked Gauss-Jordan Matrix Inversion

Input: $p \times q$ process grid, matrix $A \in \mathbb{R}^{m\times m}$ 2D-partitioned into $p \times q$ blocks, each process has a block A_{ij} of size (m/p)-by-(m/q), panel size b.

Output: Matrix $B \in \mathbb{R}^{m\times m}$, $B = A^{-1}$ distributed in the same way as A.

```
1:  $d_s = 1$, fill $B$ with 0
2:  **while** $d_s < m$ **do**
3:      Maximize $d_e$ such that $d_e - d_s + 1 \leq b$ and $D := A(d_s : d_e, d_s : d_e)$ is on a single process, calculate process grid coordinate $(r_r, r_c)$ of the process that has current diagonal block $D$
4:      Process $P_{r_r,r_c}$ computes partially pivoted LU factorization $[L, U, pivot] = LU(D)$ and $B(d_s : d_e, d_s : d_e) = D^{-1}$
5:      Processes $P_{i,r_c}, i \in [1, p]$ broadcast column panel $A_{1c}, A_{2c}$ as root processes to all processes, $P_{r_r,r_c}$ also broadcasts the LU factors of $D$ together with the column panel
6:      Processes $P_{r_r,j}, j \in [1, q]$ compute $B_{r1} = D^{-1} \times B_{r1}$ and $A_{r2} = D^{-1} \times A_{r2}$ using the LU factors of $D$
7:      Processes $P_{r_r,j}, j \in [1, q]$ broadcast row panel $B_{r1}, A_{r2}$ as root processes to all processes, $P_{r_r,r_c}$ also broadcasts the LU factors of $D$ together with the column panel
8:      Processes $P_{i,r_c}, i \in [1, p]$ compute $B_{1c} = -A_{1c} \times D^{-1}$ and $B_{2c} = -A_{2c} \times D^{-1}$ using the LU factors of $D$
9:      Processes $P_{i,j}, i \in [1, r_r], j \in [1, r_c]$ compute $B_{11} = B_{11} - A_{1c} \times B_{r1}$
10:     Processes $P_{i,j}, i \in [1, r_r], j \in [r_c, q]$ compute $B_{12} = B_{12} - A_{2c} \times B_{r1}$
11:     Processes $P_{i,j}, i \in [r_r, p], j \in [1, r_c]$ compute $A_{12} = A_{12} - A_{1c} \times A_{r2}$
12:     Processes $P_{i,j}, i \in [r_r, p], j \in [r_c, q]$ compute $A_{22} = A_{22} - A_{2c} \times A_{r2}$
13:     $d_s = d_e + 1$
14: **end while**
```

In Algorithm 3, lines 8-12 are computed in parallel, and they are the most time-consuming parts of the algorithm since $m \gg b$ in most cases. It is easy to know that in each iteration, updating each element in D, B_{11}, B_{21}, A_{12}, and A_{22} requires $2b$ flops, while updating each element in B_{r1}, A_{r2}, B_{1c}, and B_{2b} requires b flops. Therefore, each process performs no more than $2bm^2/P$ flops in each iteration. The total computation cost of PBGJ is

$$C_{PBGJ} = \frac{2m^3}{P}. \tag{3.28}$$

The communication costs of lines 5 and 7 are

$$T_5 = \alpha\left(\log_2(q) + p - 1\right) + 2\beta\frac{mb}{p}\frac{q-1}{q},$$
$$T_7 = \alpha\left(\log_2(p) + q - 1\right) + 2\beta\frac{mb}{q}\frac{p-1}{p},$$

respectively. The total communication cost of PBGJ is

$$T_{PBGJ} = \alpha\frac{m}{b}\log_2(P) + 2\beta m^2\left(\frac{p+q-2}{P}\right). \tag{3.29}$$

3.4.3 Communication-Avoiding 3D Matrix Multiplication

The experiment results in Section 2.4 have demonstrated that CA3DMM can scale to a large number of processors with good parallel performance and scalability. To achieve optimal parallel performance, we utilize the CA3DMM algorithm for parallel GEMM in our implementation.

3.5 Numerical Experiments

3.5.1 Numerical Stability

We first examine the numerical stability of various polar decomposition algorithms, including hybrid versions. We use MATLAB for the experiments in this section. SN using LU factorization and Algorithm 3 are both implemented. We do not consider using the bidiagonal reduction-based matrix inversion in our parallel implementations (to be discussed in Section 3.5.2), so we also omit it in this section. QDWH using MATLAB built-in QR function and Algorithm 2 are both implemented. CA3DMM has no impact on numerical stability, so we use the built-in matrix multiplication implementation in MATLAB. We also implement a Gauss-Jordan-based DWH (GDWH) using formulas

$$Z_k = I + c_k X_k^H X_k, \tag{3.30}$$

$$X_{k+1} = \frac{b_k}{c_k} X_k + \left(a_k - \frac{b_k}{c_k}\right) X_k Z_k^{-1}, \tag{3.31}$$

where Z_k^{-1} is computed using Algorithm 3. For comparison, we implement SN, QDWH, CDWH, and ZOLO-2 using MATLAB's built-in `lu()`, `qr()`, `chol()` functions, and black-slash solver. Test matrices are generated using MATLAB function `gallery('randsvd', n, c)` where `n` is the matrix dimension and `c` is the condition number.

We measure two relative errors:

$$E_{orth} = \|X_k^H X_k - I\|_F / \|I\|_F, \tag{3.32}$$

$$E_{cs} = \|A - X_k(H + H^H)/2\|_F / \|A\|_F, \text{ where } H = X_k^H A. \tag{3.33}$$

E_{orth} measures the orthogonality of X_k. H is the Hermitian polar factor of A, E_{cs} accurately measures the preservation of the left singular vector space since $Q = UV$ holds for U and V given by the SVD $A = U\Sigma V^H$. Table 3.2 and Table 3.3 show the measured E_{orth} and E_{cs} of different polar decomposition algorithms for 500×500 matrices with different condition numbers. All algorithms without using SN have both E_{orth} and E_{cs} reaching the level of 10^{-15}. Therefore, we categorize algorithms without using SN as *stable algorithms*, and SN-related algorithms as *quasi-stable algorithms*. SN with LU factorization has E_{orth} and E_{cs} being an order of magnitude larger than algorithms without SN. The E_{orth} and

E_{cs} of SN are independent of $\kappa_2(A)$. Switching from LU to Algorithm 3 further decreases the accuracy of SN. This is reasonable since Algorithm 3 only performs pivoting in each diagonal block. In GDWH, Z_k is symmetric positive definitive (SPD), so pivoting does not have a large impact on the accuracy of Algorithm 3.

Table 3.2: E_{orth} of different polar decomposition algorithms for 500×500 matrices with different condition numbers. "MQR QDWH" means the QDWH is implemented using MATLAB built-in `qr()` function. Errors in italics do *not* reach 10^{-15} level.

$\kappa_2(A)$	1e3	1e6	1e9	1e12	1e15
9 LU SN	8.68e-15	8.65e-15	8.65e-15	8.64e-15	8.82e-15
9 Alg1 SN	*1.89e-13*	*5.74e-13*	*5.83e-13*	*3.40e-13*	*8.84e-13*
6 MQR QDWH	1.18e-15	1.15e-15	1.15e-15	1.13e-15	1.13e-15
6 Alg2 QDWH	8.49e-16	8.25e-16	8.52e-16	8.37e-16	8.42e-16
44 SSNS	8.95e-16	8.91e-16	9.03e-16	9.05e-16	9.12e-16
4 MQR ZOLO-2	1.23e-15	1.61e-15	1.05e-15	1.36e-15	1.18e-15
4 Alg2 ZOLO-2	1.09e-15	7.41e-16	7.58e-16	1.02e-15	9.83e-16
2 MQR QDWH + 4 CDWH	7.75e-16	7.83e-16	7.89e-16	7.86e-16	7.71e-16
2 Alg2 QDWH + 4 GDWH	7.85e-16	7.75e-16	7.77e-16	7.74e-16	7.85e-16
2 Alg2 QDWH + 8 SSNS	9.01e-16	8.92e-16	8.93e-16	8.97e-16	9.08e-16
1 Alg2 ZOLO-2 + 11 SSNS	8.97e-16	9.01e-16	9.53e-16	1.02e-15	9.01e-16
2 Alg2 QDWH + 6 LU SN	8.75e-16	8.70e-16	8.71e-16	8.55e-16	8.74e-16
2 Alg2 QDWH + 6 Alg1 SN	*1.89e-13*	*5.75e-13*	*5.50e-13*	*3.55e-13*	*1.21e-13*
4 Alg1 SN + 9 SSNS	5.21e-16	5.22e-16	5.24e-16	5.25e-16	5.25e-16
4 LU SN + 9 SSNS	5.23e-16	5.20e-16	5.17e-16	5.24e-16	5.23e-16

3.5.2 Linear Algebra Operation Scalability

We implement Algorithm 2, Algorithm 3, and CA3DMM in C + MPI + OpenMP and compare their parallel scalability with the counterparts in ScaLAPACK. All codes are compiled using Intel C compiler v19.0.5 with optimization flags "-xHost -O3". MVAPICH2 2.3.2 is used as the MPI backend. ScaLAPACK routines are provided by Intel MKL v19.0.5. We tested pure MPI parallel and different numbers of MPI processes + OpenMP threads combinations for ScaLAPACK routines, pure MPI parallel gives the best performance in most cases. Therefore, all ScaLAPACK routines use one CPU core per MPI process and a 64×64 block size. We use a 512 block size for both Algorithm 2 and Algorithm 3. All

Table 3.3: E_{cs} of different polar decomposition algorithms for 500×500 matrices with different condition numbers. "MQR QDWH" means the QDWH is implemented using MATLAB built-in `qr()` function. Errors in italics do *not* reach 10^{-15} level.

$\kappa_2(A)$	1e3	1e6	1e9	1e12	1e15
9 LU SN	*1.51e-14*	*1.64e-14*	*1.83e-14*	*1.94e-14*	*2.11e-14*
9 Alg1 SN	*8.30e-13*	*1.15e-11*	*5.53e-12*	*3.23e-12*	*7.45e-12*
6 MQR QDWH	1.91e-15	1.93e-15	1.78e-15	2.13e-15	2.29e-15
6 Alg2 QDWH	1.52e-15	1.56e-15	1.47e-15	1.88e-15	2.00e-15
44 SSNS	4.14e-15	4.42e-15	4.93e-15	5.33e-15	5.88e-15
4 MQR ZOLO-2	1.32e-15	1.28e-15	1.27e-15	1.56e-15	1.46e-15
4 Alg2 ZOLO-2	1.21e-15	9.59e-16	1.08e-15	1.27e-15	1.40e-15
2 MQR QDWH + 4 CDWH	2.41e-15	1.95e-15	1.96e-15	2.27e-15	2.36e-15
2 Alg2 QDWH + 4 GDWH	2.40e-15	1.96e-15	1.97e-15	2.27e-15	2.33e-15
2 Alg2 QDWH + 8 SSNS	1.81e-15	1.75e-15	1.64e-15	2.02e-15	2.12e-15
1 Alg2 ZOLO-2 + 11 SSNS	1.12e-15	1.22e-15	1.41e-15	1.72e-15	4.19e-15
2 Alg2 QDWH + 6 LU SN	*1.42e-14*	*1.28e-14*	*1.26e-14*	*1.23e-14*	*1.27e-14*
2 Alg2 QDWH + 6 Alg1 SN	*8.77e-12*	*8.34e-12*	*1.51e-12*	*1.61e-12*	*9.53e-13*
4 Alg1 SN + 9 SSNS	*3.17e-13*	*6.45e-12*	*3.09e-12*	*8.73e-13*	*1.37e-12*
4 LU SN + 9 SSNS	*1.04e-14*	*1.33e-14*	*1.57e-14*	*1.71e-14*	*1.81e-14*

tests are performed on the Georgia Tech Hive-Phoenix cluster. Each computing node of this cluster has two Intel Xeon Gold 6226 12-core processors and 192 GB DDR4 memory. Computing nodes are connected using a 100 Gb/s InfiniBand networking.

Figure 3.3 shows the strong scaling curves of different parallel linear algebra algorithms used in one polar decomposition iteration. CA3DMM has the best parallel scalability and shortest runtime since it has the smallest asymptotic communication cost. PBGJ scales a little better than ScaLAPACK LU factorization and matrix inversion. PSBMGS2 and ScaLAPACK QR decomposition have similar scalability, but PSBMGS2 is faster even if it requires more flops. As for ScaLAPACK QR, [39] shows two important points: (1) the parallel CAQR and the ScaLAPACK QR "match in the number of flops and words transferred" but the parallel CAQR sends fewer messages than ScaLAPACK QR thanks to using TSQR for panel factorization, and (2) the parallel CAQR attains the parallel QR communication cost lower bounds. Therefore, replacing ScaLAPACK QR with a parallel CAQR implementation, for example, the QR implementation in the SLATE library, can improve

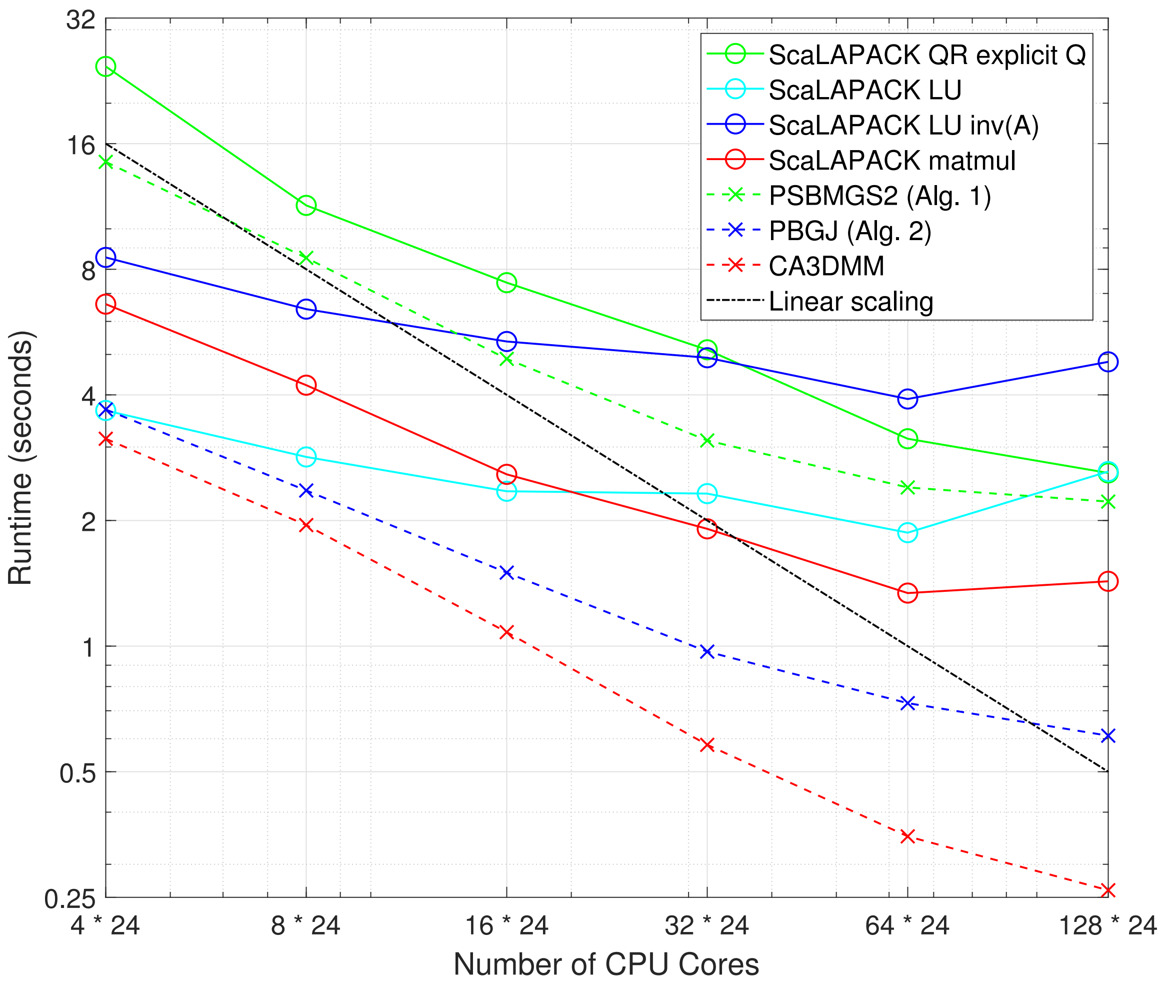


Figure 3.3: Strong scaling of different parallel linear algebra algorithms used in one polar decomposition iteration. Test matrix dimension is $n \times n$ for LU, matrix inversion, and matrix multiplication, $2n \times n$ for QR and PSBMGS2, $n = 20000$. All ScaLAPACK routines use 1 CPU core per MPI process, and their results are plotted with solid lines. PSBMGS2, PBGJ, and CA3DMM use 1 MPI process with 24 CPU cores per node, and their results are plotted with dashed lines.

the performance of QR but cannot reduce the communication complexity of QR. In other words, the parallel scalability of communication-optimal parallel QR algorithms will be very similar to that of ScaLAPACK QR. The difference in runtime and communication cost lower bounds between linear algebra operations justifies the use of HPD for scaling to a large number of nodes. The advantages of our implementations over ScaLAPACK routines show the importance of utilizing hierarchical parallelization for lower communication costs.

We do not consider the bidiagonal reduction-based matrix inversion mentioned in [40] since it is harder to implement than QR factorization but unlikely to make SN sufficiently faster than QDWH. The bidiagonal reduction-based matrix inversion has three steps:

1. Bidiagonal reduction: $A = UBV^H$ with U, V unitary and B upper bidiagonal,
2. Solve $BY = U^H$ for Y with back substitution,
3. Compute $A^{-1} = VY$.

One can use Householder reflectors to compute U and V, and the U matrix needs to be constructed explicitly for the second step. Therefore, both the bidiagonal reduction and the construction of U need $8n^3/3$ flops in the sequential implementation [1]. As a comparison, the Householder QR in QDWH requires $10n^3/3$ flops for the factorization of a $2n \times n$ matrix and another $10n^3/3$ flops for constructing the Q matrix from Householder reflectors. The bidiagonal reduction is not implemented in ScaLAPACK or other parallel linear algebra libraries. Since both the bidiagonalization and the Householder QR use Householder reflectors and communication operations are the performance bottlenecks of parallel Householder QR, the runtime of a carefully implemented parallel bidiagonalization for a $n \times n$ matrix is likely to be very similar to the runtime of a parallel Householder QR for a $2n \times n$ matrix. As a result, the runtime of one SN iteration using the bidiagonal reduction-based matrix inversion is likely to be very similar to the runtime of a QDWH iteration using a Householder QR. Meanwhile, Figure 3.1 shows that QDWH reduces the condition number of the matrix much faster than SN in the first three steps. Therefore, we think QDWH should be a better candidate for f_1 compared to SN with the bidiagonal reduction-based matrix inversion.

3.5.3 Hybrid Polar Decomposition Scalability

We implement three polar decomposition algorithms SN, QDWH, SSNS, and three HPD algorithms QDWH + SSNS, ZOLO + SSNS, QDWH + SN, using our parallel implementations of Algorithm 2, Algorithm 3, and CA3DMM. The code is compiled and tested using the same compiler and configurations as those in Section 3.5.2. For comparison, we also compile and test the POLAR library [37]. POLAR implements QDWH and CDWH using ScaLAPACK.

Figure 3.4 shows the strong scaling curves of different polar decomposition algorithms for a 20000×20000 random matrix. We set POLAR to run 2 QDWH + 4 CDWH iterations and only count the running time of these six iterations. The scaling trends of tested algorithms align with the trends in Figure 3.3. The comparison between algorithms leads us to some important conclusions.

First, we can balance performance, scalability, and numerical stability by choosing and combining different algorithms. Using SN solely is the fastest, but it is quasi-stable if the matrix inverse is calculated by LU or blocked Gauss-Jordan. Stable algorithms require a bidiagonal reduction-based matrix inversion (SN) or column orthonormalization (DWH and ZOLO), which has larger computation and communication costs, and worse parallel scalability than matrix multiplication, LU factorization, and blocked Gauss-Jordan.

Second, if an accurate solution is desired, HPD algorithms may run faster and have better parallel efficiency than direct PD methods. The 2 QDWH + 8 SSNS scheme is much faster than the 44 SSNS scheme, and its advantage over the 6 QDWH scheme becomes larger when running on a larger number of nodes. The 1 ZOLO-2 + 11 SSNS scheme is a little slower than the 2 QDWH + 8 SSNS scheme. Both schemes require two column orthonormalizations, but the advantage of doing two column orthonormalizations in parallel is canceled by three SSNS iterations in ZOLO-2. We can expect that other ZOLO-r + SSNS schemes will also have the same issue. Therefore, QDWH is usually better than ZOLO-r for the first iterative method in HPD. The runtime difference between the 2 QDWH + 6 SN and 2 QDWH + 8 SSNS schemes shows that SSNS is a better choice for the second iterative method in HPD when running on a large number of nodes.

Third, utilizing hierarchical parallelism and reducing communication costs are impor-

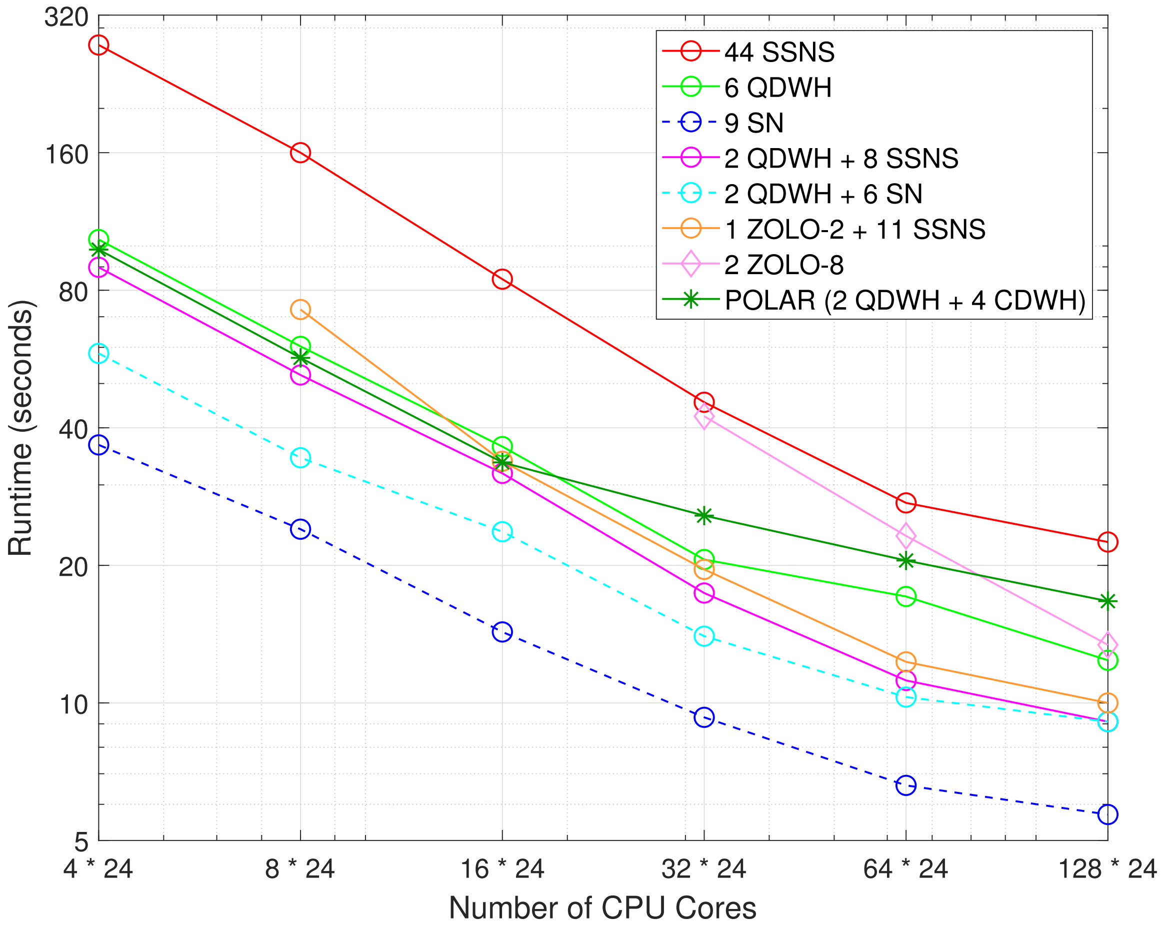


Figure 3.4: Strong scaling of different polar decomposition algorithms for a 20000×20000 matrix. SN, QDWH, ZOLO-2, and SSNS are implemented using Algorithm 3, Algorithm 2, and CA3DMM. The POLAR library implements QDWH and CDWH using ScaLAPACK. All algorithms except POLAR use one MPI process with 24 CPU cores per node. POLAR uses one MPI process per CPU core. Stable and quasi-stable algorithms are plotted with solid and dashed lines, respectively. The current test driver program for ZOLO-2 runs out of memory on four nodes.

tant for parallel algorithm scalability, even if it requires more flops. Six QDWH iterations with PSBMGS2 require $60n^3$, almost twice the $30.67n^3$ flops required by 2 QDWH + 4 CDWH in the POLAR library. However, POLAR is just a little faster than 6 QDWH with PSBMGS2 on 16 or fewer nodes, and POLAR is slower on 32 or more nodes.

3.6 Conclusions

In this work, we reviewed different types of polar decomposition algorithms and adopted the hybrid polar decomposition approach for scaling up this linear algebra operation to large supercomputers. HPD is driven by the convergence histories of different polar decomposition algorithms and the parallel scalability of linear algebra operations used in polar decomposition. We designed and implemented new parallel linear algebra algorithms to help scale up HPD with hierarchical parallelization. Numerical experiments show that our new parallel algorithms and HPD scale better than ScaLAPACK routines and ScaLAPACK-based polar decomposition implementations.

Experiment results in Section 3.5 also leave us with some future research topics. The first topic is the scalability of column orthonormalization. Although Algorithm 2 is faster than ScaLAPACK QR, its parallel efficiency drops very fast when scaling to a large number of nodes. QDWH is the best choice for obtaining high-accuracy solutions, and its scalability is bounded by column orthonormalization. The second topic is the possibility of improving the accuracy of blocked Gauss-Jordan without harming its parallel performance. SN converges rapidly and a blocked Gauss-Jordan is much cheaper than a column orthonormalization, improving the accuracy of blocked Gauss-Jordan makes SN more competitive.

CHAPTER 4
EXPLORING THE DESIGN SPACE OF DISTRIBUTED PARALLEL SPARSE MATRIX-MULTIPLE VECTOR MULTIPLICATION

4.1 Introduction

Linear algebra operations with sparse matrices are fundamental computational kernels in scientific computing, big data analysis, and artificial intelligence. While the sparsity of matrices greatly reduces computational costs, harnessing this sparsity poses a challenge. Therefore, accelerating sparse matrix linear algebra operations is crucial and extensively researched.

In sparse matrix linear algebra operations, sparse-dense matrix-matrix multiplication (SpMM) is an important building block. SpMM is used in block iterative solvers [51, 52, 53], dynamical simulations [54], non-negative matrix factorization (NNMF) [55, 56], graph neural network (GNN) training [57, 58, 59, 60], and deep neural network (DNN) training [61, 62]. SpMM calculations exhibit substantial parallelism, underscoring the need for efficient utilization of parallel resources to reduce computation time. Moreover, communication costs have been for a long time relatively more expensive than computation on both shared-memory and distributed-memory platforms. Reducing communication costs is the key to obtaining high performance in SpMM and other parallel algorithms. This work specifically concentrates on reducing *communication* costs in *distributed-memory* parallel SpMM.

A natural approach to parallelize SpMM is to treat it as sparse-dense matrix-vector multiplication (SpMV) with multiple input vectors and using the same parallelization as SpMV. This SpMV-based parallel SpMM approach can benefit from well-studied parallel SpMV algorithms. Methods such as (hyper)graph partitioning [63, 64, 65, 66] and other algorithms [67, 68] have been proposed for partitioning sparse matrices to reduce the communication costs of SpMV. A parallel SpMM implementation can directly reuse these sparse matrix partitionings and replace the local SpMV calculation with a SpMM routine to achieve good performance. However, SpMM introduces additional parallelism and allows partitioning the dense input vectors, enabling further reduction in communication costs and improvement in parallel performance.

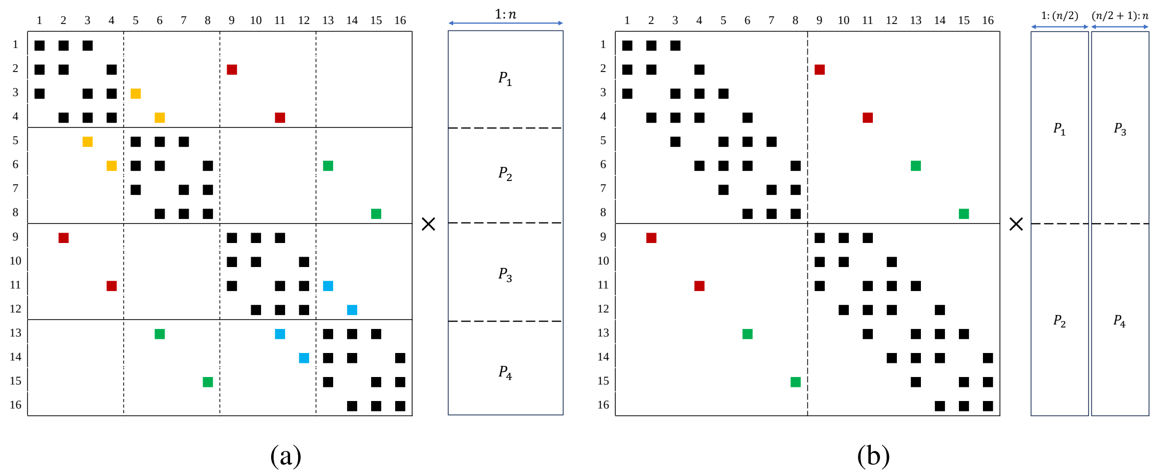


Figure 4.1: Two SpMM parallelization schemes for a 16×16 Laplacian matrix A and 4 processes: (a) parallelize over the rows of A only and (b) parallelize over both the rows of A and the columns of B. Scheme (b) needs to replicate A once but needs to replicate fewer B matrix rows (required by non-zeros in off-diagonal blocks, marked with colors).

Figure 4.1 shows an example of two SpMM parallelization schemes: partitioning the sparse matrix only like SpMV and partitioning both the sparse matrix and the input vectors. In the case of a relatively large number of input vectors, the second scheme requires a smaller communication size. Further, these two parallelization schemes are not the only two schemes for the given sparse matrix and four processes. Other schemes may further reduce the SpMM communication size. This example raises a fundamental question: *when and how should the parallelism of multiple input vectors be harnessed to reduce the communication costs of parallel SpMM?* No existing research has considered this issue.

In response to this inquiry, this work makes the following contributions.

- In Section 4.2, we first analyze the vast design space of parallelizing SpMM, compare various parallelization schemes, and position existing parallel SpMM algorithms within the design space.

- Section 4.3 formulates the communication cost models for different parallelization schemes and provides illustrative examples for these cost models.
- In Section 4.4, we propose an algorithm to optimize the process grid geometry, aiming to reduce communication costs with multiple levels of parallelism.
- Finally, in Section 4.5, we present theoretical analysis and numerical experiment results to demonstrate the efficacy of our new algorithm in reducing SpMM communication costs and achieving superior parallel performance compared to existing algorithms.

In this work, we assume that a process is not limited by memory size. Such limitations can be incorporated in practice but are omitted here for simplicity. We also assume that the number of vectors being multiplied simultaneously is large enough so that there are no additional efficiencies, e.g. from vectorization, gained by simultaneously multiplying more vectors. (When going from 1 to 4 vectors, there are great performance benefits of SpMM (time per vector), but not from, say 100 to 400 vectors, since the multiplication by 100 vectors is already highly vectorized.) In this regime, communication performance is much more important than vectorization and cache performance.

4.2 The Design Space of Parallelizing SpMM

Similar to the discussions in Section 2.2, parallel SpMM algorithms can also be categorized into 1D, 2D, or 3D algorithms. However, the algorithm discussed in Section 2.3 cannot be directly applied to SpMM since the sparsity of matrix A introduces both restrictions and opportunities in parallel algorithm design. In this section, we will explore the design space of parallelizing SpMM and position existing parallel SpMM algorithms within this design space.

4.2.1 1D Parallelization Schemes for SpMM

One-dimensional (1D) parallelization schemes are fundamental and commonly used approaches for parallelizing SpMV and SpMM. Categorized based on the partitioned dimension of the MM iteration space, 1D parallelization schemes include m-dimension, k-dimension, and n-dimension parallelization schemes.

An m-parallelization corresponds to a 1D row partitioning of both A and C. The rows of A and C are partitioned and assigned to different processes in a consistent manner. The rows of A and C owned by each process might be discontiguous. Each process needs to gather a portion of B matrix rows for its local SpMM calculation, with no additional communication required.

A k-parallelization corresponds to a 1D partitioning of both the columns of A and the rows of B. The columns of A and the rows of B are assigned to different processes in a consistent manner, and each process might have discontiguous columns of A. No communication is needed before the local SpMM computation. Following the local SpMM, the partial results of C need to be reduced and added to obtain the final C matrix.

To balance the computational workload, the rows of A owned by each process in the m-parallelization should contain approximately the same number of non-zeros. As a process may own discontiguous rows of A, various algorithms can be used to compute row partitionings of A with approximately balanced non-zero distributions. It should be noted that permuting the rows of A and assigning a row block with contiguous rows to a process is equivalent to assigning a set of possibly discontiguous rows of the original matrix to the same process. Similarly, a dual discussion holds for the k-parallelization.

An n-parallelization corresponds to a 1D column partitioning of both B and C. This method divides the original SpMM calculation into sub-tasks, each of which computes a portion of the input vectors through SpMM or SpMV operations. This approach is simpler than partitioning along the m or k dimension because different input vectors are identical in terms of both computation and communication. The only required communication involves replicating A between processes.

4.2.2 2D and 3D Parallelization Schemes for SpMM

The 1D parallelization schemes are independent of each other and can be combined to get 2D and 3D parallelization schemes. Without loss of generality, we arrange p processes as a 3D grid of size $p_m \times p_k \times p_n = p$, where $p_m, p_n, p_k \geq 1$. This 3D process grid results from combining a p_m-way m-parallelization, a p_n-way n-parallelization, and a p_k-way k-parallelization.

We first consider the 2D parallelization that partitions both the m and n dimensions. This mn-parallelization involves 1D row partitioning of A and 1D column partitioning of B, both of which are independent of each other. Thus, a p_m-way m-parallelization can be directly combined with a p_n-way n-parallelization. Similarly, a p_k-way k-parallelization can be directly combined with a p_n-way n-parallelization to obtain a kn-parallelization.

The direct combination of k-parallelization and m-parallelization will result in a 2D checkerboard partitioning of A, generally leading to an imbalance distribution of non-zeros. Below, we will introduce various algorithms that can be employed to calculate a 2D sparse matrix decomposition, ensuring a nearly balanced distribution of non-zeros and achieving an optimal or near-optimal SpMV communication cost. Nevertheless, the computation of a high-quality 2D decomposition could be computationally expensive.

The 3D mnk-parallelization can be considered as having p_n groups of processes, where each group's $p_m \times p_k$ processes perform the computation of multiplying A with n/p_n columns of B using a 2D partitioning of A. This naturally inherits all properties of mk-parallelization.

All 2D and 3D parallelization schemes inhabit an extensive design space that is spanned by two perpendicular subspaces. The first subspace involves the dimensions of a generalized 3D process grid $p_m \times p_n \times p_k = p$, where $1 \leq p_m, p_n, p_k \leq p$. The second subspace constitutes the solution space for partitioning the sparse matrix A. If p has numerous prime factors, multiple triplets satisfying $p_m \times p_n \times p_k = p$ can exist. Following the selection of the number of row and column partitions (the values of p_m and p_k), various algorithms can be used to compute a decomposition of A.

4.2.3 Existing Parallel SpMM Algorithms

In this section, we locate existing parallel SpMM algorithms in the huge design space of parallelizing SpMM, and discuss their strengths and weaknesses.

We first consider SpMV-based parallel SpMM algorithms. SpMV is one of the well-studied topics in parallel computing. Many optimizations for SpMV has been proposed, including

- New sparse matrix storage schemes (e.g., sliced ELLPACK [69], ESB [70], SELL-C-σ [71], CSR5 [72]);
- Workload partitioning and load balancing strategies (e.g., merge-based partitioning [73], adaptive partitioning for using the CSR format on GPUs [74, 75]);
- (Hyper)graph-based-partitioning schemes (e.g., [64, 65, 66]) and other partitioning algorithms (e.g., [67, 68]) for minimizing communication volume;
- Inspector-Executor (I-E) frameworks that inspect a matrix's sparsity structure and choose a matrix-specific storage format, parallelization scheme, and/or even generated source code (e.g., [76, 77, 78]).

Most of these optimizations fall into the categories of m-parallelization and mk-parallelization. For the latter, the sparse matrix can be partitioned using 2D checkerboard partitioning [66], 2D jagged-like partitioning (the sparse matrix is first partitioned into multiple row or column blocks, then each row or column block is partitioned independently) [66], 2D fine-grain partitioning (non-zeros are individually assigned to processes) [65, 66], and other methods like the Mondrian algorithm [67, 68] and the merge-based SpMV [73]. For improving the communication performance of distributed parallel SpMV, Bienz *et al.*[79]

also proposed a node aware parallel SpMV algorithm to utilize the knowledge of system topology for improving inter-node communication performance.

Driven by the need to accelerate GNN and sparse DNN, multiple new shared-memory parallel SpMM algorithms targeting multi-core CPUs [80, 81] and GPUs [82, 83, 84, 62, 59] have surfaced in recent years. These algorithms mainly focus on cache blocking, efficient vectorization, and other low-level hardware-oriented optimizations. Both new CPU SpMM algorithms use m-parallelization. Due to the difference in parallel programming models, some GPU SpMM algorithms can be categorized into either mk-parallelization or mnk-parallelization.

In terms of distributed-memory parallel SpMM, multiple studies reuse or modify parallel dense general matrix multiplication (GEMM) algorithms for SpMM. In the work of Koanantakool *et al.* [85], the authors assumed that the non-zeros in A are uniformly distributed and analyzed the communication costs of applying 1D, 2D SUMMA (mn-parallelization), and 3D SUMMA (mnk-parallelization) GEMM algorithms to SpMM directly. They further proposed three new 1.5D SpMM algorithms, which can be categorized into n-parallelization and nk-parallelization. Some application papers describe the parallel SpMM algorithms used by one or multiple higher-level driver algorithms. MPI_FAUN [56] uses a 1.5D algorithm (mk-parallelization) and CAGNET [57] uses a 2D algorithm (mn-parallelization). These algorithms are designed for handling tall-skinny B matrices. Selvitopi *et al.* [86] implemented these algorithms and another 2D algorithm (mn-parallelization) using both the bulk synchronous parallel model and the asynchronous parallel model with one-sided communication. All aforementioned GEMM-based parallel SpMM algorithms partition the sparse matrix A into equal-size blocks without considering the distribution of non-zeros. As a result, some A blocks can be empty, leading to severe load imbalance and a large number of unnecessary communications of B matrix elements.

Only a few parallel SpMM algorithms that consider the sparsity of A have been proposed. Acer *et al.* [87] proposed a generic framework that uses both graph and hypergraph partitioning for minimizing communication costs of SpMM using 1D m-parallelization. Recently, Gianinazzi *et al.* proposed an "arrow matrix decomposition" approach[88] for parallel SpMM. This approach decomposes the sparse matrix A of the form $A = \sum_{i=1}^{l} P_i B_i P_i^{\mathsf{T}}$, where P_i is a permutation matrix, B_i is an arrowhead sparse matrix with a user-specified tile size b. An L-shape decomposition is applied to B_i for parallelizing the SpMM calculation, without utilizing n-dimension parallelism. Each arrow decomposition requires a specific number of processes, which cannot be calculated easily, for running SpMM. Given that calculating an arrow decomposition is even more expensive than finding a 2D decomposition using hypergraph partitioning or the Mondrian algorithm, this method is hard to use in practice.

Some high-level multi-purpose libraries also support parallel SpMM computation. The Cyclops Tensor Framework (CTF) [25] uses the 2.5D algorithm [17] (mnk-parallelization) for parallel sparse and dense matrix multiplications. The Portable, Extensible Toolkit for Scientific Computing (PETSc) library [89, 90] also supports parallel SpMM computation. PETSc uses 1D row partitioning for sparse matrices [91], so its parallel SpMV and SpMM use 1D m-parallelization. Neither CTF nor PETSc is fine tuned for parallel SpMM.

4.3 Data Transfers in Parallel SpMM

In this section, we discuss the data transfer size, referring to the number of matrix elements requiring communication in a parallel SpMM algorithm. We will not consider any possible special properties (for example, symmetry) of A. Additionally, we assume that all p processes share one copy of A and B at the beginning, and one copy of C at the end.

4.3.1 Notation

We consider a parallel algorithm using p processes for computation. All processes have the same number of cores and the same size of memory. A network connects all processes and inter-process data exchange must go through the network. Let P be the grid of all

p processes. P can be organized as a 1D, 2D, or 3D grid, and is indexed with tuples of the same dimension. We use one-based indexing for the process grid and MATLAB colon notation to indicate a slice of processes or a slice of a matrix, for example, $P(2,:)$ refers to all processes in the second row of a 2D process grid, and $A(1:4,:)$ refers to the first four rows of matrix A. To represent row and column index sets, we use capital letters I and J, respectively. For example, if $I_1 = \{1, 2, 3, 4\}$, then $A(I_1,:)$ refers to the first four rows of matrix A. The $nnz(\cdot)$ function denotes the number of non-zeros in a matrix or matrix block. We introduce two new functions to facilitate the discussion of communication costs:

- The $nec(\cdot)$ function denotes the set of non-empty column (columns that have non-zeros) indices in a matrix block;
- The $ner(\cdot)$ function denotes the set of non-empty row (rows that have non-zeros) indices in a matrix block.

We use $|\cdot|$ to denote the size of a set. Finally, m, k, and n are the matrix dimensions defined in (Equation 1.1).

4.3.2 Data Transfers in 1D Parallelization

The m-parallelization. Denote the A matrix rows owned by process P_i as $A(I_i,:)$. The output matrix C is divided according to the row partitioning of A to avoid communication: process P_i owns $C(I_i,:)$ and computes $C(I_i,:) = A(I_i,:) \times B$. To avoid communication, we require that each row of B is owned by one process that uses this row in its local SpMM (a common practice for 1D row parallel SpMV). The only communication required in this parallelization scheme is gathering rows of matrix B. The total number of matrix elements to be transferred is

$$S_m(p) = \left(\sum_{i=1}^{p} |nec(A(I_i,:))| - k \right) n. \tag{4.1}$$

The k-parallelization. This scheme is analogous to an m-parallelization. Denote the column block of matrix A owned by process P_i as $A(:,J_i)$. Correspondingly, P_i owns

$B(J_i,:)$ and possesses $|ner(A(:,J_i))|$ rows of partial C matrix. We also require that each row of C is owned by one process with a partial result of this row to avoid communication. The total number of matrix elements to be transferred is

$$S_k(p) = \left(\sum_{i=1}^{p} |ner(A(:,J_i))| - m \right) n. \tag{4.2}$$

The n-parallelization. The B and C matrices are distributed in the same way. This scheme only requires replicating A $p-1$ times so that each process has a full copy of A. The total number of matrix elements to be transferred is

$$S_n(p) = (p-1) \cdot nnz(A). \tag{4.3}$$

4.3.3 Data Transfers in 2D and 3D Parallelization

We first consider the mn-parallelization and the kn-parallelization. As discussed in Section 4.2.2, these 2D parallelization schemes can be achieved by directly combining an existing m-parallelization or k-parallelization with an existing n-parallelization. Using the formulas in Section 4.3.2, the total number of matrix elements to be transferred in an mn-parallelization is

$$S_{mn}(p_m,p_n) = \left(\sum_{i=1}^{p_m} |nec(A(I_i,:))| - k \right) n + (p_n - 1) \cdot nnz(A), \tag{4.4}$$

and the total number of matrix elements to be transferred in a kn-parallelization is

$$S_{kn}(p_k,p_n) = \left(\sum_{i=1}^{p_k} |nec(A(:,J_i))| - m \right) n + (p_n - 1) \cdot nnz(A). \tag{4.5}$$

For an mk-parallelization, the total number of matrix elements to be transferred depends on the 2D sparse matrix decomposition algorithm. If a 2D checkerboard or a 2D jagged-like partitioning is used, the total data transfer size can be expressed in the form of

$$S_{mk}(p_m, p_k) = \sum_{i=1}^{p} |nec(A(I_i, J_i))| + \sum_{i=1}^{p} |ner(A(I_i, J_i))| - K, \tag{4.6}$$

where K is a constant that depends on the distribution of B and C for reducing communication. The total number of matrix elements to be transferred for the mnk-parallelization also depends on the 2D partitioning of A.

4.3.4 Examples of Parallel SpMM Data Transfer

Table 4.1 summarizes the frequently used notations and formulas in Section 4.3.2 and Section 4.3.3. We use an example to illustrate some the notations and formulas.

Table 4.1: Summary of notations used in Section 4.3.2 and Section 4.3.3

Notation	Meaning
$nec(\cdot)$	The set of non-empty column indices in a block
$ner(\cdot)$	The set of non-empty row indices in a block
$S_m(p)$	m-parallelization total communication size
$S_n(p)$	n-parallelization total communication size
$S_k(p)$	k-parallelization total communication size
$S_{mn}(p_m, p_n)$	mn-parallelization total communication size
$S_{kn}(p_k, p_n)$	kn-parallelization total communication size
$S_{mk}(p_m, p_k)$	mk-parallelization total communication size

We use Figure 4.1 as the example. Let $I_1 = \{1, 2, 3, 4\}$, $I_2 = \{5, 6, 7, 8\}$, $I_3 = \{9, 10, 11, 12\}$, and $I_4 = \{13, 14, 15, 16\}$. We have

$$nec(A(I_1, :)) = ner(A(:, I_1)) = \{1, 2, 3, 4, 5, 6, 9, 11\},$$
$$nec(A(I_2, :)) = ner(A(:, I_2)) = \{3, 4, 5, 6, 7, 8, 13, 15\},$$

and so on. We first consider a m-parallelization (Figure 4.1a). Process P_1 owns $B(I_1, 1 : n)$ and it needs to receive $B(\{5, 6, 9, 10\}, 1 : n)$ for computing $C(I_1, :) = A(I_1, 1 : n) \times B$. Process P_2 owns $B(I_2, 1 : n)$ and it needs to receive $B(\{3, 4, 13, 15\}, 1 : n)$ for computing

$C(I_2, :) = A(I_2, 1 : n) \times B$. Similar analysis can be applied to P_2, P_3 and P_4. The total communication size is

$$S_m(4) = \left(\sum_{i=1}^{4} |nec(A(I_i, :))| - 16 \right) n = 16n.$$

Now we consider an mn-parallelization for this matrix with $p_m = p_n = 2$ (Figure 4.1b). Let $\tilde{I}_1 = I_1 \cup I_2$ and $\tilde{I}_2 = I_3 \cup I_4$. Let P_1 and P_3 own $A(\tilde{I}_1, :)$, and let P_2 and P_4 own $A(\tilde{I}_2, :)$. Process P_1 now needs to receive $B(\{9, 11, 13, 15\}, 1 : n/2)$ for computing $C(\tilde{I}_1, 1 : n/2) = A(\tilde{I}_1, 1 : n) \times B(\tilde{I}_1, 1 : n/2)$. Similar analysis applies to P_2, P_3, and P_4. The total communication size is

$$S_{mn}(2, 2) = \left(\sum_{i=1}^{2} |nec(A(\tilde{I}_i, :))| - 16 \right) n + (2 - 1)nnz(A) = 8n + 60.$$

If $n \geq 8$, the mn-parallelization has a smaller communication cost than the m-parallelization.

4.3.5 Near-Optimal Parallel SpMM

As we discussed in Section 4.2.2, the parallelization of SpMM encompasses an extremely large solution space, defined by two perpendicular subspaces. Finding an optimal SpMM parallelization is thus very challenging. This is true not even mentioning that finding an optimal SpMM parallelization would involve hypergraph partitioning as a subproblem, which is known to be NP-hard [92].

Instead of finding an optimal SpMM parallelization, a near-optimal parallelization can be computed via a brute-force approach in a restricted design space. This involves enumerating all combinations of $p_m \times p_k \times p_n = p$, followed by computing a 1D or 2D partitioning of A that minimizes the data transfer size of parallel SpMV using $p_m \times p_k$ processes (the restricted design space), and then choosing the parallelization that has the lowest SpMM data transfer size. This approach is computationally expensive when there are many ways to factor p, or when A is very large.

In the next section, we propose an approach that is more practical than this brute-force approach, and that we also show can work well in practice.

4.4 The CRP-SpMM Algorithm

In this section, we present the Communication-Reduced Parallel SpMM (CRP-SpMM) algorithm. CRP-SpMM optimizes the process grid geometry for SpMM using the sparsity pattern of matrix A and the cost models developed in Section 4.3. It can be implemented easily by reusing existing 1D m-parallel SpMV / SpMM algorithms. In this section, we will continue to use the notation defined in Section 4.3.

4.4.1 Process Grid Geometry and Matrix Partitioning

The CRP-SpMM algorithm is grounded in the SpMM data transfer size models discussed in Section 4.3. The exhaustive approach outlined in Section 4.3.5 for identifying an optimal SpMM parallelization scheme can be very expensive. Instead of seeking the optimal solution, our objective is to employ a cost-effective method to identify a parallelization scheme with lower communication costs than the traditional 1D m-parallelization.

CRP-SpMM uses a generalized 2D mn-parallelization instead of a generalized 3D parallelization for three reasons. Firstly, the solution space of $p_m \times p_n = p$ is smaller than that of $p_m \times p_n \times p_k = p$ for the same p. Additionally, as discussed in Section 4.3.3, finding a 2D partitioning of a sparse matrix with a balanced non-zero distribution is more computationally expensive than finding a balanced 1D row partitioning. Lastly, handling irregular replications of B matrix rows is easier than dealing with irregular reductions of partial results in the C matrix. Hence, we opt for mn-parallelization over kn-parallelization. We also opt out the mk-parallelization since it is the traditional parallelization by using a 2D partitioning of A without utilizing n-dimension parallelism.

Given that we will use the generalized 2D mn-parallelization, CRP-SpMM must determine (1) a $p_m \times p_n = p$ process grid and (2) a 1D p_m-way row partitioning for the sparse matrix A. It seems that p_m of the process grid must be chosen first, followed by the 1D p_m-way partitioning. However, CRP-SpMM does these two steps together.

(Hyper)graph-partitioning-based 1D row partitioning methods can yield favorable solutions, particularly in terms of data transfer size. However, running these algorithms multiple times can be computationally expensive. One potential approach involves initially computing a p-way row partitioning of A, followed by generating different p_m-way row partitionings through the merging of row blocks from the initial p-way row partitioning. We are unaware of the existence of an algorithm for this specific task. A possible greedy approach involves iteratively merging two row blocks with the largest communication size until only p_m row blocks with approximately the same number of non-zeros remain. We leave this as a topic for future study.

Assigning discontiguous rows to the same process is equivalent to reordering the matrix rows and dividing A into multiple row blocks, each containing contiguous rows of A. The reversed Cuthill–McKee algorithm [93] and other algorithms can be used for reordering the sparse matrix and improve parallel SpMV performance [94, 95, 96]. CRP-SpMM allows the higher-level driver algorithm to select a proper reordering algorithm and other methods for calculating a 1D p-way row partitioning of A, and uses this partitioning as an input and a baseline in searching the 2D process grid.

CRP-SpMM uses a greedy algorithm to determine the 2D process grid geometry, utilizing the following formula for calculating communication size:

$$S(p_m, p_n) = r(A) \cdot \left(\sum_{i=1}^{p_m} |nec(A(I_i, :))| - k \right) n + f(A) \cdot (p_n - 1) \cdot nnz(A). \qquad (4.7)$$

Formula (Equation 4.7) differs from Formula (Equation 4.4) in two places. Firstly, Formula (Equation 4.7) takes the reuse of A into consideration. In iterative solvers and some other algorithms, A remains unchanged while B is updated between iterations. Hence, A only needs to be replicated once if $p_n > 1$. The constant $r(A)$ is used to specify the number of times A will be reused. A driver algorithm can estimate a lower bound of $r(A)$, or simply set it as 1 if the lower bound of $r(A)$ is hard to estimate. Secondly, Formula (Equation 4.7) accounts for the storage of a row or column index whenever a nonzero value is stored. Formula (Equation 4.7) contains the constant $f(A)$, defined as the total memory size required to store A divided by the total memory size needed to store only the non-zeros values (excluding their row and column indices). To illustrate, if A is stored in the compressed sparse row (CSR) format using a 4-byte integer data type for column indices and an 8-byte floating point data type for non-zeros values, $f(A) = 1.5$.

The search of a 2D process grid for minimizing Formula (Equation 4.7) is based on the following assumption.

Assumption 1. *The number of B matrix elements to be transferred in a 1D row-wise parallel SpMM increases monotonically with the number of processes.*

In other words, using a smaller p_m is likely to reduce the first term on the RHS of Formula (Equation 4.7). If this reduction outweighs the increase in the second term on the RHS, then a more favorable 2D process grid is identified. This observation also leads us to Assumption 2.

Assumption 2. *For the same sparse matrix, the optimal p_n increases when p and/or n increases.*

CRP-SpMM finds a 2D process grid that minimizes Formula (Equation 4.7) with a greedy algorithm detailed in Algorithm 4. Instead of exhaustively evaluating all possible combinations of $p_m \times p_n = p$, the greedy algorithm starts with a 1D row partitioning and iteratively increases p_n while calculating the communication size using Formula (Equation 4.7). Importantly, it maintains p_n as a factor of p through this process. Algorithm 4 computes a new p'_m-way 1D partition by merging multiple contiguous row blocks of the initial p-way 1D partitioning. This way of merging matrix blocks can, but not always, reduce the cost of replicating the rows of matrix B (the first term in the RHS of Formula (Equation 4.7)). If the matrix is derived from a structural grid, merging row blocks corresponding to adjacent subdomains can better reduce the communication costs.

In some cases, CRP-SpMM may reduce to either a 1D m-parallelization or a 1D n-parallelization. Let p_s denote the smallest prime factor of p. By using Formula (Equation 4.7), if $S(p/p_s, p_s) > S(p, 1)$, CRP-SpMM will reduce to a 1D m-parallelization. Conversely if $S(1, p) < S(p_s, p/p_s)$, CRP-SpMM will reduce to a 1D n-parallelization. This observation also gives rise to Corollary 1.

Corollary 1. *If the SpMV can be computed in a matrix-free way (for example, stencil calculation), $f(A) = 0$ and $p_n = \min(p, n)$ minimizes the SpMV / SpMM communication cost.*

Algorithm 4 Computing a 2D process grid for CRP-SpMM

```
Input: SpMM problem dimensions $m$, $k$, $n$, sparsity matrix $A$, number of processes $p$, a
    1D row partitioning $\{I_i\}_{i=1}^{p}$
Output: A 2D process grid $p_m \times p_n = p$
 1: Set: $p_m = p$, $p_n = 1$, and $p_f = -1$.
 2: Compute $S_{opt} = S(p_m, p_n)$ using Formula (Equation 4.7).
 3: Prime factorization $p = \prod_{i=1}^{s} p_i,\ p_i \geq p_{i+1}$.
 4: for $i$ = 1 to $s$ do
 5:     if ($p_i == p_f$) or ($p_n p_i > n$) then continue to next $i$.
 6:     Set: $p'_n = p_n p_i$, $p'_m = p/p'_n$.
 7:     Merge $p'_n$ contiguous row blocks of $\{I_i\}_{i=1}^{p}$ to get a new $p'_m$-way 1D partition.
 8:     Compute $S(p'_m, p'_n)$ using Formula (Equation 4.7).
 9:     if $S(p'_m, p'_n) < S_{opt}$ then
10:         Update: $p_m = p'_m$, $p_n = p'_n$, $S_{opt} = S(p'_m, p'_n)$.
11:         Set $p_f = -1$.
12:     else
13:         Set $p_f = p_i$.
14:     end if
15: end for
```

After selecting a $p_m \times p_n$ process grid, each of the p_m processes in process grid column j computes a column block $C(:, J_j) = A \times B(:, J_j)$ using 1D row-wise parallel SpMM. The matrix partitionings and communication operations for a 2D mn-parallel SpMM are detailed in Algorithm 5. Steps 1-3 in Algorithm 5 are conceptual and do not involve any

Algorithm 5 SpMM with a 2D mn-parallelization

```
Input: A 1D or 2D process grid $p_m \times p_n$, sparse matrix $A$ and dense matrix $B$ distributed on $p$ processes.
Output: 1D or 2D partitioned $C = A \times B$ distributed on $p$ processes
1: Partition $A$ into $p_m$ row blocks s.t. $nnz(A(I_i, :)) \approx nnz(A)/p_m, 1 \leq i \leq p_m$.
2: Partition $B$ into a $p_m \times p_n$ grid of equal size sub-matrices s.t. $B(I_i, J_j)$ has $k/p_m$ rows and $n/p_n$ columns.
3: Partition $C$ into a $p_m \times p_n$ grid of different size sub-matrices s.t. $C(I_i, J_j)$ has the same rows as $A(I_i, :)$ and the same columns as $B(:, J_j)$.
4: if $p_n > 1$ then process $P(i, j)$ replicates $A(I_i, :)$ in process grid row $P(i, :)$ using allgather operation.
5: Process $P(i, j)$ packs local $B$ matrix rows needed by other processes in process grid column $P(:, j)$ posts non-blocking send and receive operations.
6: Process $P(i, j)$ multiples the diagonal block of $A(I_i, :)$ with the local $B$ matrix rows.
7: Process $P(i, j)$ waits for all non-blocking receive operations to complete.
8: Process $P(i, j)$ multiples the off-diagonal block of $A(I_i, :)$ with the received $B$ matrix rows.
```

actual operation. The distributions of A, B, and C adhere to the requirements discussed in Section 4.3.2. The driver algorithm of CRP-SpMM may adjust the matrix distribution after computing a 2D process grid or redistribute A and B before calling Algorithm 5. Steps 5-8 in Algorithm 5 constitute a standard 1D m-parallelization SpMM. Additionally, Algorithm 5 assumes that all processes possess sufficient memory to store all required matrix blocks. The topic of incorporating memory constraints in our communication-reduced algorithm is left as a topic for future study.

4.4.2 Complexity Analysis of CRP-SpMM

In this section, we will analyze the number of arithmetic operations, memory usage, communication size, and communication latency of CRP-SpMM. We assume that $\min(p_m, p_n) > 1$, and we further assume butterfly network collectives for communication size and latency analysis [28], which are optimal or near-optimal in the α-β 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:

$$T_{\text{allgather}}(n, P) = \alpha \log_2(P) + \beta n \frac{P-1}{P},$$
$$T_{\text{alltoall}}(n, P) = \alpha(p-1) + \beta n.$$

We define the number of arithmetic operations F and the memory usage M as the maximum number of floating point operations and the maximum number of matrix elements (where each A matrix non-zero is counted as $f(A)$ elements), respectively, on any process in Algorithm 5. After step 5, process $P(i, j)$ stores $A(I_i, :)$ and $B(nA_i, J_j)$. Since $|nA_i| \leq k$ always holds, we have

$$F = \mathcal{O}\left(\frac{nnz(A)n}{p}\right), \tag{4.8}$$
$$M = \mathcal{O}\left(f(A) \cdot \frac{nnz(A)}{p_m} + \frac{kn}{p_n}\right). \tag{4.9}$$

We define the communication size Q and the communication latency L as the maximum number of matrix elements transferred and the maximum number of messages sent, respectively, by any process in Algorithm 5. Process $P(i, j)$ needs to receive $nnz(A)/p_m$ A matrix elements in the allgather operation and receive $|nA_i|n/p_n \leq kn/p_n$ B matrix elements in the alltoall operation, which gives

$$Q = \mathcal{O}\left(f(A) \cdot \frac{nnz(A)}{p_m} \cdot \frac{p_m - 1}{p_m} + \frac{kn}{p_n}\right), \tag{4.10}$$
$$L = \mathcal{O}\left(\log_2(p_n) + p_m\right). \tag{4.11}$$

4.4.3 CRP-SpMM and Shared-Memory Optimizations

CRP-SpMM focuses on minimizing *inter-process* communication costs in *distributed-memory* parallel SpMM. Its process grid geometry optimization is hardware-independent, presenting both advantages and disadvantages. On the positive side, CRP-SpMM can collaborate with hardware-oriented optimizations, such as techniques for enhancing vectorization, register and cache reuse, and ensuring performance portability. Some shared-memory optimization techniques, including novel sparse matrix storage schemes and inspector-executor (I-E) frameworks, also leverage sparsity structures to enhance SpMV and SpMM performance. These techniques run in parallel with the process grid geometry optimization in CRP-SpMM, offering the potential for a more substantial performance improvement when combined. However, on the negative side, there may be instances where CRP-SpMM determines a parallelization scheme in which each process only computes with a small number of input vectors in local SpMM. This scenario might be unfavorable for vectorization and could adversely impact the performance of local SpMM calculations.

4.4.4 Implementation of CRP-SpMM

We implement CRP-SpMM in C + MPI + MKL. We use the widely used compressed sparse row (CSR) format for matrix A with a 4-byte integer data type (`int`) for indices and an 8-byte floating point data type (`double`) for non-zeros. B and C are stored and used in row-major style. We note that CRP-SpMM can use other formats for A, B, and C. Algorithm 5 uses `MPI_Iallgatherv` and overlaps multiple non-blocking operations to better utilize the network bandwidth (a technique proposed in [32]). Algorithm 5 step 5 uses `MPI_Isend` and `MPI_Irecv`. For local SpMM computation, we use the MKL I-E sparse BLAS routine `mkl_sparse_d_mm`. We re-index the local A matrix block's columns to reduce memory usage and improve memory access locality. For example, before column re-indexing, a process needs to compute a local SpMM

$$\begin{bmatrix} 0 & a & b & 0 & c & 0 & d & 0 \\ 0 & 0 & e & 0 & f & 0 & 0 & g \end{bmatrix} \times [1, 2, 3, 4, 5, 6, 7, 8]^{\mathsf{T}}.$$

After column re-indexing, this process computes

$$\begin{bmatrix} a & b & c & d & 0 \\ 0 & e & f & 0 & g \end{bmatrix} \times [2, 3, 5, 7, 8]^{\mathsf{T}}.$$

Our 1D m-parallelization SpMM implementation (including re-indexing) is very similar to the parallel SpMV implementation in PETSc (Section 4.1 in [91]), while the only difference is that the PETSc's implementation uses the PETSc scalable communication layer instead of MPI routines.

4.5 Numerical Experiments

All experiments in this section are performed on the Georgia Tech PACE-Phoenix cluster. Each CPU node has two CPU sockets and 192 GB DDR4 memory. Each socket has an Intel Xeon Gold 6226 12-core processor. The compute nodes are connected with 100 Gbps InfiniBand networking.

We use 19 sparse matrices from different fields in numerical experiments. Table 4.2 shows the properties of these matrices. Matrix `Amazon` is from [57], while all other matrices are downloaded from the SuiteSparse Matrix Collection [97].

4.5.1 Reducing Communication Sizes with 2D Parallelization

In this section, we evaluate the theoretical communication sizes of 1D m-parallelization and 2D mn-parallelization SpMM to illustrate the impact of reducing SpMM communication sizes with 2D parallelization. We utilize Formula (Equation 4.7) with $r(A) = 1$ and $f(A) = 1.5$ for computing communication sizes.

We first consider the first eight matrices in Table 4.2, which are symmetric matrices obtained from 3D structural problems discretized by the finite element method (FEM). Typically, these matrices are partitioned using a domain decomposition method or a graph partitioning algorithm. For our study, we employ the widely used METIS library [63] to

Table 4.2: Properties of sparse matrices used in numerical experiments.

Matrix Name	# rows $\times 10^6$	# cols $\times 10^6$	$nnz(A)$ $\times 10^6$	Avg. nnz per row	Problem Kind
PFlow_742	0.7	0.7	37	50	3D structural
Serena	1.4	1.4	65	46	
Geo_1438	1.4	1.4	63	44	
StocF-1465	1.5	1.5	21	14	
Long_Coup_dt6	1.5	1.5	87	59	
Hook_1498	1.5	1.5	61	41	
Flan_1565	1.6	1.6	117	75	
Bump_2911	2.9	2.9	128	44	
com-Orkut	3.1	3.1	234	76	NNMF
Amazon	14.3	14.3	230	16	GNN
reddit	0.2	0.2	115	495	GNN
cage15	5.1	5.1	99	19	DNA
kmer_V2a	55.0	55.0	117	2	Protein
delaunay_n23	8.4	8.4	50	6	Graph
wb-edu	9.8	9.8	57	6	Graph
nlpkkt160	8.3	8.3	229	28	Optimization
Hardesty3	8.2	7.6	40	5	Visualization
ss	1.6	1.6	35	22	Semiconductor
circuit5M	5.6	5.6	59	11	Circuit

compute 1D row partitionings of these eight matrices. We implemented Algorithm 4 using C + OpenMP. Both METIS and our code are compiled using the Intel C/C++ compiler v19.1, and the tests are performed on a single computing node. The last 11 matrices in Table 4.2 come from different areas and lack a mesh or grid with a physical structure. As these matrices do not originate from a structured mesh or grid, we refrain from utilizing METIS for calculating 1D row partitioning (although some symmetric matrices may still employ METIS for this purpose). Instead, we use the simple 1D partitioning approach mentioned in Section 4.4.1.

Table 4.3 lists the communication sizes of 1D m-parallelization and 2D mn-parallelization SpMM with $p = 256$ processes and $n = 1024$ input vectors. Despite METIS providing high-quality 1D row partitionings for these sparse matrices derived from 3D FEM, Algorithm 4 demonstrates the capability to find better 2D process grid dimensions, resulting in further reductions in SpMM communication sizes. If a more effective algorithm for computing 1D row partitionings for different p_m values (Algorithm 4 step 7) is employed, the communication sizes of mn-parallelization can be further decreased. The cost of determining a 2D process grid is negligible in comparison to the cost of computing a 1D row partitioning using METIS.

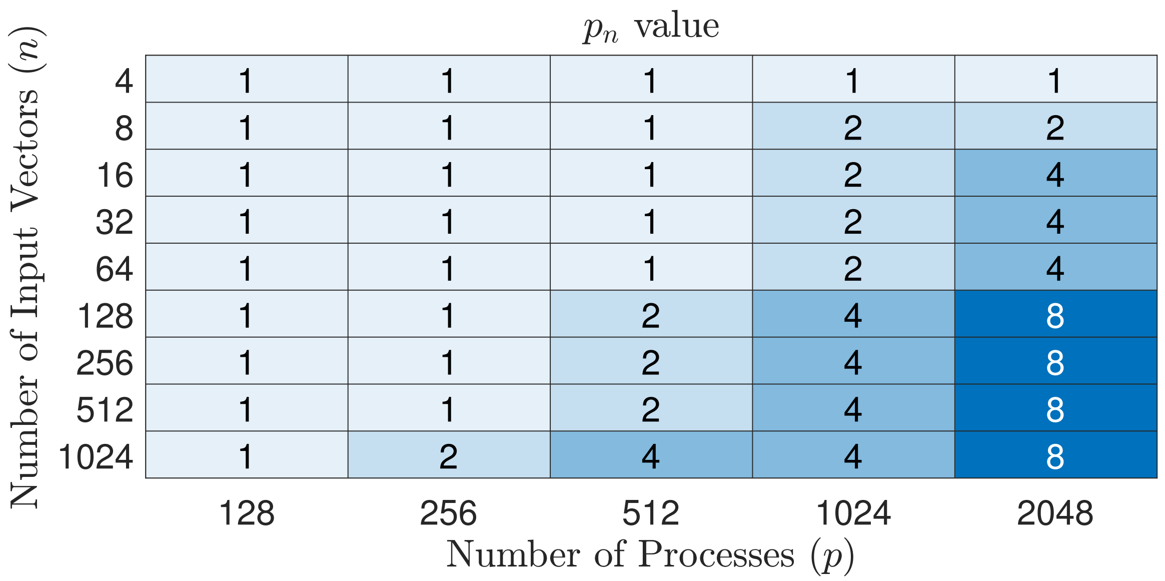
On non-FEM matrices, using a 2D parallelization scheme significantly reduces the total communication sizes for matrices `com-Orkut`, `Amazon`, `reddit`, and `ss`. Additionally, Algorithm 4 outputs a 1D process grid for matrices `delaunay_n23` and `wb-edu`. If a hypergraph partitioning algorithm is used for computing 1D row partitionings, determining the baseline p-way 1D row partitioning might be computational expensive. However, the cost of computing optimized 2D process grid dimensions using Algorithm 1 is likely to be negligible when compared to the cost of computing hypergraph partitioning.

Figure 4.2 shows the changes of 2D process grid dimensions $p_m \times p_n = p$ concerning the number of processes (p) and input vectors (n) for the `StocF-1465` matrix. The results in this figure align with Assumption 2. The optimized 2D process grid dimension

Table 4.3: The 1D partitioning (“PT”), 2D grid search (“GS”) timings (in seconds), and communication sizes (“CS”, number of double-precision words) of 1D m-parallelization (“m-para.”) and 2D mn-parallelization (“mn-para.”) for $p = 256$ processes and $n = 1024$ input vectors.

Matrix Name	1D PT Time (s)	2D GS Time (s)	m-para. CS ($\times 10^6$)	mn-para. $p_m \times p_n$	mn-para. CS ($\times 10^6$)
PFlow_742	8.08	0.01	359	128×2	315
Serena	5.78	0.01	666	128×2	625
Geo_1438	5.08	0.02	674	128×2	630
StocF-1465	4.14	0.01	442	64×4	361
Long_Coup_dt6	7.10	0.02	836	128×2	802
Hook_1498	5.13	0.01	620	128×2	557
Flan_1565	6.90	0.02	604	128×2	603
Bump_2911	13.22	0.04	1160	128×2	1136
com-Orkut	0.01	0.53	107752	8×32	25760
Amazon	0.01	2.17	120815	4×64	39391
reddit	0.01	0.14	34333	16×16	5893
cage15	0.01	0.15	29125	8×32	11803
kmer_V2a	0.01	1.83	115058	1×256	44835
delaunay_n23	0.01	0.13	10070	256×1	10070
wb-edu	0.01	0.13	737	256×1	737
nlpkkt160	0.01	0.24	22121	32×8	12566
Hardesty3	0.01	0.17	8805	8×32	3844
ss	0.01	0.06	6588	32×8	2655
circuit5M	0.01	0.11	22395	16×16	19821

experiences more rapid changes as n increases, primarily because the communication size for replicating B matrix rows increases proportionally to n. Similar observations can be applied to other sparse matrices with comparable application backgrounds.



	128	256	512	1024	2048
4	1	1	1	1	1
8	1	1	1	2	2
16	1	1	1	2	4
32	1	1	1	2	4
64	1	1	1	2	4
128	1	1	2	4	8
256	1	1	2	4	8
512	1	1	2	4	8
1024	1	2	4	4	8

Figure 4.2: Process grid dimensions for the `StocF-1465` matrix with different numbers of processes (p) and input vectors (n).

4.5.2 Parallel Performance of Parallel SpMM Algorithms

We test and compare five distributed-memory SpMM algorithms. The 1.5D A-stationary, 2D A-stationary, and 2D C-stationary algorithms are implemented using the bulk synchronous parallel model in the Combinatorial BLAS (CombBLAS) 2.0 library [86, 98]. We include our implementation of 1D m-parallelization SpMM and CRP-SpMM in the evaluation. CRP-SpMM directly reuses the 1D m-parallelization SpMM code. The CTF and PETSc libraries support distributed-memory parallel SpMM but are not specifically optimized for this task. As we explained in Section 4.2.3, the arrow matrix decomposition method is hard to use in practice, so we do not test it. CombBLAS stores sparse matrices in the compressed sparse column (CSC) format and only supports a square process grid. We compile CombBLAS and our code using the Intel C/C++ compiler v19.1 with optimization flags "-xHost -O3". Both codes use Intel MKL v19.1 for shared-memory parallel SpMM and MVAPICH2 2.3.6 for the MPI backend.

Table 4.4 lists the single SpMM execution time of all tested codes on 128 nodes for all matrices in Table 4.2. The running time of calculating 1D baseline partitioning and searching for 2D grid are also listed. All programs use 2 MPI processes per node. Each MPI process uses 12 OpenMP threads and is bonded to one CPU socket. CombBLAS uses a 16×16 process grid for all matrices. Both 1D m-parallelization and CRP-SpMM use METIS for the first 8 matrices, and both algorithms use a simple 1D row partitioning for the last 11 matrices. One-time initialization costs, including initializing MPI communicators and allocating work buffers, are not included in the reported values.

Table 4.4: Single SpMM execution time ("RT") of the best CombBLAS implementation ("CB-best"), 1D m-parallelization ("1D m-para."), and CRP-SpMM on 128 nodes with 2 MPI processes per node ($p = 256$) and $n = 256$ input vectors for all matrices in Table 4.2. The running time of calculating 1D baseline partitioning ("PT") and searching for 2D grid ("GS") are also listed. All timings are in seconds.

Matrix	CB-best	1D m-para.		CRP-SpMM		
Name	RT (s)	PT (s)	RT (s)	$p_m \times p_n$	GS (s)	RT (s)
PFlow_742	0.79	8.08	0.01	256×1	0.01	0.01
Serena	1.07	5.78	0.01		0.01	0.01
Geo_1438	1.22	5.08	0.01		0.02	0.01
StocF-1465	0.55	4.14	0.01		0.01	0.01
Long_Coup_dt6	1.59	7.10	0.02		0.02	0.02
Hook_1498	1.19	5.13	0.01		0.01	0.01
Flan_1565	2.20	6.90	0.01		0.02	0.01
Bump_2911	2.46	13.22	0.01		0.04	0.01
com-Orkut	2.31	0.01	0.62	16×16	0.30	0.30
Amazon	2.85	0.01	0.69	16×16	0.95	0.42
reddit	0.59	0.01	0.21	32×8	0.05	0.07
cage15	2.26	0.01	0.08	32×8	0.10	0.07
kmer_V2a	5.99	0.01	0.69	32×8	1.05	0.70
delaunay_n23	1.97	0.01	0.17	256×1	0.10	0.17
wb-edu	2.25	0.01	0.02	256×1	0.12	0.02
nlpkkt160	4.65	0.01	0.26	64×4	0.17	0.17
Hardesty3	1.40	0.01	0.03	16×16	0.13	0.03
ss	0.60	0.01	0.04	64×4	0.02	0.03
circuit5M	4.43	0.01	(N/A)	256×1	0.07	(N/A)

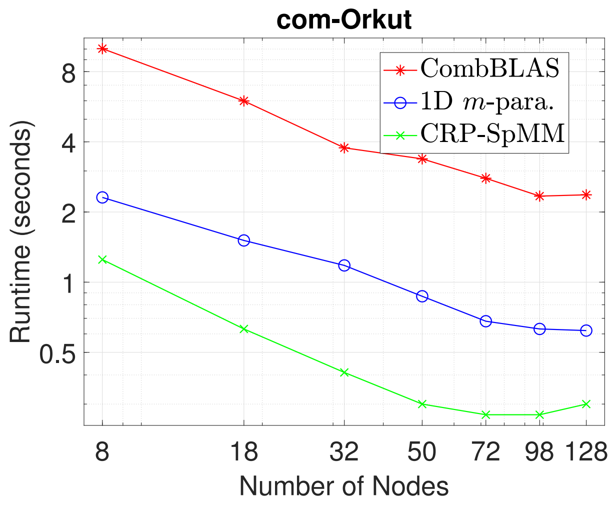
Overall, both 1D m-parallelization and CRP-SpMM greatly outperform CombBLAS across all tested matrices. As discussed in Section 4.2.3, the parallel SpMM algorithms implemented in CombBLAS are modified from parallel GEMM algorithms. For matrices `wb-edu`, `nlpkkt160`, `cage15`, and `Hardesty`, CombBLAS reported very high load imbalance ratios (defined as $p \max(nnz_i)/nnz$, where nnz_i is the number of non-zeros on process i) ranging from 15 to 18, which are much higher than other non-FEM matrices. Corresponding, the speedups of CRP-SpMM over CombBLAS on these four matrices, ranging from $28\times$ to $112\times$, are also much larger than the speedups on other non-FEM matrices. These results show that CombBLAS suffers from load imbalance and unnecessary communications due to non-uniform non-zero distributions of A. Selvitopi *et al.* implemented not only the bulk-synchronous version of these algorithms in CombBLAS, but also the asynchronous version using remote direct memory access (RDMA) operations for SpMM on GPUs [86]. Their work reveals multiple performance tradeoffs for SpMM on GPUs. Since our work does not involve GPUs, the subsequent discussion will focus on comparing 1D m-parallelization and CRP-SpMM.

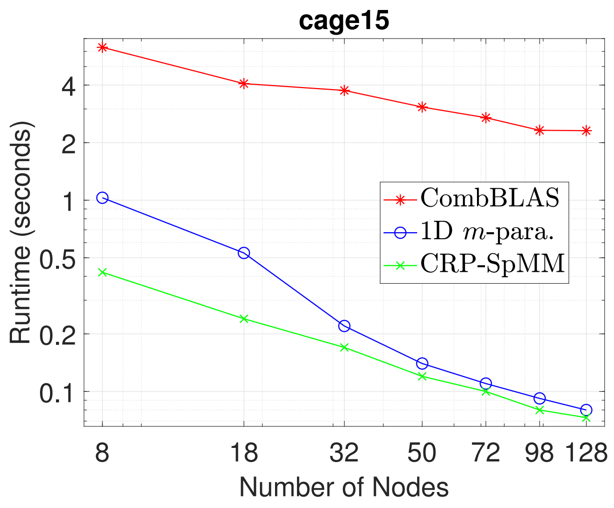
For the first 8 matrices, given that METIS already provides high-quality 1D row partitionings and the number of input vectors is not sufficiently large, CRP-SpMM reduces to 1D m-parallelization. We also tested $n = 1024$ for these 8 matrices (results omitted) on 256 nodes to evaluate the performance difference between 1D m-parallelization and the 2D mn-parallelization in CRP-SpMM. However, the running time of all 8 matrices under both parallelization schemes is remarkably small, falling within the range of 0.01 to 0.02 seconds. Such minuscule timings can be easily influenced by network performance fluctuations. Meanwhile, in Table 4.3, the differences in communication sizes between 1D and 2D parallelization schemes are marginal, making it challenging to discern a impact on execution time.

For the last 11 matrices, where both 1D m-parallelization and CRP-SpMM uses the same simple 1D row partitioning method, the speedup of CRP-SpMM over 1D m-parallelization aligns with the communication size comparison in Table 4.3. Notably, matrices `com-Orkut`, Amazon, reddit, and ss exhibit substantial speedups with CRP-SpMM over 1D m-parallelization. Additionally, two special cases are observed. For the kmer_V2a matrix, Table 4.3 shows that the communication size of 2D mn-parallelization is only slightly smaller than 1D m-parallelization. However, 2D mn-parallelization involves more communication operations and incurs a larger overhead cost. The circuit5M matrix has certain rows that contain over 5×10^6 non-zeros, resulting in one or more processes having a B matrix block larger than 4GB for local SpMM, causing a crash in the SpMM routine within Intel MKL.

Figure 4.3 shows the CombBLAS, 1D m-parallelization, and CRP-SpMM strong scaling results on matrices com-Orkut, cage15, and ss. CRP-SpMM runs much faster than CombBLAS on all three matrices. On the com-Orkut matrix, the CRP-SpMM communication size for using 128 nodes is slightly larger than that of using 98 nodes, which explains the increased runtime on 128 nodes. On the cage15 matrix, CRP-SpMM uses $p_n = 4$ and $p_n = 9$ for 8 and 18 nodes respectively, leading to a significant reduction of communication sizes compared to 1D m-parallelization. For 32 and more nodes, CRP-SpMM uses $p_n = 8$, the communication size difference between CRP-SpMM and 1D m-parallelization becomes smaller. Therefore, the speedup of CRP-SpMM over 1D m-parallelization also becomes smaller. On the ss matrix, the difference in communication size and performance between CRP-SpMM and 1D m-parallelization becomes larger on 98 and 128 nodes.

Figure 4.4 provides the runtime breakdown for 1D m-parallelization and CRP-SpMM in tests on 128 nodes for matrices com-Orkut, cage15, and ss. In both parallelization schemes, communication costs associated with replicating A and B dominate the overall runtime, with the communication costs of mn-parallelization being smaller than those of m-parallelization. On matrices com-Orkut and ss, we observe large communication size imbalances in the 1D m-parallelization reflected by the difference between the averaged and maximum running time of "Pack & Isend B". CRP-SpMM has a less severe load imbalance since it uses smaller p_m values than 1D m-parallelization. This also shows the





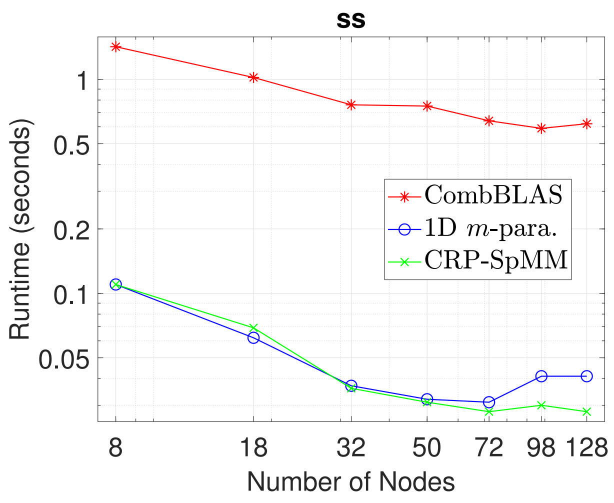


Figure 4.3: The CombBLAS, 1D m-parallelization ("1D m-para."), and CRP-SpMM strong scaling results on matrices `com-Orkut`, `cage15`, and `ss`. Each node runs 2 MPI processes. CombBLAS always uses a square process grid.

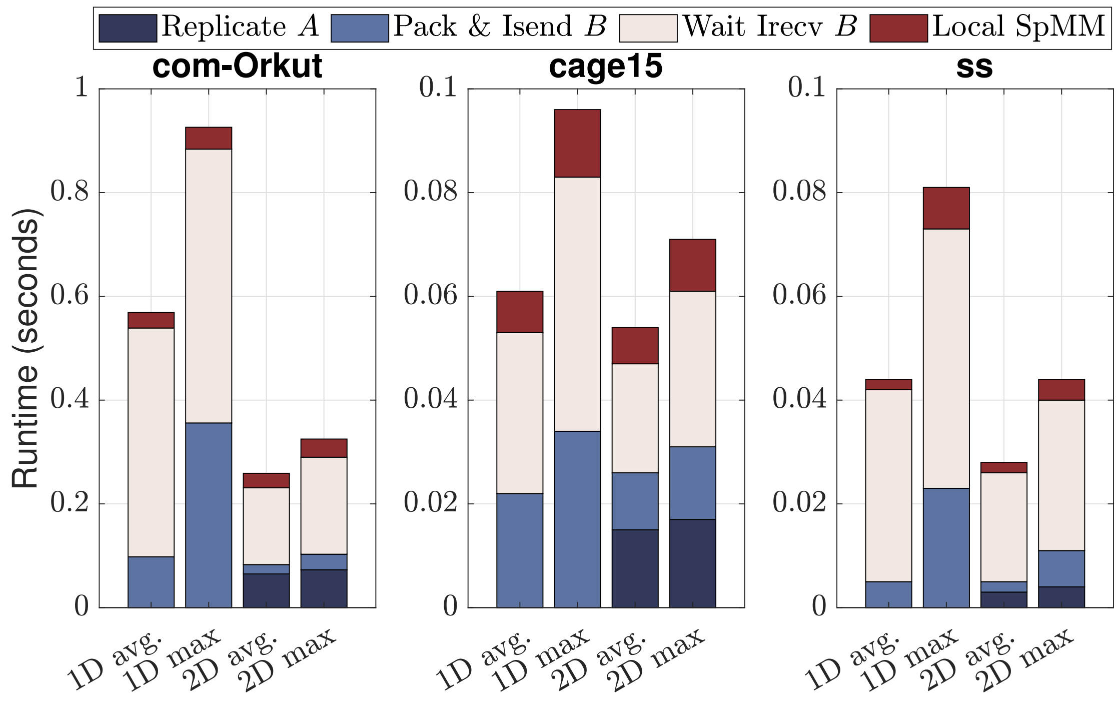


Figure 4.4: The 1D m-parallelization ("1D") and CRP-SpMM ("2D") runtime (in seconds) breakdowns for 128 node tests on matrices `com-Orkut`, `cage15`, and `ss`. The "avg." and "max" labels refer to the average and maximum value over all processes. "Replicate A", "Pack & Isend B", and "Wait Irecv B" refers to steps 4, 5, and 7 in Algorithm 5, respectively. "Local SpMM" refers to the total runtime of steps 6 and 8 in Algorithm 5.

necessary of using a better 1D partitioning algorithm to balance the communication sizes on different processes. Additionally, we observe that the cost of packing B matrix data could surpass the cost of local SpMM calculations. This issue can be alleviated by using MPI derived data types and other low-level performance optimizations. As our primary focus is not on low-level performance engineering in this work, we leave this issue for future study.

4.6 Conclusions and Future Work

In this study, we tackle the optimization of process grid geometry to harness different levels of parallelism, aiming to reduce the communication costs of parallel SpMM. We first analyze the design space of parallel SpMM algorithms and formulate communication cost models for different parallelization schemes. Based on these cost models, we propose an algorithm to optimize the process grid geometry. Our algorithm begins with a 1D row partitioning of the sparse matrix, generated through (hyper)graph partitioning or other methods. Subsequently, it employs a greedy algorithm to explore and discover more effective 2D process grid geometries. Numerical experiments show that our algorithm can find better process grid geometries, even when high-quality 1D row partitionings are used as baselines. Notably, it significantly reduces the communication sizes of SpMM in certain cases. For sparse matrices derived from structural grids or meshes and a small or moderate number of input vectors, using a 1D row parallelization with a graph partitioning or a domain decomposition partitioning is usually good enough. For a large number of input vectors or sparse matrices that arise in other areas, our algorithm is more likely to reduce the communication sizes. Furthermore, our implementation demonstrates a substantial performance advantage over existing distributed-memory parallel SpMM codes across a diverse range of process numbers and sparse matrices from different fields, exhibiting various sparsity patterns. Moreover, our theoretical analysis and experimental data offer insights into potential topics for future research.

The first topic involves designing a fast algorithm for computing 1D row partitionings for different p_m values (Algorithm 4 step 7) using an existing 1D p-way row partitioning. This may further reduce the communication sizes of mn-parallelization.

The second topic involves extending the parallel algorithm to support generalized 3D parallelization. This requires a careful redesign and implementation of the process grid geometry search algorithm to replace Algorithm 4, as well as the communication patterns in SpMM.

The third topic involves enhancing the performance of parallel SpMM through low-level optimizations. Leveraging MPI derived data types and/or MPI one-sided communication operations may help alleviate the overhead associated with manually packing and unpacking B matrix rows.

The code of this work is available in open-source form at https://github.com/scalable-matrix/CRP-SpMM. The methods proposed in this work and our codes can be integrated into higher-level algorithm libraries, for example, parallel iterative eigenvalue solver and non-negative matrix factorization libraries.

CHAPTER 5

H2PACK: HIGH-PERFORMANCE $\mathcal{H}^2$ MATRIX PACKAGE

5.1 Introduction

Many problems in scientific computing and data analytics, such as particle simulations with long-range interactions, the numerical solution of integral equations, and Gaussian process modeling led to dense *kernel matrices*. Given two sets of points, X and Y, and a non-compact kernel function $K(x, y)$, the kernel matrix $K(X, Y)$ has entries defined as $K(x_i, y_j)$ with all $(x_i, y_j) \in X \times Y$. Usually, kernel matrices have block low-rank structure, i.e., certain blocks of the matrices are numerically low-rank. For such a kernel matrix, representing these blocks in low-rank form gives a *rank-structured matrix representation* that asymptotically reduces the quadratic cost of matrix storage and matrix-vector multiplication. Different kernel matrices can be effectively stored in different rank-structured matrix representations such as $\mathcal{H}$ [99, 100], $\mathcal{H}^2$ [101, 102], and HSS [103]. In this work, we focus on the development of a library for efficiently constructing and using $\mathcal{H}^2$ matrix representations defined by non-oscillatory, translationally-invariant kernel functions with points in low-dimensional space (e.g., 2D or 3D).

$\mathcal{H}^2$ matrix representations are constructed by compressing specific matrix blocks into low-rank form via a nested approach. The compression of these blocks can be computed either *analytically* based on degenerate approximations of the kernel function such as multipole expansions and polynomial expansions, or *algebraically* based on matrix decomposition methods such as SVD, QR, and ACA [104]. It is worth noting that, when analytic compression methods are used, the fast matrix-vector multiplication of the constructed $\mathcal{H}^2$ matrix can be viewed as an algebraic variant of the fast multipole method (FMM) [105, 106, 107, 108]. Compared to algebraic compression, analytic compression usually requires less intermediate storage and computation but is limited to specific kernel functions and can give an approximation rank much larger than the numerical rank of the matrix block to be compressed. Algebraic compression, instead, is usually more effective in terms of the range of applicability and optimality of the approximation rank. Due to these differences, the matrix-vector multiplication with an $\mathcal{H}^2$ matrix constructed by algebraic methods is usually faster than FMM, since a lower-rank approximation leads to more cost reduction in the multiplication. As a sacrifice, algebraic methods usually lead to much more expensive $\mathcal{H}^2$ matrix construction than analytic methods. For example, simply evaluating all matrix entries has a quadratic cost, making many algebraic methods such as SVD unfavorable.

To balance between analytic methods and algebraic methods, we use a hybrid analytic-algebraic compression method called the *proxy point method* [109] to construct $\mathcal{H}^2$ matrix representations. For kernel functions from potential theory, such as the Laplace and Stokes kernels, Martinsson and Rokhlin [110] introduced the *proxy surface method* to efficiently compress specific kernel blocks into a low-rank form called interpolative decomposition (ID) [111]. Two variants of the proxy surface method were proposed later by Corona [112] and Minden [113]. All three methods belong to the class of the proxy point methods that has been formalized and studied in recent work [109]. Compared to algebraic methods, the proxy point method avoids forming a kernel block explicitly before compressing it and also requires far less data communication in parallel $\mathcal{H}^2$ matrix construction. Compared to analytic methods, the proxy point method can obtain better approximation ranks and is kernel-independent. As a result, it can efficiently construct an $\mathcal{H}^2$ matrix representation with linear complexity, while the constructed $\mathcal{H}^2$ matrix can have faster matrix-vector multiplications than FMM. Another common hybrid analytic-algebraic approach is to combine an analytic method with algebraic recompression [114, 115, 116]. Such an approach gives better approximation rank but is also restricted to certain kernels, like analytic methods. In comparison, the proxy point method incorporates the kernel function numerically when constructing the $\mathcal{H}^2$ matrix representation. This allows the construction to be kernel-independent.

H2Pack is a shared-memory parallel library for kernel matrices based on constructing $\mathcal{H}^2$ matrix representations using the proxy point method. The kernel functions must be non-oscillatory and *translationally-invariant* (i.e., $K(x, y) = k(x - y)$ with a univariate function $k(\cdot)$) with points in low-dimensional space. H2Pack library works for both scalar and tensor kernel functions. Presently, the library further requires the input kernel function to be symmetric, i.e, $K(x, y) = K(y, x)$, and the kernel matrix to be defined by one set of points X, i.e., $K(X, X)$. These two requirements can be easily lifted via a simple extension of H2Pack which will be addressed in the next version.

More precisely, H2Pack implements the following two components:

- $\mathcal{H}^2$ matrix construction based on the proxy point method with inputs being a kernel function $K(x, y)$, a set of points X, and an error threshold for the low-rank approximations;

- $\mathcal{H}^2$ matrix-vector multiplication.

We optimize H2Pack from both the algorithm and software perspectives. Different parallelization and load-balancing strategies are applied for different computation phases in H2Pack. Moreover, two running modes for H2Pack are available to adapt to different computing platforms and different problem settings for better performance. The ahead-of-time mode precomputes and stores all the components of an $\mathcal{H}^2$ matrix. The just-in-time mode calculates a large portion of the components dynamically when needed in $\mathcal{H}^2$ matrix-vector multiplication. These two modes provide a trade-off between storage and computation. The performance difference between the two modes depends on the memory bandwidth, CPU speed, and the complexity of the kernel function evaluations. It is worth noting that this two-mode approach has been proposed before in Refs [117, 116]. Lastly, we also exploit intrinsic functions for better vectorization to further improve the performance of H2Pack on multi-core and many-core processors.

Our numerical tests with H2Pack show that its $\mathcal{H}^2$ matrix construction cost is only around 5 to 15 times the corresponding $\mathcal{H}^2$ matrix-vector multiplication cost. Comparisons of H2Pack with two state-of-the-art FMM libraries, PVFMM [118] and FMM3D [119], show that H2Pack has asymptotically more expensive $\mathcal{H}^2$ matrix construction but faster $\mathcal{H}^2$ matrix-vector multiplications. More precisely, the $\mathcal{H}^2$ matrix construction cost in H2Pack is similar to the FMM setup costs in FMM3D and PVFMM for a low or moderate relative multiplication accuracy, e.g., 10^{-5} and 10^{-8}, and is just 2 to 5 times more expensive for a high relative multiplication accuracy, e.g., 10^{-11}. Meanwhile, the approximation ranks of blocks in H2Pack are 5 to 10 times smaller than those in PVFMM and FMM3D. As a result, the $\mathcal{H}^2$ matrix-vector multiplication in H2Pack is 1.5 to 5 times faster than in PVFMM and 5 to 25 times faster than in FMM3D in various tests. In practice, H2Pack is ideal for problems where many matrix-vector multiplications are required per configuration of data points, e.g., numerical solution of integral equations and Gaussian process modeling, so that the relatively expensive $\mathcal{H}^2$ matrix construction cost can be amortized.

Related work There are several libraries for FMM and its variants. FMM3D [119] implements the classical FMM [106, 120] for three key kernel functions in 3D from potential theory, the Laplace, Helmholtz, and Stokes kernels. PVFMM [118] implements the kernel-independent FMM [107] and works for kernel functions from potential theory. BBFMM3D [121] implements the black-box FMM [108] and works for smooth, translationally-invariant kernel functions. All these FMM libraries support OpenMP shared memory parallelization. PVFMM further supports MPI distributed memory parallelization and GPU acceleration of major FMM subroutines.

There are also several libraries for working with rank-structured matrices. H2Lib [122] constructs $\mathcal{H}^2$ matrix representations algebraically but only works for matrices from the boundary element method whose entries are kernel-defined interactions between compact basis functions in integral form. H2Lib supports OpenMP shared memory parallelization.

SMASH [116] uses a heuristic hybrid compression method to construct both $\mathcal{H}^2$ and HSS matrix representations for kernel matrices. SMASH is written in MATLAB and its C language implementation is still under development. STRUMPACK [123, 124] uses a randomized algebraic compression method to efficiently construct HSS matrix representations for a general class of dense matrices. STRUMPACK supports MPI distributed memory parallelization for fast matrix-vector multiplications and fast matrix solve. Recently, an $\mathcal{H}^2$ matrix library for GPUs has also been developed [125].

5.2 $\mathcal{H}^2$ matrix representation and $\mathcal{H}^2$ matrix-vector multiplication

Consider a kernel matrix $K(X, X)$ defined by a non-oscillatory kernel function $K(x, y)$ that is *translationally-invariant and symmetric*, and a set of points X in a low-dimensional space. This section describes an $\mathcal{H}^2$ matrix representation of $K(X, X)$, $\mathcal{H}^2$ matrix construction based on the proxy point method, and $\mathcal{H}^2$ matrix-vector multiplication. The following discussion applies to both scalar and tensor kernel functions $K(x, y)$, e.g., both the Laplace and the Stokes kernels.

5.2.1 $\mathcal{H}^2$ matrix representation

Interpolative decomposition

An interpolative decomposition (ID) [126, 111] represents or approximates a matrix $A \in \mathbb{R}^{n\times m}$ in the low-rank form UA_J, where $U \in \mathbb{R}^{n\times k}$ has bounded entries, $A_J \in \mathbb{R}^{k\times m}$ contains k rows of A, and k is the rank. An ID approximation defined this way is said to have error below the *error threshold* ε_0 if the 2-norm of each row of $A - UA_J$ is bounded by ε_0. Using an algebraic approach, an ID approximation with a given rank or a given error threshold can be calculated using the strong rank-revealing QR (SRRQR) decomposition [111] or using the pivoted QR decomposition. Specifically, an ID approximation of a kernel matrix block $K(X_0, Y_0)$ can be written as $K(X_0, Y_0) \approx UK(X_{\text{id}}, Y_0)$ where $K(X_{\text{id}}, Y_0)$ contains a subset of the rows in $K(X_0, Y_0)$ and X_{id} is a subset of X_0.

Hierarchical partitioning of X and $K(X,X)$

To construct an $\mathcal{H}^2$ matrix representation, the first step is to hierarchically partition the points in X. Assume X is in a d-dimensional space and let $\mathcal{B}$ be a box with equal-length edges that encloses X. The box $\mathcal{B}$ is partitioned into 2^d smaller same-sized boxes by bisecting all its edges. Each smaller box is further partitioned recursively in the same way until the number of points in a box is less than a prescribed constant. This hierarchical partitioning of $\mathcal{B}$ can be represented by a 2^d-ary *partition tree* $\mathcal{T}$ whose nodes correspond to the boxes. We define the root node of $\mathcal{T}$ to be at level 0, its children nodes to be at level 1, etc. We also define the leaf level to be level L.

Each level of the partition tree defines a non-overlapping partitioning of the set of points X. This partitioning is defined using the set of nodes at a given level of the partition tree. To generalize the concept of the set of nodes at a given level to the case of possibly non-perfect partition trees, let level$^+(l)$ denote the union of all the nodes in level l and all the leaf nodes above level l (toward the root). The caption of Figure 5.1 gives examples of level$^+(l)$ for an example partition tree.

Now, let X_i denote the set of points lying in box i and corresponding to node i in the tree. At any level l, $\{X_i\}_{i\in \text{level}^+(l)}$ defines a non-overlapping partitioning of the set of points X, i.e.,

$$X_i \cap X_j = \emptyset \text{ for distinct } i,j \in \text{level}^+(l) \quad \text{and} \quad \cup_{i\in\text{level}^+(l)} X_i = X.$$

For the kernel matrix itself, $\{K(X_i,X_j)\}_{i,j\in\text{level}^+(l)}$ defines a non-overlapping partitioning of $K(X,X)$. See Figure 5.1 for an example of a partition tree and the associated matrix partitioning at each level.

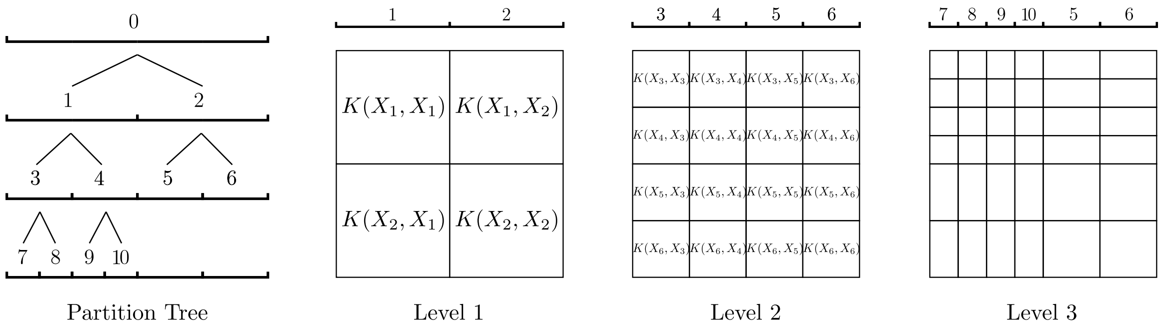


Figure 5.1: Illustration of a 3-level hierarchical partitioning of a set of points X in 1-dimensional space and the associated partitioning of a kernel matrix $K(X, X)$. In this partition tree, level$^+(1) = \{1, 2\}$, level$^+(2) = \{3, 4, 5, 6\}$, and level$^+(3) = \{7, 8, 9, 10, 5, 6\}$. In each level l, $K(X, X)$ is partitioned into non-overlapping blocks $K(X_i, X_j)$ with $i, j \in$ level$^+(l)$.

Inadmissible, admissible, and partially admissible blocks

In an $\mathcal{H}^2$ matrix representation, a kernel block $K(X_*, Y_*)$ is considered numerically low-rank if X_* is in a box and Y_* is in the *far field* of the box. The far field of a box is defined as the area of all the boxes that are least one box width away from the box. For any box i in some level l, we split boxes in level$^+(l)$ into two subsets $\mathcal{F}_i$ and $\mathcal{N}_i$ as

$$\mathcal{F}_i = \{k \in \text{level}^+(l) \mid \text{box } k \text{ is in the far field of box } i\} \quad \text{and} \quad \mathcal{N}_i = \text{level}^+(l) \setminus \mathcal{F}_i.$$

Let $Y_i = \cup_{k \in \mathcal{F}_i} X_k$ be the set of all points in the far field of box i. Then, $K(X_i, Y_i)$ for each box i with nonempty $\mathcal{F}_i$ is considered to be numerically low-rank. Thus, the numerically low-rank blocks at each level can be denoted as $K(X_i, Y_i)$ or $K(Y_i, X_i)$ for all nodes $i \in$ level$^+(l)$. Note that if $K(X, X)$ is symmetric, then $K(Y_i, X_i) = K(X_i, Y_i)^T$. See Figure 5.2 for an illustration of these low-rank blocks.

A block $K(X_i, X_j)$ that is contained in the low-rank blocks $K(X_i, Y_i)$ or $K(Y_j, X_j)$ is thus also low-rank. Based on this observation, the blocks in $\{K(X_i, X_j)\}_{i,j \in \text{level}^+(l)}$ can be categorized into three classes:

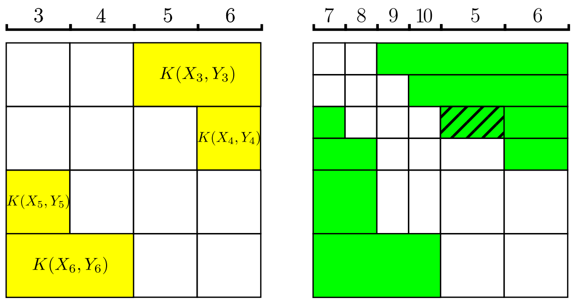
- *inadmissible blocks*, if $K(X_i, X_j)$ is not within $K(X_i, Y_i)$ and not within $K(Y_j, X_j)$ (equivalent to $X_j \cap Y_i = \emptyset$ and $X_i \cap Y_j = \emptyset$);

- *admissible blocks*, if $K(X_i, X_j)$ is within both $K(X_i, Y_i)$ and $K(Y_j, X_j)$ (equivalent to $X_j \subseteq Y_i$ and $X_i \subseteq Y_j$);

- *partially admissible blocks*, if $K(X_i, X_j)$ is within $K(X_i, Y_i)$ but not within $K(Y_j, X_j)$ (equivalent to $X_j \subseteq Y_i$ and $X_i \cap Y_j = \emptyset$), or if $K(X_i, X_j)$ is not within $K(X_i, Y_i)$ but within $K(Y_j, X_j)$ (equivalent to $X_j \cap Y_i = \emptyset$ and $X_i \subseteq Y_j$).

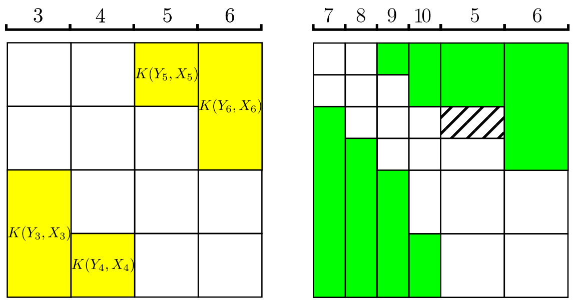
See the hatched block in Figure 5.2 for an example of a partially admissible block.

The concept of "partially admissible blocks" is new to the standard $\mathcal{H}^2$ matrix representation. More details follow later in this section.



Level 2 Level 3

(a) blocks $K(X_i, Y_i)$ with $i \in \text{level}^+(l)$, $l = 2$ or 3



Level 2 Level 3

(b) blocks $K(Y_i, X_i)$ with $i \in \text{level}^+(l)$, $l = 2$ or 3

Figure 5.2: Illustrations of the low-rank blocks $K(X_i, Y_i)$ and $K(Y_i, X_i)$ for the partition tree in Figure 5.1. The low-rank blocks are colored yellow for level 2 and green for level 3. For level 2, these blocks are labeled explicitly. In level 3, note that some of these blocks are not contiguous. The hatched block $K(X_9, X_5)$ is a partially admissible block since it is within $K(X_9, Y_9)$ in (a) but not within $K(Y_5, X_5)$ in (b).

Compression of low-rank blocks

We express a low-rank approximation of each $K(X_i, Y_i)$ in an ID form,

$$K(X_i, Y_i) \approx U_i K(X_i^{\text{id}}, Y_i), \tag{5.1}$$

where X_i^{id} is a subset of X_i and $K(X_i^{\text{id}}, Y_i)$ contains a subset of the rows in $K(X_i, Y_i)$. For a non-leaf box i with children $\{i_1, i_2, \ldots, i_s\}$, the above ID approximation is formed and computed by a nested approach (to be described in Section 5.2.2) in an $\mathcal{H}^2$ matrix representation. Precisely, the two ID components U_i and X_i^{id} are recursively defined in the nested form,

$$U_i = \begin{bmatrix} U_{i_1} & & \\ & \ddots & \\ & & U_{i_s} \end{bmatrix} R_i \quad \text{and} \quad X_i^{\text{id}} \subseteq X_{i_1}^{\text{id}} \cup X_{i_2}^{\text{id}} \ldots \cup X_{i_s}^{\text{id}} \subseteq X_i, \tag{5.2}$$

with some matrix R_i to be computed. Based on eq. (5.2), U_i for each non-leaf node is not explicitly formed but can be recovered recursively from quantities at all the descendants of node i.

Each inadmissible block $K(X_i, X_j)$ is considered to be full-rank. Each admissible block $K(X_i, X_j)$ is numerically low-rank and can be compressed as

$$K(X_i, X_j) \approx U_i K(X_i^{\text{id}}, X_j^{\text{id}}) U_j^T, \tag{5.3}$$

based on the compression of $K(X_i, Y_i)$ and $K(Y_j, X_j)$ in eq. (5.1). Each partially admissible block $K(X_i, X_j)$ can be compressed as

$$K(X_i, X_j) \approx \begin{cases} U_i K(X_i^{\text{id}}, X_j) & \text{if } K(X_i, X_j) \text{ is within } K(X_i, Y_i) \\ K(X_i, X_j^{\text{id}}) U_j^T & \text{if } K(X_i, X_j) \text{ is within } K(Y_j, X_j) \end{cases}, \tag{5.4}$$

based on the compression of $K(X_i, Y_i)$ or $K(Y_j, X_j)$ in eq. (5.1).

$\mathcal{H}^2$ matrix representation

The $\mathcal{H}^2$ matrix representation of $K(X, X)$ consists of three parts:

- dense inadmissible blocks $K(X_i, X_j)$ with both i and j being leaf nodes.
- low-rank approximations eq. (5.3) of all the admissible blocks $K(X_i, X_j)$ that are

not contained in larger admissible or partially admissible blocks.

- low-rank approximations eq. (5.4) of all the partially admissible blocks $K(X_i, X_j)$ that are not contained in larger admissible or partially admissible blocks.

Denote the three sets of the node pairs (i, j) associated with the above three sets of kernel blocks as $\mathcal{D}$, $\mathcal{A}$, and $\mathcal{A}_p$, respectively. See Figure 5.3 for an example of these three sets of blocks making up an $\mathcal{H}^2$ matrix representation. As can be easily verified, these three sets of kernel blocks exactly form a non-overlapping partitioning of $K(X, X)$. The components stored by an $\mathcal{H}^2$ matrix include:

- U_i and X_i^{id} for each leaf node i with nonempty $\mathcal{F}_i$;
- R_i and X_i^{id} for each non-leaf node i with nonempty $\mathcal{F}_i$;
- *intermediate blocks* denoted by $B_{i,j}$ for each $(i, j) \in \mathcal{A} \cup \mathcal{A}_p$. Block B_{ij} is one of blocks $K(X_i^{\text{id}}, X_j^{\text{id}})$, $K(X_i^{\text{id}}, X_j)$, or $K(X_i, X_j^{\text{id}})$ in the low-rank approximation eq. (5.3) or eq. (5.4) of $K(X_i, X_j)$;
- *inadmissible blocks* $K(X_i, X_j)$ denoted by $D_{i,j}$ for each $(i, j) \in \mathcal{D}$.

All the intermediate and inadmissible blocks can be computed using only the sets $\{X_i\}$ and $\{X_i^{\text{id}}\}$ for all i, which can be stored economically. Instead of precomputing and storing these intermediate and inadmissible blocks, they can be dynamically computed when needed, using only $\{X_i\}$ and $\{X_i^{\text{id}}\}$. This provides a trade-off between storage and computation.

More details on partially admissible blocks

In the standard $\mathcal{H}^2$ matrix representation, all the partially admissible blocks characterized above are treated as admissible blocks and are compressed into the form eq. (5.3) (instead of eq. (5.4)) where the corresponding U_i and X_i^{id} for each node i are computed by the ID approximation of $K(X_i, \tilde{Y}_i)$ with $\tilde{Y}_i$ defined as some superset of Y_i.

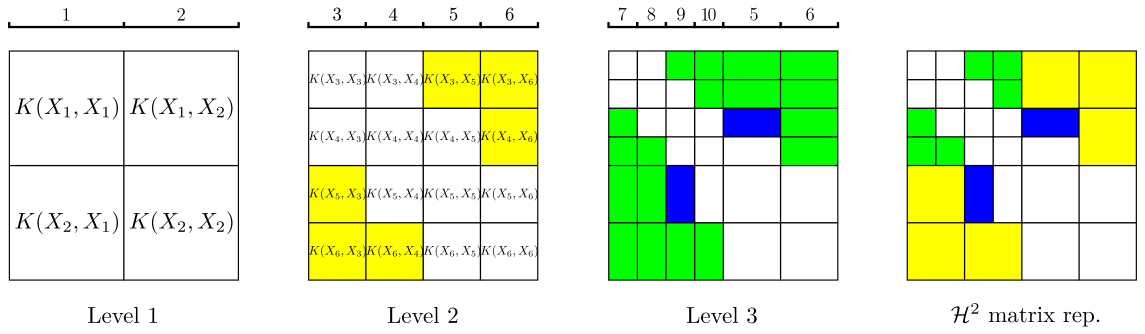


Figure 5.3: Illustration of a 3-level $\mathcal{H}^2$ matrix representation associated with the partition tree in Figure 5.1. Inadmissible blocks are white in all levels, level 2 has admissible blocks (yellow), and level 3 has admissible blocks (green) and partially admissible blocks (blue). The $\mathcal{H}^2$ matrix representation is made up of specific inadmissible and admissible blocks in levels 2 and 3 and the partially admissible blocks in level 3.

Taking the partially admissible block $K(X_5, X_9)$ in Figure 5.3 as an example, we have $Y_5 = X_3$, $\tilde{Y}_5 = X_3 \cup X_9$, and $\tilde{Y}_9 = Y_9 = X_7 \cup X_5 \cup X_6$. Note that $K(X_5, X_9)$ is within $K(X_5, \tilde{Y}_5)$ but not within $K(X_5, Y_5)$. Thus, by the ID approximation of $K(X_5, Y_5)$ and $K(X_9, Y_9)$, the block $K(X_5, X_9)$ can only be compressed into the form eq. (5.4). Meanwhile, by the ID approximation of $K(X_5, \tilde{Y}_5)$ and $K(X_9, \tilde{Y}_9)$ in the standard $\mathcal{H}^2$ matrix representation, the block $K(X_5, X_9)$ can be compressed into the form eq. (5.3).

Since $\tilde{Y}_i$ in the standard $\mathcal{H}^2$ matrix representation is defined as a superset of Y_i for each node i, $K(X_i, \tilde{Y}_i)$ has larger numerical rank than $K(X_i, Y_i)$ (can be much larger in rare cases), leading to a larger rank for the approximation of each admissible or partially admissible block $K(X_i, X_j)$. Thus, the $\mathcal{H}^2$ matrix representation using partially admissible blocks introduced in this work typically has smaller storage cost and faster matrix-vector multiplications than the standard $\mathcal{H}^2$ matrix representation. The concept of partially admissible blocks has a counterpart in FMM and is necessary for the exact equivalence between $\mathcal{H}^2$ matrix-vector multiplication and FMM [127].

5.2.2 $\mathcal{H}^2$ matrix construction

$\mathcal{H}^2$ matrix construction consists of two parts: (1) computing the ID approximation of $K(X_i, Y_i)$ for each node i with non-empty $\mathcal{F}_i$ via a nested approach and (2) computing the intermediate blocks associated with $\mathcal{A} \cup \mathcal{A}_p$ and the inadmissible blocks associated with $\mathcal{D}$. As just mentioned in the previous paragraph, the second part is optional. The nested approach to computing these ID approximations is as follows.

For a leaf node i, the ID approximation of $K(X_i, Y_i)$ is directly computed using the proxy point method (to be described in Section 5.2.3). For a non-leaf node i with children $\{i_1, i_2, \ldots, i_s\}$, the ID approximations associated with all these children nodes must be computed first. Then, since $X_i = X_{i_1} \cup \ldots \cup X_{i_s}$, $K(X_i, Y_i)$ can be split into blocks $K(X_{i_a}, Y_i)$ with $i_a \in \{i_1, i_2, \ldots, i_s\}$. By definition, the points in Y_i are in the far field of box i and thus are also in the far field of each child box i_a, i.e., $Y_i \subseteq Y_{i_a}$. As a result, the computed ID approximation $K(X_{i_a}, Y_{i_a}) \approx U_{i_a} K(X_{i_a}^{\text{id}}, Y_{i_a})$ associated with i_a gives the approximation $K(X_{i_a}, Y_i) \approx U_{i_a} K(X_{i_a}^{\text{id}}, Y_i)$. Together, $K(X_i, Y_i)$ is split and approximated as,

$$K(X_i, Y_i) = \begin{bmatrix} K(X_{i_1}, Y_i) \\ K(X_{i_2}, Y_i) \\ \vdots \\ K(X_{i_s}, Y_i) \end{bmatrix} \approx \begin{bmatrix} U_{i_1} K(X_{i_1}^{\text{id}}, Y_i) \\ U_{i_2} K(X_{i_2}^{\text{id}}, Y_i) \\ \vdots \\ U_{i_s} K(X_{i_s}^{\text{id}}, Y_i) \end{bmatrix} = \begin{bmatrix} U_{i_1} & & & \\ & U_{i_2} & & \\ & & \ddots & \\ & & & U_{i_s} \end{bmatrix} \begin{bmatrix} K(X_{i_1}^{\text{id}}, Y_i) \\ K(X_{i_2}^{\text{id}}, Y_i) \\ \vdots \\ K(X_{i_s}^{\text{id}}, Y_i) \end{bmatrix}. \tag{5.5}$$

Denoting $\hat{X}_i = X_{i_1}^{\text{id}} \cup X_{i_2}^{\text{id}} \cup \ldots \cup X_{i_s}^{\text{id}}$, an ID approximation of the last block above $K(\hat{X}_i, Y_i)$ is computed using the proxy point method as,

$$K(\hat{X}_i, Y_i) \approx R_i K(X_i^{\text{id}}, Y_i), \quad X_i^{\text{id}} \subseteq \hat{X}_i \subseteq X_i.$$

Plugging this approximation into eq. (5.5), we get the ID approximation $K(X_i, Y_i) \approx U_i K(X_i^{\text{id}}, Y_i)$ with U_i defined in the nested form eq. (5.2) using the computed R_i.

5.2.3 The proxy point method

The $\mathcal{H}^2$ matrix construction above is dominated by the ID approximation of $K(X_i, Y_i)$ for leaf nodes i and $K(\hat{X}_i, Y_i)$ for non-leaf nodes i. All these approximated kernel blocks share the same form $K(X_*, Y_*)$ where X_* is a set of points in a box $\mathcal{X}$ and Y_* is a set of points in a compact subdomain $\mathcal{Y}$ of the far field of $\mathcal{X}$, as illustrated in Figure 5.4. In general, Y_* has far more points than X_*. The proxy point method [109] can efficiently construct an ID approximation of $K(X_*, Y_*)$ with $X_* \times Y_*$ lying in a pair of compact domains $\mathcal{X} \times \mathcal{Y}$ as follows.

First select a set of so-called *proxy points* Y_p in $\mathcal{Y}$ following the selection scheme Algorithm 6 (to be described later). Then compute an ID approximation of $K(X_*, Y_p)$ algebraically using the pivoted QR decomposition as $K(X_*, Y_p) \approx U_* K(X_*^{\text{id}}, Y_p)$ with $X_*^{id} \subseteq X_*$. Using the computed U_* and X_*^{id}, the ID approximation of $K(X_*, Y_*)$ is then directly defined as $K(X_*, Y_*) \approx U_* K(X_*^{\text{id}}, Y_*)$. In most cases, Y_p has far fewer points that Y_* and thus the above proxy point method is far more efficient than the direct ID approximation of $K(X_*, Y_*)$. Numerically, when a relative error threshold ε_{id} is used for the algebraic ID approximation of $K(X_*, Y_p)$, the defined ID approximation of $K(X_*, Y_*)$ usually has relative error approximately ε_{id}.

Selection of the proxy points

The selection scheme given in Algorithm 6 was proposed in Ref. [109]. The basic idea is to first discretize $K(x, y)$ in $\mathcal{X} \times \mathcal{Y}$ into matrix $K(X_1, Y_1)$. Steps 2 and 3 in this algorithm compresses this matrix as $K(X_1, Y_1) \approx K(X_1, Y_p) K(X_p, Y_p)^{-1} K(X_p, Y_1)$ with $O(\varepsilon_p)$ error. Due to the low-rank property of $K(x, y)$, it can be proved that, if $|X_1|$ and $|Y_1|$ are sufficiently large,

$$\begin{aligned} &K(x,y) \approx K(x,Y_p)K(X_p,Y_p)^{-1}K(X_p,y) + O(\varepsilon_p), \quad (x,y) \in \mathcal{X} \times \mathcal{Y}, \\ \xrightarrow{\text{plug in } X_*, Y_*} \; &K(X_*,Y_*) \approx K(X_*,Y_p)K(X_p,Y_p)^{-1}K(X_p,Y_*) + O(\varepsilon_p). \end{aligned}$$

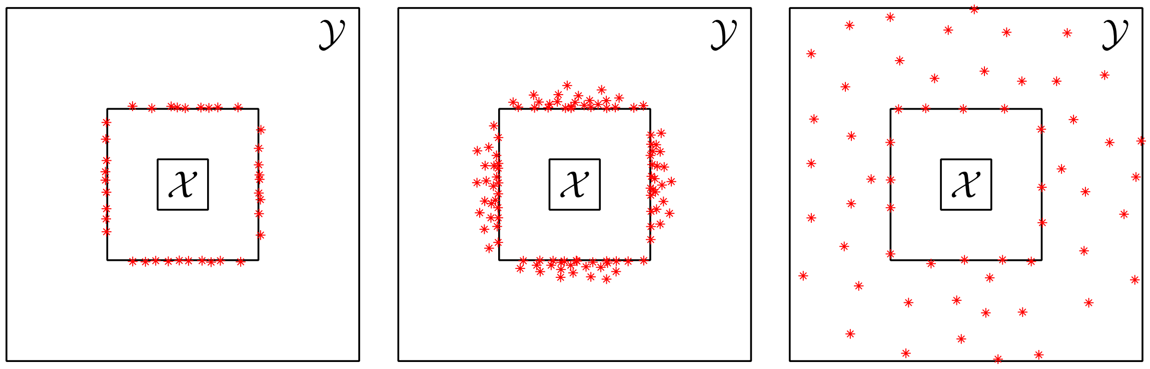
The proxy point method exactly computes an ID approximation of $K(X_*, Y_p)$ and thus can also be viewed as a recompression of the above $O(\varepsilon_p)$-accuracy approximation of $K(X_*, Y_*)$. Usually, the parameter ε_p can be set to one or two orders of magnitudes smaller than the error threshold specified for the proxy point method. The sizes of X_1 and Y_1 should be large enough to guarantee the accuracy $O(\varepsilon_p)$ of the above function approximation to $K(x, y)$, and also to guarantee well-boundedness of this specific vector function $K(X_p, Y_p)^{-1}K(X_p, y)$ in $\mathcal{Y}$ which is critical for the accuracy of the proxy point method. More explanations can be found in [109].

This selection scheme is computationally expensive and only depends on $K(x, y)$ and $\mathcal{X} \times \mathcal{Y}$. With more sample points X_1 and Y_1, the set of proxy points Y_p selected by Algorithm 6 is more effective in terms of controlling the accuracy of the proxy point method based on Y_p, but Algorithm 6 becomes more expensive. In H2Pack, the numbers of sample points in X_1 and Y_1 in Algorithm 6 are heuristically chosen. We used $|X_1| = 1000$ and $|Y_1| = 15000$ for the various kernel functions and pairs of domains that were tested numerically (see Section 5.4). An adaptive choice of the number of sample points can be developed and applied if necessary. The ID approximation of $K(X_1, Y_1)$ at Step 2 of Algorithm 6 is computed using a randomized method [128] instead of the pivoted QR decomposition, for better efficiency. Figure 5.4 illustrates several examples of the selected proxy points for different kernel functions.

Applying Algorithm 6 to select proxy points for each ID approximation in $\mathcal{H}^2$ matrix construction is expensive and impractical. Instead, we can reuse a set of selected proxy points Y_p for all the ID approximations associated with nodes in one level of the construc-

Algorithm 6 Proxy point selection scheme

```
Input: $K(x,y)$, $\mathcal{X}$, $\mathcal{Y}$, $\varepsilon_p$.
Output: proxy points $Y_p$.
1: Sample domains $\mathcal{X}$ and $\mathcal{Y}$ to obtain two sets of uniformly distributed points $X_1$ and $Y_1$
   with high point density, respectively.
2: Compute an ID approximation $K(X_1,Y_1) \approx U_1 K(X_p,Y_1)$ with error threshold
   $\varepsilon_p\sqrt{|Y_1|}$.
3: Compute a pivoted QR decomposition $K(X_p,Y_1)P = Q(R_1,R_2)$ where $P$ is a permutation matrix,
   $Q$ is an orthogonal matrix, and $R_1$ is an $|X_p| \times |X_p|$ upper-triangular matrix.
4: Let $Y_p$ be the subset of points in $Y_1$ that corresponds to the $|X_p|$ columns of $R_1$ after
   permutation.
```



(a) $K(x,y) = \log(|x-y|)$ (b) $K(x,y) = \exp(-|x-y|^2)$ (c) $K(x,y) = \exp(-0.1|x-y|^2)$

Figure 5.4: Examples of the proxy points selected by Algorithm 6 for various kernel functions with $\mathcal{X} = [-1,1]^2$, $\mathcal{Y} = [-7,7]^2 \setminus [-3,3]^2$, and $\varepsilon_p = 10^{-10}$. The three sets have 37, 103, and 58 proxy points, respectively.

tion. Specifically, note that all the boxes in the same level are of the same size and $K(x, y)$ is assumed to be translationally-invariant. Thus, at each level l, we select $\mathcal{X}$ as a box in level l and $\mathcal{Y}$ as a large compact subdomain of the far field of $\mathcal{X}$, and apply Algorithm 6 with $\mathcal{X} \times \mathcal{Y}$ to select a set of proxy points Y_p^l. For each node i in level l, let z_i be a translation vector such that $X_i + z_i$ lies in $\mathcal{X}$ and $Y_i + z_i$ lies in $\mathcal{Y}$ ($\mathcal{Y}$ should be selected large enough to contain $Y_i + z_i$ for each node i). Since $K(X_i, Y_i) = K(X_i + z_i, Y_i + z_i)$, we can apply the proxy point method with the shifted proxy points $Y_p^l - z_i$ to compute the ID approximation of $K(X_i, Y_i)$ (or $K(\hat{X}_i, Y_i)$).

As a result, at each level, we only need to construct a set of proxy points Y_p^l for just one pair of domains $\mathcal{X} \times \mathcal{Y}$. The corresponding proxy points for all the nodes in one level can be obtained by proper translation of Y_p^l. Also, another option is to precompute and store multiple sets of proxy points for box domains $\mathcal{X}$ of different sizes (with sufficiently large domains $\mathcal{Y}$) given a kernel function. In $\mathcal{H}^2$ matrix construction, we simply need to load the corresponding proxy point set based on the box domain size in each level. Combining the proxy point method with the $\mathcal{H}^2$ matrix construction described in the last subsection, the overall $\mathcal{H}^2$ matrix construction for a kernel matrix $K(X, X)$ is summarized in Algorithm 7.

Algorithm 7 $\mathcal{H}^2$ matrix construction for $K(X, X)$

```
1:  • construct a hierarchical partitioning of $X$ which gives a $L$-level partition tree $\mathcal{T}$.
2:  for $l = L, L-1, \dots, 1$ do
3:      • construct a set of proxy points $Y_p^l$ for just one box in level $l$.
4:      parfor all nodes $i$ in level $l$ do (dynamic scheduling)
5:          if $i$ is a leaf node then
6:              • compute $U_i$ and $X_i^{\text{id}}$ from an ID approximation of $K(X_i, Y_i)$ using the proxy point method with a proper translation of $Y_p^l$.
7:          else if $i$ is a non-leaf node with children $\{i_1, i_2, \dots, i_s\}$ then
8:              • construct $\hat{X}_i^{\text{id}} = X_{i_1}^{\text{id}} \cup \dots \cup X_{i_s}^{\text{id}}$.
9:              • compute $R_i$ and $X_i^{\text{id}}$ from an ID approximation of $K(\hat{X}_i, Y_i)$ using the proxy point method with a proper translation of $Y_p^l$.
10:         end if
11:     end parfor
12: end for
13: • (optional, can be dynamically computed) compute the inadmissible blocks $D_{i,j}$ for all $(i, j) \in \mathcal{D}$ and compute the intermediate blocks $B_{i,j}$ for all $(i, j) \in \mathcal{A} \cup \mathcal{A}_p$.
```

5.2.4 $\mathcal{H}^2$ matrix-vector multiplication

Consider computing $b = K(X,X)q$. For each node $i \in \mathcal{T}$, let q_i and b_i denote the subvectors of q and b, respectively, corresponding to the point subset X_i in X. The $\mathcal{H}^2$ matrix-vector multiplication algorithm [101], summarized in Algorithm 8, traverses all three sets of kernel blocks $K(X_i, X_j)$ corresponding to $\mathcal{D}$, $\mathcal{A}$, and $\mathcal{A}_p$ in the $\mathcal{H}^2$ matrix representation and accumulates the products $K(X_i, X_j)q_j$.

First, initialize the result vector b to zero. For each inadmissible block $K(X_i, X_j)$ with $(i,j) \in \mathcal{D}$, the dense matrix computation is straightforward: $b_i = b_i + K(X_i, X_j)q_j$. For each admissible block $K(X_i, X_j)$ with $(i,j) \in \mathcal{A}$, the computation

$$b_i = b_i + K(X_i, X_j)q_j \approx b_i + U_i B_{i,j} U_j^T q_j,$$

can be computed in three steps $U_j^T q_j$, $B_{i,j}(U_j^T q_j)$, and $b_i = b_i + U_i \left(B_{i,j}\left(U_j^T q_j\right)\right)$ giving three phases in $\mathcal{H}^2$ matrix-vector multiplication: forward transformation, intermediate multiplication, and backward transformation.

Forward transformation

This phase computes $y_j = U_j^T q_j$ for all the nodes $j \in \mathcal{T}$. Note that y_j can be used for all the admissible blocks with columns defined by X_j. For each leaf node j, y_j is directly computed. For each non-leaf node j with children $\{j_1, j_2, \ldots, j_s\}$, y_j is recursively computed using $y_{j_1}, y_{j_2}, \ldots, y_{j_s}$ associated with the children as

$$y_j = U_j^T q_j = R_j^T \begin{bmatrix} U_{j_1}^T & & \\ & \ddots & \\ & & U_{j_s}^T \end{bmatrix} \begin{bmatrix} q_{j_1} \\ \vdots \\ q_{j_s} \end{bmatrix} = R_j^T \begin{bmatrix} U_{j_1}^T q_{j_1} \\ \vdots \\ U_{j_s}^T q_{j_s} \end{bmatrix} = R_j^T \begin{bmatrix} y_{j_1} \\ \vdots \\ y_{j_s} \end{bmatrix}.$$

Intermediate multiplication

This phase computes $z_{i,j} = B_{i,j}y_j$ for each admissible block $K(X_i, X_j)$ with $(i,j) \in \mathcal{A}$. Note that all the $z_{i,j}$ sharing the node i are to be multiplied by U_i and added to b_i as

$$b_i = b_i + \textstyle\sum_{(i,j)\in\mathcal{A}} U_i z_{i,j} = b_i + U_i \left(\textstyle\sum_{(i,j)\in\mathcal{A}} z_{i,j}\right), \quad \text{for each node } i \in \mathcal{T}.$$

Only multiplying U_i once, it is more efficient to first sum over all these $z_{i,j}$, then apply U_i, and lastly add to b_i. Thus, for each node $i \in \mathcal{T}$, this phase further computes $z_i = \sum_{(i,j)\in\mathcal{A}} z_{i,j}$.

Backward transformation

This phase computes $b_i = b_i + U_i z_i$ for each node $i \in \mathcal{T}$. For a non-leaf node i with children $\{i_1, i_2, \ldots, i_s\}$, b_i is recursively accumulated as

$$b_i = b_i + U_i z_i = b_i + \begin{bmatrix} U_{i_1} & & \\ & \ddots & \\ & & U_{i_s} \end{bmatrix} R_i z_i = b_i + \begin{bmatrix} U_{i_1}[R_i z_i]_{i_1} \\ \vdots \\ U_{i_s}[R_i z_i]_{i_s} \end{bmatrix}$$

where $[R_i z_i]_{i_a}$ denotes the subvector of $R_i z_i$ associated with U_{i_a}. Thus, $b_i = b_i + U_i z_i$ is reduced to $b_{i_a} = b_{i_a} + U_{i_a}[R_i z_i]_{i_a}$ with all the children i_a. Meanwhile, $b_{i_a} = b_{i_a} + U_{i_a} z_{i_a}$ needs to be computed as well. Only multiplying U_{i_a} once, it is more efficient to first overwrite z_{i_a} by $z_{i_a} = z_{i_a} + [R_i z_i]_{i_a}$ and then multiply U_{i_a} by z_{i_a}. Recursively, this phase traverses the tree from the root to the leaves to overwrite each z_i by $z_i = z_i + [R_p z_p]_i$ with p the parent of i. As a result, for each leaf node i, z_i accumulates the intermediate multiplication results from all its ancestors. Adding $U_i z_i$ to b_i for all the leaf nodes in $\mathcal{T}$ finishes this phase. See the lines 21-28 in Algorithm 8 for the exact calculation.

For each partially admissible block $K(X_i, X_j)$ with $(i,j) \in \mathcal{A}_p$, its multiplication by

p_j,

$$b_i = b_i + U_i B_{i,j} q_j \quad \text{or} \quad b_i = b_i + B_{i,j} U_j^T q_j$$

can be similarly computed following the above process for the admissible blocks. In fact, these multiplications can be merged into the above three phases for admissible blocks.

$\mathcal{H}^2$ matrix-matrix multiplication

Consider computing $C = K(X, X)Q$. It is straightforward to extend the above $\mathcal{H}^2$ matrix-vector multiplication to the multiplication by multiple vectors simultaneously. We only need to replace vectors q_i, b_i, y_i, and z_i in Algorithm 8 by matrices Q_i, C_i, Y_i, and Z_i, respectively, where Q_i and C_i are the row subsets of Q and C associated with X_i.

5.3 Parallel Implementation

5.3.1 Parallelization and load-balancing

In Section 5.2, we presented $\mathcal{H}^2$ matrix construction ($\mathcal{H}^2$-construction) and $\mathcal{H}^2$ matrix-vector multiplication ($\mathcal{H}^2$-matvec). For the parallel implementation of these two operations, we consider calculation dependencies associated with each node in the partition tree. In $\mathcal{H}^2$-construction, the first step is to compute specific ID approximations associated with each node i with nonempty $\mathcal{F}_i$. In this step, the ID approximation at a non-leaf node cannot be computed until the ID approximations at all its children nodes are computed, corresponding to a post-order traversal of the partition tree. The optional step of computing inadmissible and intermediate blocks has no restriction on calculation orders for each block. In $\mathcal{H}^2$-matvec, the forward transformation phase has the same calculation order as the ID approximations in $\mathcal{H}^2$-construction, i.e., the calculation of y_i for a non-leaf node i requires the calculation of $\{y_{i_k}\}$ with the children $\{i_k\}$ of i. The backward transformation phase has calculation order reverse to that of the forward transformation phase, corresponding to a pre-order traversal of the partition tree. Meanwhile, there is no restriction on the

Algorithm 8 $\mathcal{H}^2$ matrix-vector multiplication

```
 1: • Initialize result vector $b$ to zero.
 2: • Initialize temporary vectors $z_i, \forall i \in \mathcal{T}$ to zero.
    ▷ Step 1: Forward transformation
 3: for $l = L, L-1, \ldots, 1$ do
 4:     parfor all nodes $i$ in level $l$ do (greedy static partitioning)
 5:         if $i$ is a leaf node then
 6:             $y_i = U_i^T q_i$.
 7:         else
 8:             $y_i = R_i^T (y_{i_1}^T, y_{i_2}^T, \ldots, y_{i_s}^T)^T$ with children $\{i_1, i_2, \ldots, i_s\}$ of node $i$.
 9:         end if
10:     end parfor
11: end for
    ▷ Step 2: Intermediate multiplication
12: parfor all $(i, j) \in \mathcal{A}$ do (hybrid load balancing)
13:     $z_i = z_i + B_{i,j} y_j$.
14: end parfor
15: parfor all $(i, j) \in \mathcal{A}_p$ do (hybrid load balancing)
16:     if $K(X_i, X_j) \approx U_i B_{i,j}$ then
17:         $z_i = z_i + B_{i,j} q_j$.
18:     else (note: $K(X_i, X_j) \approx B_{i,j} U_j^T$)
19:         $b_i = b_i + B_{i,j} y_j$.
20:     end if
21: end parfor
    ▷ Step 3: Backward transformation
22: for $l = 1, 2, \ldots, L$ do
23:     parfor all non-leaf node $i$ in level $l$ do (greedy static partitioning)
24:         $z_{i_a} = z_{i_a} + [R_i z_i]_{i_a}$ with all children $i_a \in \{i_1, i_2, \ldots, i_s\}$ of node $i$.
25:     end parfor
26: end for
27: parfor all leaf node $i$ in $\mathcal{T}$ do (hybrid load balancing)
28:     $b_i = b_i + U_i z_i$.
29: end parfor
    ▷ Step 4: Dense multiplication
30: parfor all $(i, j) \in \mathcal{D}$ do (hybrid load balancing)
31:     $b_i = b_i + D_{i,j} q_j$.
32: end parfor
```

calculation order in the intermediate and dense multiplication phases, since the matrix-vector multiplications by different $B_{i,j}$ and $D_{i,j}$ are completely independent.

Based on the above observations, the calculations in $\mathcal{H}^2$-construction and $\mathcal{H}^2$-matvec can be categorized into two types. The first type is *level-by-level calculation*, where the calculation at node i rely on the calculations at nodes on the level above or below. The second type is *independent calculation*, where the calculations associated with different $B_{i,j}$ or $D_{i,j}$ are independent. We apply different strategies to parallelize these two types of calculations within the OpenMP framework.

Level-by-level calculations Let the calculation at node i in a level-by-level computation phase be referred to as *task* i. In the following phases, task i needs the results of multiple tasks in a lower level or the result of a task in an upper level:

- the ID approximations in $\mathcal{H}^2$-construction (lines 2-12 in Algorithm 7),
- the forward transformation in $\mathcal{H}^2$-matvec (lines 3-11 in Algorithm 8),
- the backward transformation in $\mathcal{H}^2$-matvec (lines 22-29 in Algorithm 8).

In these computational tasks, the accessed matrices $K(X_i, Y_p)$, $K(\hat{X}_i, Y_p)$, U_i, and R_i usually have size smaller than 1000×1000. For such small matrices, the column-pivoted QR and matrix-vector multiplications usually have poor parallel performance when using a large number of processors. Instead of parallelizing these elementary computational kernels, we choose to parallelize across the tasks in each level of $\mathcal{T}$. Specifically, we parallelize the for-loops in line 4 of Algorithm 7 and in lines 4, 23, and 27 of Algorithm 8 with OpenMP.

We use different load-balancing strategies for the three computation phases listed above. In $\mathcal{H}^2$-construction, since the size of $K(\hat{X}_i, Y_p)$ at each non-leaf node is not known in advance, we use OpenMP dynamic scheduling to balance the workload in the parallel loop of ID approximations in each level. In $\mathcal{H}^2$-matvec, the performance bottleneck of the forward and backward transformations is the transfer of U_i and R_i from memory to processors. Since the sizes of U_i and R_i are known at this stage, we use a greedy static partitioning scheme to approximately balance the total sizes of matrices that each processor needs to load from memory.

Independent calculations Let calculations associated with a block $B_{i,j}$ or $D_{i,j}$ in an independent computation phase be referred to as *task* (i, j) with (i, j) in the node pair sets $\mathcal{A} \cup \mathcal{A}_p$ or $\mathcal{D}$. In the following phases, all tasks are independent and can be performed in parallel without restriction:

- the optional construction of $B_{i,j}$ and $D_{i,j}$ in $\mathcal{H}^2$-construction (lines 13 in Algorithm 7),
- the intermediate multiplication phase in $\mathcal{H}^2$-matvec (lines 12-21 in Algorithm 8),
- the dense multiplication phase in $\mathcal{H}^2$-matvec (lines 30-32 in Algorithm 8).

Note that, for each $B_{i,j}$ or $D_{i,j}$, both the computation cost of forming it and the communication cost of transferring it from memory to a processor are proportional to its block size which is known after the ID approximations in $\mathcal{H}^2$-construction.

We first consider exploiting the symmetry property of these blocks $B_{i,j}$ and $D_{i,j}$. Since $K(X, X)$ is symmetric, $B_{i,j} = B_{j,i}^T$ if (i, j) is in $\mathcal{A} \cup \mathcal{A}_p$ (corresponding to an admissible or partially admissible block) and $D_{i,j} = D_{j,i}^T$ if (i, j) is in $\mathcal{D}$ (corresponding to an inadmissible block). Thus, for each pair of (i, j) and (j, i) in $\mathcal{A} \cup \mathcal{A}_p$, only $B_{i,j}$ is computed and the following two matrix-vector multiplications in the intermediate multiplication phase will be performed on one processor simultaneously:

$$z_i = z_i + B_{i,j} y_j, \quad z_j = z_j + B_{i,j}^T y_i.$$

The same approach applies to each pair of (i, j) and (j, i) in $\mathcal{D}$ with blocks $D_{i,j}$.

We use a hybrid approach for parallelizing and load-balancing the independent calculations. In this hybrid approach, a static partitioning is used for approximately balancing the workload on each processor and a dynamic task scheduler is used for polishing the load balance. We use the construction of blocks $B_{i,j}$ to illustrate this approach. For P processors, we partition all tasks into kP disjoint task units ($1 \leq k \leq 20$ is a prescribed constant) with a greedy algorithm such that the total size of matrix blocks in each task unit is approximately the same. Each processor has $k-1$ initial task units, which leads to approximately the same computation time for initial task units on each processor. The last P task units form a task pool for dynamic task scheduling. After finishing all its $k-1$ initial task units, a processor starts to steal task units one by one from the task pool until all task units have been consumed. If $k = 1$, the hybrid approach is equivalent to a static task partitioning scheme. The construction of blocks $D_{i,j}$, the intermediate multiplication phase, and the dense multiplication phase are all parallelized in the same way.

Combining the utilization of the symmetry property and the hybrid parallelization approach causes a new problem. In the intermediate and dense multiplication phases, two or more processors may update the same z_i or b_i simultaneously, leading to incorrect results. Three solutions to this problem are available. The first is to discard utilizing the symmetry property and then to partition the corresponding tasks in a way that each z_i and b_i can be updated by only one processor. This approach is unfavorable since it leads to more data transfer between memory and processors and higher computation cost. The second solution is to use atomic operations for updating z_i and b_i. However, atomic operations are much slower than their non-atomic counterparts. In H2Pack, we use the third solution that each processor uses local copies of z_i and b_i to accumulate local matrix-vector multiplication results. All local copies of z_i/b_i are summed after the intermediate/dense multiplication phase to form the actual z_i/b_i. The additional cost for summing local copies of z_i and b_i is negligible compared to the main phases in $\mathcal{H}^2$-matvec.

For multiplying multiple vectors simultaneously, H2Pack provides a separate $\mathcal{H}^2$ matrix-

matrix multiplication ($\mathcal{H}^2$-matmul) function. In $\mathcal{H}^2$-matmul, vectors q_i, b_i, y_i, and z_i in $\mathcal{H}^2$-matvec are replaced by blocks Q_i, C_i, Y_i, and Z_i, and the matrix-vector multiplications in $\mathcal{H}^2$-matvec are replaced by matrix-matrix multiplications. The multiplicand matrix Q could be stored in either row-major or column-major format, with the output matrix C stored in the same format. $\mathcal{H}^2$-matmul adopts almost the same parallelization and load-balancing scheme as $\mathcal{H}^2$-matvec. One exception is that $\mathcal{H}^2$-matmul does not utilize the symmetry property of $B_{i,j}$ and $D_{i,j}$ blocks. Instead, independent calculation tasks with $B_{i,j}$ and $D_{i,j}$ are partitioned in a way that each Z_i and C_i is only updated by one processor. Processor-local Z_i and C_i copies are not used since they could require a large amount of memory.

5.3.2 Performance optimizations

We optimize H2Pack for state-of-the-art multi-core and many-core architectures. We first introduce the H2Pack kernel function interface in Section 5.3.2. Next, we discuss two running modes of H2Pack in Section 5.3.2. Then, we illustrate the use of intrinsic functions for better vectorization in Section 5.3.2.

Kernel function interface

The performance of H2Pack relies on the performance of evaluating the kernel function. $\mathcal{H}^2$-construction and $\mathcal{H}^2$-matvec using just-in-time mode (to be discussed in Section 5.3.2) both need to evaluate a large number of kernel matrix blocks. H2Pack provides an optimized implementation of the 3D Laplace kernel $K(x, y) = 1/|x-y|$ which can be modified easily for other kernel functions. In the following, we use the 3D Laplace kernel as an example to show the H2Pack kernel function interface.

H2Pack provides a C language interface. A driver program must provide a pointer to a *kernel matrix evaluation* (KME) function to use H2Pack. Listing 5.1 shows a KME function for the 3D Laplace kernel. Lines 2-4 in Listing 5.1 are parameters of a KME function:

two sets of point coordinates `coord0` and `coord1` and the kernel matrix `kmat` to be returned. Input `coord0` is a $3\times$`n0` row-major matrix with leading dimension `ld0` and contains the coordinates of `n0` points. Each column of `coord0` stores a point coordinate. The same storage scheme applies to `coord1`. The function returns an `n0`$\times$`n1` kernel matrix stored in a row-major matrix `kmat` with leading dimension `ldm`. Note that a KME function should be single-threaded and should only use variables or memory that can be updated by the current thread.

The above design of a KME function is in order to facilitate the vectorization of multiple kernel function evaluations. It would be easier for users to program a function evaluating the kernel function for just a single pair of points. However, such a single-value function must be compiled together with H2Pack so that the compiler can auto-vectorize multiple kernel function evaluations. Using KME functions is more flexible: H2Pack only needs to be compiled once for different KME functions, and a KME function can be auto-vectorized by the compiler (line 14 in Listing Listing 5.1) or manually vectorized (to be discussed in Section 5.3.2).

Ahead-of-time and just-in-time running modes

H2Pack provides two running modes of $\mathcal{H}^2$-construction and $\mathcal{H}^2$-matvec: (1) ahead-of-time (AOT) mode computes and stores all $B_{i,j}$ and $D_{i,j}$ in $\mathcal{H}^2$-construction, and (2) just-in-time (JIT) mode computes each $B_{i,j}$ and $D_{i,j}$ when needed in $\mathcal{H}^2$-matvec without storing them. These two modes give flexibility in how to obtain performance on different computing platforms for different kernel functions. We note that any implementations of $\mathcal{H}^2$ matrix representations based on ID approximations, e.g., the SMASH library and the STRUMPACK library, can also use both AOT and JIT modes.

The AOT mode is designed to avoid redundant calculation when the cost of kernel function evaluation is high. To form $B_{i,j}$ and $D_{i,j}$, a large number of kernel function evaluations are needed. Kernel functions with transcendental arithmetic (e.g., the Gaussian

Listing 5.1: Sample KME function for the 3D Laplace kernel

```
void Laplace_3D_krnl_eval(
    const double *coord0, const int ld0, const int n0,
    const double *coord1, const int ld1, const int n1,
    double * restrict kmat, const int ldm
)
{
    const double *x0 = coord0 + ld0 * 0, *x1 = coord1 + ld1 * 0;
    const double *y0 = coord0 + ld0 * 1, *y1 = coord1 + ld1 * 1;
    const double *z0 = coord0 + ld0 * 2, *z1 = coord1 + ld1 * 2;
    for (int i = 0; i < n0; i++)
    {
        double x0i = x0[i], y0i = y0[i], z0i = z0[i];
        double *kmat_i = kmat + i * ldm;
        #pragma omp simd  // Requires the compiler to vectorize this loop
        for (int j = 0; j < n1; j++)
        {
            double dx = x1[j] - x0i;
            double dy = y1[j] - y0i;
            double dz = z1[j] - z0i;
            double r2 = dx * dx + dy * dy + dz * dz;
            double rinv = (r2 == 0.0) ? 0.0 : 1.0 / sqrt(r2);
            kmat_i[j] = rinv;
        }
    }
}
```

kernel $K(x, y) = \exp(|x - y|^2)$ and the logarithm kernel $K(x, y) = \log(|x - y|)$) have much higher evaluation cost than those without transcendental arithmetic. In such cases, using AOT mode could be more efficient than using JIT mode for $\mathcal{H}^2$-matvec. As a trade-off, AOT mode has much larger storage cost than JIT mode due to the storage of $B_{i,j}$ and $D_{i,j}$.

The performance bottleneck of $\mathcal{H}^2$-matvec in AOT mode is the transfer of $B_{i,j}$ and $D_{i,j}$ from memory to processors. Two optimizations are proposed for $\mathcal{H}^2$-matvec in AOT mode, targeting the intermediate and dense multiplication phases. First, we optimize for non-uniform memory access (NUMA) architectures. The memory for $B_{i,j}$ and $D_{i,j}$ blocks used by a processor is bound to its NUMA node to reduce memory access latency and to fully utilize memory bandwidth of all NUMA nodes in a computer. Second, we implement a bi-matrix-vector multiplication (BMV) function that computes Ax_0 and $A^T x_1$ with a matrix A and two input vectors x_0, x_1 simultaneously to avoid loading the same $B_{i,j}$ or $D_{i,j}$ block twice from memory or processor cache. This function is not available in any existing optimized linear algebra library.

The JIT mode is designed to reduce the storage cost of an $\mathcal{H}^2$ matrix representation. The total size of all $B_{i,j}$ and $D_{i,j}$ blocks is usually 10 to 100 times larger than that of other $\mathcal{H}^2$ matrix components. For a given memory size, by not storing $B_{i,j}$ and $D_{i,j}$ blocks, H2Pack in JIT mode can handle problems with far more points. We use the cache-blocking technique to optimize the intermediate and dense multiplication phases in JIT mode. Specifically, we partition $B_{i,j}$ or $D_{i,j}$ into multiple subblocks such that each subblock and the coordinates associated with this subblock can fit in processor L2 data cache. A small processor-private buffer is used for each processor to temporarily store a dynamically generated subblock. Once a subblock is generated, we use this subblock and the BMV function to compute two matrix-vector multiplications immediately. Since only the point coordinates need to be transferred from memory to processors, the intermediate and dense multiplication phases also have much smaller memory bandwidth pressure in JIT

Listing 5.2: Sample BKM function for the 3D Laplace kernel

```c
 1 void Laplace_3D_bi_krnl_matvec(
 2     const double *coord0, const int ld0, const int n0,
 3     const double *coord1, const int ld1, const int n1,
 4     const double *xin0, const double *xin1,
 5     double * restrict xout0, double * restrict xout1
 6 )
 7 {
 8     const double *x0 = coord0 + ld0 * 0, *x1 = coord1 + ld1 * 0;
 9     const double *y0 = coord0 + ld0 * 1, *y1 = coord1 + ld1 * 1;
10     const double *z0 = coord0 + ld0 * 2, *z1 = coord1 + ld1 * 2;
11     for (int i = 0; i < n0; i++)
12     {
13         double x0i = x0[i], y0i = y0[i], z0i = z0[i], xin1_i = xin1[i];
14         double sum_i = 0.0;
15         #pragma omp simd  // Requires the compiler to vectorize this loop
16         for (int j = 0; j < n1; j++)
17         {
18             double dx = x1[j] - x0i;
19             double dy = y1[j] - y0i;
20             double dz = z1[j] - z0i;
21             double r2 = dx * dx + dy * dy + dz * dz;
22             double rinv = (r2 == 0.0) ? 0.0 : 1.0 / sqrt(r2);
23             sum_i    += rinv * xin0[j];
24             xout1[j] += rinv * xin1;
25         }
26         xout0[i] += sum_i;
27     }
28 }
```

mode than in AOT mode.

We further design a *matrix-free* approach for $\mathcal{H}^2$-matvec in JIT mode using a *bi-kernel matvec* (BKM) function (note that $\mathcal{H}^2$-matmul does not use the BKM function). For two point sets X_0 and X_1, a BKM function calculates two matrix-vector multiplications $K(X_0, X_1)x_0$ and $K(X_1, X_0)x_1$ at the same time without explicitly storing ("matrix-free") any subblock of $K(X_0, X_1)$ or $K(X_1, X_0)$. Compared to using a KME function, using a BKM function eliminates the transferring of the dynamically generated subblocks of $B_{i,j}$ and $D_{i,j}$ between a processor and its L2 data cache. Listing 5.2 shows a sample BKM function for the 3D Laplace kernel. Lines 2-4 in Listing 5.2 are parameters of a BKM function: two sets of point coordinates `coord0` and `coord1` stored in the same way as in the KME function, two input vectors `xin0`, `xin1`, and two output vectors `xout0`, `xout1`. Input `xin0` stores x_0 and `xout0` stores the result of $K(X_0, X_1)x_0$. Input `xin1` stores x_1 and `xout0` stores the result of $K(X_1, X_0)x_1$. The only difference between a KME function and a BKM function is that once a kernel function value is calculated (line 21 in both Listing Listing 5.1 and Listing 5.2), a KME function stores this value to a matrix but a BKM function consumes this value and discards it immediately. If the kernel function evaluation is cheap (for example, for the 3D Laplace kernel) and we have fast processors but moderate memory bandwidth, $\mathcal{H}^2$-matvec in JIT mode using a BKM function could be faster than $\mathcal{H}^2$-matvec in AOT mode.

Vector intrinsics

Effectively vectorizing the KME and BKM functions is critical to high performance of H2Pack. In general, KME and BKM functions for scalar kernels (kernels that return a single value for a pair of points) using the same framework as Listing 5.1 and Listing 5.2 can be auto-vectorized by compilers. As an alternative, H2Pack provides a set of *vector wrapper functions* independent of the processor instruction set for manually vectorizing calculations with intrinsic functions. The optimized KME and BKM functions provided in H2Pack for the 3D Laplace kernel use these vector wrapper functions. Currently the vector wrapper functions supports AVX, AVX2, and AVX-512 instruction sets on x86 architecture. (Other architectures can also be supported in the future.)

Listing 5.3 shows a sample KME function for the 3D Laplace kernel using vector wrapper functions. This function has two major parts in its inner loop: a manually vectorized loop using vector wrapper functions (lines 15-28) and a remainder loop (lines 30-39) using scalar operations. All vector wrapper functions are named as `vec_{opname}_{d/s}`, where `opname` is the operation name and the suffix indicates the floating point data type (*double (d)* or *float (s)*). Constant value `SIMD_LEN_D` indicates the number of double words in each `vec_d` vector data type. This constant is determined according to the processor instruction set and H2Pack compilation options. Vector wrapper functions used in Listing

Listing 5.3: Sample KME function for the 3D Laplace kernel using vector wrapper functions

```
void Laplace_3D_krnl_eval_vec(
    const double *coord0, const int ld0, const int n0,
    const double *coord1, const int ld1, const int n1,
    double * restrict kmat, const int ldm
)
{
    const double *x0 = coord0 + ld0 * 0, *x1 = coord1 + ld1 * 0;
    const double *y0 = coord0 + ld0 * 1, *y1 = coord1 + ld1 * 1;
    const double *z0 = coord0 + ld0 * 2, *z1 = coord1 + ld1 * 2;
    int n1_vec = (n1_vec / SIMD_LEN_D) * SIMD_LEN_D;
    for (int i = 0; i < n0; i++)
    {
        double *kmat_i = kmat + i * ldm;
        // Vectorized loop
        vec_d x0i_v = vec_set1_d(x0[i]);
        vec_d y0i_v = vec_set1_d(y0[i]);
        vec_d z0i_v = vec_set1_d(z0[i]);
        for (int j = 0; j < n1_vec; j += SIMD_LEN_D)
        {
            vec_d dx_v = vec_sub_d(vec_loadu_d(x1 + j), x0i_v);
            vec_d dy_v = vec_sub_d(vec_loadu_d(y1 + j), y0i_v);
            vec_d dz_v = vec_sub_d(vec_loadu_d(z1 + j), z0i_v);
            vec_d r2_v = vec_mul_d(dx_v, dx_v);
            r2_v = vec_fmadd_d(dy_v, dy_v, r2_v);
            r2_v = vec_fmadd_d(dz_v, dz_v, r2_v);
            vec_d rinv_v = vec_frsqrt_d(r2_v);
            vec_storeu_d(kmat_i + j, rinv_v);
        }
        // Remainder loop
        double x0i = x0[i], y0i = y0[i], z0i = z0[i];
        for (int j = n1_vec; j < n1; j++)
        {
            double dx = x1[j] - x0i;
            double dy = y1[j] - y0i;
            double dz = z1[j] - z0i;
            double r2 = dx * dx + dy * dy + dz * dz;
            double rinv = (r2 == 0.0) ? 0.0 : 1.0 / sqrt(r2);
            kmat_i[j] = rinv;
        }
    }
}
```

Listing 5.3 are the most useful vector wrapper functions for programming KME and BKM functions. A detailed list of all vector wrapper functions and their usage can be found in the H2Pack user manual. For BKM functions, H2Pack automatically pads artificial points in `coord0`, `coord1` and pads extra zeros in `xin0`, `xin1` to guarantee that `n0` and `n1` are multiples of $SIMD\_LEN\_D$. The padding aims to simplify the programming of BKM functions since the remainder loop can be eliminated.

Calculating the reciprocal square root (RSQRT) is an expensive step in evaluating $1/|x-y|$, which appears in many kernel functions. We thus implement a fast RSQRT function with x86 intrinsic functions based on the approach proposed in Ref. [129]. In this fast RSQRT function, a dedicated intrinsic function is first used to calculate an approximate RSQRT value with relative error less than 4×10^{-4}. Then, two Newton-Raphson iterations are performed using the approximate RSQRT value as an initial guess to obtain a more accurate RSQRT result with $O(10^{-14})$ relative error. The same or similar approaches to computing RSQRT have also been used in some existing FMM implementations [118, 130].

5.4 Numerical Experiments

We consider two types of point distributions: random distributions on the unit sphere in 3D (*sphere* point sets) and random distributions in the unit ball in 3D (*ball* point sets). For experiments in Sections 5.4.1 and 5.4.5, we use an Intel Skylake node on the Stampede2 supercomputer at Texas Advanced Computing Center. This node has two sockets and 192 GB DDR4 memory. Each socket has an Intel Xeon Platinum 8160 processor with 24 cores and 2 hyperthreads per core. For experiments in Section 5.4.2, we use an Intel Skylake node described above and an Intel Knights Landing node. The latter has an Intel Xeon Phi 7210 many-core processor with 64 cores and 4 hyperthreads per core, 16 GB MCDRAM high-bandwidth memory, and 96 GB DDR4 memory. H2Pack is compiled using Intel C compiler 2018.0.2 with optimization flags "-xHost -O3" on both nodes. Intel MKL 2018.0.2 is used in H2Pack to perform general matrix-vector multiplications (xGEMV) and general matrix-matrix multiplications (xGEMM). Double precision floating point is used for storage and calculations in H2Pack.

5.4.1 Accuracy tests

We first measure the accuracy of $\mathcal{H}^2$ matrix representations constructed by H2Pack under different settings. We consider three kernel functions:

- 3D Laplace kernel: $K(x,y) = \frac{1}{|x-y|}$,
- 3D Gaussian kernel: $K(x,y) = \exp(-|x-y|^2)$,
- 3D Stokes kernel: $K(x,y) = \frac{1}{|x-y|}I + \frac{(x-y)(x-y)^T}{|x-y|^3}$.

For each $\mathcal{H}^2$-matvec, denoted as an approximation $A_{\mathcal{H}^2}v \approx b = K(X,X)v$, its relative error is measured as

$$\text{relerr} = \frac{\sqrt{\sum_{i\in S}(b_i - (A_{\mathcal{H}^2}v)_i)^2}}{\sqrt{\sum_{i\in S} b_i^2}}, \tag{5.6}$$

where S is a set of 10000 indices randomly chosen from 1 to the length of b and the entries $\{b_i\}_{i\in S}$ are computed via direct matrix-vector multiplication.

Table 5.1 shows the relative error of $\mathcal{H}^2$-matvec for the two types of point sets with different prescribed relative error thresholds for the ID approximation in $\mathcal{H}^2$-construction. Both the *sphere* and *ball* point sets contain 1×10^6 points. All the multiplicand vectors for $\mathcal{H}^2$-matvec have entries randomly and uniformly generated between $[-1, 1]$. Each reported relative error is the average result obtained by 10 independent $\mathcal{H}^2$-matvec tests. The prescribed relative error threshold varies from 1×10^{-2} to 1×10^{-12}. As can be observed, for all tested kernel functions and types of point sets, the relative errors of $\mathcal{H}^2$-matvec are controlled by the prescribed threshold. Further, when the relative error threshold is above 1×10^{-8}, the actual relative error is usually an order of magnitude smaller than the threshold.

Table 5.1: Relative error of $\mathcal{H}^2$-matvec in H2Pack for several kernel functions with different prescribed relative error thresholds for the ID approximation in $\mathcal{H}^2$-construction ("ID approx. relerr"). Both sphere and ball points sets are tested. Each point set contains 1×10^6 points.

ID approx. relerr		1.00E-2	1.00E-4	1.00E-6	1.00E-8	1.00E-10	1.00E-12
3D Laplace	sphere	8.42E-4	3.68E-6	4.30E-8	6.35E-10	2.97E-11	9.20E-13
	ball	8.21E-4	4.13E-6	4.54E-8	8.05E-10	4.27E-11	5.30E-13
3D Gaussian	sphere	3.38E-3	1.89E-5	2.35E-7	4.25E-9	1.73E-11	2.38E-13
	ball	3.57E-3	1.53E-5	1.46E-7	1.13E-9	1.09E-11	3.85E-12
3D Stokes	sphere	1.26E-3	7.06E-6	6.02E-8	3.73E-10	2.46E-12	2.71E-12
	ball	1.42E-3	7.72E-6	3.20E-7	2.61E-9	2.69E-11	2.76E-12

5.4.2 Scalability tests

We now demonstrate the strong scalability (fixed problem size) of H2Pack. We test the 3D Laplace kernel with a ball point set of size 2×10^5 points and with 1×10^{-6} prescribed matvec relative error. Under this setting, the constructed $\mathcal{H}^2$ matrix representation in AOT mode can be completely stored in the 16 GB MCDRAM high-bandwidth memory of the Knights Landing node. On the Skylake node, we run H2Pack using one thread per core on all 48 cores. On the Knights Landing node, we run H2Pack using one thread per core on all 64 cores. Figure 5.5 shows the timings of $\mathcal{H}^2$-construction ("build") and $\mathcal{H}^2$-matvec ("matvec") of H2Pack in AOT and JIT modes on the two different nodes.

For $\mathcal{H}^2$-construction, JIT mode is faster than AOT mode on both types of compute nodes since AOT mode additionally calculates and stores $B_{i,j}$ and $D_{i,j}$ blocks. For both modes, however, $\mathcal{H}^2$-construction does not fully scale to all the cores on both types of nodes. The reason is that the performance of $\mathcal{H}^2$-construction is limited by memory bandwidth. The major computational kernel in $\mathcal{H}^2$-construction in both modes is the column-pivoted QR factorization to compute ID approximations (lines 6 and 9 in Algorithm 7). On both nodes, this computational kernel takes more than 95% and 50% of $\mathcal{H}^2$-construction time in JIT mode and AOT mode, respectively. Meanwhile, this computational kernel has a low computation-to-memory-access ratio and thus its performance is determined by the

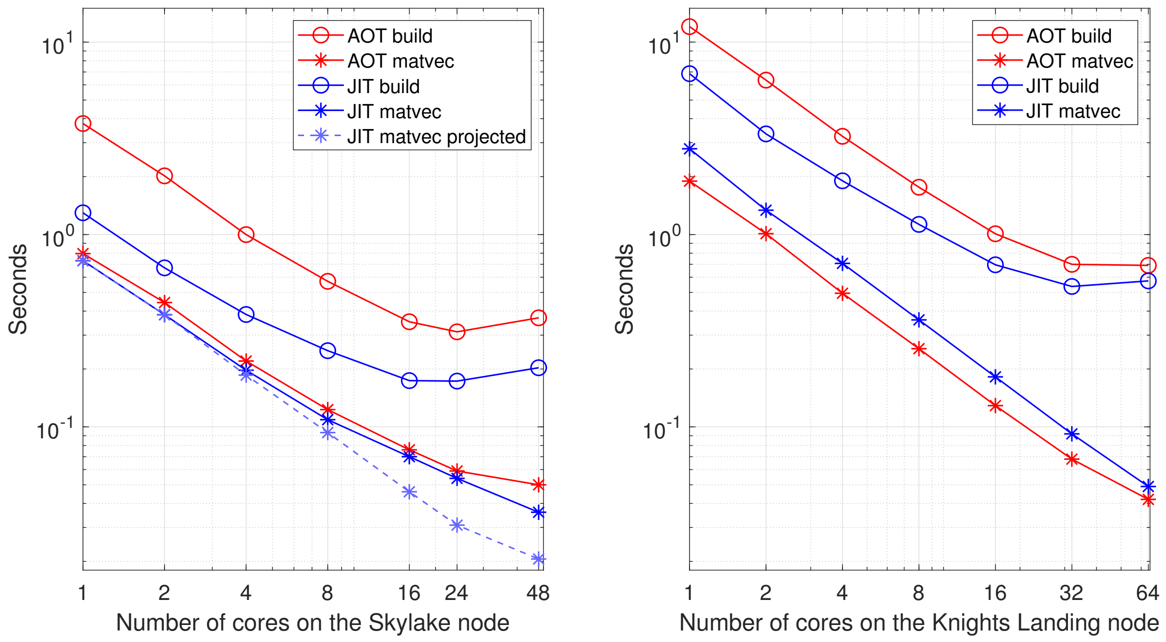


Figure 5.5: H2Pack $\mathcal{H}^2$ construction (“build”) and $\mathcal{H}^2$ matvec (“matvec”) timings in AOT mode and JIT mode on an Intel Skylake node (left) and an Intel Knights Landing node (right) using different numbers of cores. Projected $\mathcal{H}^2$-matvec time in JIT mode assuming all processors always run at 3.5GHz (“JIT matvec projected”) on the Skylake node is also plotted as a reference. A *ball* point set with 2×10^5 points and a 10^{-6} prescribed matvec relative error are used.

memory access bandwidth. Intel VTune (Intel performance profiling software) reports that the achieved memory bandwidth of this computational kernel is more than 80% of the peak memory bandwidth when using all cores on both nodes.

For $\mathcal{H}^2$-matvec, JIT mode is faster than AOT mode on the Skylake node while AOT mode is faster than JIT mode on the Knights Landing node, which is due to hardware differences between the Skylake node and the Knights Landing node. The Knights Landing node has high memory bandwidth but its single core performance is only moderate. On this node, the parallel efficiencies of $\mathcal{H}^2$-matvec in JIT and AOT modes are 89.0% and 70.5%, respectively, showing excellent scalability. Intel VTune reports that $\mathcal{H}^2$-matvec in AOT mode only utilizes about 65% of the peak MCDRAM memory bandwidth on the Knights Landing node when using all 64 cores. The Skylake node has powerful processor cores with moderate memory bandwidth. On this node, the parallel efficiencies of $\mathcal{H}^2$-matvec in JIT and AOT modes are only 33.1% and 42.3%, respectively. Intel VTune reports that $\mathcal{H}^2$-matvec in AOT mode achieved 79% and 90% of the peak memory bandwidth when using 24 and 48 cores on the Skylake node, suggesting that the lower parallel efficiency in AOT mode than in JIT mode is caused by the memory bandwidth limit. Furthermore, the lower parallel efficiency in JIT mode on the Skylake node than on the Knights Landing node (i.e., 33.1% v.s. 89.0%) is due to Intel Turbo Boost technology on Intel Xeon Platinum processors. If only one core is active on a Xeon Platinum 8160 processor (on the Skylake node), this core runs at 3.5 GHz. The more active cores, the lower the clock frequency of the cores. If all 24 cores are active, all the cores run at only 2.0 GHz. In comparison, all cores on the Knights Landing node always run at 1.4 GHz. This decrease of core frequency reduces the parallel efficiency of $\mathcal{H}^2$-matvec in JIT mode which requires a large number of kernel function evaluations. In Figure 5.5, we also plot the projected execution time for $\mathcal{H}^2$-matvec in JIT mode on the Skylake node assuming that all its cores always run at 3.5 GHz. The projected parallel efficiency of $\mathcal{H}^2$-matvec in JIT mode is 72.5% when using 48 cores.

Lastly, we also measure the performance of $\mathcal{H}^2$-matvec in JIT mode in terms of giga floating-point operations per second (GFLOPS). To measure this performance, we note that evaluating one value of the 3D Laplace kernel requires 19 floating-point operations (8 for the squared distance, 1 for the approximate RSQRT and 10 for two Newton iterations). On the Skylake node, $\mathcal{H}^2$-matvec in JIT mode achieved 1047 GFLOPS (34.9% of the peak performance). On the Knights Landing node, $\mathcal{H}^2$-matvec in JIT mode achieved 651 GFLOPS (24.5% of the peak performance).

5.4.3 Performance improvements by BKM and vectorization

The efficient evaluation of kernel functions is crucial to the overall performance of $\mathcal{H}^2$-construction and $\mathcal{H}^2$-matvec (in JIT mode). In this section, we study the performance improvements brought by the BKM interface and vector wrapper functions. Table 5.2 shows

Table 5.2: Timing results (in seconds) of $\mathcal{H}^2$-construction and $\mathcal{H}^2$-matvec using different implementations of KME and BKM functions for 3D Gaussian kernel: no vectorization ("no-vec"), automatic vectorization by the Intel C compiler ("auto-vec"), and manual vectorization by vector wrapper functions ("wrap-vec"). The JIT mode and relative error threshold 10^{-6} are used in all the tests.

#pts $\times 10^5$	ball			sphere		
	1	4	16	1	4	16
$\mathcal{H}^2$-construction						
KME no-vec	0.046	0.142	0.583	0.051	0.127	0.400
KME auto-vec	0.045	0.131	0.574	0.049	0.123	0.396
KME wrap-vec	0.043	0.128	0.566	0.050	0.123	0.394
$\mathcal{H}^2$-matvec						
KME no-vec	0.086	0.211	0.504	0.035	0.117	0.499
KME auto-vec	0.028	0.075	0.188	0.012	0.041	0.172
KME wrap-vec	0.018	0.057	0.146	0.008	0.035	0.119
BKM no-vec	0.092	0.264	0.735	0.040	0.138	0.586
BKM auto-vec	0.028	0.076	0.209	0.012	0.041	0.178
BKM wrap-vec	0.013	0.037	0.118	0.006	0.029	0.089

the timing results of $\mathcal{H}^2$-construction and $\mathcal{H}^2$-matvec using different implementations of KME and BKM functions for the 3D Gaussian kernel $K(x, y) = \exp(-|x - y|^2)$: no vectorization, automatic vectorization by the Intel C compiler, and our manual vectorization by vector wrapper functions.

As explained in Section 5.4.2, $\mathcal{H}^2$-construction in JIT mode is dominated by the column-pivoted QR factorization, and thus only gains minor performance improvements from vectorization. Meanwhile, both automatic and manual vectorization of KME and BKM functions can lead to 300%-400% speedup in $\mathcal{H}^2$-matvec, with the manual vectorization being 20%-50% faster than the automatic vectorization. Comparing KME and BKM functions for $\mathcal{H}^2$-matvec, using KME functions without vectorization or with automatic vectorization can be even faster than using BKM functions. This is because the dense matrix-vector multiplication after evaluating a kernel block by KME functions is vectorized in the BLAS library. On the other hand, based on the manual vectorization, using BKM functions is 20%-35% faster than using KME functions.

5.4.4 Comparison between $\mathcal{H}^2$-matvec and $\mathcal{H}^2$-matmul

In this section, we compare the performance of $\mathcal{H}^2$-matvec and $\mathcal{H}^2$-matmul in H2Pack for multiplying multiple vectors. In the latter, the vectors are assumed to be available at the same time and the multiplications are performed simultaneously. Figure 5.6 shows the timings of $\mathcal{H}^2$-matvec and $\mathcal{H}^2$-matmul to multiply different numbers of vectors in both AOT and JIT modes. The results are qualitatively similar for other kernel functions and point sets. Column-major format for the matrix of vectors was used; the timings for row-major format are very similar. The corresponding $\mathcal{H}^2$-construction in AOT and JIT modes take 2.54 seconds and 1.11 seconds, respectively. The runtime of $\mathcal{H}^2$-matmul increases much more slowly with the number of vectors compared to $\mathcal{H}^2$-matvec. This indicates that calculating $B_{i,j}$ and $D_{i,j}$ blocks in JIT mode or transferring these blocks from the main memory to the processor cache in AOT mode are very expensive compared to the actual multiplication. For a single vector, $\mathcal{H}^2$-matmul is slower than $\mathcal{H}^2$-matvec because the symmetry property of $B_{i,j}$ and $D_{i,j}$ is not exploited (see Section 5.3.1). In AOT mode, the performance of $\mathcal{H}^2$-matmul is further affected by NUMA. $\mathcal{H}^2$-matvec uses a fixed workload partitioning and $B_{i,j}$ and $D_{i,j}$ blocks are optimized for this fixed partitioning using the first-touch policy. $\mathcal{H}^2$-matmul uses a different workload partitioning, making it hard to optimize for NUMA without almost doubling the storage.

5.4.5 Comparison with fast multipole methods

In this section, we compare the performance of H2Pack with two fast multipole method (FMM) libraries: the FMM3D library implements the standard FMM and the PVFMM library implements the kernel-independent FMM (KIFMM). We note that FMM3D works for the 3D Laplace, Stokes, and Helmholtz kernels, PVFMM works for kernel functions from potential theory, and H2Pack can work for non-oscillatory kernel functions in general. For all the libraries, we use the same sets of points and test using the 3D Laplace kernel. The number of points in X ranges from 1×10^5 to 1.6×10^6. For all three libraries,

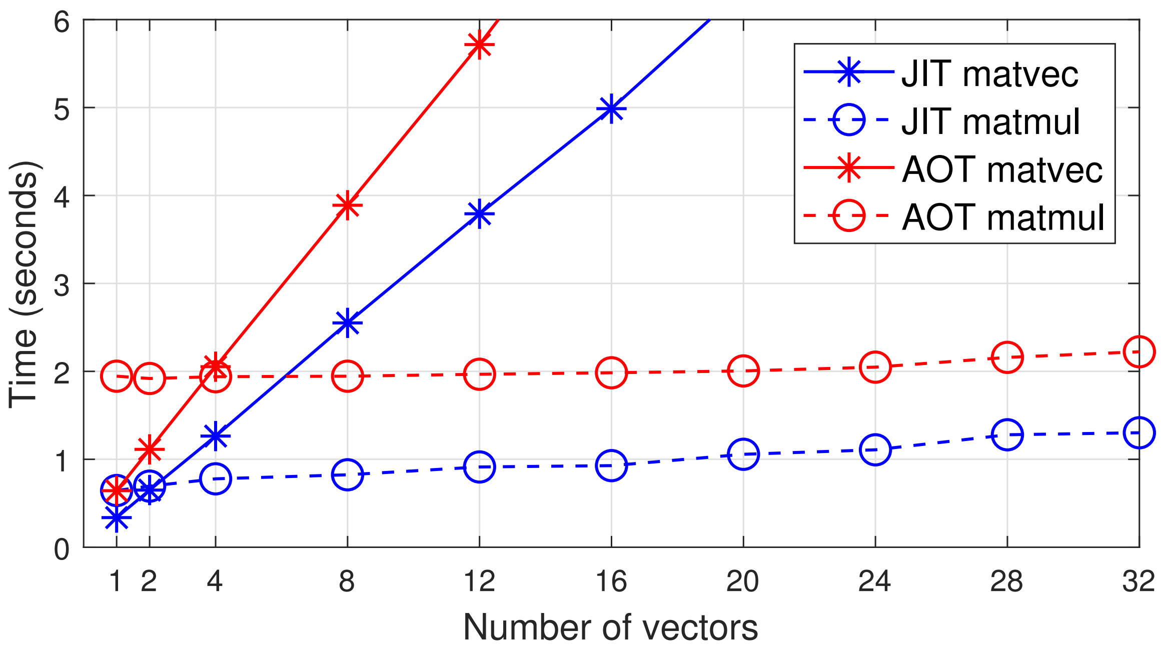


Figure 5.6: Timings of $\mathcal{H}^2$-matvec and $\mathcal{H}^2$-matmul (in seconds) in AOT and JIT modes for multiplying different numbers of vectors on the Skylake node. The test settings are: 3D Laplace kernel, a *ball* point set with 1.6×10^6 points, and a prescribed relative error threshold 10^{-6}.

we specify that a box is further partitioned into smaller boxes if it contains more than 400 points in the hierarchical partitioning of X. For H2Pack, JIT mode is used. All three libraries are compiled using Intel C/C++/Fortran compilers and Intel MPI 2018.0.2 with optimization flags "-xHost -O3". Intel MKL 2018.0.2 is used to perform optimized general matrix-vector multiplications (xGEMV), general matrix-matrix multiplications (xGEMM), and fast Fourier transformations that appear in these three libraries. Double precision floating point is used for storage and calculations in all three libraries.

We run all three libraries using one thread per core on all 48 cores on a Skylake node. Tables 5.3 to 5.5 show the test results corresponding to relative multiplication accuracy of approximately 10^{-5}, 10^{-8}, and 10^{-11}, respectively. The tables show results for the following quantities:

- **Precomputation cost**. The runtime of specific precomputations in H2Pack and PVFMM that can be reused for different sets of points but not for different accuracy requirements and for different kernel functions. (FMM3D does not have precomputations.) In H2Pack, the precomputation involves computing the proxy points. In PVFMM, the precomputation involves computing fixed translation operators in KIFMM and storing them into a file.

- **Setup cost**. The runtime of all the computations other than the precomputations above before matrix-vector multiplications, i.e., hierarchical partitioning of X in all the libraries and $\mathcal{H}^2$-construction in H2Pack.
- **Peak memory**. The peak memory usage recorded by the operating system during the entire program execution (precomputation, setup, and matrix-vector multiplication).
- **Storage cost**. The storage cost of the translation operators in PVFMM and of the $\mathcal{H}^2$ matrix components in H2Pack. (FMM3D does not report its storage cost.)
- **Runtime and relative error of the multiplication**. These results are averaged over 5 multiplications by random vectors for each point set X. The relative error is defined in Equation 5.6.
- **Degree and rank**. The "degree" in PVFMM and FMM3D is an input parameter characterizing the number of expansion terms used for analytic compression of kernel matrix blocks. In PVFMM, a degree of k corresponds to a rank-$6k^2$ analytic approximation of each block to be compressed in the equivalent $\mathcal{H}^2$ matrix representation. In FMM3D, a degree of k corresponds to the approximation rank being $(k+1)^2$. For H2Pack, the resulting maximum and average ranks of all the low-rank approximations in each constructed $\mathcal{H}^2$ matrix are listed.

From the results, the cost for $\mathcal{H}^2$-construction ("setup") in H2Pack scales linearly in the number of points and increases with higher relative multiplication accuracy. For points in a unit ball, the H2Pack setup cost can be much more expensive than the setup costs in PVFMM and FMM3D. However, the setup cost of H2Pack is much cheaper for points on the unit sphere than in the unit ball. This is due to the smaller approximation ranks for all the blocks compressed in $\mathcal{H}^2$-construction.

The maximum and average approximation ranks in H2Pack are all much smaller than those in PVFMM and FMM3D. The approximation ranks in H2Pack are different with different point distributions, while PVFMM and FMM3D have fixed approximation ranks for both types of point distributions. As a result, H2Pack is the fastest library for matrix-vector multiplications among the three and this efficiency advantage becomes even greater when dealing with points on the unit sphere, i.e., around 5 times faster than PVFMM and 25 times faster than FMM3D.

The storage cost of H2Pack is proportional to the number of points and the approximation ranks in the constructed $\mathcal{H}^2$ matrices. In comparison, the storage cost of PVFMM changes very mildly under different problem settings. H2Pack has much smaller storage cost for small problems compared with PVFMM but ultimately can have larger storage cost when the number of points or the relative accuracy increases. For example, H2Pack begins to have more storage cost for 8×10^5 points in the unit ball with relative accuracy 10^{-11}. FMM3D does not report its storage cost but theoretically only has very small storage cost for temporary components.

It is worth noting that the peak memory recorded by the operating system depends on the actual implementations of these libraries and can only be used as a rough reference for comparing the three different methods. As can be noted, H2Pack has its peak memory increasing much faster than PVFMM and eventually has larger peak memory than PVFMM when dealing with large numbers of points and high relative accuracy, e.g., 8×10^5 points in the unit ball with relative accuracy 10^{-11}. Meanwhile, FMM3D also has increasing peak memory with more points but has the smallest peak memory among the three libraries when dealing with a large number of points.

Compared to FMM3D, both H2Pack and PVFMM have relatively expensive precomputations. For H2Pack, the precomputation involves the kernel-related proxy point selection.

However, for a given kernel function, the selected proxy points in H2Pack can be saved to a file and reused in future computations. For certain kernel functions such as the Laplace and Stokes kernels, H2Pack can also apply the proxy surface method [110] to generate the proxy points with negligible computation cost. For PVFMM, the precomputation involves computing fixed translation operators and its complexity depends on the kernel function and the degree parameter (which controls the relative accuracy). These precomputed results in PVFMM are stored in files for reuse. Since the precomputations in H2Pack and PVFMM can be reused when the kernel function and the relative accuracy are fixed, the precomputation costs typically make no impact in practice.

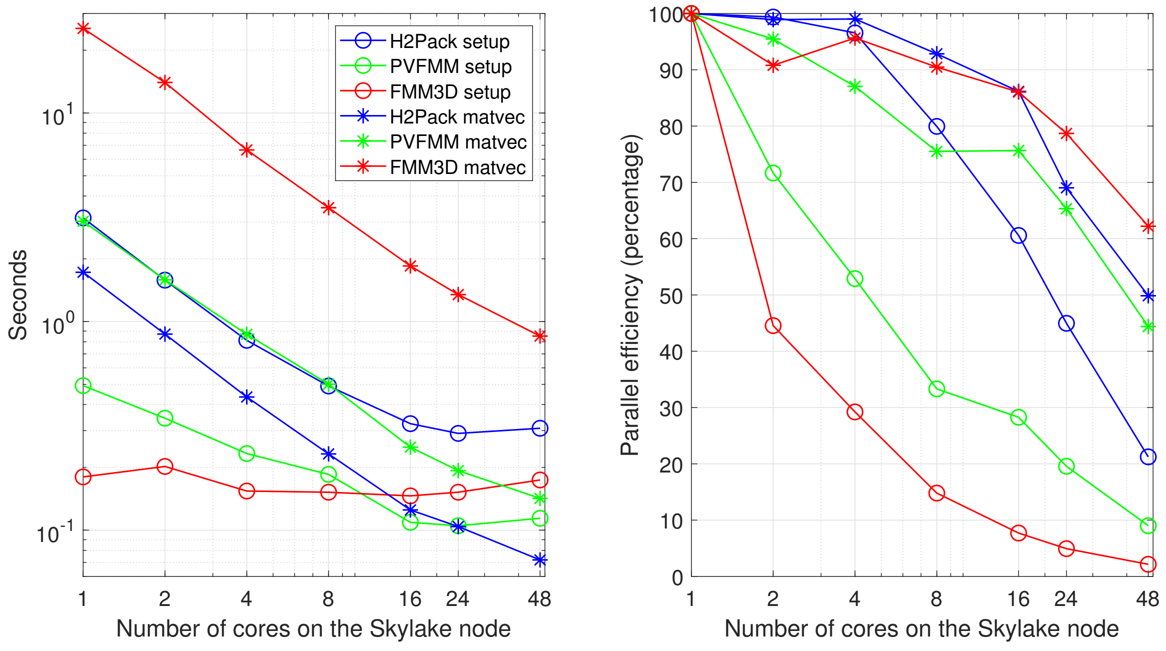


Figure 5.7: Setup and matvec timings (in seconds) and parallel efficiency (in percentage) using different numbers of cores on a Skylake node for FMM3D, PVFMM, and H2Pack. A *ball* point set with 4×10^5 points and a 10^{-8} prescribed matvec relative error threshold are used.

Figure 5.7 shows the timings and parallel efficiencies of the “setup” and “matvec” procedures of the three tested libraries. For H2Pack, the results in Figure 5.7 are similar to the results in Figure 5.5, but the parallel efficiency of $\mathcal{H}^2$-matvec when using 48 cores is higher in Figure 5.7 (49.9% v.s. 33.1%) due to more points and a larger parallelism. The setup procedure is not parallelized in FMM3D and not fully parallelized in PVFMM, leading to poor parallel efficiencies in FMM3D and PVFMM for the setup. Although FMM3D has slightly better parallel efficiency in matvec compared to PVFMM and H2Pack, its absolute matvec time is much larger than the matvec time of PVFMM and H2Pack.

To summarize, the numerical comparisons above show that FMM libraries typically have less cost for setup and storage but also typically have slower matrix-vector multiplications than H2Pack. Thus, FMM libraries are more suitable for problems where only a few matrix-vector multiplications are required per set of points, e.g., particle simulations. Meanwhile, H2Pack is more suitable for problems where many matrix-vector multiplications are required per set of points, e.g., numerical solution of integral equations and Gaussian processes, so that the relatively expensive $\mathcal{H}^2$ construction cost can be amortized by many multiplications.

Table 5.3: Numerical results of the three libraries with relative accuracy around 10^{-5}. "Precomp" refers to the precomputations in H2Pack and PVFMM. "Mem" refers to the peak memory usage recorded by the operating system. "Storage" refers to the storage cost of translation operators in PVFMM and that of $\mathcal{H}^2$ matrix components in H2Pack.

H2Pack							
#pts $\times 10^5$	precomp(s)	setup(s)	matvec(s)	mem(MB)	storage(MB)	relerr	max/avg rank
sphere 1	0.185	0.062	0.005	519	15	1.63E-05	29/15
sphere 2	0.194	0.082	0.010	436	30	1.91E-05	28/15
sphere 4	0.187	0.125	0.018	569	59	2.12E-05	28/15
sphere 8	0.239	0.223	0.037	839	119	2.34E-05	28/15
sphere 16	0.278	0.444	0.075	1345	234	2.64E-05	28/15
ball 1	0.159	0.073	0.010	669	43	1.68E-05	69/40
ball 2	0.162	0.160	0.019	823	89	1.86E-05	69/35
ball 4	0.157	0.173	0.035	836	163	2.14E-05	71/39
ball 8	0.194	0.249	0.090	1181	308	2.56E-05	70/38
ball 16	0.189	0.784	0.149	2418	723	2.84E-05	70/37

PVFMM								
#pts $\times 10^5$	precomp(s)	setup(s)	matvec(s)	mem(MB)	storage(MB)	relerr	degree	rank
sphere 1	1.161	0.069	0.020	1164	1138	7.51E-06	5	150
sphere 2	1.117	0.087	0.027	1388	1186	8.85E-06	5	150
sphere 4	1.114	0.134	0.056	1799	1271	6.41E-06	5	150
sphere 8	1.163	0.328	0.132	2753	1461	9.91E-06	5	150
sphere 16	1.115	0.686	0.230	4528	1845	9.70E-06	5	150
ball 1	1.113	0.042	0.035	1128	1134	1.99E-05	5	150
ball 2	1.113	0.083	0.030	1355	1186	1.49E-05	5	150
ball 4	1.114	0.113	0.075	1751	1252	1.85E-05	5	150
ball 8	1.113	0.221	0.267	2547	1394	2.98E-05	5	150
ball 16	1.115	0.691	0.219	4518	1832	1.54E-05	5	150

FMM3D						
#pts $\times 10^5$	setup(s)	matvec(s)	mem(MB)	relerr	degree	rank
sphere 1	0.041	0.120	238	8.81E-06	15	256
sphere 2	0.085	0.163	414	8.88E-06	15	256
sphere 4	0.183	0.329	747	9.41E-06	15	256
sphere 8	0.441	0.626	1397	8.61E-06	15	256
sphere 16	1.025	1.259	2784	9.56E-06	15	256
ball 1	0.042	0.167	302	6.55E-06	15	256
ball 2	0.081	0.168	353	6.85E-06	15	256
ball 4	0.170	0.192	554	6.77E-06	15	256
ball 8	0.443	1.266	1830	6.84E-06	15	256
ball 16	0.955	1.261	2025	6.75E-06	15	256

Table 5.4: Numerical results of the three libraries with relative accuracy around 10^{-8}.

H2Pack

#pts $\times 10^5$	precomp(s)	setup(s)	matvec(s)	mem(MB)	storage(MB)	relerr	max/avg rank
sphere 1	0.349	0.095	0.006	778	47	1.20E-08	77/39
sphere 2	0.419	0.143	0.012	717	90	1.43E-08	78/36
sphere 4	0.612	0.199	0.023	908	176	1.74E-08	78/36
sphere 8	0.727	0.326	0.047	1320	352	1.83E-08	79/37
sphere 16	0.948	0.589	0.097	2046	687	2.04E-08	77/36
ball 1	0.417	0.135	0.021	857	137	1.71E-08	194/96
ball 2	0.352	0.247	0.044	1294	331	1.68E-08	201/78
ball 4	0.339	0.312	0.078	1652	561	2.19E-08	203/94
ball 8	0.500	0.417	0.141	2246	984	2.58E-08	206/90
ball 16	0.438	1.190	0.340	4642	2362	3.07E-08	205/78

PVFMM

#pts $\times 10^5$	precomp(s)	setup(s)	matvec(s)	mem(MB)	storage(MB)	relerr	degree	rank
sphere 1	2.981	0.048	0.030	1397	1211	2.35E-08	8	384
sphere 2	2.967	0.089	0.052	1597	1266	2.46E-08	8	384
sphere 4	2.966	0.138	0.122	2112	1358	1.75E-08	8	384
sphere 8	2.966	0.471	0.201	3288	1574	2.57E-08	8	384
sphere 16	2.967	0.658	0.420	4927	1994	2.70E-08	8	384
ball 1	2.970	0.043	0.041	1233	1205	3.57E-08	8	384
ball 2	2.957	0.087	0.058	1530	1264	2.48E-08	8	384
ball 4	2.955	0.115	0.118	1948	1331	3.55E-08	8	384
ball 8	2.959	0.336	0.330	2800	1477	4.07E-08	8	384
ball 16	2.954	0.883	0.454	5052	1971	3.97E-08	8	384

FMM3D

#pts $\times 10^5$	setup(s)	matvec(s)	mem(MB)	relerr	degree	rank
sphere 1	0.041	0.156	298	1.10E-08	21	484
sphere 2	0.084	0.295	454	1.21E-08	21	484
sphere 4	0.184	0.535	860	1.14E-08	21	484
sphere 8	0.418	1.099	1615	1.19E-08	21	484
sphere 16	1.027	2.138	3235	1.28E-08	21	484
ball 1	0.042	0.172	366	1.13E-08	21	484
ball 2	0.081	0.210	359	1.10E-08	21	484
ball 4	0.171	0.863	632	1.11E-08	21	484
ball 8	0.452	1.037	2113	1.11E-08	21	484
ball 16	0.926	1.322	2330	1.18E-08	21	484

Table 5.5: Numerical results of the three libraries with relative accuracy around 10^{-11}.

H2Pack

#pts $\times 10^5$	precomp(s)	setup(s)	matvec(s)	mem(MB)	storage(MB)	relerr	max/avg rank
sphere 1	0.962	0.159	0.009	1222	114	5.66E-12	165/73
sphere 2	1.018	0.203	0.019	1160	236	6.06E-12	165/70
sphere 4	1.151	0.292	0.035	1616	437	7.90E-12	165/69
sphere 8	1.022	0.493	0.072	2398	864	8.43E-12	165/69
sphere 16	1.728	0.903	0.144	4035	1707	9.13E-12	166/68
ball 1	0.906	0.590	0.035	1676	391	2.35E-12	444/184
ball 2	0.873	0.847	0.082	2270	796	6.65E-12	450/109
ball 4	0.792	1.502	0.177	3590	1604	9.93E-12	444/168
ball 8	1.011	2.438	0.305	5426	2709	1.86E-11	450/167
ball 16	0.941	4.057	0.633	9442	5539	2.50E-11	449/109

PVFMM

#pts $\times 10^5$	precomp(s)	setup(s)	matvec(s)	mem(MB)	storage(MB)	relerr	degree	rank
sphere 1	9.700	0.055	0.054	1953	1445	1.29E-11	12	864
sphere 2	9.559	0.104	0.126	2196	1517	1.47E-11	12	864
sphere 4	9.558	0.159	0.234	2555	1624	9.75E-12	12	864
sphere 8	9.562	0.600	0.491	3535	1893	1.44E-11	12	864
sphere 16	9.575	1.014	0.890	5496	2392	1.69E-11	12	864
ball 1	9.547	0.056	0.060	1527	1434	2.76E-11	12	864
ball 2	9.578	0.158	0.151	2086	1510	2.22E-11	12	864
ball 4	9.652	0.180	0.181	2514	1578	2.73E-11	12	864
ball 8	9.595	0.410	0.430	3552	1880	4.31E-11	12	864
ball 16	9.607	0.784	1.123	5544	2351	2.17E-11	12	864

FMM3D

#pts $\times 10^5$	setup(s)	matvec(s)	mem(MB)	relerr	degree	rank
sphere 1	0.034	0.272	278	9.90E-12	29	900
sphere 2	0.078	0.472	553	1.08E-11	29	900
sphere 4	0.167	0.899	907	1.09E-11	29	900
sphere 8	0.375	1.698	1780	1.12E-11	29	900
sphere 16	0.917	3.541	3366	1.11E-11	29	900
ball 1	0.037	0.238	208	9.55E-12	29	900
ball 2	0.098	0.522	678	1.07E-11	29	900
ball 4	0.163	0.654	728	1.10E-11	29	900
ball 8	0.346	2.117	994	1.08E-11	29	900
ball 16	0.947	3.106	4502	1.14E-11	29	900

5.5 Conclusion

H2Pack provides linear-scaling matrix-vector multiplication for kernel matrices defined by non-oscillatory kernel functions. Such multiplications are needed on their own in many applications, but can also be used in iterative solvers for kernel matrix systems. The critical step for linear-scaling matrix-vector multiplication is constructing the $\mathcal{H}^2$ matrix representation of the kernel matrix. In H2Pack, this is done by using the recently-developed proxy point method. The advantages of using the proxy point method are (1) greater generality compared to other methods (e.g., it works for Gaussian kernels), and (2) more effective block low-rank compression compared to analytic methods such as those used in FMM. The latter is what makes H2Pack matrix-vector multiplication faster than kernel summation in FMM libraries. On the other hand, constructing the $\mathcal{H}^2$ matrix representation in H2Pack is often more expensive than the setup phase in FMM libraries.

We have focused on translationally-invariant kernels. This allows the proxy points for each box (in a given level of the partition tree) to be translates of each other, thus reducing the overall cost of proxy point selection. We have also focused on 2D and 3D problems, as is common for FMM libraries. H2Pack can be extended to higher dimensions if a cheap method of selecting proxy points in higher dimensions is available.

In standard $\mathcal{H}^2$ matrix representations, blocks of the matrix are either admissible (represented as a low-rank block) or inadmissible (represented as a dense block). In H2Pack, we introduce the concept of partially admissible blocks. Such blocks arise with non-uniform distributions of points, leading to non-perfect partition trees. By treating partially admissible blocks in the appropriate way (rather than as either admissible or inadmissible), the representation of these blocks is more efficient. The same technique exists in FMM libraries but not in existing $\mathcal{H}^2$ matrix libraries.

H2Pack has been optimized for high-performance on shared-memory parallel computers. Important considerations are vectorization of kernel function evaluations, reducing memory traffic, and load balancing. Just-in-time and ahead-of-time modes are provided to trade computation with storage and memory traffic. Vectorization of kernel function evaluations is particularly important in just-in-time mode, and a kernel function interface is described. Numerical tests show good scaling of H2Pack matrix-vector multiplication with the number of cores. For constructing the $\mathcal{H}^2$ matrix representations, the performance with large numbers of cores is limited by the high memory bandwidth requirement of the column-pivoted QR factorization used in the code.

CHAPTER 6
CONCLUSION

One of the major challenges of scaling matrix computations to massively parallel computers is reducing communication costs. Communication-avoiding and communication-reducing algorithms address this challenge. While many algorithms and theoretical analyses for parallel dense-dense matrix multiplication have been proposed in the past fifty years, there is still a gap between the theoretically communication-optimal algorithm and the parallel implementation. Parallelizing dense-sparse multiplication is more challenging than parallelizing dense-dense matrix multiplication since the sparsity introduces a vast design space of parallelization. Many methods have also been proposed for compressing and using kernel matrices, a special classic of dense matrices. The lack of a high-performance, multipurpose library hinders the efficient use of kernel matrices in many applications. In this dissertation, we make important steps to address these issues. Here, we summarize the main contributions of this dissertation and future work.

6.1 Communication-Avoiding 3D Matrix Multiplication

In this work, we propose 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 allows CA3DMM to reduce to 1D, 2D, or 3D algorithms for different problem dimensions and different numbers of processes to achieve optimal or near-optimal communication costs. Experimental results show that CA3DMM outperforms state-of-the-art methods, further supporting the theoretical analysis.

The current design of CA3DMM has two major limitations, and we plan to address these issues in our future work. The first issue is memory usage. CA3DMM consumes more memory than 2D algorithms to reduce communication costs. We plan to make the

CA3DMM algorithm more flexible to operate under a given memory size constraint and trade between memory usage and communication costs. The second issue is reducing the matrix distribution conversion costs since CA3DMM employs special 2D distributions for input and output matrices. Converting from a natural 1D/2D distribution or a widely used 2D block-cyclic distribution to the internal distributions could be expensive.

6.2 Hybrid Polar Decomposition

In this work, we review different iterative methods for computing polar decomposition and propose multiple hybrid approaches for scaling up polar decomposition on large parallel computers. The HPD approach utilizes the differences in the scalability of different basic linear algebra operations and the differences in the convergence behaviors of different iterative PD methods. By adopting the CA3DMM algorithm and a new parallel matrix column orthonormalization algorithm, HPD demonstrates better parallel performance and scalability than existing ScaLAPACK-based parallel PD implementations.

Experiment results also suggest some future research topics for us. Firstly, the parallel efficiency of the new column orthonormalization algorithm is not satisfactory, while column orthonormalization is necessary for high-accuracy PD calculation. Secondly, if we can improve the accuracy of the blocked Gauss-Jordan algorithm, the scaled Newton method can be more competitive since it runs much faster than the QDWH method.

6.3 Communication-Reduced Parallel SpMM

In this work, we analyze the vast design space of parallel SpMM algorithms, formulate communication cost models for different parallelization schemes, and propose CRP-SpMM for optimizing the process grid geometry and reducing communication costs. CRP-SpMM reuses existing 1D parallel algorithms for SpMV and can benefit from research in (hyper)graph partitioning and other sparse matrix partitioning methods. Experimental results show that CRP-SpMM can find better process grids that reduce the total communication size even when high-quality 1D row partitionings are used as baselines. Our implementation also significantly outperforms existing distributed-memory parallel SpMM codes.

We plan to improve CRP-SpMM in the following aspects in future research. Firstly, the current algorithm for computing new 1D partitionings from the baseline partitioning should be enhanced to calculate new 1D partitionings with smaller communication costs. Secondly, we are considering extending CRP-SpMM to support generalized 3D parallelization, which requires careful redesign and implementation of the process grid search algorithm. Shifting from a generalized 2D algorithm to a generalized 3D algorithm may further reduce communication costs and improve parallel performance. Lastly, we can adopt low-level optimizations to enhance the performance of CRP-SpMM.

6.4 H2Pack

We present the H2Pack library, a high-performance multi-purpose library for kernel matrices defined by translationally invariant kernels and low-dimensional (2D/3D) problems. We discuss multiple optimization techniques used in H2Pack for efficient $\mathcal{H}^2$ matrix construction and multiplication of a $\mathcal{H}^2$ matrix with a dense vector or matrix. Numerical experimental results show that H2Pack can achieve high accuracy and outperforms state-of-the-art FMM libraries in certain application scenarios.

We are working on adopting the algorithms and techniques in H2Pack to a new package designed for Gaussian Process calculations. In the training of GPs, the kernel function parameters are updated rapidly, and the kernel matrix changes accordingly. Therefore, a faster $\mathcal{H}^2$ matrix construction algorithm for both low-dimensional and high-dimensional data points and its parallel implementation need to be developed.

REFERENCES

[1] G. H. Golub and C. F. Van Loan, *Matrix Computations*, Forth. The Johns Hopkins University Press, 2013.

[2] C. D. Meyer, *Matrix Analysis and Applied Linear Algebra*. USA: SIAM, 2000.

[3] N. J. Higham, *Functions of Matrices: Theory and Computation*. Philadelphia, PA, USA: SIAM, 2008, pp. xx+425.

[4] Y. Saad, *Numerical Methods for Large Eigenvalue Problems*. SIAM, 2011.

[5] 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*, May 2015, pp. 804–811.

[6] J. Kepner *et al.*, "Mathematical foundations of the GraphBLAS", in *2016 IEEE High Performance Extreme Computing Conference (HPEC)*, Sep. 2016, pp. 1–9.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] J.-W. Hong and H.-T. Kung, "I/O complexity: The red-blue pebble game", ser. STOC '81, Milwaukee, Wisconsin, USA: ACM, 1981, pp. 326–333.

[12] 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.

[13] 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.

[14] 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, pp. 1–155, 2014.

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

[16] 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.

[17] 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, Aug. 2011, pp. 90–109.

[18] J. Demmel *et al.*, "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.

[19] 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.

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

[21] 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.

[22] 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, 1992, pp. 120–127.

[23] 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, Denver, Colorado: ACM, Nov. 2019.

[24] 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.

[25] 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*, May 2013, pp. 813–824.

[26] V. Strassen, "Gaussian elimination is not optimal", *Numerische Mathematik*, vol. 13, no. 4, pp. 354–356, Aug. 1969.

[27] 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, Vienna, Austria: ACM, 1997, pp. 44–51.

[28] R. Thakur, R. Rabenseifner, and W. Gropp, "Optimization of collective communication operations in MPICH", *International Journal of High Performance Computing Applications*, vol. 19, no. 1, pp. 49–66, Feb. 2005.

[29] 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, A1325–A1349, Jan. 2013.

[30] 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, Nov. 2019, pp. 1–11.

[31] 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, A477–A503, 2020.

[32] 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.

[33] Q. Xu *et al.*, "SPARC: Simulation package for *Ab-initio* real-space calculations", *SoftwareX*, vol. 15, p. 100 709, 2021.

[34] D. Sukkari, H. Ltaief, and D. Keyes, "A high performance QDWH-SVD solver using hardware accelerators", *ACM Transactions on Mathematical Software*, vol. 43, no. 1, pp. 1–25, Aug. 2016.

[35] D. Sukkari, H. Ltaief, A. Esposito, and D. Keyes, “A QDWH-based SVD software framework on distributed-memory manycore systems”, *ACM Transactions on Mathematical Software*, vol. 45, no. 2, pp. 1–21, Jun. 2019.

[36] S. Li, J. Liu, and Y. Du, “A high performance implementation of Zolo-SVD algorithm on distributed memory systems”, *Parallel Computing*, vol. 86, pp. 57–65, Aug. 2019.

[37] H. Ltaief, D. Sukkari, A. Esposito, Y. Nakatsukasa, and D. Keyes, “Massively parallel polar decomposition on distributed-memory systems”, *ACM Transaction on Parallel Computing*, vol. 6, no. 1, Jun. 2019.

[38] G. Ballard, J. Demmel, O. Holtz, and O. Schwartz, “Communication-optimal parallel and sequential Cholesky decomposition”, *SIAM Journal on Scientific Computing*, vol. 32, no. 6, pp. 3495–3523, 2010.

[39] J. Demmel, L. Grigori, M. Hoemmen, and J. Langou, “Communication-optimal parallel and sequential QR and LU factorizations”, *SIAM Journal on Scientific Computing*, vol. 34, no. 1, A206–A239, 2012.

[40] R. Byers and H. Xu, “A new scaling for Newton’s iteration for the polar decomposition and its backward stability”, *SIAM Journal on Matrix Analysis and Applications*, vol. 30, no. 2, pp. 822–843, Jan. 2008.

[41] Y. Nakatsukasa, Z. Bai, and F. Gygi, “Optimizing Halley’s iteration for computing the matrix polar decomposition”, *SIAM Journal on Matrix Analysis and Applications*, vol. 31, no. 5, pp. 2700–2720, Sep. 2010.

[42] Y. Nakatsukasa and R. W. Freund, “Computing fundamental matrix decompositions accurately via the matrix sign function in two iterations: The power of Zolotarev’s functions”, *SIAM Review*, vol. 58, no. 3, pp. 461–493, Jan. 2016.

[43] J. Chen and E. Chow. “A stable scaling of Newton-Schulz for improving the sign function computation of a Hermitian matrix”. Preprint ANL/MCS-P5059-0114, Argonne National Laboratory. (2014).

[44] T. Fukaya, Y. Nakatsukasa, Y. Yanagisawa, and Y. Yamamoto, “CholeskyQR2: A simple and communication-avoiding algorithm for computing a tall-skinny QR factorization on a large-scale parallel system”, in *2014 5th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*, Nov. 2014, pp. 31–38.

[45] E. Hutter and E. Solomonik, “Communication-avoiding Cholesky-QR2 for rectangular matrices”, in *2019 IEEE 33rd International Parallel and Distributed Processing Symposium*, IEEE, May 2019, pp. 89–100.

[46] N. J. Higham, “Computing the polar decomposition-with applications”, *SIAM Journal on Scientific and Statistical Computing*, vol. 7, no. 4, pp. 1160–1174, 1986.

[47] A. Kiełbasiński and K. Zietak, “Numerical behaviour of Higham’s scaled method for polar decomposition”, *Numerical Algorithms*, vol. 32, pp. 105–140, Apr. 2003.

[48] T. J. Dekker and W. Hoffmann, “Rehabilitation of the Gauss-Jordan algorithm”, *Numerische Mathematik*, vol. 54, no. 5, pp. 591–599, Sep. 1989.

[49] N. Melab, E.-G. Talbi, and S. Petiton, “A parallel adaptive version of the block-based Gauss-Jordan algorithm”, in *Proceedings of the 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing. IPPS/SPDP 1999*, San Juan, Puerto Rico: IEEE, Apr. 1999, pp. 350–354.

[50] N. J. Higham and R. S. Schreiber, “Fast polar decomposition of an arbitrary matrix”, *SIAM Journal on Scientific and Statistical Computing*, vol. 11, no. 4, pp. 648–655, 1990.

[51] A. V. Knyazev, “Toward the optimal preconditioned eigensolver: Locally optimal block preconditioned conjugate gradient method”, *SIAM Journal on Scientific Computing*, vol. 23, no. 2, pp. 517–541, 2001.

[52] Y. Zhou and Y. Saad, “Block Krylov–Schur method for large symmetric eigenvalue problems”, *Numerical Algorithms*, vol. 47, pp. 341–359, 4 2008.

[53] P. Maris *et al.*, “Large-scale ab initio configuration interaction calculations for light nuclei”, *Journal of Physics: Conference Series*, vol. 403, no. 1, p. 012 019, Dec. 2012.

[54] X. Liu, E. Chow, K. Vaidyanathan, and M. Smelyanskiy, “Improving the performance of dynamical simulations via multiple right-hand sides”, in *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, IEEE, May 2012, pp. 36–47.

[55] J. Kim and H. Park, “Fast nonnegative matrix factorization: An active-set-like method and comparisons”, *SIAM Journal on Scientific Computing*, vol. 33, no. 6, pp. 3261–3281, 2011.

[56] R. Kannan, G. Ballard, and H. Park, “MPI-FAUN: An MPI-based framework for alternating-updating nonnegative matrix factorization”, *IEEE Transactions on Knowledge and Data Engineering*, vol. 30, no. 3, pp. 544–558, 2018.

[57] A. Tripathy, K. Yelick, and A. Buluç, “Reducing communication in graph neural network training”, in *Proceedings of the International Conference for High Perfor-*

mance Computing, Networking, Storage and Analysis, Atlanta, GA, USA: IEEE, Nov. 2020, pp. 1–14.

[58] Y. Hu *et al.*, “FeatGraph: A flexible and efficient backend for graph neural network systems”, in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, Atlanta, GA, USA: IEEE, Nov. 2020, pp. 1–13.

[59] G. Huang, G. Dai, Y. Wang, and H. Yang, “GE-SpMM: General-purpose sparse matrix-matrix multiplication on GPUs for graph neural networks”, in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, Atlanta, GA, USA: IEEE, Nov. 2020, pp. 1–12.

[60] M. F. Balin, K. Sancak, and Ü. V. Çatalyürek, “MG-GCN: A scalable multi-GPU GCN training framework”, in *Proceedings of the 51st International Conference on Parallel Processing*, ACM, Aug. 2023, pp. 1–11.

[61] E. Elsen, M. Dukhan, T. Gale, and K. Simonyan, “Fast sparse ConvNets”, in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, Jun. 2020, pp. 14 617–14 626.

[62] T. Gale, M. Zaharia, C. Young, and E. Elsen, “Sparse GPU kernels for deep learning”, in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, Atlanta, GA, USA: IEEE, Nov. 2020, pp. 1–14.

[63] G. Karypis and V. Kumar, “A fast and high quality multilevel scheme for partitioning irregular graphs”, *SIAM Journal on Scientific Computing*, vol. 20, no. 1, pp. 359–392, 1998.

[64] Ü. V. Çatalyürek and C. Aykanat, “Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication”, *IEEE Transactions on Parallel and Distributed Systems*, vol. 10, no. 7, pp. 673–693, 1999.

[65] Ü. V. Çatalyürek, “A fine-grain hypergraph model for 2D decomposition of sparse matrices”, in *Proceedings 15th International Parallel and Distributed Processing Symposium. IPDPS 2001*, San Francisco, CA, USA: IEEE, Apr. 2001, pp. 1199–1204.

[66] Ü. V. Çatalyürek, C. Aykanat, and B. Uçar, “On two-dimensional sparse matrix partitioning: Models, methods, and a recipe”, *SIAM Journal on Scientific Computing*, vol. 32, no. 2, pp. 656–683, 2010.

[67] B. Vastenhouw and R. H. Bisseling, “A two-dimensional data distribution method for parallel sparse matrix-vector multiplication”, *SIAM Review*, vol. 47, no. 1, pp. 67–95, 2005.

[68] A.-J. N. Yzelman and R. H. Bisseling, “Cache-oblivious sparse matrix-vector multiplication by using sparse matrix partitioning methods”, *SIAM Journal on Scientific Computing*, vol. 31, no. 4, pp. 3128–3154, 2009.

[69] A. Monakov, A. Lokhmotov, and A. Avetisyan, “Automatically tuning sparse matrix-vector multiplication for GPU architectures”, in *High Performance Embedded Architectures and Compilers*, Berlin, Heidelberg: Springer Berlin Heidelberg, Jan. 2010, pp. 111–125.

[70] X. Liu, M. Smelyanskiy, E. Chow, and P. Dubey, “Efficient sparse matrix-vector multiplication on x86-based many-core processors”, in *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, Eugene, Oregon, USA: ACM, Jun. 2013, pp. 273–282.

[71] M. Kreutzer, G. Hager, G. Wellein, H. Fehske, and A. R. Bishop, “A unified sparse matrix data format for efficient general sparse matrix-vector multiplication on modern processors with wide SIMD units”, *SIAM Journal on Scientific Computing*, vol. 36, no. 5, pp. C401–C423, 2014.

[72] W. Liu and B. Vinter, “CSR5: An efficient storage format for cross-platform sparse matrix-vector multiplication”, in *Proceedings of the 29th ACM on International Conference on Supercomputing*, Newport Beach, California, USA: ACM, Jun. 2015, pp. 339–350.

[73] D. Merrill and M. Garland, “Merge-based parallel sparse matrix-vector multiplication”, in *SC ’16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, Salt Lake City, UT, USA, Nov. 2016, pp. 678–689.

[74] J. L. Greathouse and M. Daga, “Efficient sparse matrix-vector multiplication on GPUs using the CSR storage format”, in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, New Orleans, LA, USA: ACM, Nov. 2014, pp. 769–780.

[75] A. Ashari, N. Sedaghati, J. Eisenlohr, S. Parthasarath, and P. Sadayappan, “Fast sparse matrix-vector multiplication on GPUs for graph applications”, in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, New Orleans, LA, USA: ACM, Nov. 2014, pp. 781–792.

[76] K. Cheshmi, S. Kamil, M. M. Strout, and M. M. Dehnavi, "Sympiler: Transforming sparse matrix codes by decoupling symbolic analysis", in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '17, Denver, Colorado: ACM, Nov. 2017.

[77] M. M. Strout, M. Hall, and C. Olschanowsky, "The sparse polyhedral framework: Composing compiler-generated inspector-executor code", *Proceedings of the IEEE*, vol. 106, no. 11, pp. 1921–1934, 2018.

[78] K. Cheshmi, Z. Cetinic, and M. M. Dehnavi, "Vectorizing sparse matrix computations with partially-strided codelets", in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '22, Dallas, Texas: IEEE, Nov. 2022.

[79] A. Bienz, W. D. Gropp, and L. N. Olson, "Node aware sparse matrix-vector multiplication", *Journal of Parallel and Distributed Computing*, vol. 130, pp. 166–178, 2019.

[80] H. M. Aktulga, A. Buluç, S. Williams, and C. Yang, "Optimizing sparse matrix-multiple vectors multiplication for nuclear configuration interaction calculations", in *2014 IEEE International Symposium on Parallel and Distributed Processing*, Phoenix, AZ, USA: IEEE, May 2014, pp. 1213–1222.

[81] S. E. Kurt, A. Sukumaran-Rajam, F. Rastello, and P. Sadayyapan, "Efficient tiled sparse matrix multiplication through matrix signatures", in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, Atlanta, GA, USA: IEEE, Nov. 2020, pp. 1–14.

[82] C. Hong *et al.*, "Efficient sparse-matrix multi-vector product on GPUs", in *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*, Tempe, AZ, USA: ACM, Jun. 2018, pp. 66–79.

[83] C. Yang, A. Buluç, and J. D. Owens, "Design principles for sparse matrix multiplication on the gpu", in *Euro-Par 2018 Parallel Processing: 24th International Conference on Parallel and Distributed Computing*, Turin, Italy: Springer-Verlag, Aug. 2018, pp. 672–687.

[84] P. Jiang, C. Hong, and G. Agrawal, "A novel data transformation and execution strategy for accelerating sparse matrix multiplication on GPUs", in *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, San Diego, CA, USA: ACM, Feb. 2020, pp. 376–388.

[85] P. Koanantakool *et al.*, "Communication-avoiding parallel sparse-dense matrix-matrix multiplication", in *2016 IEEE International Parallel and Distributed Processing Symposium*, Chicago, IL, USA: IEEE, May 2016, pp. 842–853.

[86] O. Selvitopi, B. Brock, I. Nisa, A. Tripathy, K. Yelick, and A. Buluç, "Distributed-memory parallel algorithms for sparse times tall-skinny-dense matrix multiplication", in *Proceedings of the ACM International Conference on Supercomputing*, New York, NY, USA: ACM, Jun. 2021, pp. 431–442.

[87] S. Acer, O. Selvitopi, and C. Aykanat, "Improving performance of sparse matrix dense matrix multiplication on large-scale parallel systems", *Parallel Computing*, vol. 59, no. C, Nov. 2016.

[88] L. Gianinazzi *et al.*, "Arrow matrix decomposition: A novel approach for communication-efficient sparse matrix multiplication", in *Proceedings of the 29th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, Edinburgh, United Kingdom: ACM, Feb. 2024, pp. 404–416.

[89] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith, "Efficient management of parallelism in object oriented numerical software libraries", in *Modern Software Tools in Scientific Computing*, E. Arge, A. M. Bruaset, and H. P. Langtangen, Eds., Birkhäuser Press, 1997, pp. 163–202.

[90] S. Balay *et al.*, "PETSc/TAO users manual", Argonne National Laboratory, Tech. Rep. ANL-21/39 - Revision 3.18, 2022.

[91] J. Zhang *et al.*, "The PetscSF scalable communication layer", *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 4, pp. 842–853, 2022.

[92] T. Lengauer, *Combinatorial Algorithms for Integrated Circuit Layout*. Chichester, U.K.: Wiley, 1990.

[93] E. Cuthill and J. McKee, "Reducing the bandwidth of sparse symmetric matrices", in *Proceedings of the 1969 24th National Conference*, ACM, 1969, pp. 157–172.

[94] L. Oliker, X. Li, P. Husbands, and R. Biswas, "Effects of ordering strategies and programming paradigms on sparse matrix computations", *SIAM Review*, vol. 44, no. 3, pp. 373–393, 2002.

[95] J. C. Pichel, F. F. Rivera, M. Fernández, and A. Rodríguez, "Optimization of sparse matrix-vector multiplication using reordering techniques on GPUs", *Microprocessors and Microsystems*, vol. 36, no. 2, pp. 65–77, 2012.

[96] J. D. Trotter *et al.*, "Bringing order to sparsity: A sparse matrix reordering study on multicore CPUs", in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '23, Denver, CO, USA: Association for Computing Machinery, Nov. 2023.

[97] T. A. Davis and Y. Hu, "The University of Florida sparse matrix collection", *ACM Transactions on Mathematical Software*, vol. 38, no. 1, Dec. 2011.

[98] A. Azad, O. Selvitopi, M. T. Hussain, J. Gilbert, and A. Buluç, "Combinatorial BLAS 2.0: Scaling combinatorial algorithms on distributed-memory systems", *IEEE Transactions on Parallel and Distributed Systems*, 2021.

[99] W. Hackbusch, "A sparse matrix arithmetic based on $\mathcal{H}$-matrices. Part I: Introduction to $\mathcal{H}$-matrices", *Computing*, vol. 62, no. 2, pp. 89–108, 1999.

[100] W. Hackbusch and B. N. Khoromskij, "A sparse $\mathcal{H}$-matrix arithmetic. Part II: Application to multi-dimensional problems", *Computing*, vol. 64, no. 1, pp. 21–47, 2000.

[101] W. Hackbusch, B. N. Khoromskij, and S. A. Sauter, "On $\mathcal{H}^2$-matrices", in *Lectures on Applied Mathematics: Proceedings of the Symposium Organized by the Sonderforschungsbereich 438 on the occasion of Karl-Heinz Hoffmann's 60th birthday, Munich, June 30 - July 1, 1999*, Berlin: Springer, 2000, pp. 9–29.

[102] W. Hackbusch and S. Börm, "Data-sparse approximation by adaptive $\mathcal{H}^2$-matrices", *Computing*, vol. 69, no. 1, pp. 1–35, 2002.

[103] S. Chandrasekaran, M. Gu, and T. P. Pals, "A fast ULV decomposition solver for hierarchically semiseparable representations", *SIAM Journal on Matrix Analysis and Applications*, vol. 28, no. 3, pp. 603–622, 2006.

[104] M. Bebendorf and S. Rjasanow, "Adaptive low-rank approximation of collocation matrices", *Computing*, vol. 70, no. 1, pp. 1–24, 2003.

[105] L. F. Greengard and V. Rokhlin, "A fast algorithm for particle simulations", *Journal of Computational Physics*, vol. 73, no. 2, pp. 325–348, 1987.

[106] L. F. Greengard and V. Rokhlin, "A new version of the fast multipole method for the Laplace equation in three dimensions", *Acta Numerica*, vol. 6, pp. 229–269, 1997.

[107] L. Ying, G. Biros, and D. Zorin, "A kernel-independent adaptive fast multipole algorithm in two and three dimensions", *Journal of Computational Physics*, vol. 196, no. 2, pp. 591–626, 2004.

[108] W. Fong and E. Darve, "The black-box fast multipole method", *Journal of Computational Physics*, vol. 228, no. 23, pp. 8712–8725, 2009.

[109] X. Xing and E. Chow, “Interpolative decomposition via proxy points for kernel matrices”, *SIAM Journal on Matrix Analysis and Applications*, vol. 41, pp. 221–243, 2020.

[110] P.-G. Martinsson and V. Rokhlin, “A fast direct solver for boundary integral equations in two dimensions”, *Journal of Computational Physics*, vol. 205, no. 1, pp. 1–23, 2005.

[111] M. Gu and S. C. Eisenstat, “Efficient algorithms for computing a strong rank-revealing QR factorization”, *SIAM Journal on Scientific Computing*, vol. 17, no. 4, pp. 848–869, 1996.

[112] E. Corona, P.-G. Martinsson, and D. Zorin, “An $O(N)$ direct solver for integral equations on the plane”, *Applied and Computational Harmonic Analysis*, vol. 38, no. 2, pp. 284–317, 2015.

[113] V. Minden, A. Damle, K. L. Ho, and L. Ying, “Fast spatial Gaussian process maximum likelihood estimation via skeletonization factorizations”, *Multiscale Modeling & Simulation*, vol. 15, no. 4, pp. 1584–1611, 2017.

[114] S. Börm and L. Grasedyck, “Hybrid cross approximation of integral operators”, *Numerische Mathematik*, vol. 101, no. 2, pp. 221–249, 2005.

[115] M. Bebendorf and S. Kunis, “Recompression techniques for adaptive cross approximation”, *The Journal of Integral Equations and Applications*, vol. 21, no. 3, pp. 331–357, 2009.

[116] D. Cai, E. Chow, L. Erlandson, Y. Saad, and Y. Xi, “SMASH: Structured matrix approximation by separation and hierarchy”, *Numerical Linear Algebra with Applications*, vol. 25, no. 6, e2204, 2018.

[117] P.-G. Martinsson and V. Rokhlin, “An accelerated kernel-independent fast multipole method in one dimension”, *SIAM Journal on Scientific Computing*, vol. 29, no. 3, pp. 1160–1178, 2007.

[118] D. Malhotra and G. Biros, “PVFMM: A parallel kernel independent fmm for particle and volume potentials”, *Communications in Computational Physics*, vol. 18, no. 3, pp. 808–830, 2015.

[119] Z. Gimbutas, L. Greengard, J. Magland, M. Rachh, and V. Rokhlin, *FMM3D*, 2019.

[120] L. F. Greengard and J. Huang, “A new version of the fast multipole method for screened Coulomb interactions in three dimensions”, *J. Comput. Phys.*, vol. 180, no. 2, pp. 642–658, 2002.

[121] R. Wang, *BBFMM3D*, 2018.

[122] S. Börm, *H2Lib*, 2017.

[123] P. Ghysels, X. S. Li, F.-H. Rouet, S. Williams, and A. Napov, "An efficient multicore implementation of a novel HSS-structured multifrontal solver using randomized sampling", *SIAM Journal on Scientific Computing*, vol. 38, no. 5, S358–S384, 2016.

[124] F.-H. Rouet, X. S. Li, P. Ghysels, and A. Napov, "A distributed-memory package for dense hierarchically semi-separable matrix computations using randomization", *ACM Trans. Math. Softw.*, vol. 42, no. 4, 27:1–27:35, 2016.

[125] W. Boukaram, G. Turkiyyah, and D. Keyes, "Hierarchical matrix operations on GPUs: Matrix-vector multiplication and compression", *ACM Transaction on Mathematical Software*, vol. 45, no. 1, 3:1–3:28, 2019.

[126] H. Cheng, Z. Gimbutas, P.-G. Martinsson, and V. Rokhlin, "On the compression of low rank matrices", *SIAM Journal on Scientific Computing*, vol. 26, no. 4, pp. 1389–1404, 2005.

[127] X. Xing, "The proxy point method for rank-structured matrices", Ph.D. dissertation, Georgia Institute of Technology, 2019.

[128] N. Halko, P. Martinsson, and J. Tropp, "Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions", *SIAM Review*, vol. 53, no. 2, pp. 217–288, 2011.

[129] K. Nitadori, J. Makino, and P. Hut, "Performance tuning of N-body codes on modern microprocessors: I. Direct integration with a hermite scheme on x86_64 architecture", *New Astronomy*, vol. 12, no. 3, pp. 169–181, 2006.

[130] A. Chandramowlishwaran, S. Williams, L. Oliker, I. Lashuk, G. Biros, and R. Vuduc, "Optimizing and tuning the fast multipole method for state-of-the-art multicore architectures", in *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, 2010, pp. 1–12.