The Dissertation Committee for Chen-Han Yu
certifies that this is the approved version of the following dissertation:

The Science of High Performance Algorithms for Hierarchical Matrices

Committee:

George Biros, Supervisor

Robert A. van de Geijn, Co-supervisor

Don Batory

Georgios-Alex Dimakis

Per-Gunnar Martinsson

The Science of High Performance Algorithms for Hierarchical Matrices

by

Chen-Han Yu

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

August 2018

Dedicated to my beloved family.

Acknowledgments

This dissertation is based upon work supported by 30-2122-5870 a gift from Qualcomm; by AFOSR grant FA9550-17-1-0190; by NSF grants CCF-1337393 and SSI-1550493; by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Applied Mathematics program under Award Numbers DE-SC0010518 and DE-SC0009286; by NIH grant 10042242; by DARPA grant W911NF-115-2-0121; and by the Technische Universität München—Institute for Advanced Study, funded by the German Excellence Initiative (and the European Union Seventh Framework Program under grant agreement 291763), as well as the German Academic Exchange Service (DAAD). Any opinions, findings, and conclusions or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the Qualcomm, AFOSR, DOE, NIH, DARPA, and NSF. Computing time on the Texas Advanced Computing Centers Stampede system was provided by an allocation from TACC and the NSF.

> *Whatever you do, or dream you can do, begin it. Boldness has genius and power and magic in it.*
>
> —Johann Wolfgang von Goethe

I may be the person who began the journey, but it is these amazing people who gave me the power to finish it. I would like to start by presenting you these fantastic

people who have contributed their love, input, feedback, and put their faith in me during my Ph.D. period.

In 2012, my last year at National Taiwan University, I decided to pursue my Ph.D. degree in Computer Science after receiving my master degree in Applied Mathematics. Austin is about 8,000 miles away from my hometown Taipei. I would like to say thank you first to my beloved family and my adviser Prof. Weichung Wang who help me prepare the new journey and give me their blessing.

I am glad that I started my new life in Prof. Robert van de Geijn's research group. Although Robert always told me that he would like to retire, he has been the most supportive co-supervisor in the past five years. We have a group (`FLAME` and `SHPC`) of active and wonderful people here at the south corner of the Gates Dell Complex (GDC) working on high-performance linear algebra and tensor computation. With Dr. Jianyu Huang, Woody Austin, Field Van Zee, and Dr. Tyler Smith's help, I was able to extend the BLAS-like Library Instantiation Software Framework (`BLIS`) to N-body operator such as κ-nearest neighbor search.

> *Why this topic? The magical $\mathcal{O}(N)$ and $\mathcal{O}(N \log N)$ complexity of $\mathcal{H}$-matrix methods have irresistible power.*

I still recall how surprised I was when Prof. George Biros presented Fast Multiple Methods (`FMM`), a class of hierarchical matrix ($\mathcal{H}$-matrix) methods, in my first year of the graduate studies. Despite later I found that *it is anything but trivial to achieve this magical complexity for a dense N-by-N matrix-vector multiplication*, still I enjoy my research and the time spent with George. In addition to efforts associated with my research, I am fortunate to have George as my mentor. George not only guided me throughout my career as a graduate student, but he also taught me how to convey my ideas through writing. It is only through his constant encouragement and pressure to do better than I have been able to achieve all that I have.

George runs a wonderful laboratory `PADAS` at Peter O'Donnell Jr Building

(POB). I had the honor to work with the best people in the field of parallel algorithms and supercomputing. I want to extend my sincerest thanks to Dr. William March, Dr. Bo Xiao, Severin Reiz, James Levitt, and Dr. Dhairya Malhotra, who dedicated their time to work with me. I learn a lot from you guys.

Finally, I cannot go without acknowledging everyone I consider family. In particular, I would like to thank my significant other Yanyi Song, little brother Chen-Hsuan Yu, my parents Shih-Kuang Yu and Chao-Jung Lee. Thank you for sharing my best moment of this work and providing me undivided support when I needed the most. I am glad that I can finally repay them for everything they have done for me during this long process. For those who are still around and for those who unfortunately have to go, from the bottom of my heart, thank you all.

Chen-Han Yu

The University of Texas at Austin
August 2018

The Science of High Performance Algorithms for Hierarchical Matrices

Publication No. _____

Chen-Han Yu, Ph.D.
The University of Texas at Austin, 2018

Supervisor: George Biros
Co-Supervisor: Robert A. van de Geijn

Many matrices in scientific computing, statistical inference, and machine learning exhibit sparse and low-rank structure. Typically, such structure is exposed by appropriate matrix permutation of rows and columns, and exploited by constructing an hierarchical approximation. That is, the matrix can be written as a summation of sparse and low-rank matrices and this structure repeats recursively. Matrices that admit such hierarchical approximation are known as **hierarchical matrices** ($\mathcal{H}$-matrices in brief). $\mathcal{H}$-matrix approximation methods are more general and scalable than solely using a sparse or low-rank matrix approximation. Classical numerical linear algebra operations on $\mathcal{H}$-matrices—multiplication, factorization, and eigenvalue

decomposition—can be accelerated by many orders of magnitude.

Although the literature on $\mathcal{H}$-matrices for problems in computational physics (low-dimensions) is vast, there is less work for generalization and problems appearing in machine learning. Also, there is limited work on high-performance computing algorithms for pure algebraic $\mathcal{H}$-matrix methods. This dissertation tries to address these open problems on building hierarchical approximation for kernel matrices and generic symmetric positive definite (SPD) matrices. We propose a general tree-based framework (`GOFMM`) for appropriately permuting a matrix to expose its hierarchical structure. `GOFMM` supports both static and dynamic scheduling, shared memory and distributed memory architectures, and hardware accelerators. The supported algorithms include kernel methods, approximate matrix multiplication and factorization for large sparse and dense matrices.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

This document[1] aims to extend the *science* of high-performance algorithms for hierarchical matrices (**$\mathcal{H}$-matrices**). We present a systematic study on designing Geometry Oblivious Fast Multiple Method (`GOFMM`) [82] and its distributed-memory variant `MPI-GOFMM`, which are novel parallel frameworks for creating $\mathcal{H}$-matrix approximation of dense symmetric positive definite (SPD) matrices.

`GOFMM` can compress an N-by-N SPD matrix with $\mathcal{O}(N \log N)$ work. Once

[1] Several of my prior publications contribute to this document. In [82] (Chenhan D. Yu, James Levitt, Severin Reiz, and George Biros. Geometry-oblivious FMM for compressing dense SPD matrices.), I introduced `GOFMM` and $\mathcal{H}$-matrix schemes that can be used to compress arbitrary SPD matrices. In [84] (Chenhan D. Yu, William B. March, Bo Xiao, and George Biros. `INV-ASKIT`: a parallel fast direct solver for kernel matrices.), I introduced `INV-ASKIT` an $\mathcal{O}(N \log^2 N)$ direct solver for kernel matrices. In [83] (Chenhan D. Yu, William B March, and George Biros. An N log N parallel fast direct solver for kernel matrices.), I introduced an $\mathcal{O}(N \log N)$ hybrid direct-iterative solver for kernel matrices. In [81] (Chenhan D. Yu, Jianyu Huang, Woody Austin, Bo Xiao, and George Biros. Performance optimization for the κ-nearest neighbors kernel on x86 architectures.), I introduced an efficient algorithm for exhausted κ-nearest neighbor search on modern CPUs. By fusing neighbor search with GEMM in the assembly level, the proposed method is free from extra working space and suffer from smaller memory latency. In [55] (William B. March, Bo Xiao, Chenhan D. Yu, and George Biros. An algebraic parallel treecode in arbitrary dimensions.), In introduced an efficient algorithm for kernel summation on modern GPUs. This further shows that fusing additional operations with GEMM does not compromise its cache behavior, and such optimization can be applied to other N-body primitives. In [54] (William B. March, Bo Xiao, Sameer Tharakan, Chenhan D. Yu, and George Biros. Robust treecode approximation for kernel machines.), I conducted experiments for kernel ridge regression using `ASKIT` and apply efficient neighbor search and kernel summation kernels to reduce the training and inference time. In [53] (William B. March, Bo Xiao, Sameer Tharakan, Chenhan D. Yu, and George Biros. A kernel-independent FMM in general dimensions.), I introduce a hybrid algorithm that efficiently schedules kernel summation tasks to available CPUs and GPUs. In [56] (William B. March, Bo Xiao, Chenhan D. Yu, and George Biros. `ASKIT`: An efficient, parallel library for high-dimensional kernel summations.), I design and conduct experiments to empirically analyze the performance of `ASKIT`.

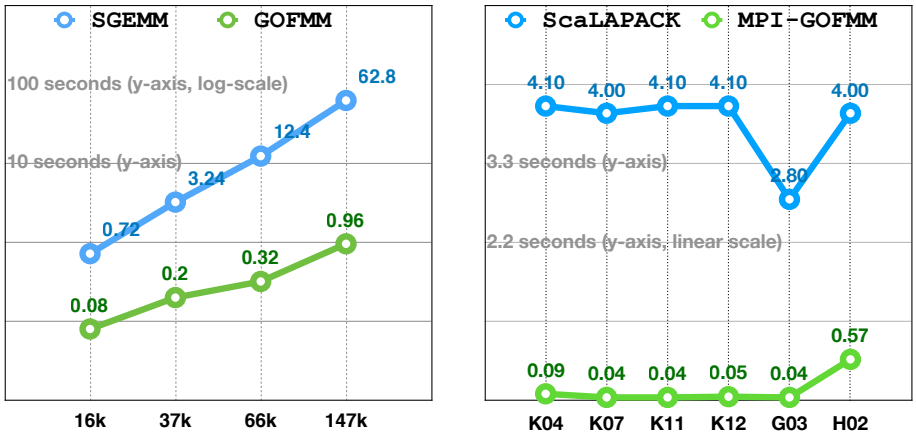

Figure 1.1: The left figure compares the runtime of `GOFMM` (in green) with `MKL` `SGEMM` on two 12-core Intel Xeon E5-2690 v3 "Haswell" processors, where **y-axis** denotes runtime in seconds (log-scale) and **x-axis** denotes matrix size N. Each `SGEMM` computes $K \times w$ ($K \in \mathbb{R}^{N \times N}$, $w \in \mathbb{R}^{N \times 2048}$), where K is derived from a constrained PDE optimization problem. `GOFMM` compresses K such that $\widetilde{K} \times w$ can be computed in $\mathcal{O}(N \log N)$ work and achieve accuracy of 1E-2 to 4E-4 (relative error) in single precision. The right figure compares `GOFMM` with `ScaLAPACK` with four `MPI` processes. Each CPU node (process) has two 24-core Intel Xeon Platinum 8160 "Skylake" processors. The **y-axis** denotes runtime in seconds (linear-scale) and **x-axis** denotes different SPD matrices (see Section 4.3 for details).

compressed, matrix-vector multiplication and solution of linear system can be computed with $\mathcal{O}(N \log N)$ or even $\mathcal{O}(N)$ work [82, 83]. As an example, in Figure 1.1 we report timings for r matrix-vector multiplications. The blue plots denote the runtime of an N-by-N matrix times an N-by-r matrix with $\mathcal{O}(rN^2)$ work, using highly optimized dense matrix libraries: Basic Linear Algebra Subprograms (`BLAS`) and Scalable Linear Algebra Package (`ScaLAPACK`). To be precise, the matrix multiplication calls `SGEMM` (general matrix-matrix multiplication in single precision), which is part of Intel's Math Kernel Library (`MKL`). The green plots denote our $\mathcal{O}(rN \log N)$ fast multiplication using `GOFMM`. $\mathcal{H}$-matrix multiplication grants up to 60× speedup (left figure) on a shared-memory multi-core CPU and up to 100× speedup (right figure) on a distributed-memory CPU cluster.

1.1 Hierarchical Matrices

Informally, we say that a matrix $\widetilde{K}$ admits a good **hierarchically low-rank plus sparse structure**, i.e., $\widetilde{K}$ is an $\mathcal{H}$-matrix [34, 9], if

$$\widetilde{K} = D + S + UV, \tag{1.1}$$

where D is **block-diagonal** with **every block being an $\mathcal{H}$-matrix**, U and V are **low-rank** (see Section 1.5 for an explanation), and S is **sparse** (with a few non-zero entries). To be precise, matrix $\widetilde{K}$ is partitioned (with respect to row and column indices) by a binary tree[2] recursively such that

$$\widetilde{K}_{\alpha\alpha} = \begin{bmatrix} \widetilde{K}_{\mathtt{ll}} & 0 \\ 0 & \widetilde{K}_{\mathtt{rr}} \end{bmatrix} + \begin{bmatrix} 0 & S_{\mathtt{lr}} \\ S_{\mathtt{rl}} & 0 \end{bmatrix} + \begin{bmatrix} 0 & U_{\mathtt{l}}V_{\mathtt{r}} \\ U_{\mathtt{r}}V_{\mathtt{l}} & 0 \end{bmatrix}, \tag{1.2}$$

where `l` and `r` are **left** and **right** children of tree node α. Each node α contains a set of matrix column indices (or row indices due to symmetry), and the two children evenly split this set such that $\alpha = \mathtt{l} \cup \mathtt{r}$. *This recursive partitioning can be performed at most* $\log_2(N)$ *times, generating* $\mathcal{O}(N)$ *leaf nodes (diagonal blocks). Combining these two terms together gives a taste of the* $\mathcal{O}(N \log N)$ *complexity of the fast matrix-vector multiplication we present in Figure 1.1.* Although such matrices are rare in real-world applications, it is quite common to find matrices that can be approximated arbitrarily well by an $\mathcal{H}$-matrix.

It is easy to observe the complexity saving in the derivation above, but there are many questions that one must answer in order to develop and generalize efficient algorithms for $\mathcal{H}$-matrix based approximations. For example, one must first exploit the order of the symmetric row and column permutation, because the $\mathcal{H}$-matrix structure is *not invariant* to the matrix permutation.

[2] We overload notation α, β, `l` and `r` to denote tree nodes and the matrix indices those tree nodes own.

1.2 Problem Statement

Let input $K \in \mathbb{R}^{N \times N}$ be a Symmetric Positive Definite (SPD) matrix, with

$$K = K^T,\ w^T K w > 0,\ \forall w \in \mathbb{R}^N,\ \text{and } w \neq 0. \tag{1.3}$$

Alternatively, we also consider K a kernel matrix, where the entries of K can be evaluated with a specific kernel function $\mathcal{K}$. For such matrices, the input is not a matrix, but only the points $\{x_i\}_{i=1}^N$ so that the entry K_{ij} can be computed as $\mathcal{K}(x_i, x_j)$ on the fly. Examples of kernel functions are radial basis functions, Green's functions, and angle similarity functions. Since matrix K can be dense, it requires $\mathcal{O}(N^2)$ storage, $\mathcal{O}(N^2)$ work for a matrix-vector multiplication (`MATVEC`), and $\mathcal{O}(N^3)$ work for a direct linear solver (`SOLVE`).

Given p parallel processes and assuming the evaluation of a single matrix entry K_{ij} via user-defined function $K(i, j)$ requires $\mathcal{O}(1)$ work, we wish to construct an $\mathcal{H}$-matrix $\widetilde{K}$ in the form of Equation 1.2 with the following properties:

- constructing $\widetilde{K}$ requires $\mathcal{O}(N \log N)$ work and $\mathcal{O}(\frac{N}{p} \log N)$ time in parallel;
- a `MATVEC` with $\widetilde{K}$ requires $\mathcal{O}(N \log N)$ work and $\mathcal{O}(\frac{N}{p} \log N)$ time in parallel;
- an `SOLVE` with $(\lambda I + \widetilde{K})$ also requires $\mathcal{O}(\frac{N}{p} \log N)$ time in parallel; and
- $\|\widetilde{K} - K\| \leq \epsilon \|K\|$, where $0 < \epsilon < 1$ is a user-defined error tolerance.

In other words, given any SPD matrix K, our task is to construct an $\mathcal{H}$-matrix compression $\widetilde{K}$ in parallel such that the relative error $\|K - \widetilde{K}\|/\|K\|$ is small. The cost of constructing $\widetilde{K}$ and performing linear algebra operations with $\widetilde{K}$ must take less than $\mathcal{O}(N \log N)$ work and $\mathcal{O}(\frac{N}{p} \log N)$ time in parallel.

Beyond satisfying the properties above, we further want to design algorithms that achieve portable high performance on different computing platforms including

non-uniform memory access (NUMA) multi-core, many-core (e.g. GPUs, Intel Xeon Phi), heterogeneous architectures, and distributed-memory environments.

1.3 Significance and Challenges

Dense SPD and kernel matrices appear in scientific computing, statistical inference, machine learning, and data analytics. They appear in Cholesky and LU factorizations [29], in Schur complement matrices for saddle point problems [11], in Hessian operators in optimization [62], in kernel methods for statistical learning [39, 30], and in N-body methods and integral equations [32, 34].

These applications usually require (1) row-wise reduction (or selection) of matrix K, (2) solving linear systems with K, and (3) solving for eigenpairs of K. However, these operations can be prohibitively expensive for large matrix size N, since K can be **dense** (or become dense as the computation progresses). This *complexity barrier* has limited the use of related methods for large-scale problems [51, 23]. Consequently, one must settle on approximation.

Why $\mathcal{H}$-matrix approximation? With points and kernel functions, matrix approximation can be reasoned according to the distribution of the points and the characteristic of the kernel function. For example, consider the Gaussian kernel,

$$\mathcal{K}(x_i, x_j) = \exp\left(-\frac{1}{2}\frac{\|x_i - x_j\|_2^2}{h^2}\right), \tag{1.4}$$

where h is the *kernel bandwidth*. For small h, matrix K approaches the identity matrix whereas, for large h, matrix K approaches the rank-one constant matrix (with every entry equal to one). The first regime suggests sparse approximations while the second regime suggests global low-rank approximations. But for the majority of h values, K is neither sparse nor globally low-rank. An *hierarchical representation in Equation 1.1—combining both sparse and low-rank approximation—is more robust*

than solely using either one of the approximation.

Three challenges: There are three challenges one must address first before constructing any $\mathcal{H}$-matrix approximation. One important observation is that *hierarchical low-rank structure is not invariant to row and column permutations.* Therefore any algorithm for constructing approximation $\widetilde{K}$ must

- appropriately permute matrix K to expose the low-rank structure UV and sparsity S in the off-diagonal blocks of Equation 1.2; and
- be able to discover sparsity S from a possibly dense matrix with less than $\mathcal{O}(N \log N)$ work; and
- must take $\mathcal{O}(N \log N)$ work to compute the low-rank decomposition UV.

Recall that matrix K has N^2 entries, and our goal is to construct an approximation $\widetilde{K}$ in $\mathcal{O}(N \log N)$ work. Thus, *necessary guidance in each step is important to avoid visiting all N^2 entries.*

The need for geometry (spatial) information: Existing algorithms rely on the matrix entries K_{ij} being "interactions" (e.g. kernel functions) between **(data) points** $\{x_i\}_{i=1}^N$ in $\mathbb{R}^d$ and permute K either by clustering the points (typically using some tree data-structure in the form of Equation 1.2) or by using graph partitioning techniques (if K is sparse). Furthermore, the construction of the sparse correction S uses the **nearest-neighbor** structure of the input points. The low-rank matrices U, V can be either analytically computed using expansions of the kernel function, or semi-algebraically computed using fictitious points (or equivalent points) or using algebraic sampling-based methods that use geometric information. *In a nutshell, geometric information is used in all aspects of an $\mathcal{H}$-matrix method.*

Challenges in generalization: In many cases, however, such points and kernel functions are not available. For example, dense graph Laplacian operators and their inverses in data analysis (e.g., social networks, protein interactions). Additional

examples include frontal matrices and Schur complements in the factorization of sparse matrices; Hessian operators in optimization; and kernel methods in machine learning without points (e.g., word sequences and diffusion on graphs [14, 47]). *How to generalize $\mathcal{H}$-matrix methods to arbitrary SPD matrices without knowing the data points and kernel functions (geometric-oblivious) is the greatest innovation of this dissertation.*

Challenges in parallelization: The key components of $\mathcal{H}$-matrix methods are tree traversals, which inherently exhibit parallelism diminishing and load-balancing issues. As a result, parallelism must be fully exploited in multiple levels and in different granularities to be scalable. $\mathcal{H}$-matrix also involves sparse matrix computation, where the sparsity is not known until runtime. Exchanging sparse data with unknown sparsity makes parallelization in the distributed-memory environment even more difficult. *How to systematically and fully exploit the parallelism of $\mathcal{H}$-matrix methods is one of the main topic discussed in this dissertation.*

1.4 Toward a Solution

We present a systematic study on designing `GOFMM`, novel parallel algorithms for creating $\mathcal{H}$-matrix approximation of generic SPD matrices. `GOFMM` is inspired by the rich literature of algorithms for matrix sketching, $\mathcal{H}$-matrices, and fast multipole methods (`FMM`). In the following, we briefly illustrate the roadmap toward a solution for high-performance $\mathcal{H}$-matrix algorithms. We summarize the established works in Section 1.5. Innovation and contribution are summarized in Section 1.6. A thorough review of related work can be found in Chapter 5.

Established works on $\mathcal{H}$-matrix approximation: `GOFMM` extends our previous work Algebraic Skeletonization Kernel Independent Treecode (`ASKIT`) [56,

55, 54, 53]. Given tolerance ϵ, `ASKIT` constructs an $\mathcal{H}$-matrix approximation

$$\widetilde{K}_{\alpha\alpha} = \begin{bmatrix} \widetilde{K}_{\mathtt{ll}} & 0 \\ 0 & \widetilde{K}_{\mathtt{rr}} \end{bmatrix} + \begin{bmatrix} 0 & S_{\mathtt{lr}} \\ S_{\mathtt{rl}} & 0 \end{bmatrix} + \begin{bmatrix} 0 & UV_{\mathtt{lr}} \\ UV_{\mathtt{rl}} & 0 \end{bmatrix} \tag{1.5}$$

for generic kernel matrices, where data points $\{x_i\}_{i=1}^N$ and kernel function $\mathcal{K}$ are required as input. To address the three challenges of $\mathcal{H}$-matrix approximation, `ASKIT` makes use of a spatial ball tree [63] to determine the matrix partitioning, recursive Interpolative Decomposition (`ID`) [36] to approximate UV, and the nearest-neighbor of the data points to approximate S as well as improving the sampling quality while constructing the UV factor. The previous work [54] shows that $\mathcal{H}$-matrix approximation is more robust than the globally low-rank approximation (e.g. Nystrom methods [75, 28, 36]). As a result, [54] yields better classification accuracy in kernel ridge regression tasks.

Limitations of `ASKIT`: As first of its kind, `ASKIT` has several limitations. For example, `ASKIT` does not fully exploit the parallelism at different granularities. As a result, `ASKIT` often suffers from diminishing parallelism during tree traversals. `ASKIT` uses synchronous collective communication routines such as `Alltoallv()`, which inherently introduces unnecessary global barriers. `ASKIT` also does not contain any architecture-dependent design to fully exploit the underlying architecture.

Solution: `GOFMM` introduces geometry-oblivious distances (Section 2.4) to further generalize `ASKIT` to generic SPD matrices. Geometry-oblivious distance d_{ij} is well defined on SPD matrices for arbitrary row index i and column index j. It can be used in matrix partition and neighbor search to replace Euclidean distances. We discuss the effectiveness of `GOFMM` in Section 2.4.

`GOFMM` systematically exploits the parallelism of the whole algorithms by building a dependency graph of tasks that composes several tree traversals at runtime. By **"task"** we refer to a computation that occurs when we visit a tree node during a

traversal. With a dependency graph, scheduling can be done in *static* or *dynamic* fashion (Section 1.5). We implement a light-weight runtime system with dynamic Heterogeneous Earliest Finish Time (HEFT) [71] policy using `OpenMP` threads. Each worker (thread) in the runtime system can use more than one physical core with either a nested `OpenMP` constructing, or by employing a device (accelerator) as a slave. Distributed data dependencies (i.e. message passing and communication) can also be handled by the runtime system. We break down global synchronous communication into several asynchronous sending and receiving tasks, which effectively prevent communication from blocking other workers.

`GOFMM` is fully templatized and abstracted to facilitate architecture-dependent optimization. By substituting the reference implementation of each **task**, specialized kernels such as nearest-neighbor search and kernel summation can be integrated into `GOFMM` to achieve high performance. These specialized kernels (N-body operations) are inspired by the `BLIS` framework [72, 85], which provides matrix-free matrix-matrix multiplication like operations. We discuss how these operations can be designed systematically to achieve high performance on different architectures in Section 4.2.

1.5 Background

Treecodes and fast multipole methods (`FMM`), algorithms to construct $\mathcal{H}$-matrices, originally were developed for astrophysics N-body problems and integral equations. Algebraic variants led the way to the abstraction of $\mathcal{H}$-matrix methods and the application to the factorization of sparse systems arising from the discretization of elliptic PDEs [34, 9, 3, 31, 37, 76].

We review the sparse and low-rank matrix structure that directly contribute to the complexity of $\mathcal{H}$-matrix `MATVEC` in Subsection 1.5.1. We then review how treecodes (algorithms built upon tree structures and traversals) address the three challenges of constructing $\mathcal{H}$-matrix approximation for 2D kernel matrices in Subsection 1.5.2. We

briefly summarize the $\mathcal{H}$-matrix classification in Subsection 1.5.3.

1.5.1 Matrix Rank and Sparsity

In a purely algebraic point of view, $\mathcal{H}$-matrix is built upon two types of special matrix structure: low-rank and sparse. Here we briefly explain how algebraic fast `MATVEC` (with less than $\mathcal{O}(N^2)$ work) can be achieved by exploiting these properties. Details on how to discover and decompose these factors are described in Chapter 2. Related work is reviewed in Chapter 5.

We say a matrix $K \in \mathbb{R}^{N\times N}$ has rank-s if there exists decomposition $U \in \mathbb{R}^{N\times s}$ and $V \in \mathbb{R}^{s\times N}$ such that $UV = K$ (see [12, 8] for a formal definition). In the case where the smallest such s is N, matrix K is a *full-rank* matrix. Otherwise, we say matrix K is low-rank. For example,

$$K = UV = \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{bmatrix} \left(\begin{bmatrix} 1 & \\ & 10^{-3} \end{bmatrix} \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{bmatrix} \right) \tag{1.6}$$

is the singular value decomposition (SVD) of K. The diagonal matrix in the middle has all positive entries, indicating that K has rank-2 (i.e. *full-rank*). By definition, a rank-2 matrix K cannot be decomposed into UV decomposition with $s \leq 1$. However, given tolerance ϵ, it is possible to create a rank-s ϵ-approximation of K such that $\|UV - K\| \leq \epsilon\|K\|$. For example, we can simply drop the second largest singular value from the decomposition above such that

$$K \approx UV = \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{bmatrix} = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix} \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix}. \tag{1.7}$$

Low-rank factors UV become a rank-1 approximation of K. The absolute 2-norm

error of the low-rank approximation UV is bounded by

$$\|UV - K\|_2 = \left\| \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{bmatrix} \begin{bmatrix} 0 & 0 \\ 0 & 10^{-3} \end{bmatrix} \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{bmatrix} \right\|_2 \leq \mathcal{O}(10^{-3}). \tag{1.8}$$

If low-rank factors UV are known, then (approximate) `MATVEC` of K can be computed fast in two steps: (1) an $\mathcal{O}(sN)$ `MATVEC` with U, and (2) an $\mathcal{O}(Ns)$ `MATVEC` with V.

Fast `MATVEC` is also available if K is sparse (or there is a sparse ϵ-approximation of K). We say a matrix is sparse if only nnz out of N^2 entries of K are non-zero. If these nnz non-zero entries are stored in a compact format (e.g. coordinate format (COO triplets), and compressed sparse column (CSC) [27]), then `MATVEC` of K can be computed in $\mathcal{O}(nnz)$ by traversing all non-zero entries.

Neither sparse nor low-rank: Recall that many SPD or kernel matrices are neither sparse nor low-rank. For example,

$$K = \begin{bmatrix} 1 & 2 & 1 & 2 \\ 2 & 5 & 2 & 5 \\ 1 & 2 & 1 & 2 \\ 2 & 5 & 2 & 5 \end{bmatrix} = S + UV = \begin{bmatrix} 1 & & & \\ & 1 & 1 & \\ & 1 & 1 & \\ & & & 1 \end{bmatrix} + \begin{bmatrix} 1 & 0 \\ 2 & 1 \\ 1 & 0 \\ 2 & 1 \end{bmatrix} \begin{bmatrix} 1 & 2 & 1 & 2 \\ 0 & 1 & 0 & 1 \end{bmatrix} \tag{1.9}$$

is dense and full-rank. Still, it is possible to decouple the sparse correction from the low-rank factors. Recall that $\mathcal{H}$-matrix methods decouple the sparse correction from the low-rank factors recursively in the form of Equation 1.2. As a result, we can compute their fast `MATVEC` separately.

1.5.2 Spatial Tree

For kernel matrices with points in 2D and 3D, quad-tree (with 4 children) or oct-tree (with 8 children) that resemble spatial decomposition are typically used to permute and expose the $\mathcal{H}$-matrix structure. Figure 1.2 is an example that shows how 16

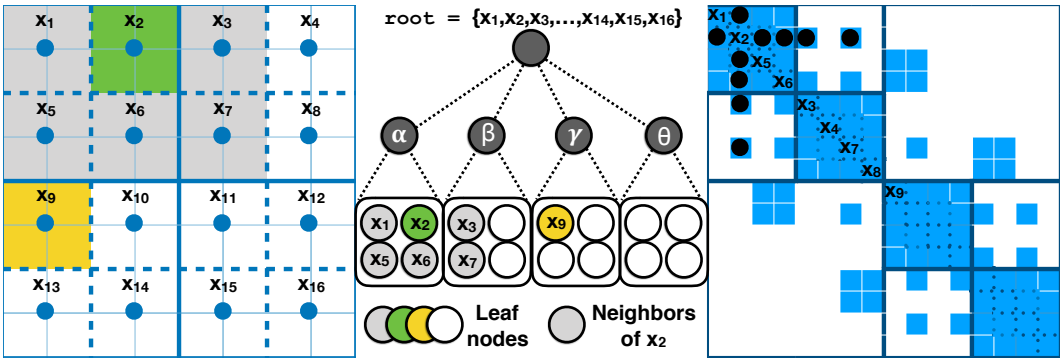

Figure 1.2: Spatial partition (clustering) on 2D-grid points (left), corresponding quad-tree, and its approximate distance-based kernel matrix: grid points $\{x_i\}_{i=1}^{16}$ are first partitioned with solid blue lines into 4 quadrants, and repartitioned with dotted lines into 16 sub-quadrants. Such spatial hierarchy is captured and encoded as a two-level quad-tree (with 4 children). Neighbors of x_2 are defined to be the points that fall into the 5 adjacent sub-quadrants colored in gray. For example, neighbor x_1 is **near** to x_2. Hence, the corresponding entries K_{12} and K_{21} in the distance-based kernel matrix must be evaluated directly. On the other hand, K_{92} and K_{29} can be approximated, because x_9 is **far** from x_2. Base on this **near**/**far** definition, the all entries that cannot be approximated are colored in blue, which creating a sparse matrix S in Equation 1.2.

2D-grid points get clustered recursively by a quad-tree. The first level of clustering is denoted by the solid blue lines in all figures. The 2D space is divided into 4 quadrants on the left. In the middle, the root node is split into 4 children. On the right, the whole matrix is partitioned, and there are four diagonal blocks. The second level of clustering is denoted by the dotted lines. The 2D space ends up with 16 quadrants. Each leaf node represents a sub-quadrant, containing exactly one point. In the matrix view (on the right), the 4 diagonal blocks are repartitioned into 16 blocks. Each sub-block contains one diagonal entry.

Near-far decomposition: While the spatial hierarchy is captured and encoded by the quad-tree, we can now decide which entry K_{ij} cannot be approximated. That is K_{ij} must be evaluated as $\mathcal{K}(x_i, x_j)$ directly. For the majority of distance-

based kernel matrices, low-rank approximation typically does not perform well[3] on entry K_{ij} with small pairwise distance $\|x_i - x_j\|_2$. As a result, for each point x_i, we identify its neighbors to capture these entries. x_i and x_j are defined to be **near** if they are neighbors to each other. Otherwise, they are **far**.

In Figure 1.2, we use point x_2 as an example. With the quad-tree, neighbors of x_2 are defined to be the points that fall into the 5 adjacent sub-quadrants (leaf nodes) colored in gray. Entries K_{12} and K_{21} are evaluated directly because x_1 is a neighbor of x_2. If we identify all neighbor pairs, then all direct evaluated entries form a sparse matrix in Figure 1.2.

Treecode: We now explain how `MATVEC` of a kernel matrix can be approximated by traversing the spatial tree downward. The following computation

$$u_i = \sum_{j=1}^{N} \mathcal{K}(x_i, x_j) w_j = \sum_{x_j \in \texttt{root}} \mathcal{K}(x_i, x_j) w_j,\ 1 \le i \le N. \tag{1.10}$$

is called **kernel summation**, which computes the weighted sum of all interactions between x_i and all points in the `root` node ($\texttt{root} \equiv \{x_j\}_{j=1}^{N}$).

To explain the relation between the tree traversal and the formula above, let us take point x_2 in Figure 1.2 and its summation u_2 as an example. Observe that

$$\begin{aligned} u_2 = \sum_{x_j \in \texttt{root}} \mathcal{K}(x_2, x_j) w_j = &\sum_{x_j \in \alpha} \mathcal{K}(x_2, x_j) w_j + \sum_{x_j \in \beta} \mathcal{K}(x_2, x_j) w_j + \\ &\sum_{x_j \in \gamma} \mathcal{K}(x_2, x_j) w_j + \sum_{x_j \in \theta} \mathcal{K}(x_2, x_j) w_j. \end{aligned} \tag{1.11}$$

can be decomposed by the quad-tree recursively. The recursion resembles a tree traversal, and the process stops if the tree node contains no neighbor of x_2. For example, node γ does not contain any neighbor of x_2. We can then approximate all contribution from γ as $\widetilde{w}_\gamma$ (see Chapter 2 for details). Similarly, the contribution of

[3]This can be proved for some distance-based kernel functions.

θ can be approximated by $\widetilde{w}_\theta$. Node α and β contain neighbors of x_2. Thus, they must be furthered decomposed through recursion. Let $\mathcal{N}(i)$ denotes all the neighbors of x_i, the full expansion can be written as two parts:

$$u_2 = \sum_{x_j \in \texttt{root}} \mathcal{K}(x_2, x_j) w_j \approx \sum_{x_j \in \mathcal{N}(i)} \mathcal{K}(x_i, x_j) w_j + (\widetilde{w}_4 + \widetilde{w}_8 + \widetilde{w}_\gamma + \widetilde{w}_\theta). \tag{1.12}$$

The **near** contribution is directly evaluated. In Equation 1.2, it is represented as the sparse correction S. The **far** contribution is approximated by constructing low-rank decomposition for each tree nodes in different levels. In Equation 1.2, it is represented as the low-rank factor UV.

1.5.3 $\mathcal{H}$-matrix Classification

Recall the decomposition $K = D + S + UV$ in Equation 1.1 and Equation 1.2. If S is zero the approximation is called a hierarchical off-diagonal low-rank (`HODLR`) scheme. In addition to S being zero, if the $\mathcal{H}$-matrix decomposition of D is used to construct U, V we have a hierarchical semi-separable (`HSS`) scheme. If S is not zero we have a generic $\mathcal{H}$-matrix; but if the U, V terms are constructed in a nested way then we have an $\mathcal{H}^2$-matrix or an `FMM` depending on more technical details. `HSS` and `HODLR` matrices lead to very efficient approximation algorithms for $(\lambda I + K)^{-1}$. However, $\mathcal{H}^2$ and `FMM` compression schemes better control the maximum rank of the U and V matrices than `HODLR` and `HSS` schemes. For the latter, the rank of U and V can grow with N [16] and the complexity bounds are no longer valid. Recently, there have been algorithms to effectively compress `FMM` and $\mathcal{H}^2$-matrices [21, 80].

1.6 Contribution

This dissertation contributes to the field of computer science and computational science in the following ways:

- **Geometry-oblivious $\mathcal{H}$-matrix methods [82, 83, 84]:** By using the Gramian vector space for SPD matrices, we defined distances between matrix indices of K using only matrix values. Using these geometry-oblivious distances, we introduced `GOFMM` and $\mathcal{H}$-matrix schemes that can be used to compress arbitrary SPD matrices. `GOFMM` takes $\mathcal{O}(N \log N)$ work to compress and factorize an $N \times N$ SPD matrix. The only required input to our algorithm is a routine that returns submatrix K_{IJ}, for arbitrary row and column index sets I and J. Once K is compressed and factorized, `MATVEC`s of $\widetilde{K}$ and `SOLVE`s of $\lambda I + \widetilde{K}$ can be applied with $\mathcal{O}(N \log N)$ work.

- **Parallel $\mathcal{H}$-matrix algorithms [82, 83, 84, 56, 53, 55, 54]:** With `GOFMM`, a purely algebraic $\mathcal{H}$-matrix method, we abstract the class of $\mathcal{H}$-matrix methods in terms of several tasks and tree traversals, which allow us to systematically exploit the *out-of-order* parallelism of our methods with a self-contained runtime system. We show that dependency analysis and scheduling can resolve the dynamic workload due to adaptive ranks and the parallelism-diminishing issue during tree traversals. As a result, `GOFMM` can achieve higher performance on shared- and distributed-memory heterogenous architectures comparing to other methods that use level-by-level tree traversals and synchronous message passing.

- **High-performance N-body operators [81, 55, 83, 56, 53]:** Despite our method only requiring raw matrix values as input, the framework can use geometry information to improve its accuracy and performance if provided. For example, `GOFMM` has general support to kernel matrices. While data points are provided, highly optimized N-body operators are exploited to boost the performance and reduce storage requirement.

- We conduct extensive experiments to demonstrate the feasibility of the proposed approach. We test our code on 22 different matrices related to machine learning,

stencil PDEs, spectral PDEs, inverse problems, and graph Laplacian operators. We perform numerical experiments on Intel Skylake, Haswell, Ivy-bridge, and KNL, Qualcomm ARM, and NVIDIA Pascal architectures. Finally, we compare with three state-of-the-art codes: `HODLR`, `STRUMPACK`, and `ASKIT`.

To be specific, the dissertation results in the following conference and journal papers:

- Chenhan D Yu, James Levitt, Severin Reiz, and George Biros. Geometry-oblivious FMM for compressing dense SPD matrices. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 53. ACM, 2017

- Chenhan D Yu, William B March, and George Biros. An n log n parallel fast direct solver for kernel matrices. In *Parallel and Distributed Processing Symposium (IPDPS), 2017 IEEE International*, pages 886–896. IEEE, 2017

- Chenhan D. Yu, William B. March, Bo Xiao, and George Biros. INV-ASKIT: a parallel fast direct solver for kernel matrices. In *Proceedings of the IPDPS16*, Chicago, USA, May 2016

- Chenhan D. Yu, Jianyu Huang, Woody Austin, Bo Xiao, and George Biros. Performance optimization for the k-nearest neighbors kernel on x86 architectures. In *Proceedings of SC15*, pages 7:1–7:12. ACM, 2015

- William B. March, Bo Xiao, Chenhan D. Yu, and George Biros. Askit: An efficient, parallel library for high-dimensional kernel summations. *SIAM Journal on Scientific Computing*, 38(5):S720–S749, 2016

- William B. March, Bo Xiao, Sameer Tharakan, Chenhan D. Yu, and George Biros. A kernel-independent FMM in general dimensions. In *Proceedings of SC15*, The SCxy Conference series, Austin, Texas, November 2015. ACM/IEEE

- William B. March, Bo Xiao, Chenhan Yu, and George Biros. An algebraic parallel treecode in arbitrary dimensions. In *Proceedings of IPDPS 2015*, 29th IEEE International Parallel and Distributed Computing Symposium, May 2015

- William B. March, Bo Xiao, Sameer Tharakan, Chenhan D. Yu, and George Biros. Robust treecode approximation for kernel machines. In *Proceedings of the 21st ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 1–10, Sydney, Australia, August 2015

Togather, these advance the field.

1.7 Limitations

Not all SPD and kernel matrices admit a good hierarchical low-rank decomposition. Typically, this is related to the intrinsic dimensionality of the dataset at different scales. So our method may fail to achieve the desired tolerance ϵ or the desired $\mathcal{O}(N \log N)$ complexity. `GOFMM` requires the ability to evaluate arbitrary (and not contiguous) matrix entries, an option that may not be always available. If access of K is only available through matrix-free interfaces, these assumptions may not be satisfied. Other algorithms, like `STRUMPACK`, have inherent support for such matrix-free compression. Our method guarantees that $\widetilde{K}$ is symmetric, but positive definiteness may be compromised when relative error $\|K - \widetilde{K}\|/\|K\|$ is large.

$\mathcal{H}$-matrix factorization: Since there is no guarantee on positive definiteness, our $\mathcal{H}$-matrix factorization may be numerically unstable. That is, even if our method can compress the matrix, the second potential point of failure is the choice of regularization parameter λ for factorizing $(\lambda I + \widetilde{K})$. While $\widetilde{K}$ may exhibit negative eigenvalues, $(\lambda I + \widetilde{K})$ may be ill-conditioned (or actually rank deficient) if λ is too small. Our algorithms [84, 83] can numerically detect the instability, but it is not clear how to fix it while maintaining the log-linear complexity of the algorithm. Also,

our methods in [84, 83] can only be a preconditioner but not the exact factorization if the sparse correction S in Equation 1.2 is not zero.

Performance bottleneck: Although the asymptotic complexity analysis of our compression phase is $\mathcal{O}(N \log N)$, but the constant is actually huge due to several memory-bound operations. The main performance bottleneck is the pivoted QR factorization (`GEQP3` in `LAPACK`), which computes the skeleton of each tree node. The tree-like dependencies and synchronizations also result in parallelism diminishing. Second, the performance of our distributed evaluation ($\mathcal{H}$-matrix multiplication) is affected by the distribution of the interaction lists. Since `GOFMM` does not implement a distributed job-stealing mechanism for load-balancing, the distribution of the interaction lists reflects the distributed workload and message sizes. In the case that the distribution is skew and not local, asynchronous communication cannot be fully overlapped. Finally, the optimal complexity estimate assumes that the rank of UV matrices that appear in our algorithm is independent of the problem size N. In many applications, this assumption is not valid.

1.8 Outline of the dissertation

This dissertation is structured as followed:

We start by reviewing the three challenges of constructing $\mathcal{H}$-matrices and the properties of low-rank and sparse matrices in Chapter 2. We review how these matrix properties relate to the geometry (distribution of points) and end with a revisit but with a geometry-oblivious interpretation that generalizes to arbitrary SPD matrices.

In Chapter 3, we introduce the four phases of our $\mathcal{H}$-matrix method (`GOFMM`): compression, evaluation (`MATVEC`), factorization, and linear solver (`SOLVE`). We present task-based and distributed-memory parallel algorithms. We end the chapter with a theory section to summarize the accuracy and complexity of `GOFMM`.

In Chapter 4, we present implementation, optimization details, and experi-

mental results. The chapter begins with a review of GOFMM's self-contained runtime system. Then we briefly discuss how k-nearest neighbor search and kernel summation can be accelerated by GEMM-like N-body operators in a matrix-free fashion. We present parallel efficiency with strong and weak scaling experiments. Finally, we present results on kernel ridge regression tasks using real-world datasets.

In Chapter 5, we discuss related work, reviewing different low-rank approximation methods, matrix permutation strategies, $\mathcal{H}$-matrix variants, $\mathcal{H}$-matrix software, task scheduling, and high-performance linear algebra operations.

In Chapter 6, we end with concluding remarks and ideas for future research.

Chapter 2

Metric Trees, Neighbor Search, and Skeletonization

[1]We saw a treecode example in Figure 1.2, where a spatial quad-tree is used to partition 2D-grid points. The corresponding symmetric matrix permutation encodes the spatial distribution of data points $\{x_i\}_{i=1}^{N}$, exposing **weak (far) kernel interactions** between loosely coupled clusters, enabling low-rank matrix approximation in the off-diagonal blocks. The sparse correction is captured by the neighbor structure, which evaluates **strong (near) kernel interactions** directly without approximation. To further abstract and generalize the method, we need to know how such spatial trees and neighbors can be defined and computed for data points in an arbitrary d-dimensional space. At the end of the chapter, we generalize these key concepts to generic SPD matrices, where *no points but only the raw matrix values are provided.*

[1]Several of my prior publications contribute to this chapter. In [82] (Chenhan D. Yu, James Levitt, Severin Reiz, and George Biros. Geometry-oblivious FMM for compressing dense SPD matrices.), I introduced `GOFMM` and $\mathcal{H}$-matrix schemes that can be used to compress arbitrary SPD matrices. In [81] (Chenhan D. Yu, Jianyu Huang, Woody Austin, Bo Xiao, and George Biros. Performance optimization for the κ-nearest neighbors kernel on x86 architectures.), I introduced an efficient algorithm for exhausted κ-nearest neighbor search on modern CPUs. By fusing neighbor search with GEMM in the assembly level, the proposed method is free from extra working space and suffer from smaller memory latency. In [56] (William B. March, Bo Xiao, Chenhan D. Yu, and George Biros. `ASKIT`: An efficient, parallel library for high-dimensional kernel summations.), I design and conduct experiments to empirically analyze the performance of `ASKIT`.

In this chapter, we begin with a review on how a semi-algebraic $\mathcal{H}$-matrix method `ASKIT` [56] addresses the three challenges of constructing an $\mathcal{H}$-matrix representation for a kernel matrix. We focus on how and why geometry information is used in addressing each challenge such that our algorithms can be geometry-oblivious at the end. In Section 2.1, we use a binary metric tree to generalize the concept of spatial partitioning to an arbitrary d-dimensional space. In Section 2.2, we review why κ-nearest neighbors are used in high-dimensional space and how to iteratively approximate these neighbors using a tree-based greedy search. Nested interpolative decomposition (`ID`) [36] is reviewed in Section 2.3. We use `ID` to create low-rank approximation UV in Equation 1.1 with nested basis. In Section 2.4, we end the chapter with a revisit on how treecodes can be generalized without using geometry information (points). *We show that the capability to define a proper distance metric purely on matrix indices instead of points is the key.*

Notation: In Table 2.1, we summarize the main notation used in the dissertation. The first category contains all user-defined parameters. The second category relates to matrix indexing and tree node representation. The third category contains auxiliary notation used to describe kernel matrices.

Related contributions: This chapter contributes to the following conference and journal publications.

- Chenhan D Yu, James Levitt, Severin Reiz, and George Biros. Geometry-oblivious FMM for compressing dense SPD matrices. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 53. ACM, 2017

- Chenhan D. Yu, Jianyu Huang, Woody Austin, Bo Xiao, and George Biros. Performance optimization for the k-nearest neighbors kernel on x86 architectures. In *Proceedings of SC15*, pages 7:1–7:12. ACM, 2015

- William B. March, Bo Xiao, Chenhan D. Yu, and George Biros. Askit: An efficient,

Notation	Description
N	problem size
$K(i,j)$	a virtual function to access K_{ij} of an N-by-N SPD matrix K
m	maximum number of points at leaf nodes
s	maximum number of skeleton points (maximum rank of off-diagonal blocks)
κ	neighbor size
ϵ	tolerance in adaptive skeletonization
b	**budget**
p	number of `MPI` processes
r	number of right-hand sides
λ	regularization for solving $\lambda I + \widetilde{K}$
α, β, γ, θ	tree nodes (indices owned by tree nodes)
`l`, `r`	left and right children of the tree node
$\widetilde{\alpha}$, $\widetilde{\beta}$, $\widetilde{\gamma}$, $\widetilde{\theta}$	skeleton of tree nodes (subsets of indices or points owned by tree nodes)
d	dimension of input points
$\mathcal{X}$	$\mathbb{R}^{d\times N}$, point dataset $\{x_i\}_{i=1}^N$
$\mathcal{X}(\alpha)$, or $\mathcal{X}_\alpha$	points owned by node α, i.e. $\mathcal{X}(\alpha) = \{x_i \vert \forall i \in \alpha\}$
$\mathcal{K}(x_i, x_j)$	kernel function that takes two points as input
$\widetilde{K}$	$\mathcal{H}$-matrix approximation of K
$K_{\alpha\beta}$	submatrix $K(\alpha,\beta)$ in `MATLAB`-tyle matrix stretching

Table 2.1: The table presents the main notation used. The first category contains all user-defined parameters. The second category relates to matrix indexing and tree node representation. The third category contains auxiliary notation used to describe kernel matrices. Occasionally we will use `MATLAB`-style matrix representations.

parallel library for high-dimensional kernel summations. *SIAM Journal on Scientific Computing*, 38(5):S720–S749, 2016

2.1 Metric Trees

To generalize spatial tree partitioning to an arbitrary d-dimensional space, `ASKIT` [56] uses a binary balanced ball tree instead of the 2^d-way tree that creates a hypercube spatial partition. Such 2^d-way trees are not preferred for high dimensional problems ($d > 3$) because the complexity scales exponentially with d. *Also, notice that spatial trees are used to exploit the low-rank matrix structure representing weak interactions between loosely coupled clusters, and it can be shown that if the low-rank submatrices*

Algorithm 2.1 $[\texttt{l}, \texttt{r}] = \texttt{BallTreeSplit}(\alpha)$

$x_c = \sum_{x_i \in \alpha} x_i / |\alpha|$; /* Find the centroid of α by averaging over all points. */
$x_{\texttt{l}} = \text{argmax}(d_{ic} = \{\|x_i - x_c\|_2 | x_i \in \alpha\})$;
$x_{\texttt{r}} = \text{argmax}(d_{ip} = \{\|x_i - x_{\texttt{l}}\|_2 | x_i \in \alpha\})$;
$[\texttt{l}, \texttt{r}] = \text{medianSplit}(\{x_i^T (x_{\texttt{l}} - x_{\texttt{r}}) / \|x_{\texttt{l}} - x_{\texttt{r}}\|_2 | x_i \in \alpha\})$; /* Project and split. */

of K can be decomposed by a hierarchical p-way tree partition, then a binary tree is sufficient to exploit the same low-rank structure by repartitioning.

We first summarize how `ASKIT` creates a balanced binary ball tree by projecting high-dimensional points down to low-dimensional space. We end the section with an introduction to a general binary metric tree, where, instead of using the physical coordinates of points to compute the projection and centroid directly, the relative pairwise distances between points are used to split the points.

Binary ball trees: The root node is assigned with the full set of points, i.e., $\texttt{root} \equiv \{x_i\}_{i=1}^N$, and the tree is constructed recursively by splitting points owned by a tree node evenly between two child nodes in three steps. In Algorithm 2.1, we partition a tree node α by estimating the farthest pair of points $(x_{\texttt{l}}, x_{\texttt{r}})$ belonging to it. As a result, points in α are split into two loosely coupled clusters (children) with almost the same amount of points.

Centroid x_c is computed first by averaging over $\{x_i | x_i \in \alpha\}$ in each dimension. Then $x_{\texttt{l}}$ is selected to be the farthest point from x_c using Euclidean distances. Finally, $x_{\texttt{r}}$ is selected to be the farthest point from $x_{\texttt{l}}$. As a result, $x_{\texttt{l}}$ and $x_{\texttt{r}}$ approximate the two farthest points that belong to node α. We project all the points in node α on to the line drawn by $x_{\texttt{l}}$ and $x_{\texttt{r}}$ and perform a median split over the projected values. This will split all points into two children `l` and `r` evenly. We recursively split each tree node until every leaf node contains no more than m points, where m is a user-defined parameter (see Table 2.1) that controls the depth of the tree.

Binary metric trees: Notice that Algorithm 2.1 computes the projection and centroid using the physical coordinates of points directly. However, the fact

Algorithm 2.2 $[\mathtt{l}, \mathtt{r}] = \mathtt{MetricTreeSplit}(\alpha)$

Let $\mathcal{C} \subset \alpha$ contains $\mathcal{O}(1)$ samples;
$x_{\mathtt{l}} = \text{argmax}\left(\{\sum_{c \in \mathcal{C}} d_{ic}/|\mathcal{C}||i \in \alpha\}\right)$; /* Largest averaged distance */
$x_{\mathtt{r}} = \text{argmax}(\{d_{i\mathtt{l}}|i \in \alpha\})$;
$[\mathtt{l}, \mathtt{r}] = \mathtt{medianSplit}(\{d_{i\mathtt{l}} - d_{i\mathtt{r}}|i \in \alpha\})$; /* Relative position */

is that we only need relative positions between points for splitting points in the geometry space. To avoid direct-use of these coordinates in $\mathbb{R}^d$ but still approximately split the node according to the two farthest points, we introduce a binary metric tree in Algorithm 2.2, where only pairwise distance $d_{ij} = \|x_i - x_j\|_2$ (or any properly defined distance metric) between index i and j is used.

Instead of computing the centroid x_c directly in Algorithm 2.2, the first farthest point $x_{\mathtt{l}}$ is approximated by the one with the largest averaged distance over a set of samples $\mathcal{C} \subset \alpha$. Again $x_{\mathtt{r}}$ is selected to be the farthest point from $x_{\mathtt{l}}$. Finally, the relative position of x_i between $x_{\mathtt{l}}$ and $x_{\mathtt{r}}$ is computed by $d_{i\mathtt{l}} - d_{i\mathtt{r}}$. We perform a median split on this relative position to get an even split.

2.2 Nearest-Neighbor Search

We have seen that neighbors of 2D-grid points (Figure 1.2) involve points falling in the eight adjacent partitions created by the quad-tree. Naturally, for 3D-grid points, neighbors will involve the 26 adjacent partitions[2] created by the oct-tree (with $2^3 = 8$ children). However, we do not create a 2^d-way tree for data points in an arbitrary d-dimensional space, because the corresponding complexity scales exponentially with d. Recall that we use a binary metric tree instead. The drawback is that binary trees do not produce a regular space partition. As a result, defining neighbors based on adjacent partitions can be difficult. As an alternative, we use κ-nearest neighbors of each point, which does not rely on the spatial partition created by the metric tree.

[2]The number of adjacent partitions is computed by $3 \times 3 \times 3 - 1$.

Algorithm 2.3 $\mathcal{N}(\alpha) =$ `ANN(`α`)`

for each $i \in \alpha$ **do** $\mathcal{N}(1:\kappa, i) =$ `k-select(`$\{$`pair(`d_{ij}, j`)`$|\forall j \in \alpha\}$`)`;

Algorithm 2.4 `MergeNeighbors(`$\mathcal{N}$`, `$\mathcal{N}'$`)`

for each $j = 1 : N$ **do**
$\mathcal{N}'(1:2\kappa, j) =$`sort([`$\mathcal{N}(1:\kappa, j)$`;`$\mathcal{N}'(1:\kappa, j)$`]);`
$\mathcal{N}(1:\kappa, j) = \mathcal{N}'(1:\kappa, j)$;

Definition 1. *Given a set of N reference points $\mathcal{X} \equiv \{x_i \in \mathbb{R}^d\}_{i=1}^N$, and a query point x in an arbitrary d-dimensional space, $\mathcal{N}(x)$ (the κ-nearest neighbors of point x) is the set of κ points such that $\forall x_p \in \mathcal{N}(x)$ we have*

$$\|x - x_j\|_2 \geq \|x - x_p\|_2, \ \forall x_j \in \mathcal{X} \setminus \mathcal{N}(x). \tag{2.1}$$

When we compute the nearest neighbors for all points $x_j \in \mathcal{X}$, the problem is called the all-nearest-neighbor (`ANN`) problem.

Notice that an exhaustive search such as the one presented in Algorithm 2.3 for all N points requires N^2 distance evaluations, hence $\mathcal{O}(dN^2)$ work. In low dimensions (say $d < 10$), regular spatial trees can compute the exact κ-nearest neighbors using $\mathcal{O}(N)$ distance evaluations [66]. But in higher dimensions, it is known that tree-based algorithms end up having quadratic complexity [74]. To circumvent this problem, *we must abandon the concept of exact searches and settle for approximate searches.*

State-of-the-art methods for problems in high dimensions use randomization methods, for example, tree-based methods [61, 24, 44, 77] or hashing based methods [6, 2]. In Algorithm 2.5, we present a randomized tree-based greedy search inspired by [78] to approximate κ-nearest neighbors iteratively.

Each approximate iteration in Algorithm 2.5 takes $\mathcal{O}(dN \log N)$ work to build a random projection tree and $\mathcal{O}(dN)$ work to perform an exhaustive search in each leaf node. Instead of splitting the tree node according to the relative position between two

Algorithm 2.5 `RandomizedAllNearestNeighbors()`

while $\mathcal{N}$ does not converge **do**
  `[Preorder]` **for each** $\alpha \in$ `owned_nodes` **do**
    randomly select $x_{\mathtt{l}} \neq x_{\mathtt{r}}$ from node α;
    $[\mathtt{l}, \mathtt{r}] =$ `medianSplit`$(\{d_{i\mathtt{l}} - d_{i\mathtt{r}} | x_i \in \alpha\})$;
  `[Anyorder]` **for each leaf node** α **do** `ANN`(α); `/* Exhausted search */`
  `MergeNeighbors`$(\mathcal{N}, \mathcal{N}')$;

farthest points, we randomly select $x_{\mathtt{l}}$ and $x_{\mathtt{r}}$ to create different spatial partitioning. For each leaf node α, we perform an exhaustive κ-nearest neighbor search for all points in α using `ANN`(α) (Algorithm 2.5). Although the searching scope is limited to the candidates in the same tree node, different spatial partitioning computed by the random split will gradually increase the overall search space.

At the end of the iteration, `MergeNeighbors()` in Algorithm 2.4 merges new κ candidates to update the existing candidates. The process first concatenates and sorts the two lists according to the `key` values (pairwise distances). Finally, the sorted list is truncated to length κ and copied back to overwrite the existing candidates.

2.3 Skeletonization

We detail how low-rank factors U and V in Equation 1.2 are computed in our $\mathcal{H}$ matrix methods. We approximate off-diagonal blocks of K with a nested interpolative decomposition (`ID` in brief) [36]. We first define the concepts of *skeletonization* and *skeleton* for each tree node in the following.

Definition 2. *Let β be the indices owned by a tree node and $I = \{1, ..., N\} \backslash \beta$ be the set complement. The* skeletonization *of β is a rank-s approximation of its off-diagonal blocks $K_{I\beta}$ using the `ID`, which we write as*

$$K_{I\beta} \approx K_{I\tilde{\beta}} P_{\tilde{\beta}\beta}, \tag{2.2}$$

where $\widetilde{\beta} \subset \beta$ *is the* skeleton *of* β. $K_{I\widetilde{\beta}} \in \mathbb{R}^{(N-|\beta|)\times s}$ *is a column submatrix of* $K_{I\beta}$, *and* $P_{\widetilde{\beta}\beta} \in \mathbb{R}^{s\times|\beta|}$ *is a matrix of interpolation coefficients.*

`ID` tries to select the best pair of skeleton $\widetilde{\beta}$ and coefficient matrix $P_{\widetilde{\beta}\beta}$ by solving the following optimization problem

$$\text{argmin}_{\widetilde{\beta}\subset\beta, P\in\mathbb{R}^{s\times|\beta|}} \|K_{I\beta} - K_{I\widetilde{\beta}}P\|_2. \tag{2.3}$$

In [36], it is shown that $\widetilde{\beta}$ can be computed by a rank-revealing QR factorization (`GEQP3` in `LAPACK`) on $K_{I\beta}$, and $P_{\widetilde{\beta}\beta}$ has a least square solution $K^{\dagger}_{I\widetilde{\beta}}K_{I\beta}$, where $K^{\dagger}_{I\widetilde{\beta}}$ is the pseudo-inverse[3] of $K_{I\widetilde{\beta}}$. To be precise, `GEQP3` computes

$$K_{I\beta} = \begin{bmatrix} K_{I\widetilde{\beta}} & K_{I[\beta\setminus\widetilde{\beta}]} \end{bmatrix} = \begin{bmatrix} Q_{I\widetilde{\beta}} & Q_{I[\beta\setminus\widetilde{\beta}]} \end{bmatrix} \begin{bmatrix} R_{11} & R_{12} \\ & R_{22} \end{bmatrix}, \tag{2.4}$$

where R_{11} and R_{22} are upper triangular matrices and $Q_{I\widetilde{\beta}}$ contains the first s orthonormal columns. Replacing $K^{\dagger}_{I\widetilde{\beta}}$ with the pseudo-inverse of $Q_{I\widetilde{\beta}}R_{11}$ yields

$$P_{\beta\widetilde{\beta}} = K^{\dagger}_{I\widetilde{\beta}}K_{I\beta} = R_{11}^{-1}Q^T_{I\widetilde{\beta}} \begin{bmatrix} Q_{I\widetilde{\beta}} & Q_{I[\beta\setminus\widetilde{\beta}]} \end{bmatrix} \begin{bmatrix} R_{11} & R_{12} \\ & R_{22} \end{bmatrix} = \begin{bmatrix} I & R_{11}^{-1}R_{12} \end{bmatrix}. \tag{2.5}$$

The system above can be solved by a triangular solver (`TRSM` in `BLAS`) because R_{11} is an upper triangular matrix.

Randomized `ID`: Notice that a full `ID` takes $\mathcal{O}((N-|\beta|)|\beta|^2)$ work for a node β due to the full QR factorization. Computing `ID` for all leaf nodes requires at least $\mathcal{O}(N^2)$ work. To efficiently compute this approximation, we select a sample subset $I' \subset I$ using a combination of neighbor-based importance sampling and uniform sampling (if there are not enough neighbors). The `ID` process remains the same, but

[3]The pseudo-inverse is also known as the Moore–Penrose inverse. To be specific $K^{\dagger}_{I\widetilde{\beta}} = (K^T_{I\widetilde{\beta}}K_{I\widetilde{\beta}})^{-1}K^T_{I\widetilde{\beta}}$. With a full QR factorization of $K_{I\widetilde{\beta}}$, $K^{\dagger}_{I\widetilde{\beta}} = R^{-1}Q^T$.

we use row samples $K_{I'\beta}$ instead of the full off-diagonal block $K_{I\beta}$ in Equation 2.3. This idea follows the observation that using neighbors as row samples (see [56, 82]) produces accurate skeletonizations and also does not necessarily depend on the off-diagonal block size but rather on the local intrinsic dimensionality of the dataset.

Nested skeletons: For an internal node α, we form the skeletonization in the same way, except that the columns are also sampled using the skeletons of the children of α. That is, the `ID` is computed for $K_{I'[\widetilde{\mathtt{lr}}]}$ instead of $K_{I'\alpha}$, where $[\widetilde{\mathtt{lr}}] = \widetilde{\mathtt{l}} \cup \widetilde{\mathtt{r}}$ contains the skeletons of the children of α. To be precise,

$$K_{I'[\widetilde{\mathtt{lr}}]} \approx K_{I'\widetilde{\alpha}} P_{\widetilde{\alpha}[\widetilde{\mathtt{lr}}]}, \text{ and } P_{\alpha[\widetilde{\mathtt{lr}}]} = K^{\dagger}_{I'[\widetilde{\mathtt{lr}}]} K_{I'\widetilde{\alpha}} \tag{2.6}$$

is the corresponding least square solution. In this way, the skeletons are nested: $\widetilde{\alpha} \subset \widetilde{\mathtt{l}} \cup \widetilde{\mathtt{r}}$. As a consequence of the nesting property, we can use $P_{\widetilde{\mathtt{ll}}}$ and $P_{\widetilde{\mathtt{rr}}}$ to construct an approximation of the full block $K_{I\alpha}$:

$$K_{I'\alpha} \approx K_{I'[\widetilde{\mathtt{lr}}]} \begin{bmatrix} P_{\widetilde{\mathtt{ll}}} & \\ & P_{\widetilde{\mathtt{rr}}} \end{bmatrix} \approx K_{I'\widetilde{\alpha}} P_{\widetilde{\alpha}[\widetilde{\mathtt{lr}}]} \begin{bmatrix} P_{\widetilde{\mathtt{ll}}} & \\ & P_{\widetilde{\mathtt{rr}}} \end{bmatrix}. \tag{2.7}$$

Then we have a *telescoping* expression for the full coefficient matrix:

$$P_{\widetilde{\alpha}\alpha} = P_{\widetilde{\alpha}[\widetilde{\mathtt{lr}}]} \begin{bmatrix} P_{\widetilde{\mathtt{ll}}} & \\ & P_{\widetilde{\mathtt{rr}}} \end{bmatrix}. \tag{2.8}$$

Notice that during the computation we never explicitly form $P_{\widetilde{\alpha}\alpha}$ for any internal node, but instead use the telescoping expression during evaluation. Nested skeletons are essential for $\mathcal{O}(N)$ `MATVEC` and $\mathcal{O}(N \log N)$ `SOLVE`. We discuss how the nested property is exploited in Section 3.1 and Section 3.3.

Adaptive rank: The rank s is chosen adaptively such that $\sigma_{s+1}(K_{I'[\widetilde{\mathtt{lr}}]}) < \tau$, where $\sigma_{s+1}(K_{I'[\widetilde{\mathtt{lr}}]})$ is the estimated $s+1$ singular value of submatrix $K_{I'[\widetilde{\mathtt{lr}}]}$ and τ

is related to a user-specified tolerance ϵ. Details regarding the least square problem and how τ is selected based on ϵ are discussed in Appendix A.

2.4 Geometry-Oblivious Interpretation

In this section, we introduce the machinery for using `GOFMM` in a geometry-oblivious manner. Throughout the following discussion, we refer to a set of matrix indices $I = \{1, \ldots, N\}$, where index i corresponds to the ith row (or column) of the matrix K in the original ordering. Our objective is to find a permutation of I so that K can be approximated by an $\mathcal{H}$-matrix. The key is to define a distance between a pair of indices i, $j \in I$, denoted as d_{ij}. Using the distances, we then perform a hierarchical clustering of I, which is used to define the permutation and determine which interactions go into the sparse correction S (using nearest neighbors).

Other than $d_{ij} = \|x_i - x_j\|_2$, the point-based Euclidian distance defined in the geometric space, we define two additional geometry-oblivious distance metrics in the Gramian vector space, which do not rely on points $\{x_i\}_{i=1}^N$. With a properly defined distance metric d_{ij} that only requires the raw matrix values, we can generalize $\mathcal{H}$-matrix methods to generic SPD matrices.

Theorem 1. *Given a symmetric positive definite (SPD) matrix $K \in \mathbb{R}^{N \times N}$, the distance metric $d_{ij} = K_{ii} + K_{jj} - 2K_{ij}$ (**Gram-ℓ^2, or kernel distance in brief**) and $d_{ij} = 1 - K_{ij}^2/(K_{ii}K_{jj})$ (**Gram-angle, or angle distance**) are well-defined.*

Proof. Since K is SPD, it is the *Gramian matrix* of some set of **unknown Gramian vectors**, $\{\phi_i\}_{i=1}^N \subset \mathbb{R}^N$ ([67], Proposition 2.16, page 44). That is, $K_{ij} = (\phi_i, \phi_j)$, where $(\cdot, \cdot)$ denotes the ℓ^2 inner product in $\mathbb{R}^N$.

The **Gram-ℓ^2** distance is defined to be the square ℓ^2 distance of the Gram vectors $\{\phi_i\}_{i=1}^N$. Computing the kernel distance only requires three entries of K:

$$d_{ij} = \|\phi_i\|^2 + \|\phi_j\|^2 - 2(\phi_i, \phi_j) = K_{ii} + K_{jj} - 2K_{ij} \geq 0. \tag{2.9}$$

Similarly, **Gram-angle** distance is defined to be the sine function value (cosine similarity) in inner product spaces. We define the distance as $d_{ij} = \sin^2(\angle(\phi_i, \phi_j)) \in [0, 1]$. As a result, computing an angle distance only requires three entries of K:

$$d_{ij} = 1 - \cos^2(\angle(\phi_i, \phi_j)) = 1 - K_{ij}^2/(K_{ii}K_{jj}). \tag{2.10}$$

Observe that Equation 2.10 is chosen so that d_{ij} is small for nearly collinear Gramian vectors, large for nearly orthogonal Gramian vectors, and d_{ij} is inexpensive to compute. □

Although the value d_{ij} we defined above may seem arbitrary, *we only compare values for the purpose of ordering, so any equivalent metric will do.* To reiterate for emphasis, d_{ij} defines proper distances (metrics) because K is SPD. By replacing the distance metric, we can apply metric tree partitioning (Algorithm 2.2), neighbor search (Algorithm 2.5), and perform important sampling in the skeletonization process without using any points and geometry information. That is, *as long as the distance metric d_{ij} is properly defined on an SPD matrix, we can apply $\mathcal{H}$-matrix methods or algebraic `FMM` to compress the matrix.*

Effectiveness (Figure 2.1): To examine the effectiveness of these geometry-oblivious distances, we test different permutations to discuss the different distances in `GOFMM`. In each set of experiments, we present relative error (blue plots) and average rank (green bars) for five different schemes.

The first two schemes use lexicographic (the original order) or random order to recursively permute K. Since there is no *distance* defined, these two schemes can only use an hierarchical semi-separable (`HSS`) approximation, where no sparse correction is introduced in the off-diagonal blocks. The **Angle** and **Kernel** distances use the corresponding Gram-angle and Gram-ℓ^2 distances we define above. Finally, we also use the standard Euclidean distance from points while available.

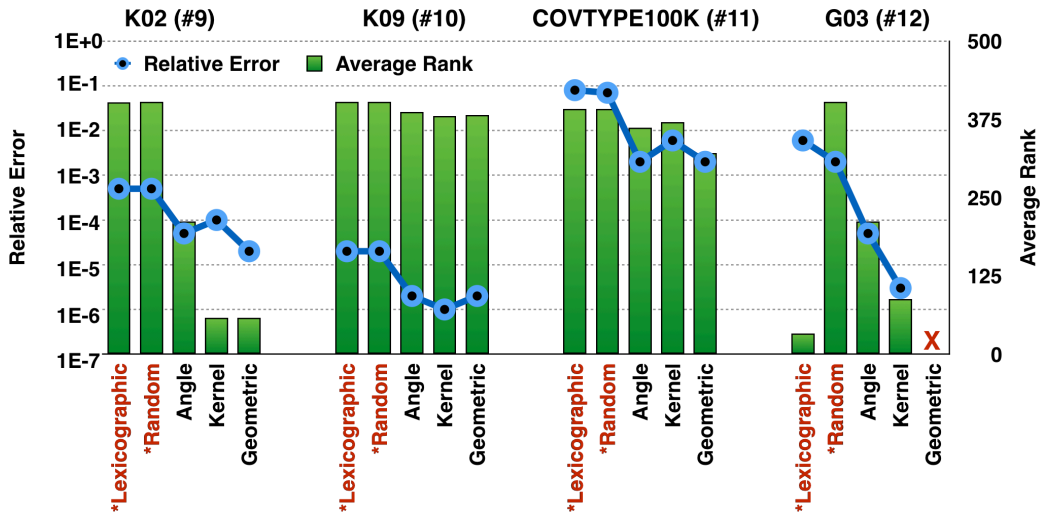

Figure 2.1: Accuracy (left y-axis) and rank (right, x-axis) comparison: **Lexicographic**, **Random**, **Kernel 2-norm**, **Angle** and **Geometric**. We use τ1E−7, s512, m64. For methods that define *distance*, we use k32 and 3% budget. **G03** is a graph Laplacian; thus, using **Geometric** distance is impossible.

Overall, we can observe that the distance metric is important in discovering the low-rank structure and improving accuracy. For example, while compressing matrix **K02**, **Kernel** and **Geometric** distances show much lower average compression rank than others. For **K09** and the Gaussian kernel matrix (**COVTYPE**), although the average ranks are not significantly different, distance-based methods usually achieve higher accuracy. Finally, we observe for matrix **G03** where no coordinate information exists (no points), our geometry-oblivious methods can still compress the matrix. Although the lexicographic permutation has very low rank, the error is large. This is because the uniform samples for the low-rank approximation are poor. **Angle** and **Kernel** distances use neighbors for importance sampling, which greatly improves the quality of the low-rank approximation.

2.5 Summary

We review how ASKIT uses a binary ball tree, κ-nearest neighbors, and randomized nested interpolative decompositions (ID) to exploit the spatial locality and generalize $\mathcal{H}$-matrix approximation to arbitrary kernel matrices in a d-dimensional space. By introducing geometry-oblivious distances, we further generalize the approach above to address the three challenges of build $\mathcal{H}$-matrix representations for generic SPD matrices. We show that as long as the distance metric is appropriately defined, metric tree partitioning (permutation) and nearest neighbors can effectively expose the $\mathcal{H}$-matrix structure.

Chapter 3

Hierarchical Matrix Algorithms

[1]Given an SPD matrix $K \in \mathbb{R}^{N \times N}$, user-defined tolerance ϵ, and p parallel processes, we aim to construct a distributed $\mathcal{H}$-matrix approximation $\widetilde{K}$ such that

$$u = Kw \approx \widetilde{K}w, \quad \text{for} \quad w \in \mathbb{R}^{N \times r}, \tag{3.1}$$

can be computed in parallel with $\mathcal{O}(N \log N)$ work and roughly $\mathcal{O}(\frac{N}{p} \log N)$ time where matrix K, potential u, and weight w are also distributed on p parallel processes. With $\mathcal{H}$-matrix $\widetilde{K}$ and an user-defined regularization parameter λ, we would further like to solve the following system

$$u \approx (\lambda I + \widetilde{K})w, \quad \text{for} \quad u \in \mathbb{R}^{N \times r}, \tag{3.2}$$

[1]Several of my prior publications contribute to this chapter. In [82] (Chenhan D. Yu, James Levitt, Severin Reiz, and George Biros. Geometry-oblivious FMM for compressing dense SPD matrices.), I introduced `GOFMM` and $\mathcal{H}$-matrix schemes that can be used to compress arbitrary SPD matrices. In [83] (Chenhan D. Yu, William B March, and George Biros. An N log N parallel fast direct solver for kernel matrices.), I introduced an $\mathcal{O}(N \log N)$ hybrid direct-iterative solver for kernel matrices. In [81] (Chenhan D. Yu, Jianyu Huang, Woody Austin, Bo Xiao, and George Biros. Performance optimization for the κ-nearest neighbors kernel on x86 architectures.), I introduced an efficient algorithm for exhausted κ-nearest neighbor search on modern CPUs. By fusing neighbor search with GEMM in the assembly level, the proposed method is free from extra working space and suffer from smaller memory latency. In [56] (William B. March, Bo Xiao, Chenhan D. Yu, and George Biros. `ASKIT`: An efficient, parallel library for high-dimensional kernel summations.), I design and conduct experiments to empirically analyze the performance of `ASKIT`.

directly in parallel with $\mathcal{O}(N \log N)$ work and roughly $\mathcal{O}(\frac{N}{p} \log N)$ time by factorizing the full-rank matrix $(\lambda I + \widetilde{K})$.

$\mathcal{H}$-matrix algorithms that compute Equation 3.1 involve two phases: **compression** and **evaluation** (`MATVEC`). Solving Equation 3.2 requires two additional phases **factorization** and **solve** (`SOLVE`). We first review the key components of `GOFMM` in Section 3.1. Then, we discuss how it can be parallelized in Section 3.2. We present parallel $\mathcal{H}$-matrix factorization algorithms in Section 3.3. Finally, we discuss the theory and complexity of our algorithms in Section 3.4.

Related contributions: This chapter contributes to the following conference and journal publications.

- Chenhan D Yu, James Levitt, Severin Reiz, and George Biros. Geometry-oblivious FMM for compressing dense SPD matrices. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 53. ACM, 2017

- Chenhan D Yu, William B March, and George Biros. An n log n parallel fast direct solver for kernel matrices. In *Parallel and Distributed Processing Symposium (IPDPS), 2017 IEEE International*, pages 886–896. IEEE, 2017

- Chenhan D. Yu, William B. March, Bo Xiao, and George Biros. INV-ASKIT: a parallel fast direct solver for kernel matrices. In *Proceedings of the IPDPS16*, Chicago, USA, May 2016

- William B. March, Bo Xiao, Chenhan D. Yu, and George Biros. Askit: An efficient, parallel library for high-dimensional kernel summations. *SIAM Journal on Scientific Computing*, 38(5):S720–S749, 2016

Task	Operations	flops
SPLI(α)	split node α into l and r using Algorithm 2.2;	$\lvert\alpha\rvert$
ANN(α)	update $\mathcal{N}(\alpha)$ with Algorithm 2.3;	m^2
FindNear(α)	Near(β) = {MortonID(i) : $\forall i \in \mathcal{N}(\beta)$};	κm mops

FindFar(β,α)	**if** $\alpha \cap Near(\beta) \neq \phi$ using MortonID **then**
	  FindFar(β,l); FindFar(β,r);
	else if MortonID(β) > MortonID(α) **then**
	  Far(β) = Far(β) $\cup \alpha$;
		κm mops
		-
		$\lvert$Far(β)$\rvert$ mops

MergeFar(α)	Far(α) = Far(l) $\cap$ Far(r); Far(l) = Far(l)\Far(α); Far(r) = Far(r)\Far(α);	$\approx \lvert$Far(α)$\rvert$ $\lvert$Far(α)$\rvert$ $\lvert$Far(α)$\rvert$
SKEL(α)	I' =ImportantSample($\mathcal{N}(\alpha)$); **if** α **is leaf then** $[R_{11}, R_{12}, \widetilde{\alpha}]$ = GEQP3($K_{I'\alpha}$); **else** $[R_{11}, R_{12}, \widetilde{\alpha}]$ = GEQP3($K_{I'[\widetilde{\mathtt{lr}}]}$);	$4s$ mops $\frac{8}{3}m^3$ $\frac{64}{3}s^3$
COEF(α)	**if** α **is leaf then** $P_{\widetilde{\alpha}\alpha}$ = TRSM(R_{11},R_{12}); **else** $P_{\widetilde{\alpha}[\widetilde{\mathtt{lr}}]}$ = TRSM(R_{11}, R_{12});	$\frac{1}{2}s^2m$ s^3
N2S(α)	**if** α is leaf **then** $\widetilde{w}_\alpha = P_{\widetilde{\alpha}\alpha} w_\alpha$; **else** $\widetilde{w}_\alpha = P_{\widetilde{\alpha}[\widetilde{\mathtt{lr}}]}[\widetilde{w}_\mathtt{l}; \widetilde{w}_\mathtt{r}]$;	$2msr$ $2s^2r$
S2S(β)	$\widetilde{u}_\beta = \sum_{\alpha \in \mathtt{Far}(\beta)} K_{\widetilde{\beta}\widetilde{\alpha}} \widetilde{w}_\alpha$;	$2s^2r\lvert$Far(β)$\rvert$
S2N(β)	**if** α **is leaf then** $u_\beta = P^T_{\widetilde{\beta}\beta} \widetilde{u}_\beta$; **else** $[\widetilde{u}_\mathtt{l}; \widetilde{u}_\mathtt{r}] += P^T_{\widetilde{\beta}[\widetilde{\mathtt{lr}}]} \widetilde{u}_\beta$;	$2msr$ $2s^2r$
L2L(β)	$u_\beta += \sum_{\alpha \in \mathtt{Near}(\beta)} K_{\beta\alpha} w_\alpha$;	$2m^2r\lvert$Near(β)$\rvert$

Table 3.1: Tasks and their costs in floating point operations (flops) or memory operations (mops): SPLI (tree splitting), ANN (all nearest-neighbors), FindNear, FindFar, MergeFar, SKEL (skeletonization), and COEF (interpolation) occur in the compression phase. Interactions N2S (nodes to skeletons), S2S (skeletons to skeletons), S2N (skeletons to nodes), and L2L (leaves to leaves) occur in the evaluation phase (MATVEC).

3.1 Algebraic Fast Multipole Methods

In this section, we introduce GOFMM as a purely algebraic variant of Fast Multipole Methods (FMM). Algorithm 3.1 compresses a generic SPD matrix K to an $\mathcal{H}$-matrix $\widetilde{K}$ with $\mathcal{O}(N \log N)$ work. After the compression, MATVEC (Algorithm 3.2) with $\widetilde{K}$ take as little as $\mathcal{O}(N)$ work depending on the sparsity. The only required input to GOFMM is a routine $K(I, J)$ that returns a submatrix K_{IJ}, for arbitrary row and column

Algorithm 3.1 `Compress`(K)

1: `[Preorder]` **for each node** α **do** `SPLI`(α)`;` `/* metric tree partition */`
2: $\mathcal{N}$ =`RadonmizedAllNearestNeighbors();` `/* Algorithm 2.5 */`
3: `[Anyorder]` **for each leaf node** β **do** `FindNear`(β)`;` `/* sparse correction */`
4: `[Anyorder]` **for each leaf node** β **do** `FindFar`(β`,root);` `/* pruning */`
5: `[Postorder]` **for each node** α **do** `MergeFar`(α);
6: `[Postorder]` **for each node** α **do** `SKEL`(α);
7: `[Anyorder]` **for each node** α **do** `COEF`(α)`;` `/* interpolation coefficients */`

index sets I and J.

Tree traversals: We use **three types of tree traversals** to describe the algorithms in `GOFMM`: `postorder`, `preorder`, and `anyorder`. While different tasks in our algorithms **read**/**write** different tree nodes; tasks with data dependencies from children to parents result in `postorder` traversals, tasks with dependencies from parents to children result in `preorder` traversals; independent tasks result in `anyorder` traversals. Here, by **"task"** we refer to a computation that occurs when we visit a tree node during a traversal. We list all tasks required by the **compression** phase (Algorithm 3.1) and **evaluation** phase (Algorithm 3.2) in Table 3.1.

Compression: As described in Section 2.4, `GOFMM` **compression** starts by creating the binary metric tree with Algorithm 3.1 that represents the binary partitioning (and encodes a symmetric permutation of matrix K). This requires *distance* d_{ij} and a `preorder` traversal of the first task `SPLI`(α) in Table 3.1. `MetricTreeSplit()` in Algorithm 2.2 approximately splits the two farthest indices into two children. Once the metric tree is built, we compute the approximate neighbors for each matrix index iteratively with `RandomizedAllNearestNeighbors()` in Algorithm 2.5.

Interaction lists: `GOFMM` requires that every tree node maintains three lists. For a tree node α, these lists are the node neighbor list $\mathcal{N}(\alpha)$, near interaction list `Near`(α), and far interaction list `Far`(α). Node neighbor lists are used to construct near interaction lists and perform important sampling. Near interaction lists record all submatrices that cannot be approximated. Far interaction lists record all low-rank

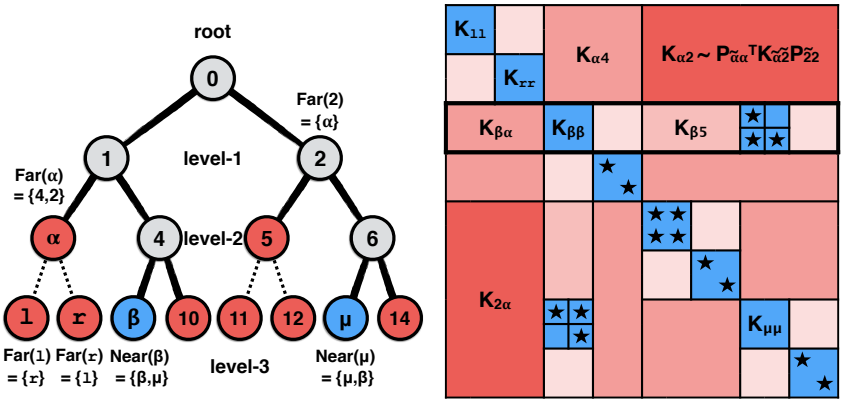

Figure 3.1: A partitioning tree (left) and corresponding hierarchically low-rank plus sparse matrix (right). The off-diagonal blocks are combinations of low-rank matrices (pink) and sparse matrices (blue). The ★ symbol denotes an entry that cannot be approximated (because the corresponding interaction is between neighbors). The solid edges in the tree mark the path traversed by `FindFar`(β,`root`). Since $K_{\beta\alpha}$ does not contain any neighbor interactions (★), this traversal adds α to `Far`(β). In this example, `FindFar(l,root)` computes `Far(l)` $= \{\texttt{r}, 4, 2\}$, and `FindFar(r,root)` computes `Far(r)` $= \{\texttt{l}, 4, 2\}$. `MergeFar`(α) then moves `Far(l)` $\cap\, Far($`r`$)$ into `Far`(α) so that `Far`(α) $= \{4, 2\}$, `Far(l)` $= \{\texttt{r}\}$ and `Far(r)` $= \{\texttt{l}\}$.

submatrices that can be approximated.

Morton index: In order to compute these lists fast, we store the Morton index of each tree node and matrix index. The Morton index is a bit array with a fixed length that encodes the path from the root to a tree node or index i. We use `MortonID(`α`)` and `MortonID(`i`)` to denote the Morton indices of a tree node α and matrix index i. Let `0` and `1` be the left and right path, the path to reach leaf node μ in Figure 3.1 is `110`. The path to reach internal node α is `000`. To differ nodes with the same path encoding, the depth of the tree node is also encoded. For example,

$$\begin{aligned} \texttt{MortonID}(\alpha) &= \texttt{000 010}, \text{ and} \\ \texttt{MortonID}(\texttt{l}) &= \texttt{000 011} \end{aligned} \tag{3.3}$$

in Figure 3.1 are distinguished by their depths `010` and `011`. The Morton index of an

index i is the Morton index of the leaf node that contains it. With Morton indices, checking whether a target tree node α or index i is the descendant of β can be done in $\mathcal{O}(1)$ work by computing the `XOR` value between their shifted Morton indices.

Node neighbor list $\mathcal{N}(\alpha)$: Recall that we first construct a list of κ nearest-neighbor for each index $i \in \alpha$ iteratively using a greedy search described in Algorithm 2.5. Then for each leaf node α, we construct $\mathcal{N}(\alpha)$ by merging all neighbors of $i \in \alpha$. For internal tree nodes, the list is merged from both children recursively [53]. To prevent the length of $\mathcal{N}(\alpha)$ from growing exponentially during the merging process, $\mathcal{N}(\alpha)$ is truncated to $\mathcal{O}(s)$, where s is a user-defined maximum rank of the low-rank approximation. To be precise, the merged lists are first sorted according to their keys (distances) and truncated to length $4s$. These $4s$ neighbors serve as important samples during the randomized low-rank compression step (skeletonization).

Near interaction list `Near`(α): Leaf nodes α and β are considered near if $\alpha \cap \mathcal{N}(\beta)$ is nonempty (i.e., $K_{\alpha\beta}$ contains at least one neighbor ($\bigstar$) in Figure 3.1). The `Near` interaction list is defined only for leaf nodes and contains only leaf nodes. For each leaf node β, `Near`(β) is constructed using task `FindNear`(β) in Table 3.1. For each neighbor $i \in \mathcal{N}(\beta)$, `FindNear`(β) adds `MortonID`(i) to `Near`(β). For each node $\alpha \in$`Near`(β), submatrix $K_{\beta\alpha}$ will be evaluated directly without compression.

Notice that the length of `Near`(β) determines the number of direct evaluations (number of blue blocks in Figure 3.1) in the off-diagonal blocks. To prevent the cost from growing too fast, we introduce a user-defined parameter **budget** b such that

$$|\texttt{Near}(\beta)| < b \times (N/m). \tag{3.4}$$

While looping over neighbor $i \in \mathcal{N}(\beta)$, instead of directly adding `MortonID`(i) to `Near`(β), we only mark it with a ballot. We insert candidates to `Near`(β) according to their votes until Equation 3.4 is reached. To enforce symmetry of $\widetilde{K}$, we loop over all lists and enforce the following: if $\alpha \in$ `Near`(β) then $\beta \in$ `Near`(α).

Far interaction list `Far`(α): Hierarchical tree partitioning provides a systematic way of viewing a matrix in different resolutions. For example,

$$K = \begin{bmatrix} K_{\alpha\alpha} & K_{\alpha\beta} \\ K_{\beta\alpha} & K_{\beta\beta} \end{bmatrix} = \begin{bmatrix} \begin{bmatrix} K_{\mathtt{ll}} & K_{\mathtt{lr}} \\ K_{\mathtt{rl}} & K_{\mathtt{rr}} \end{bmatrix} & \begin{bmatrix} K_{\mathtt{l}\beta} \\ K_{\mathtt{r}\beta} \end{bmatrix} \\ \begin{bmatrix} K_{\beta\mathtt{l}} & K_{\beta\mathtt{r}} \end{bmatrix} & \begin{bmatrix} K_{\beta\beta} \end{bmatrix} \end{bmatrix} \tag{3.5}$$

are different matrix views of K from coarse- to fine-grain. In terms of matrix approximation, *treecodes only zoom into the children levels with a higher resolution (repartition to increase matrix rank or sparsity) if the coarse-grain approximation fails to provide acceptable accuracy.* The `Far`(α) interaction lists record all low-rank interactions between node α and other tree nodes in different levels.

`Far`(α) is constructed in two steps in Algorithm 3.1, representing submatrices in the off-diagonal blocks that can be compressed. First, for each leaf node β, we invoke `FindFar`(β, `root`) in Table 3.1. Upon visiting node α, we check whether α is a parent of any leaf node in `Near`(β) by computing the `XOR` value between `MortonID`(α) and all Morton indices in `Near`(α). If so, we recurse to the two children of α; otherwise, we add `MortonID`(α) to `Far`(β) (i.e. $K_{\beta\alpha}$ can be approximated).

The second step is a `postorder` traversal on task `MergeFar` in Table 3.1. This process merges the common tree nodes from two children lists `Far`(`l`) and `Far`(`r`) to create larger off-diagonal blocks for approximation. These common tree nodes are removed from the children and inserted to their parent list `Far`(α). In Figure 3.1, tasks `FindFar` can be identified by the smallest square pink blocks, and `MergeFar` merges small pink blocks into larger blocks.

The skeletonization process described in Section 2.3 is separated into two tasks in Table 3.1 for each tree node α. Task `SKEL`(α) selects skeleton $\widetilde{\alpha}$ (in the critical path), and task `COEF`(α) computes the coefficients $P_{\widetilde{\alpha}[\widetilde{\mathtt{lr}}]}$. Notice that in Algorithm 3.1 only task `SKEL`(α) needs to be executed in `postorder`, but `COEF`(α)

Algorithm 3.2 `Evaluate`(u,w)

1: `[Postorder]` **for each node** α **do** `N2S`(α)`;`
2: `[Anyorder]` **for each node** β **do** `S2S`(β)`;`
3: `[Preorder]` **for each node** β **do** `S2N`(β)`;`
4: `[Anyorder]` **for each leaf node** β **do** `L2L`(β)`;`

can be in `anyorder` as long as `SKEL`(α) is finished. Such parallelism can only be specified at the task level, which later inspires our *out-of-order* task-based parallelism.

Evaluation: Following `ASKIT-FMM` [53], a semi-algebraic `FMM`, we present in Algorithm 3.2 a four-step process for computing the fast `MATVEC` with $\widetilde{K}$. The idea is to approximate each submatrix `MATVEC` $u_\beta += K_{\beta\alpha} w_\alpha$ in far interaction list `Far`(β) using a two-sided `ID` described in Section 2.3 such that

$$K_{\beta\alpha} w_\alpha \approx P_{\widetilde{\beta}\beta}^T K_{\widetilde{\beta}\widetilde{\alpha}} P_{\widetilde{\alpha}\alpha} w_\alpha, \tag{3.6}$$

where $P_{\widetilde{\alpha}\alpha}$ and $P_{\widetilde{\beta}\beta}$ are the interpolative coefficient matrices. That is, columns of $K_{\beta\alpha}$ are interpolated by multiplying the basis $K_{\widetilde{\beta}\widetilde{\alpha}}$ with the interpolative coefficients $P_{\widetilde{\alpha}\alpha}$. Similarly, rows of $K_{\beta\alpha}$ are interpolated by multiplying $P_{\widetilde{\beta}\beta}^T$. In Algorithm 3.2, Equation 3.6 is computed by three tree traversals with tasks `N2S` (nodes to skeletons), `S2S` (skeletons to skeletons), and `S2N` (skeletons to nodes). All direct evaluations are computed independently by task `L2L` (leaves to leaves).

Upward telescoping: Recall that the interpolative decomposition (`ID`) we computed for a tree node α in Equation 2.7 is nested ($\widetilde{\alpha} \subset \widetilde{\mathtt{l}} \cap \widetilde{\mathtt{r}}$). Consequently, the coefficient matrix has a *telescoping* expression

$$P_{\widetilde{\alpha}\alpha} = P_{\widetilde{\alpha}[\widetilde{\mathtt{l}}\widetilde{\mathtt{r}}]} \begin{bmatrix} P_{\widetilde{\mathtt{l}}\mathtt{l}} & \\ & P_{\widetilde{\mathtt{r}}\mathtt{r}} \end{bmatrix} \tag{3.7}$$

to be exploited in the evaluation phase (`MATVEC`). By replacing $P_{\widetilde{\alpha}\alpha} w_\alpha$ in Equation 3.6

with its telescoping expansion, we get a recursive relationship

$$\widetilde{w}_\alpha = P_{\widetilde{\alpha}\alpha} w_\alpha = P_{\widetilde{\alpha}[\widetilde{\mathtt{l}}\widetilde{\mathtt{r}}]} \begin{bmatrix} P_{\widetilde{\mathtt{l}}\mathtt{l}} & \\ & P_{\widetilde{\mathtt{r}}\mathtt{r}} \end{bmatrix} \begin{bmatrix} w_\mathtt{l} \\ w_\mathtt{r} \end{bmatrix} = P_{\widetilde{\alpha}[\widetilde{\mathtt{l}}\widetilde{\mathtt{r}}]} \begin{bmatrix} \widetilde{w}_\mathtt{l} \\ \widetilde{w}_\mathtt{r} \end{bmatrix} \tag{3.8}$$

between the *skeleton weights* $\widetilde{w}$ (interpolated weights). Due to the dependencies, $\widetilde{w}_\alpha$ must be evaluated after the evaluation of $w_{\widetilde{\mathtt{l}}}$ and $w_{\widetilde{\mathtt{r}}}$ with a `postorder` traversal.

In Algorithm 3.2, these skeleton weights are computed by task `N2S(`α`)` (nodes to skeletons) described in Table 3.1. Task `N2S(`α`)` computes $\widetilde{w}_\alpha = P_{\widetilde{\alpha}\alpha} w_\alpha$ for each leaf node, and $\widetilde{w}_\alpha = P_{\widetilde{\alpha}[\widetilde{\mathtt{l}}\widetilde{\mathtt{r}}]}[\widetilde{w}_\mathtt{l}; \widetilde{w}_\mathtt{r}]$ for each internal node. Recall that in the compression phase, we have computed $P_{\widetilde{\alpha}\alpha}$ for each leaf node and $P_{\widetilde{\alpha}[\widetilde{\mathtt{l}}\widetilde{\mathtt{r}}]}$ for each internal node with task `COEF(`α`)`.

Lump sum: The second step of the evaluation phase (Algorithm 3.2) is to multiply the *skeleton basis* $K_{\widetilde{\beta}\widetilde{\alpha}}$ in Equation 3.6 with skeleton weights $\widetilde{w}_\alpha$ and accumulate this partial contribution to *skeleton potentials* $\widetilde{u}$ for each node:

$$\widetilde{u}_\beta = \sum_{\alpha \in Far(\beta)} K_{\widetilde{\beta}\widetilde{\alpha}} \widetilde{w}_\alpha. \tag{3.9}$$

In Algorithm 3.2, this lump sum is computed by task `S2S(`β`)` (skeletons to skeletons) in Table 3.1. Notice that the computation above depends on all skeleton weights $\widetilde{w}_\alpha$ in the far interaction lists `Far(`β`)`. As soon as $\widetilde{w}_\alpha$ is computed in task `N2S(`α`)`, task `S2S(`α`)` can be executed in any order.

Downward telescoping: The third step of the evaluation phase is to perform interpolation on the left by multiplying $P^T_{\widetilde{\beta}\beta}$ in Equation 3.6. By replacing $P^T_{\widetilde{\beta}\beta}$ with its telescoping expansion, task `S2N(`β`)` (skeletons to nodes) accumulates

$$\begin{bmatrix} \widetilde{u}_\mathtt{l} \\ \widetilde{u}_\mathtt{r} \end{bmatrix} += P^T_{\widetilde{\beta}[\widetilde{\mathtt{l}}\widetilde{\mathtt{r}}]} \widetilde{u}_\beta \tag{3.10}$$

to its children. In the leaf node, task `S2N`(β) accumulates $u_\beta = P_{\tilde{\beta}\beta}^T \widetilde{u}_\beta$ directly to the output. Together with task `N2S` and `S2S`, these three tasks compute all `MATVEC`s for the *far* interactions (pink blocks in Figure 3.1). All `MATVEC`s with $K_{\beta\alpha}$ recorded in near interaction list `Near`(β) (blue blocks) are computed by `L2L`(β) (leaves to leaves) and directly accumulated to output u_β. The only possible race condition is the concurrent write on u_β between task `L2L` and `S2N`. Once the concurrent write is resolved (by duplication, mutex, or atomic operations), `L2L` tasks are independent from the other tasks.

3.2 Parallel Treecodes

`GOFMM` is parallelized using `MPI+OpenMP`, following the algorithms for distributed-memory treecodes described in [55] and the task-based runtime scheduler design in [82]. The latter is generalized to perform out-of-order execution in the distributed-memory setting. We mainly discuss how distributed processes communicate in the compression and evaluation phase. More implementation details regarding the runtime system and dependency analysis are presented in Section 4.1.

Given p processes, we partition the matrix indices using a distributed binary tree, where the root node contains all indices (see Figure 3.2). Computations involving distributed tree nodes above level-$\log p$ requires load-balancing and communication. Both computation and communication are described as tasks with dependencies. Tasks satisfying all dependencies are dispatched according to EFT (Earliest Finish Time) policy. We discuss scheduling details in Section 4.1. *Out-of-order* execution allows us to systematically maintain load-balance and overlap communication with computation. In the following, we summarize the notation that we use to express parallelism and summarize the distributed **compression** phase (Algorithm 3.4) and distributed **evaluation** phase (Algorithm 3.7).

Data distribution: `GOFMM` permutes the matrix indices to expose low-rank

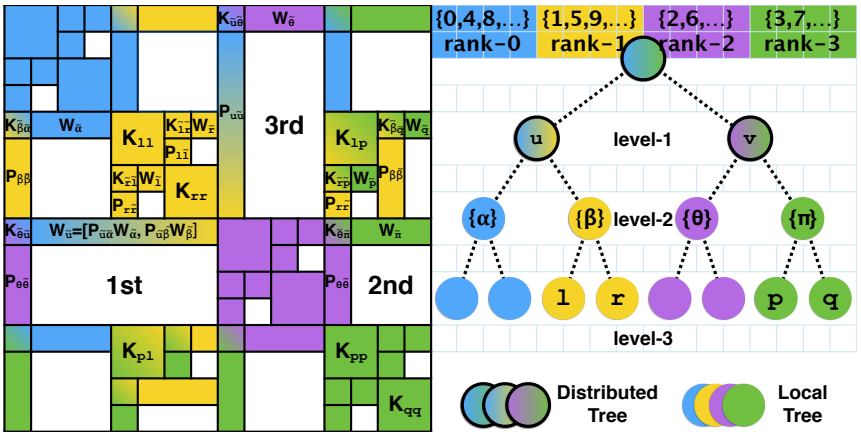

Figure 3.2: A 4-processes distributed metric tree (right), and its algebraic FMM compression (left): each color represents an `MPI` process. Computation on nodes and factors below level-log(p) (with single color) are local. Mixed-colored nodes and factors above level-log(p) require communication.

matrix structure. This is equivalent to permuting and redistributing the ownership of matrix indices among `MPI` processes. Since we do not have control over the distribution of K, we only repartition the ownership of the N column/row indices among an array of p `MPI` processes. Depending on the application, it may be possible (and desirable) to repartition data that are required for the evaluation of K_{ij}. For this reason, we provide a repartitioning interface that assumes the user provides an array with fixed-size memory per matrix index. This is conceptually equivalent to redistributing points although the data structure can contain anything. `GOFMM` only invokes an opaque callback function $K(i, j)$ without accessing the content in the data structure directly.

To describe our algorithms, we adopt a notation inspired by [64], which is used by the library `Elemental` (`http://libelemental.org/`). We use five possible index partitioning (with possible replications) among `MPI` processes. We use `[*]` or `[STAR]` to denote that every process has a local copy of all indices and their corresponding data. When the indices and data are solely owned by one unique processes, we use

Algorithm 3.3 `DistSPLI`(α)

if `Size`(α)$= 1$ **then** `SPLI`(α)`;` **return**; `/* in Table 3.1 */`
else `/** Above level-log(p) in the distributed tree */`

case `Rank`(α) $< \frac{1}{2}$`Size`(α) `:`	**case** `Rank`(α) $\geq \frac{1}{2}$`Size`(α) `:`
`CommSplit(Comm(`α`),Comm(l));`	`CommSplit(Comm(`α`),Comm(r));`
`[l,`$\bar{\texttt{l}}$`]=MetricTreeSplit(`α`);`	`[`$\bar{\texttt{r}}$`,r]=MetricTreeSplit(`α`);`
`partner` $=$ `Rank`(α) $+ \frac{1}{2}$`Size`(α)`;`	`partner` $=$ `Rank`(α) $- \frac{1}{2}$`Size`(α)`;`
`Sendrecv(`$\bar{\texttt{l}}$`,` $\bar{\texttt{r}}$`, partner);`	`Sendrecv(`$\bar{\texttt{r}}$`,` $\bar{\texttt{l}}$`, partner);`
`SPLI(l` $\cup$ $\bar{\texttt{r}}$`);`	`SPLI(r` $\cup$ $\bar{\texttt{l}}$`);`

the `[CIRC]` distribution. `[CYC]` denotes the 1D cyclic partitioning, which we assume to be the input partitioning. `[IDS]` (distributed tree partition) is the repartitioning (permutation) for `GOFMM`, `[LET]` (tree partition with local essential data dependencies) is a term that comes from classical N-body methods and is a superset of `[IDS]`, i.e., the data for every index can be replicated in multiple `MPI` processes.

Matrix distribution and redistribution: To define the partitioning of a matrix we need to define partitioning for both rows and columns. A distributed matrix is represented by a pair of the distribution above. For example, nearest neighbor pairs $\mathcal{N}$ are stored as a dense $\kappa \times N$ matrix where columns are distributed among processes without replication. We use $\mathcal{N}$`[*,CYC]`$\in \mathbb{R}^{\kappa \times N}$ to denote that row indices $(1:\kappa)$ are duplicated on each process but the column indices are distributed with a 1D cyclic partitioning. Similarly, we use $\mathcal{N}$`[*,IDS]` to denote the distribution of neighbors according to the metric tree partitioning. In the following, we overload the `"="` operator to also denote redistribution. In most cases, we implement this redistribution using a sequence of collective all-to-all communication operators.

Metric tree partition: Algorithm 3.3 describes how the distributed metric tree shown in Figure 3.2 is formed. We assume that all indices were originally distributed in `[CYC]` (1D cyclic) order. The metric tree permutes them into `[IDS]` order (with no replication). At the end of the compression phase, each process will exchange the indices they need for **evaluation** (according to the interaction lists)

and each process ends up with the [LET] distribution (with replication).

Hierarchical communicators: Each distributed tree node above level-$\log(p)$ has a unique MPI communicator to facilitate collective communication. A node α is assigned with a local communicator Comm(α). We also use Size(α) and Rank(α) to denote the local MPI rank and size in Comm(α). *While different tasks may be assigned according to the local rank Rank(α), we describe our distributed algorithms side-by-side as shown in Algorithm 3.3. The left column shows algorithms for processes with Rank(α) $< \frac{1}{2}$Size(α) in Comm(α), and the right column represents the other group of processes. We use this algorithm format in the remainder of this chapter.*

While splitting a tree node α into l and r child, MPI processes are evenly divided into left and right sub communicators Comm(l) and Comm(r) according to Rank(α). Algorithm 2.2 is invoked to split the owned indices into [c,$\bar{\text{c}}$] using "geometry-oblivious" distance, where c can be left (l) or right child (r). The index set l is kept by processes with Rank(α) $< \frac{1}{2}$Size(α), and r is kept by the other group of the processes. $\bar{\text{l}}$ and $\bar{\text{r}}$ are exchanged between "partner" processes (defined by the variable partner in Algorithm 3.3). We continue using recurrence on the children nodes with the assigned sub-communicator. After reaching level-$\log(p)$ the splitting continues without communication until we reach the leaf level (i.e., $|\alpha| \leq m$).

All nearest-neighbor: At line 4 of Algorithm 3.4, we compute κ-nearest neighbors for each matrix index in parallel (using the same distance metric as Algorithm 3.3). Neighbors will be used for sampling and constructing the near interaction lists. Our distributed implementation extends the shared-memory implementation in Algorithm 2.5 and makes use of Algorithm 3.3. In each iteration, we create a randomized metric tree using geometric-oblivious distances, where the leaf node size is 4κ (larger than κ to reduce the number of tree iterations). Exhaustive neighbor search (task ANN in Table 3.1) is performed on all indices that belong to the same leaf node (by which generate κ candidates). Neighbor candidates are redistributed

Algorithm 3.4 `DistCompress(`K`)`

```
1: root= {1, ..., N}[CYC];          /* initialized with 1D cyclic distribution */
2: [preorder] for each α ∈ owned_nodes do
3:    DistSPLI(α);                                          /* Algorithm 3.3 */
4: N[*,CYC] = AllNearestNeighbors(K);
5: N[*,IDS] = N[*,CYC];                          /* redistribute neighbors */
6: [postorder] for each α ∈ owned_nodes do
7:    DistSKEL(α);                                          /* Algorithm 3.5 */
8: [anyorder] for each α ∈ owned_nodes do
9:    COEF(α);                                                /* Table 3.1 */
10: [anyorder] for each α ∈ owned_leafs then
11:   FindNear(α);                          /* build near interaction lists */
12: for each α ∈ owned_leafs and β ∈ Near(α) do
13:   sbuff[Owner(β)] ∩= pair(β, α)    /* pack near interactions into pairs */
14: Alltoallv(sbuff, rbuff);          /* exchange all near interaction pairs */
15: for each pair(β, α) ∈ rbuff do
16:   β_near ∩= {α};                          /* symmetrize near interactions */
17: [postorder] for each α ∈ owned_nodes do
18:   DistMergeFar(α);        /* build far interaction lists (Algorithm 3.6) */
19: for each α ∈owned_nodes and β ∈ α_far do
20:   sbuff[Owner(β)] ∩= pair(β, α)     /* pack far interactions into pairs */
21: Alltoallv(sbuff, rbuff);          /* exchange all far interaction pairs */
22: for each pair(β, α) ∈ rbuff do
23:   Far(β) ∩= {α};                            /* symmetrize far interactions */
```

Algorithm 3.5 `DistSKEL(`α`)`

```
if Rank(α)= 0 then I =ImportantSample(N[*,α]\α);
if Size(α)= 1 then SKEL(α); return;
else /** Above level-log(p) in the tree */
```

`if Rank(`α`)= 0 then`	`if Rank(`α`)=` $\frac{1}{2}$`Size(`α`) then`
`  Recv(`$\widetilde{\mathbf{r}}$`);`	`  Send(`$\widetilde{\mathbf{r}}$`);`
`  SKEL(`α`);`	`  NOP;`

from `[IDS]` (randomized tree distribution) back to `[CYC]` (1D cyclic) distribution. Finally, we merge existing neighbors with the new candidates, trim the list down to κ neighbors again (Algorithm 2.4), and iterate until we converge.

Distributed `GOFMM` compression (Algorithm 3.4): With a metric tree and neighbors $\mathcal{N}$ we first redistribute $\mathcal{N}$ from `[CYC]` to `[IDS]` distribution. As a

result, we can access neighbors of a local node α as $\mathcal{N}$[*,IDS](:,α). Neighbors are used as importance samples in skeletonization (Algorithm 3.5) to improve the quality of the low-rank compression; neighbors are also used to construct the near interaction lists in Algorithm 3.4 at 11.

Skeletonization: Algorithm 3.5 compresses the off-diagonal blocks of K using a nested ID described in Section 2.3. This recursive process greedily subselects skeleton indices $\widetilde{\alpha}$ for node α from children's skeleton pivots $\mathtt{J} = \widetilde{\mathtt{l}} \cup \widetilde{\mathtt{r}}$ (thus called "nested"). Recall that skeletons $\widetilde{\alpha}$ are selected to be the first s column pivots of GEQP3, and row indices I are sampled from $\mathcal{N}(\alpha) \setminus \alpha$. This process requires gathering $\widetilde{\mathtt{r}}$ from its sibling (Rank(α)=Size(α)/2). In a distributed environment, we evaluate submatrix K_{IJ} on rank 0. Note that the coefficient matrix P is computed locally with task COEF in any order after the skeletonization is completed.

Symmetric interaction lists: If we stop at skeletonization, we obtain a hierarchical semi-separable (HSS) approximation of K, where no sparse correction is introduced in the off-diagonal blocks. To create a symmetric FMM approximation, we need to further construct two interaction lists and redistribute indices from IDS to [LET] (local essential tree) distribution that satisfies all vital data dependencies required by the both near and far interaction lists.

Near interaction list Near(α) is constructed by task FindNear(α) in Table 3.1, and the length |Near(α)| is controlled by the user-defined budget b such that |Near(α)| $< b(N/m)$. Notice that κ-nearest neighbors are not necessarily symmetric. As a result, Near(α) may not be symmetric (i.e., $S_{\mathtt{lr}} \neq S_{\mathtt{rl}}^T$). We must ensure that if $\beta \in$Near(α), then $\alpha \in$Near(β). That is, for each node α owned by the current MPI rank, we must send a pair (α, β) to the MPI rank that owns β (denoted as Owner(β). Line 12 to line 16 in Algorithm 3.4 symmetrize the interaction lists. We pack all pairs to p messages by traversing all near interaction lists and invoke Alltoallv() calls to exchange and symmetrize the near interaction lists.

Algorithm 3.6 `DistMergeFar(`α`)`

```
if α isleaf then return FindFar(root,α);
if Size(α)= 1 then MergeFar(α); return                    /* Table 3.1 */
else /** Above level-log(p) in the tree */
```

if `Rank(`α`)= 0` **then**	**if** `Rank(`α`)=` $\frac{1}{2}$`Size(`α`)` **then**
`Sendrecv(Far(l),Far(r),Comm(`α`))`	`Sendrecv(Far(r),Far(l),Comm(`α`))`
`Far(`α`) = Far(l)` $\cap$ `Far(r)`	`Far(`α`) = Far(l)` $\cap$ `Far(r)`
`Far(l) = Far(l)`$\backslash$`Far(`α`)`	`Far(r) = Far(r)`$\backslash$`Far(`α`)`

Far interaction list `Far(`α`)` is constructed with *two* stages in Algorithm 3.4 (line 19 to line 23), which we also symmetrize using `Alltoallv()` calls. The *first* stage computes `Far` interaction lists for all leaf nodes by several top-down `preorder` traversals. The *second* stage computes `Far` interaction lists for all internal nodes by recursively merging children's `Far` interaction lists upward.

We first review `FindFar(`β`,`α`)` in Table 3.1, which we use to construct `Far(`α`)` for a leaf node α by partially traversing the metric tree according to its near interaction list `Near(`α`)`. Then we describe the distributed merging process in Algorithm 3.6, which creates `Far` interaction lists for internal nodes. This merging process results in larger low-rank off-diagonal blocks in Figure 3.2.

`FindFar(`β`,`α`)` is called in Algorithm 3.6 to construct `Far(`α`)`. Throughout the recursion, argument α does not change, but β changes according to a `preorder` ordering of the tree nodes. The recursion continues to the left and right child (`l` and `r`) if β is an ancestor of any leaf node in `Near(`α`)`. This condition is checked by comparing `MortonID(`β`)` with {`MortonID(i)`|`i` $\in$`Near(`α`)`} in $\mathcal{O}(|$`Near(`α`)`$|)$. Otherwise, the recursion terminates, and we append β to `Far(`α`)` if `MortonID(`β`)` is larger. As a result, we only preserve far interaction pairs in the upper triangular part. Later in Algorithm 3.6, we merge these lists to form larger off-diagonal blocks in Figure 3.2. The lower triangular part is reflected and taken care of by the symmetrization.

The merging process involves point-to-point communication while visiting the

Algorithm 3.7 u[IDS,*]=DistEvaluate(w[IDS,*])

```
1: [anyorder] for each owned_leaf_node α and β ∈ Near(α) do
2:    sbuff[Owner(β)] ∩= β;
3: Alltoallv(sbuff,rbuff);
4: for each rank p and α ∈ rbuff[p] do
5:    sbuff[p] ∩= w_α;
6: Alltoallv(sbuff,rbuff);
7: for each rank p and β ∈ rbuff[p] do
8:    w_β =rbuff[p][β];
9: [anyorder] for each leaf α do
10:   L2L(α);
11: [postorder] for each owned_node α do
12:   DistN2S(root,w);
13: for each α and β ∈ Far(α) do
14:   sbuff[Owner(β)] ∩= β;
15: Alltoallv(sbuff,rbuff);
16: for each rank p and α ∈ rbuff[p] do
17:   sbuff[p] ∩= w̃_α;
18: Alltoallv(sbuff,rbuff);
19: for each rank p and β ∈ rbuff[p] do
20:   w̃_β =r[p][β];
21: [anyorder] for each α ∈ owned_nodes do
22:   S2S(α);
23: [preorder] for each α ∈ owned_nodes do
24:   DistS2N(root,u);
25: return u[IDS,*];
```

distributed tree nodes in the postorder traversal. While α is distributed, `Far(l)` is owned by rank `0` in `Comm(`α`)`, `Far(l)` is stored on the process with rank `Size(`α`)/2`. These two `MPI` ranks exchange their interaction lists, performing set interaction and update `Far(l)` and `Far(r)`. `Far(`α`)` is stored on rank `0`, which handles all computation for the distributed tree node α during the evaluation.

Evaluation phase: `DistEvaluate` (Algorithm 3.7) computes the `MATVEC` of $\widetilde{K}$ with a distributed vector w ("the weights"). Given weights w`[IDS]` $\in \mathbb{R}^{N\times r}$ (redistributed from its initial distribution `[CYC]`), `MATVEC` of $\widetilde{K}$ involves four steps: `L2L`, `N2S`, `S2S`, and `S2N` listed in Table 3.1. In tasks `N2S` and `S2N`, computation

Algorithm 3.8 `DistN2S`(α)

if `Size`(α)$=1$ **then** `N2S`(α); `/* Table 3.1 */`
else `/** Above level-log(p) in the tree */`

if `Rank`(α)$=0$ **then**	**if** `Rank`(α)=`Size`(α)`/2` **then**
`Recv`($\widetilde{w}_{\mathtt{r}}$);	`Send`($\widetilde{w}_{\mathtt{r}}$);
`N2S`(α)	`NOP`;

Algorithm 3.9 `DistS2N`(α)

if `Size`(α)$=1$ **then** `S2N`(α); `/* Table 3.1 */`
else `/** Above level-log(p) in the tree */`

if `Rank`(α)$=0$ **then**	**if** `Rank`(α)=`Size`(α)`/2` **then**
$[\widetilde{u}_{\mathtt{l}}^{'}; \widetilde{u}_{\mathtt{r}}^{'}] = P^{T}_{\widetilde{\alpha}[\widetilde{\mathtt{l}}\widetilde{\mathtt{r}}]} \widetilde{u}_{\alpha}$;	`NOP`;
`Send`($\widetilde{u}_{\mathtt{r}}^{'}$);	`Recv`($\widetilde{u}_{\mathtt{r}}^{'}$);
$\widetilde{u}_{\mathtt{l}} += \widetilde{u}_{\mathtt{l}}^{'}$;	$\widetilde{u}_{\mathtt{r}} += \widetilde{u}_{\mathtt{r}}^{'}$;

on distributed tree nodes above level-log(p) requires point-to-point communication using local communicator. Their distributed versions are described in `DistN2S` (Algorithm 3.8) and `DistS2N` (Algorithm 3.9). Nevertheless tasks `L2L` and `S2S` on all tree nodes may also require all-to-all communication (line 3, 6, 15, and 18) to redistribute w and intermediate results $\widetilde{w}$ from `[IDS]` to `[LET]` distribution. These `Alltoallv()` calls not only scale poorly on distributed systems but also prevent out-of-order execution between these four steps. We use Figure 3.2 to illustrate the data dependencies that result from the different interaction lists. We then discuss how our runtime system leverages asynchronous communication and distributed data dependencies.

Consider the first submatrix (blue) from Figure 3.2 to understand how skeleton weights $\widetilde{w}$ are computed in task `DistN2S` (Algorithm 3.8). $\widetilde{w}_{\alpha}$ and $P_{\widetilde{\mathtt{u}}[\widetilde{\alpha}\widetilde{\beta}]}$ are computed and owned by process blue, computing $\widetilde{w}_{\mathtt{u}}$ on process blue requires $\widetilde{w}_{\beta}$ from the right child (which is on process green). In Algorithm 3.8, $\widetilde{w}$ is sent from `MPI` rank `Size`(α)`/2` to rank `0` using `Comm`(α).

Together with the second submatrix (green) in Figure 3.2, task `S2S`(θ) requires

$\widetilde{w}_\theta$ on process blue and $\widetilde{w}_\pi$ on process green to compute $\widetilde{u}_\theta$ on process purple. That is, the owner process of θ gathers all skeleton weights in $\widetilde{\theta}$ before executing task `S2S(`θ`)`. To satisfy all dependencies resulted from the far interaction lists, we pack and exchange all skeleton weights from line 15 to 18 in Algorithm 3.7. Similarly, task `L2L` requires all processes to gather corresponding weights in the near interaction lists. These distributed data dependencies are handled similarly from line 3 to line 6.

Finally, let us consider the third submatrix (purple) in Figure 3.2 to explain `DistS2N(u)` (Algorithm 3.9). While $P^T_{\widetilde{\mathtt{u}}[\widetilde{\alpha}\widetilde{\beta}]}\widetilde{u}_{\mathtt{u}}$ is computed by process blue, updating $\widetilde{u}_\beta$ (yellow) requires sending $\widetilde{u}'_\beta$ from `MPI` rank `0` to rank `Size(`α`)/2` using `Comm(`α`)`. Once reaching level-$\log p$, no other communication is required. While reaching the leaf node, $P^T_{\widetilde{\alpha}\alpha}\widetilde{u}_\alpha$ is accumulated with the direct evaluation u_α, which is computed by task `L2L(`α`)`.

3.3 Parallel $\mathcal{H}$-matrix Factorization

When K admits a pure hierarchical low-rank approximation (with $S_{\mathtt{l}} = 0$ and $S_{\mathtt{r}} = 0$ in Equation 1.2), then we can efficiently approximate $K^{-1}_{\alpha\alpha}$ using the Sherman-Morrison-Woodbury formula along with recursion:

$$\begin{aligned}
\widetilde{K}_{\alpha\alpha} &= D_\alpha + U_\alpha V_\alpha = D_\alpha(I + W_\alpha V_\alpha), \text{ and } W_\alpha = D_\alpha^{-1}U_\alpha. \\
\widetilde{K}^{-1}_{\alpha\alpha} &= (I + W_\alpha V_\alpha)^{-1} D_\alpha^{-1} \\
&= (I - W_\alpha (I + V_\alpha W_\alpha)^{-1} V_\alpha) D_\alpha^{-1} \\
&= (I - W_\alpha Z_\alpha^{-1} V_\alpha) D_\alpha^{-1}, \text{ where } Z_\alpha = I + V_\alpha W_\alpha.
\end{aligned} \tag{3.11}$$

Recursion is used to decompose D_α. After obtaining the factorization of D_α, we compute $W_\alpha = D_\alpha^{-1}U_\alpha$ for a rank-$2s$ matrix U_α and factorize the smaller reduced system $Z_\alpha = I + V_\alpha W_\alpha \in \mathbb{R}^{2s\times 2s}$. The scheme can be easily extended to invert $\lambda I + K_{\alpha\alpha}$, where $\lambda \geq 0$ is an user-defined regularization parameter.

We have sketched how factors U_α and V_α can be computed by a nested interpolative decomposition (`ID`) in `GOFMM`. To be specific, we have

$$U_\alpha = \begin{bmatrix} P_{\tilde{\mathtt{l}}\mathtt{l}}^T & \\ & P_{\tilde{\mathtt{r}}\mathtt{r}}^T \end{bmatrix} = \begin{bmatrix} P_{\mathtt{l}\tilde{\mathtt{l}}} & \\ & P_{\mathtt{r}\tilde{\mathtt{r}}} \end{bmatrix}, \text{ and } V_\alpha = \begin{bmatrix} & K_{\tilde{\mathtt{l}}\tilde{\mathtt{r}}} P_{\tilde{\mathtt{r}}\mathtt{r}} \\ K_{\tilde{\mathtt{r}}\tilde{\mathtt{l}}} P_{\tilde{\mathtt{l}}\mathtt{l}} & \end{bmatrix} \tag{3.12}$$

due to the symmetry of K. We now discuss how to recursively apply these factors to the Sherman-Morrison-Woodbury formula in Equation 3.11 and solve $\lambda I + K_{\alpha\alpha}$ directly based on the `GOFMM` compression (without $S_\mathtt{l}$ and $S_\mathtt{r}$):

$$\widetilde{K}_{\alpha\alpha} = D_\alpha + U_\alpha V_\alpha = \begin{bmatrix} \widetilde{K}_{\mathtt{l}\mathtt{l}} & \\ & \widetilde{K}_{\mathtt{r}\mathtt{r}} \end{bmatrix} + \begin{bmatrix} P_{\mathtt{l}\tilde{\mathtt{l}}} & \\ & P_{\mathtt{r}\tilde{\mathtt{r}}} \end{bmatrix} \begin{bmatrix} & K_{\tilde{\mathtt{l}}\tilde{\mathtt{r}}} P_{\tilde{\mathtt{r}}\mathtt{r}} \\ K_{\tilde{\mathtt{r}}\tilde{\mathtt{l}}} P_{\tilde{\mathtt{l}}\mathtt{l}} & \end{bmatrix}. \tag{3.13}$$

For simplicity, we describe the case where $\lambda = 0$, but all the algorithms we describe trivially generalize to the $\lambda \neq 0$ cases.

We assume that all the tree nodes have been skeletonized. That is, for each tree node α, skeletons $\widetilde{\alpha}$ and its coefficient matrix $P_{\widetilde{\alpha}\alpha}$ or $P_{\widetilde{\alpha}[\tilde{\mathtt{l}}\tilde{\mathtt{r}}]}$ have been computed during the compression phase. The factorization of $\widetilde{K}_{\alpha\alpha}$ proceeds using a `postorder` traversal of the binary metric tree. At the leaf level, we directly factorize diagonal block $K_{\alpha\alpha} \in \mathbb{R}^{m \times m}$ using `LAPACK` LU factorization `GETRF`. For an internal node α, $\widetilde{K}_{\mathtt{l}\mathtt{l}}$ and $\widetilde{K}_{\mathtt{r}\mathtt{r}}$ are factorized from the left-hand side such that

$$\widetilde{K}_{\alpha\alpha} = D_\alpha (I + W_\alpha V_\alpha) = \begin{bmatrix} \widetilde{K}_{\mathtt{l}\mathtt{l}} & \\ & \widetilde{K}_{\mathtt{r}\mathtt{r}} \end{bmatrix} \left(I + \begin{bmatrix} \hat{P}_{\mathtt{l}\tilde{\mathtt{l}}} & \\ & \hat{P}_{\mathtt{r}\tilde{\mathtt{r}}} \end{bmatrix} \begin{bmatrix} & K_{\tilde{\mathtt{l}}\tilde{\mathtt{r}}} P_{\tilde{\mathtt{r}}\mathtt{r}} \\ K_{\tilde{\mathtt{r}}\tilde{\mathtt{l}}} P_{\tilde{\mathtt{l}}\mathtt{l}} & \end{bmatrix} \right), \tag{3.14}$$

where we define $\hat{P}_{\mathtt{l}\tilde{\mathtt{l}}} = \widetilde{K}_{\mathtt{l}\mathtt{l}}^{-1} P_{\mathtt{l}\tilde{\mathtt{l}}}$ and $\hat{P}_{\mathtt{r}\tilde{\mathtt{r}}} = \widetilde{K}_{\mathtt{r}\mathtt{r}}^{-1} P_{\mathtt{r}\tilde{\mathtt{r}}}$ to be the diagonal blocks of factor W_α (notice the **"hat"** notation).

Observe that the $\hat{P}_{\alpha\widetilde{\alpha}}$ factor requires "inverting" $\widetilde{K}_{\alpha\alpha}^{-1}$, which in turn requires traversing all the descendants of tree node α (the subtree rooted at α) and recursively applying Equation 3.14. (We introduced this scheme in [84] and resulted in

$\mathcal{O}(N \log^2 N)$ complexity.) But as we will see shortly this subtree traversal is not necessary if we exploit the telescoping expansion [83].

Reduced systems: Once we have obtained factors of W_α and V_α from both children ($\hat{P}_{\mathtt{l}\widetilde{\mathtt{l}}}$, $\hat{P}_{\mathtt{r}\widetilde{\mathtt{r}}}$, $P_{\widetilde{\mathtt{l}}\mathtt{l}}$, and $P_{\widetilde{\mathtt{r}}\mathtt{r}}$), we use Equation 3.11 (the SMW formula) to invert $I + W_\alpha V_\alpha$. This inverse requires solving a $2s$-by-$2s$ reduced system Z_α, which regarding the block decomposition of α, can be written as

$$Z_\alpha = I + V_\alpha W_\alpha = \begin{bmatrix} I & K_{\widetilde{\mathtt{l}}\widetilde{\mathtt{r}}} P_{\widetilde{\mathtt{r}}\mathtt{r}} \hat{P}_{\mathtt{r}\widetilde{\mathtt{r}}} \\ K_{\widetilde{\mathtt{r}}\widetilde{\mathtt{l}}} P_{\widetilde{\mathtt{l}}\mathtt{l}} \hat{P}_{\mathtt{l}\widetilde{\mathtt{l}}} & I \end{bmatrix}. \tag{3.15}$$

Since node α is the parent of node `l` and `r`, factor $\hat{P}_{\mathtt{l}\widetilde{\mathtt{l}}}$ and $\hat{P}_{\mathtt{r}\widetilde{\mathtt{r}}}$ have already been computed in the previous recursion during the `postorder` traversal. $K_{\widetilde{\mathtt{l}}\widetilde{\mathtt{r}}} P_{\widetilde{\mathtt{r}}\mathtt{r}} \hat{P}_{\mathtt{r}\widetilde{\mathtt{r}}}$ and $K_{\widetilde{\mathtt{r}}\widetilde{\mathtt{l}}} P_{\widetilde{\mathtt{l}}\mathtt{l}} \hat{P}_{\mathtt{l}\widetilde{\mathtt{l}}}$ are computed by `GEMM`, and the reduced system Z_α is factorized by `GETRF`.

Telescoping: The key to avoiding $\mathcal{O}(N \log^2 N)$ work is to exploit the *"telescoping"* relation between $\hat{P}_{\mathtt{l}\widetilde{\mathtt{l}}}$, $\hat{P}_{\mathtt{r}\widetilde{\mathtt{r}}}$ and $\hat{P}_{\alpha\widetilde{\alpha}}$. Recall from Section 2.3 that the interpolative coefficients $P_{\alpha\widetilde{\alpha}}$ can be "telescoped" from $P_{[\widetilde{\mathtt{l}}\widetilde{\mathtt{r}}]\widetilde{\alpha}}$, $P_{\mathtt{l}\widetilde{\mathtt{l}}}$, and $P_{\mathtt{r}\widetilde{\mathtt{r}}}$ due to the nested skeletons created in `ID`. As a result, replacing $P_{\alpha\widetilde{\alpha}}$ with its telescoping expansion (in gray) yields

$$D_\alpha^{-1} \boxed{P_{\alpha\widetilde{\alpha}}} = \begin{bmatrix} \widetilde{K}_{\mathtt{l}\mathtt{l}}^{-1} & \\ & \widetilde{K}_{\mathtt{r}\mathtt{r}}^{-1} \end{bmatrix} \boxed{\begin{bmatrix} P_{\mathtt{l}\widetilde{\mathtt{l}}} & \\ & P_{\mathtt{r}\widetilde{\mathtt{r}}} \end{bmatrix} P_{[\widetilde{\mathtt{l}}\widetilde{\mathtt{r}}]\widetilde{\alpha}}} = \begin{bmatrix} \hat{P}_{\mathtt{l}\widetilde{\mathtt{l}}} & \\ & \hat{P}_{\mathtt{r}\widetilde{\mathtt{r}}} \end{bmatrix} P_{[\widetilde{\mathtt{l}}\widetilde{\mathtt{r}}]\widetilde{\alpha}} = W_\alpha P_{[\widetilde{\mathtt{l}}\widetilde{\mathtt{r}}]\widetilde{\alpha}}. \tag{3.16}$$

Since $\hat{P}_{\alpha\widetilde{\alpha}} = \widetilde{K}_{\alpha\widetilde{\alpha}}^{-1} P_{\alpha\widetilde{\alpha}} = (I + W_\alpha V_\alpha)^{-1} D_\alpha^{-1} P_{\alpha\widetilde{\alpha}}$, we can replace $D_\alpha^{-1} P_{\alpha\widetilde{\alpha}}$ with Equation 3.16. Observe that $\hat{P}_{\alpha\widetilde{\alpha}}$ can also be telescoped from $\hat{P}_{\mathtt{l}\widetilde{\mathtt{l}}}$, $\hat{P}_{\mathtt{r}\widetilde{\mathtt{r}}}$, and $P_{[\widetilde{\mathtt{l}}\widetilde{\mathtt{r}}]\widetilde{\alpha}}$ as

$$\hat{P}_{\alpha\widetilde{\alpha}} = (I - W_\alpha Z_\alpha^{-1} V_\alpha) W_\alpha P_{[\widetilde{\mathtt{l}}\widetilde{\mathtt{r}}]\widetilde{\alpha}}. \tag{3.17}$$

We no longer need to solve $\widetilde{K}_{\mathtt{l}\mathtt{l}}^{-1}$ and $\widetilde{K}_{\mathtt{r}\mathtt{r}}^{-1}$ in Equation 3.17. Instead we only need to solve a $2s$-by-$2s$ system Z_α. Thus, no tree traversal is required. In the leaf level

Algorithm 3.10 `Factorize`(α)

if α is leaf **then**
  `GETRF`($\lambda I + K_{\alpha\alpha}$);
  $\hat{P}_{\alpha\tilde{\alpha}}$ = `Solve`(α,$P_{\alpha\tilde{\alpha}}$); /* Algorithm 3.11 */
else
  Form W_α and V_α with Equation 3.14, and Z_α with Equation 3.15;
  `GETRF`(Z_α);
  $\hat{P}_{\alpha\tilde{\alpha}}$ = `Solve`(α,$W_\alpha P_{[\tilde{\mathtt{lr}}]\tilde{\alpha}}$); /* Algorithm 3.11 */

Algorithm 3.11 w_α =`Solve`(α,u_α)

if α is leaf **then**
  Solve $w_\alpha = (\lambda I + K_{\alpha\alpha})^{-1} u_\alpha$; /* GETRS() */
else
  Solve $w_\alpha = u_\alpha - W_\alpha Z_\alpha^{-1} V_\alpha u_\alpha$; /* GETRS() */

(base case), $\hat{P}_{\alpha\tilde{\alpha}}$ is computed directly from $K_{\alpha\alpha}^{-1} P_{\alpha\tilde{\alpha}}$.

Factorization: Given formulas Equation 3.14, Equation 3.15 Equation 3.17, and the skeletonization computed in `GOFMM`, we compute the factors needed for the direct solver in a `postorder` traversal of the metric tree with task `Factorize`(α) (Algorithm 3.10). If α is a leaf node, we factorize $\lambda I + K_{\alpha\alpha}$ directly using an LU factorization (`GETRF`). Otherwise, we form and factorize the reduced system Z_α given that $\hat{P}_{\mathtt{l}\tilde{\mathtt{l}}}$ and $\hat{P}_{\mathtt{r}\tilde{\mathtt{r}}}$ have been computed in the previous recursion. Finally, $\hat{P}_{\alpha\tilde{\alpha}}$ is telescoped using Equation 3.17, which involves several matrix multiplications (`GEMM`) and solving a $2s$-by-$2s$ linear system Z_α (`LAPACK` LU solve `GETRS`).

This algorithm improves on the one in [84] by removing the extra subtree traversals that result in $\mathcal{O}(N \log^2 N)$ work. Instead, Algorithm 3.10 exploits the nested structure of $\hat{P}_{\alpha\tilde{\alpha}}$ resulting in $\mathcal{O}(N \log N)$ work for the factorization. In some of our largest runs, this resulted in over 3× speedup without any change in the accuracy.

Solve: Given factors W_α, V_α, and the LU factorization of Z_α computed and stored in each tree node, we now describe how to apply $\widetilde{K}_{\alpha\alpha}^{-1}$ to a vector u with a `postorder` traversal of the task `Solve`(α,u_α) (Algorithm 3.11). If α is a leaf

node, we can directly invoke an LU solver (`LAPACK GETRS`) to obtain $w_\alpha = K_{\alpha\alpha}^{-1} u_\alpha$. Otherwise, the SMW formula

$$w_\alpha = (I + W_\alpha V_\alpha)^{-1} w_\alpha = I - W_\alpha Z_\alpha^{-1} V_\alpha u_\alpha \tag{3.18}$$

is called to solve the reduced system. The LU factorization of Z_α has been computed in Algorithm 3.11. Thus, Z_α^{-1} can be applied in $\mathcal{O}(s^2)$ work using `GETRS`.

Distributed direct solver: The parallelization is essentially identical to the distributed-memory scheme proposed in Section 3.2. With p parallel processes, each subtree at level-$\log p$ is assigned to a distributed-memory process (or a worker). Parallel factorization phase Algorithm 3.12 and solve (Algorithm 3.13) are presented as several distributed tree traversals. Tasks involving tree nodes above level-$\log p$ (distributed tree) require communication using Message Passing Interface (`MPI`). Factors $\hat{P}_{\alpha\tilde{\alpha}}$, $P_{\tilde{\alpha}\alpha}$, u_α, and w_α above level-$\log p$ are distributed in `[IDS]` distribution (metric tree order) by default.

In Figure 3.3, we summarize the distributed-memory algorithms. The four colors represent four different `MPI` ranks and the nodes they own. Regarding distributing a matrix (on the left), each `MPI` rank holds a diagonal block and all factors in the same row, column and color (i.e., in `[IDS]` distribution). The tree on the right shows that the four tree nodes at level-2 are uniquely assigned to `MPI` ranks, but tree nodes above that level are shared among ranks. Each distributed tree node α inherits the local communicator `Comm(`α`)`, skeletons $\tilde{\alpha}$, and coefficient matrix $P_{\tilde{\alpha}[\tilde{1}\tilde{r}]}$ from `GOFMM`. To facilitate the factorization, factors $\hat{P}_{\alpha\tilde{\alpha}}$ and $P_{\tilde{\alpha}\alpha}$ are explicitly created and distributed in `[IDS]` distribution.

Distributed factorization: Task `DistFactorize(`α`)` (Algorithm 3.12) is called while visiting tree node α in a `postorder` tree traversal during the factorization. If node α is below level-$\log p$, task `Factorize(`α`)` (Algorithm 3.10) is called to factorize the node locally. Otherwise, the algorithm is divided into two cases side by side.

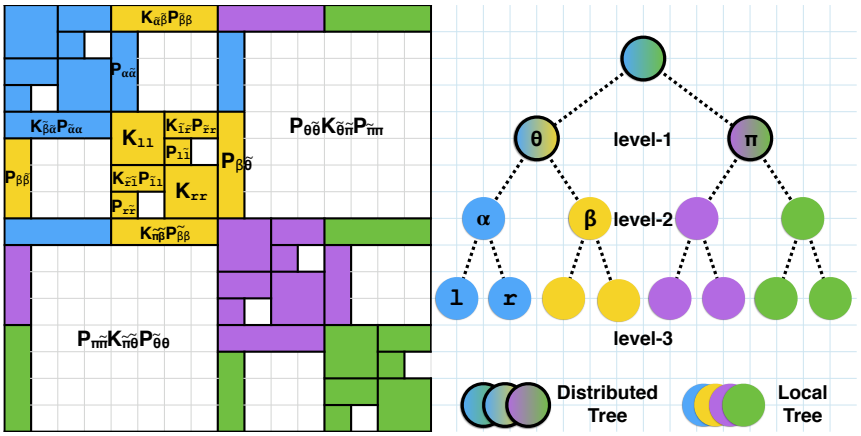

Figure 3.3: The top four levels of the tree and the corresponding blocks of the matrix $\tilde{K}$. The nodes belonging to each process are highlighted in a single color. Each process factorizes its own portion of the tree independently. We also highlight the factors used in the direct solver construction and show which process owns which factor. Each process owns a diagonal block and all factors in the same column and the same row. For example, the yellow process owns factors $\hat{P}_{\beta\tilde{\beta}}$, $K_{\tilde{\alpha}\tilde{\beta}}$, and $P_{\tilde{\beta}\beta}$ at level-1; similarly it owns factors $\hat{P}_{\beta\tilde{\theta}}$, $K_{\tilde{\pi}\tilde{\beta}}$, $P_{\tilde{\theta}\beta}$ at level-0.

Algorithm 3.12 `DistFactorize`(α)

if `Size`$(\alpha)=1$ **then** `Factorize`(α).
else `/** Above level-log(p) in the distributed tree */`

case `Rank`$(\alpha) < \frac{1}{2}$`Size`(α) **:**	**case** `Rank`$(\alpha) \geq \frac{1}{2}$`Size`$(\alpha)$ **:**
`Sendrecv(`$\tilde{\mathtt{l}}$`,`$\tilde{\mathtt{r}}$`,Size(`α`)/2);`	`Sendrecv(`$\tilde{\mathtt{r}}$`,` $\tilde{\mathtt{l}}$`,0,Comm(`α`));`
`Reduce(`$P_{\tilde{\mathtt{l}}\tilde{\mathtt{l}}}\hat{P}_{\tilde{\mathtt{l}}\tilde{\mathtt{l}}}$`,0,Comm(l));`	`Reduce(`$P_{\tilde{\mathtt{r}}\tilde{\mathtt{r}}}\hat{P}_{\tilde{\mathtt{r}}\tilde{\mathtt{r}}}$`,0,Comm(l));`
`Recv(`$P_{\tilde{\mathtt{r}}\tilde{\mathtt{r}}}\hat{P}_{\tilde{\mathtt{r}}\tilde{\mathtt{r}}}$`,Size(`$\alpha$`)/2,Comm(`$\alpha$`))`	`Send(`$P_{\tilde{\mathtt{r}}\tilde{\mathtt{r}}}\hat{P}_{\tilde{\mathtt{r}}\tilde{\mathtt{r}}}$`,0,Comm(`$\alpha$`));`
`GETRF(`Z_α`);`	`NOP;`
$\hat{P}_{\alpha\tilde{\alpha}}$ =`DistSolve(`α`,`$W_\alpha P_{[\tilde{\mathtt{l}}\tilde{\mathtt{r}}]\tilde{\alpha}}$`);`	$\hat{P}_{\alpha\tilde{\alpha}}$ =`DistSolve(`α`,`$W_\alpha P_{[\tilde{\mathtt{l}}\tilde{\mathtt{r}}]\tilde{\alpha}}$`);`

`MPI` processes with `Rank(`α`)<Size(`α`)/2` require skeletons $\tilde{\mathtt{r}}$ owned by `MPI` rank `0` in `Comm(`α`)` to compute $K_{\tilde{\mathtt{r}}\tilde{\mathtt{l}}}$, and vice versa. This point-to-point exchange is handled by a `Sendrecv()` using `Comm(`α`)`. To compute factors for the reduced system Z_α, factor $P_{\tilde{\mathtt{l}}\tilde{\mathtt{l}}}\hat{P}_{\tilde{\mathtt{l}}\tilde{\mathtt{l}}}$ is reduced from all processes in `Comm(l)`, and factor $P_{\tilde{\mathtt{r}}\tilde{\mathtt{r}}}\hat{P}_{\tilde{\mathtt{r}}\tilde{\mathtt{r}}}$ is reduced from all processes in `Comm(l)`. To factorize Z_α on rank `0` in `Comm(`α`)`, rank `0` must receive factor $P_{\tilde{\mathtt{r}}\tilde{\mathtt{r}}}\hat{P}_{\tilde{\mathtt{r}}\tilde{\mathtt{r}}}$ from rank `Size(`α`)/2` using `Comm(`α`)`. Finally, Z_α is

Algorithm 3.13 w_α[IDS]=DistSolve(α,u_α[IDS])

if Size(α)= 1 **then** w_α =Solve(α,u_α).
else /** Above level-log(p) in the distributed tree */
if α is at level $\log p$ **then**

case Rank(α) $< \frac{1}{2}$Size(α) :	**case** Rank(α) $\geq \frac{1}{2}$Size(α) :
Reduce($u_{\widetilde{\mathtt{r}}} = K_{\widetilde{\mathtt{r}}\widetilde{\mathtt{l}}} P_{\widetilde{\mathtt{l}}\mathtt{l}} u_{\mathtt{l}}$,0,Comm(l));	Reduce($u_{\widetilde{\mathtt{l}}} = K_{\widetilde{\mathtt{l}}\widetilde{\mathtt{r}}} P_{\widetilde{\mathtt{r}}\mathtt{r}} u_{\mathtt{r}}$,0,Comm(r));
Recv($u_{\widetilde{\mathtt{l}}}$,Size($\alpha$)/2,Comm($\alpha$))	Send($u_{\widetilde{\mathtt{l}}}$,0,Comm($\alpha$));
GETRS(Z_α,[$u_{\widetilde{\mathtt{l}}}$;$u_{\widetilde{\mathtt{r}}}$]);	NOP;
Send($u_{\widetilde{\mathtt{r}}}$,Size($\alpha$)/2,Comm($\alpha$))	Recv($u_{\widetilde{\mathtt{r}}}$,0,Comm($\alpha$));
Bcast($u_{\widetilde{\mathtt{l}}}$,0,Comm(l));	Bcast($u_{\widetilde{\mathtt{r}}}$,0,Comm(r));
$w_{\mathtt{l}} = u_{\mathtt{l}} - \hat{P}_{\mathtt{l}\widetilde{\mathtt{l}}} u_{\widetilde{\mathtt{l}}}$;	$w_{\mathtt{r}} = u_{\mathtt{r}} - \hat{P}_{\mathtt{r}\widetilde{\mathtt{r}}} u_{\widetilde{\mathtt{r}}}$;

factorized using GETRF, and $\hat{P}_{\alpha\widetilde{\alpha}}$ is solved by task DistSolve(α,u_α).

Distributed solve: Task DistSolve(α,u_α) (Algorithm 3.13) is called in two instances. When it is called by task DistFactorize, we are solving $\hat{P}_{\alpha\widetilde{\alpha}}$ using Equation 3.17. When it is called alone while visiting tree node α in a postorder tree traversal, then we are solving $\widetilde{K}_{\alpha\alpha}^{-1} u_\alpha$ for an unknown input u_α.

The distributed-memory part of Algorithm 3.13 for tree nodes above level-$\log p$ is again divided into two instances. While input u_α is distributed in [IDS] distribution, MPI processes with Rank(α)<Size(α)/2 need to reduce $u_{\widetilde{\mathtt{r}}} = K_{\widetilde{\mathtt{r}}\widetilde{\mathtt{l}}} P_{\widetilde{\mathtt{l}}\mathtt{l}} u_{\mathtt{l}}$, and MPI processes with Rank(α)≥Size(α)/2 need to reduce $u_{\widetilde{\mathtt{l}}} = K_{\widetilde{\mathtt{l}}\widetilde{\mathtt{r}}} P_{\widetilde{\mathtt{r}}\mathtt{r}} u_{\mathtt{r}}$. Recall that the LU factorization of Z_α is stored on rank 0 in Comm(α). As a result, rank 0 must receive $u_{\widetilde{\mathtt{l}}}$ from rank Size(α)/2 using Comm(α). GETRS will update both $u_{\widetilde{\mathtt{l}}}$ and $u_{\widetilde{\mathtt{r}}}$, and $u_{\widetilde{\mathtt{r}}}$ is sent back to rank Size(α)/2 using Comm(α). The final update involves $\hat{P}_{\mathtt{l}\widetilde{\mathtt{l}}}$[IDS,*] and $\hat{P}_{\mathtt{r}\widetilde{\mathtt{r}}}$[IDS,*]. As a result, $u_{\widetilde{\mathtt{l}}}$ and $u_{\widetilde{\mathtt{r}}}$ must be redistributed from [CIRC] to [*] distribution with a Bcast() using Comm(l) and Comm(r). Once reaching level $\log p$, Algorithm 3.11 is called to work on the local subtree.

3.4 Complexity Analysis

ihe worst case compression cost in Algorithm 3.2 is $\mathcal{O}(N^2)$, when $|\texttt{Near}(\alpha)| = \frac{N}{m}$ for each tree node α. The best case occurs when each near interaction list $\texttt{Near}(\alpha)$ only contains tree node α itself. We fix the rank s and leaf size m. The tree has $\mathcal{O}(\frac{N}{m})$ leaf nodes and $\mathcal{O}(\frac{N}{m})$ interior nodes, so in the best case, overall `N2S` has $\mathcal{O}(2sN + 2s^2(\frac{N}{m}))$ work, `S2S` has $\mathcal{O}(2s^2(\frac{N}{m}))$ work, `S2N` has $\mathcal{O}(2sN + 2s^2(\frac{N}{m}))$ work, and `L2` has $\mathcal{O}(2m^2(\frac{N}{m}))$. When s and m are held constant, the total work is $O(N)$ per right-hand side. In `GOFMM`, this is controlled by the **budget**.

Time and space: The complexity analysis of `GOFMM` is based on the work for high dimensional kernel matrices presented in [56, 53]. One minor difference is that in those works the authors do not enforce symmetry—whereas we do so in `GOFMM`. This difference changes the constants but not the asymptotics. The analysis below is only for the synchronous scheme for the matrix multiply. Here, we assume that K_{ij} can be evaluated with $\mathcal{O}(1)$ work. Let $\mathcal{D} = \log \frac{N}{m}$ be the tree depth, $n = \frac{N}{p}$ be the number of indices owned per `MPI` process in `[CYC]` and `[IDS]` distribution, t_s be the latency, t_w be the reciprocal of the bandwidth, κ be the number of neighbors, and b be the "budget" parameter. For simplicity, let $m \geq s$ and $\frac{s}{m} = \mathcal{O}(1)$. Parallel tree partition and skeletonization take

$$(t_s + t_w) \log^2 p \log N + t_w \kappa n \log p + s^3 \left(\frac{n}{m} + \log p \right) \tag{3.19}$$

time and $\mathcal{O}(\kappa n + s^2 n + s^2 \log p)$ space.

The size of the interaction lists is decided by the number of neighbors κ and the budget b. As a result, we have upper bound

$$\begin{aligned} |\texttt{Near}(\alpha)| &= \mathcal{O}(b\frac{N}{m}), \\ |\texttt{Far}(\alpha)| &= \mathcal{O}(b\mathcal{D}\frac{N}{m}) \end{aligned} \tag{3.20}$$

in the worst case[2]. With the upper bound of the list size, we now estimate the worst case complexity on redistributing indices from `[IDS]` to `[LET]` (to satisfy all distributed data dependencies introduced by the interaction lists). This cost is

$$t_s p + t_w b\mathcal{D}n^2 \tag{3.21}$$

with additional storage of size $b\mathcal{D}n^2$ per process.

We now compare asynchronous and synchronous evaluation. Algorithm 3.7 involves four `Alltoallv()` calls with complexity shown in Equation 3.21. For simplicity, we assume the total direct evaluations computed in task `L2L` and `S2S` are bounded as bN^2. All low-rank approximations including task `N2S` and `S2N` take $\mathcal{O}(2sN)$. The total communication and work for asynchronous and synchronous evaluation are the same. While bN^2 direct evaluations can be embarrassingly parallel after synchronous `Alltoallv()`, the main advantage of asynchronous evaluation is to overlap communication and resolve the parallelism diminishing issue in the `postorder` tree traversal of task `N2S`. With sufficient budget, communication can be hidden. We observe up to 2.5× speedup comparing to synchronous evaluation in Section 4.5.

$\mathcal{H}$-matrix factorization and solve: Let $T^f(N)$ denote the complexity of Algorithm 3.10, and $T^s(N)$ of Algorithm 3.11, each for problem size N. Since `Solve` computes either an LU solve or matrix-vector multiply in each step, we have

$$T^s(N) = 2T^s(\frac{N}{2}) + \mathcal{O}(Ns + s^2) = \mathcal{O}(sN \log N). \tag{3.22}$$

Notice that solving $\hat{P}_{\alpha\widetilde{\alpha}} = \widetilde{K}_{\alpha\alpha}^{-1} P_{\alpha\widetilde{\alpha}}$ only takes $\mathcal{O}(s^2 N)$ work. Using the complexity above, we derive $T^f(N)$ as

$$T^f(N) = 2T^f(\frac{N}{2}) + s^2 N + s^3 = \mathcal{O}(s^2 N \log N). \tag{3.23}$$

[2]Given that the neighbors we compute are unsymmetric, the near interaction list can be 2× longer after symmetrization in the worst case.

To summarize, both Algorithm 3.10 and Algorithm 3.11 take $\mathcal{O}(N \log N)$ work.

The communication cost for the solving phase is $\mathcal{O}(s \log p)$ per level. To traverse the whole tree, $\mathcal{O}(s \log^2 p)$ is required per right-hand side. However, during the factorization, the solving phase does not recur. Thus, instead of $\mathcal{O}(s^2 \log^2 p)$ for s right-hand sides, there is only $\mathcal{O}(s^2 \log p)$ communication per level. The overall cost for the full factorization is $\mathcal{O}(s^2 \log^2 p)$ since there is $\log p$ distributed levels.

In addition to the cost of `GOFMM` we discuss above, $\mathcal{H}$-matrix direct solver requires maintaining factors U, V, and $I+WV$ for each tree node. That is $\mathcal{O}(2sN+s^2)$ space per level. Therefore, the overall memory requirement is

$$\mathcal{O}\left(\left(2sN + s^2\right) \log \frac{N}{m}\right). \tag{3.24}$$

Using `GSKS` (General Stride Kernel Summation described in Section 4.2) can reduce $sN \log \frac{N}{m}$ space requirement to $\mathcal{O}(1)$ by evaluating and multiplying V on the fly for kernel matrices. Recomputing W with Equation 3.17 can reduce another $sN \log \frac{N}{m}$ space to sN. Using both schemes yields $\mathcal{O}(s^2 \log \frac{N}{m} + sN)$ storage with $\mathcal{O}(s^2 N \log N)$ work (with a larger constant but still $\mathcal{O}(N \log N)$ asymptotically).

3.5 Summary

We present `GOFMM` as a purely algebraic Fast Multipole Method. By describing the algorithms as several tree traversals and tasks, we show that `GOFMM` compresses and factorizes a generic SPD matrix with $\mathcal{O}(N \log N)$ work and $\mathcal{O}(\frac{N}{p} \log N)$ time in parallel. The only required input is a routine $K(I, J)$ that returns a submatrix K_{IJ}, for arbitrary row and column index sets I and J. Once compressed and factorized, `MATVEC` and `SOLVE` with the $\mathcal{H}$-matrix representation takes $\big(N \log N)$ work and $\mathcal{O}(\frac{N}{p} \log N)$ time in parallel.

`GOFMM` is also developed as a framework to help exploit the parallelism of

tree-based algorithms by a semi-auto runtime dependency analysis. As we will see in the next chapter, with reading and writing activities annotated for each task, the system can better discover parallelism with dynamic data dependencies. The runtime system also manages the software and hardware mapping for users, which systematically resolves thread binding, communication overlapping, nested parallelism, and scheduling issues.

Chapter 4

Implementation and Experimental Results

[1]In Section 4.1, we present a lightweight, self-contained runtime system that is attached to GOFMM. In Section 4.2, we discuss how GOFMM employs high-performance N-body operators to accelerate neighbor search and kernel summation while approximating and factorizing kernel matrices. In Section 4.3, we detail our experimental setups, which we use to examine the accuracy and efficiency of our methods.

We demonstrate the robustness and effectiveness of our geometry-oblivious FMM (Section 4.4), the scalability of our $\mathcal{H}$-matrix algorithms, and the runtime system by comparing with other parallel schemes (Section 4.5), and the accuracy and efficiency while applying our methods to kernel ridge regression task (Section 4.6).

[1]Several of my prior publications contribute to this chapter. In [82] (Chenhan D. Yu, James Levitt, Severin Reiz, and George Biros. Geometry-oblivious FMM for compressing dense SPD matrices.), I introduced GOFMM and $\mathcal{H}$-matrix schemes that can be used to compress arbitrary SPD matrices. In [83] (Chenhan D. Yu, William B March, and George Biros. An N log N parallel fast direct solver for kernel matrices.), I introduced an $\mathcal{O}(N \log N)$ hybrid direct-iterative solver for kernel matrices. In [81] (Chenhan D. Yu, Jianyu Huang, Woody Austin, Bo Xiao, and George Biros. Performance optimization for the κ-nearest neighbors kernel on x86 architectures.), I introduced an efficient algorithm for exhausted κ-nearest neighbor search on modern CPUs. By fusing neighbor search with GEMM in the assembly level, the proposed method is free from extra working space and suffer from smaller memory latency. In [56] (William B. March, Bo Xiao, Chenhan D. Yu, and George Biros. ASKIT: An efficient, parallel library for high-dimensional kernel summations.), I design and conduct experiments to empirically analyze the performance of ASKIT.

Related contributions: This chapter contributes to the following conference and journal publications.

- Chenhan D Yu, James Levitt, Severin Reiz, and George Biros. Geometry-oblivious FMM for compressing dense SPD matrices. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 53. ACM, 2017

- Chenhan D Yu, William B March, and George Biros. An n log n parallel fast direct solver for kernel matrices. In *Parallel and Distributed Processing Symposium (IPDPS), 2017 IEEE International*, pages 886–896. IEEE, 2017

- Chenhan D. Yu, William B. March, Bo Xiao, and George Biros. INV-ASKIT: a parallel fast direct solver for kernel matrices. In *Proceedings of the IPDPS16*, Chicago, USA, May 2016

- Chenhan D. Yu, Jianyu Huang, Woody Austin, Bo Xiao, and George Biros. Performance optimization for the k-nearest neighbors kernel on x86 architectures. In *Proceedings of SC15*, pages 7:1–7:12. ACM, 2015

4.1 Runtime System

Recursive `preorder` and `postorder` traversals inherently encode **R**ead/**W**rite dependencies between tasks while visiting different tree nodes. Following the execution flow of the compression (Algorithm 3.1) and evaluation phase (Algorithm 3.2), we can describe dependencies between various tasks. However, due to the dynamic granularity of tasks we need a data flow analysis at runtime to help exploit the available parallelism.

For example, dependencies between tasks `N2S` and `S2S` cannot be resolved at compile time, because the read-after-write (**RAW**) dependencies on $\widetilde{w}_\alpha$ are computed

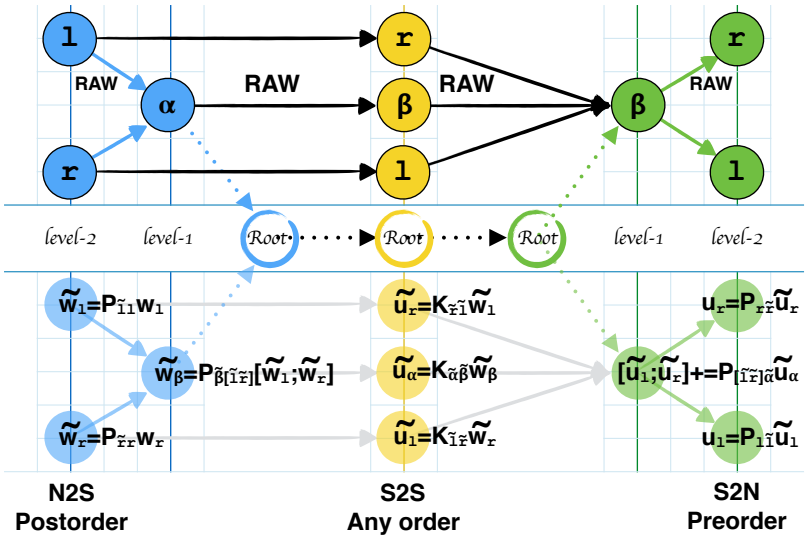

Figure 4.1: Dependency graph for steps 1–3 of Algorithm 3.2 (step 4 is completely independent of steps 1–3): each tree node denotes a task, and the arrows between nodes imply a dependency. Here `Near`(α) only contains itself (`HSS`). For example, yellow node β has a **RAW** dependency following blue α, because `S2S`(β) computes $\widetilde{u}_\beta = \sum_{\alpha \in \texttt{Near}(\beta)} K_{\widetilde{\beta}\widetilde{\alpha}} \widetilde{w}_\alpha$. When `Near`($\beta$) contains more than just itself. The dependencies are unknown at compile time and thus, `omp task depend` fails to describe the dependencies between `N2S` and `S2S`.

by neighbors $\mathcal{N}(\alpha)$. In order to build dependencies at runtime as a direct acyclic graph (DAG), we perform *symbolic* execution on Algorithm 3.1 and Algorithm 3.2 to analyze these data dependencies. For simplicity, below we discuss the **evaluation phase** for the `HSS` case (the `FMM` case is more involved). That is, `Near`(α) only contains α itself, and `Far`(α) only includes the direct sibling of α.

Dependency analysis: The runtime dependency analysis is performed by traversing all tasks in sequence and annotating their **R**ead/**W**rite activities on factors or variables carried by each tree node. A pair of consecutive read-after-write (**RAW**) or write-after-read (**WAR**) from two different tasks will result in data dependencies. In `GOFMM`, the analysis is done ahead of the computation.

Let us take Figure 4.1 as an example. We use three symbolic tree traversals[2].

[2]Execution order from left to right: dependencies are easier to follow if one rotates the page by 90°

in Algorithm 3.2 to depict the task dependencies between task `N2S`, `S2S`, and `S2N`. During the symbolic traversals, only data dependencies between tree nodes are annotated with no other computation is performed. In the first traversal (`postorder`) we find that skeleton weight $\widetilde{w}_{\mathtt{l}}$ is written by task `N2S(l)`. Going from $\widetilde{w}_{\mathtt{l}}$ to $\widetilde{w}_{\beta}$, we annotate that $\widetilde{w}_{\mathtt{l}}$ is read by task `N2S(`β`)`, i.e.

$$\widetilde{w}_{\beta} = P_{\widetilde{\beta}[\widetilde{\mathtt{l}}\widetilde{\mathtt{r}}]} \begin{bmatrix} \widetilde{w}_{\mathtt{l}} \\ \widetilde{w}_{\mathtt{r}} \end{bmatrix}. \tag{4.1}$$

This read-after-write (**RAW**) dependency is an edge from node `l` to β in the DAG.

Inter-task dependencies between task `N2S` and `S2S` are discovered during the *symbolic* execution of the yellow tree. At node β (in yellow), the computation $\widetilde{u}_{\alpha} = K_{\widetilde{\alpha}\widetilde{\beta}}\widetilde{w}_{\beta}$ will read $\widetilde{w}_{\beta}$ in task `S2S(`α`)`. Again this is a **RAW** dependency; hence there is an edge from node β (blue) to node α (yellow). The whole dependency graph for steps 1–3 of the evaluation phase (Algorithm 3.2) is built after the green `postorder` tree traversal of task `S2N`. The traversal on task `L2L` (step 4) in Algorithm 3.2 is independent of steps 1–3. Runtime data flow analysis can fully expose the task parallelism in finer granularity. Although this runtime analysis has some overhead, the amount is almost negligible ($< 1\%$) compared to the total execution time.

Dynamic scheduling: With a dependency graph, scheduling can be done *statically* or *dynamically*. Due to unknown adaptive rank s at compile time, we implement a dynamic Heterogeneous Earliest Finish Time (HEFT) [71] schedule using `OpenMP` threads. Each worker (thread) in the runtime system can use more than one physical core with either a nested `OpenMP` constructing or by employing a device (accelerator) as a slave. Tasks that satisfy all dependencies in the dependency graph will be dispatched to a "ready" queue. Each worker keeps consuming tasks in its ready queue until no tasks are left.

---

counter-clockwise

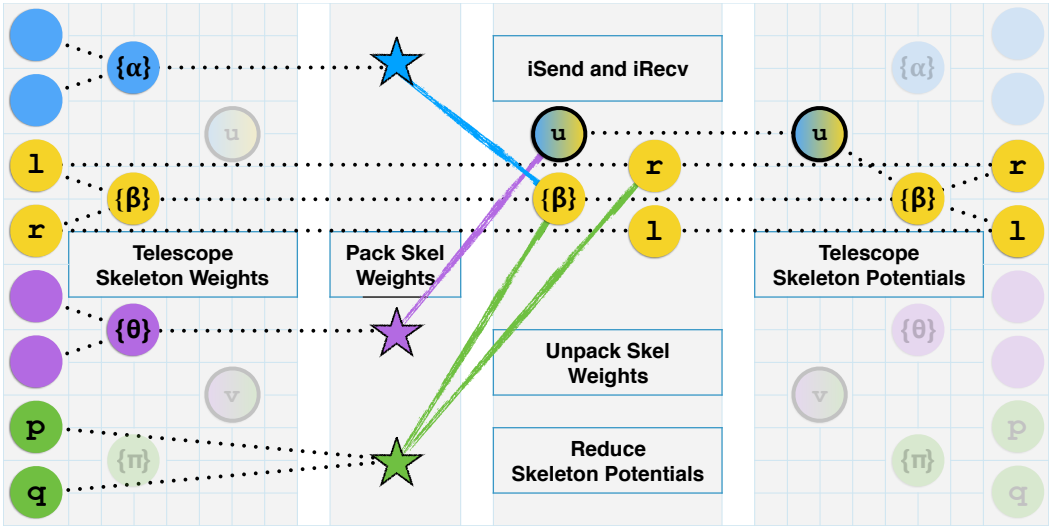

Figure 4.2: Distributed dependency graph for process yellow during the evaluation (without task `L2L()`): The whole graph has four stages: an upward pass to compute skeleton weights, packing skeleton weights to $p-1$ messages according to the far interaction list, exchanging and unpacking messages, and finally a downward pass to compute the aggregate far contribution.

Although we can estimate a cost for each task[3] in Table 3.1, the execution time of a task on a regular worker (or one with an accelerator) depends on the problem and can only be determined at runtime. The HEFT schedule is implemented using an estimated finish time of all pending tasks in a specific worker's ready queue. Each task dispatched from the dependency graph is assigned to a ready queue such that the maximum estimated finish time of each queue is minimized. For the case where the estimation is inaccurate, we also implement a job stealing mechanism.

Asynchronous message passing: To overlap communication with computation in a distributed-memory environment without reducing the maximum parallelism obtained from the dependency graph, `Alltoallv()` in Algorithm 3.7 must be broken down into at least[4] $(p-1)^2$ `Isend()` and `Irecv()` tasks handled by the runtime

[3] We divide costs for tasks by the theoretical peak `FLOPS` of the target architecture and a discount factor. For memory-bound tasks we use the theoretical `MOPS` instead.

[4] In the case that a single message size is large, we should further divide the message into several sub-messages to reduce the critical path.

system. To prevent blocking any worker (core), asynchronous messages are received by periodic polling using `Iprobe()`. This approach is known as **Dynamic Sparse Data Exchange (DSDE)** [38]. To prevent any `MPI` rank from early termination before all messages are received, we modify the nonblocking-consensus solution of DSDE to fit into our client-server based runtime system. We use a distributed dependency graph (Figure 4.2) to explain this distributed mechanism.

Edges and nodes in Figure 4.2 denote all tasks and dependencies required in each step to solely compute the process yellow portion. Dimmed tasks do not participate in the evaluation of process yellow. The leftmost tree denotes the upward traversal of `DistN2S` tasks, and the rightmost tree denotes the downward traversal of `DistS2N` tasks. In the middle, the four circle nodes denote `S2S` tasks, and the three star-shaped tasks represent the incoming messages from process blue, purple and green. In the following, we focus on the case of task `S2S` and left `L2L`, which has a similar implementation.

Listener: To implement non-blocking periodic polling, each regular worker (thread) can be promoted to consume `Irecv()` tasks. A pair of `Isend()` and `Irecv()` tasks are created simultaneously during the dependency analysis phase (synchronous) with other regular tasks. Tasks can result in **W**rite dependency on `Isend()` or **R**ead dependency on `Irecv()`. During the execution epoch, the former gets dispatched to regular workers while all incoming dependencies are satisfied (i.e. `Isend()` is treated as regular tasks), but the later can only be picked up and consumed by listeners.

Algorithm 4.1 describes how listeners perform non-blocking periodic polling to consume both `Irecv()` and regular tasks. A listener continually probes for messages and checks if their `[SRC,TAG]` match any `Irecv()` task in the list. If so, messages will be received and unpacked by the listener. Otherwise, listeners behave the same as regular workers, trying to dispatch a regular task from its own ready queue or steal one from others. Finally, the task (either an `Irecv()` or the regular one) will

Algorithm 4.1 `Listen(termination_signal)`

```
1: while in the runtime epoch do
2:    [SRC,TAR]=Iprobe(ANY_SRC_TAG);
3:    if task=list.find[SRC](TAG) then
4:       Recv(rbuff0,SRC,TAG);
5:       /** Receive buff1, ...  */
6:       task.Unpack(rbuff0,rbuff1,...);
7:    else StealAndConsumeFromOthers();
8:    if termination_signal then
9:       Ibarrier(&request) only once;
10:      if Test(request) then return;
```

be executed to release other dependent tasks. Notice that increasing the number of listeners may potentially reduce the time of receiving and unpacking messages (given sufficient bandwidth). Listeners can still consume regular tasks even if there is no incoming message. As a result, we typically use several listeners.

Termination: Listeners terminate when a global consensus on variable `termination_signal` is reached. This signal is set by regular workers while all tasks in the local dependency graph are completed. While the signal is set on a process, the first reacting listener will issue an asynchronously distributed barrier (`Ibarrier()`) and periodically test for global consensus. This guarantees all incoming message will be received and handled before global termination.

4.2 N-body Operators

In the majority of the tree-based $\mathcal{H}$-matrix methods, computation on each tree node often contains a series of N-body (or linear algebra) computation primitives. For example, while K is a kernel matrix defined on data points $\{x_i\}_{i=1}^{N}$, dense kernel summation appearing in tasks `L2L` and `S2S` dominates the runtime of the **evaluation** phase. Nearest-neighbor search (`ANN`) appears in the **compression** phase, which also takes a significant amount of runtime. In this section, we discuss the insights of

high-performance kernel summation and neighbor search.

Insights: Given data points $\{x_i\}_{i=1}^N \in \mathbb{R}^d$, tasks `L2L`, `S2S`, and `ANN` require access to an $|\alpha| \times |\beta|$ submatrix $K_{\alpha\beta}$, where α and β can be the same or two different sets of points. If $K_{\alpha\beta}$ is computed and stored, then it typically requires $\mathcal{O}(d|\alpha||\beta|)$ floating point operations (`flops`) and $\mathcal{O}(|\alpha||\beta|)$ memory operations (`mops`). Notice that `mops` typically have a larger constant in the complexity estimation. As a result, these tasks can be memory-bound when dimension d is small because $K_{\alpha\beta}$ is explicitly created and stored in the slow memory (DRAM).

Notice that tasks `L2L` and `S2S` compute the `MATVEC` of $K_{\alpha\beta}$ and only output an $\mathcal{O}(|\alpha|)$ vector. Similarly, task `ANN` selects κ neighbors for each row of $K_{\alpha\beta}$ and only output $\mathcal{O}(\kappa|\alpha|)$ neighbors. These outputs have much smaller `mops` and space requirement than $\mathcal{O}(|\alpha||\beta|)$. That is, performing these operators in a matrix-free manner (without explicitly created and stored) can significantly reduce the memory latency and the space consumption.

State-of-the-art algorithms: For Euclidean distance (i.e., ℓ_2 norm) based kernel functions, the state-of-the-art algorithms for computing tasks `L2L`, `S2S`, and `ANN` were to split the whole computation into two parts. Let $\mathcal{X}_\alpha \in \mathbb{R}^{d\times|\alpha|}$ and $\mathcal{X}_\beta \in \mathbb{R}^{d\times|\beta|}$ be the data points owned by node α and β. We first compute all pairwise distances between $\mathcal{X}_\alpha$ and $\mathcal{X}_\alpha$ using `GEMM`. This is done by expanding $\|x_j - x_i\|_2^2$ as

$$\|x_i - x_j\|_2^2 = \|x_i\|_2^2 + \|x_j\|_2^2 - 2x_i^\top x_j. \tag{4.2}$$

Computing $\|x_i\|_2^2$ scales as $\mathcal{O}(|\alpha| + |\beta|)$. The inner product terms $x_i^\top x_j$ can be computed by a `GEMM` call that, depending on the problem size and the dimension d, can achieve near peak performance; the complexity of this calculation is $\mathcal{O}(d|\alpha||\beta|)$.

Kernel summation: In [55], we presented `GSKS` (General Stride Kernel Summation), a matrix-free kernel summation that performs fusing optimization.

Arch	d	4	20	36	68	132	260
Haswell	MKL+VML	31	53	72	115	190	305
16K	GSKS	321	465	512	634	687	680
KNL	MKL+VML	12	93	132	416	636	916
16K	GSKS	703	888	1067	1246	1334	1449
Haswell	MKL+VML	32	56	80	110	198	296
8K	GSKS	301	448	515	558	620	543
KNL	MKL+VML	11	93	103	166	506	753
8K	GSKS	479	888	903	975	1220	1345
Haswell	MKL+VML	30	52	70	110	180	284
4K	GSKS	250	359	384	420	477	468
KNL	MKL+VML	11	56	76	116	370	578
4K	GSKS	341	445	464	510	858	1015

Table 4.1: Gaussian kernel summation efficiency of $16K \times 16K \times d$, $8K \times 8K \times d$, and $4K \times 4K \times d$ in GFLOPS. GSKS can be found in https://github.com/ChenhanYu/ks. The reference implementation uses MKL DGEMM and VML VEXP.

Different from the best-known method that computes

$$K_{\alpha\beta} w_\beta = \texttt{GEMV}(\mathcal{K}(\texttt{GEMM}(\mathcal{X}_\alpha^T, \mathcal{X}_\beta)), w_\beta), \tag{4.3}$$

GSKS fuses $\mathcal{K}$ (kernel function) and GEMV (reduction) into GEMM (semi-ring rank-d update). GSKS can be applied to kernels with abs, max, or min as long as they can be written as a semi-ring rank-d update and element-wise transformations.

With a BLIS-like framework [72], matrix-matrix multiplication ($C = \mathcal{X}_\alpha^T \mathcal{X}_\beta$) is divided into subproblems. A small subproblem that fits matrix C into registers is implemented in vectorized assembly to maximize FLOPS throughput. The idea is to directly perform kernel evaluation and the GEMV on C while it is still in the registers and only store back a vector w. In short, for a typical kernel summation that involves a GEMM call with $\mathcal{O}(d|\alpha||\beta|)$ flops and $\mathcal{O}(d(|\alpha| + |\beta|) + |\alpha||\beta|)$ mops in the best-known method, GSKS only requires $\mathcal{O}(d(|\alpha| + |\beta|))$ mops. This helps the computation become less memory bound even with small d.

We implement this idea in AVX2 and AVX512 for Intel Haswell and KNL

architectures. We present the performance of these two different approaches in Table 4.1. The `MKL+VML` approach implements the best-known algorithm Equation 4.3, where Intel `MKL DGEMM` is called to compute $x_i^\top x_j$ and Intel Vector Mathematical Library (`VML`) is used to vectorize the exponential function in the Gaussian kernel

$$\mathcal{K}(x_i, x_j) = \exp\left(-\frac{1}{2}\frac{\|x_i - x_j\|_2^2}{h^2}\right). \tag{4.4}$$

Due to the $\mathcal{O}(|\alpha||\beta|)$ memory saving, `GSKS` is about $3 \sim 30$x faster than the best known method on KNL for large problem size and $d < 68$. (For small problem size, `GEMM` in LIBSXMM `https://github.com/hfp/libxsmm` may be slightly faster than MKL `GEMM`, but it is still memory bound when d is small.) We see that using `GSKS` significantly outperforms using the standard approach.

Nearest neighbor kernel: We presented `GSKNN` (General Stride κ-nearest neighbors) [81], which fuses Euclidean distance evaluations with κ-neighbor selection (with a max heap). To be specific, `GSKNN` computes

$$\mathcal{N}(\alpha) = \kappa\texttt{-select}(\mathcal{K}(\texttt{GEMM}(\mathcal{X}_\alpha^T, \mathcal{X}_\beta))), \tag{4.5}$$

where kernel $\mathcal{K}$ computes pairwise Euclidean distances between data points and κ`-select` computes the κ smallest neighbor pairs with a max heap. Using the same fusing optimization, $\|x_i\|_2^2$ and $\|x_j\|_2^2$ in Equation 4.2 are directly accumulated to the output of `GEMM` (semi-ring rank-d update) to compute the pairwise Euclidean distances in registers[5]. These distances are the **keys** of our candidate pairs and the indices in tree node β serve as the **values**. In different κ`-select` algorithms, candidates are reordered according to the full (or partial) order of the **keys**. At the same time, **values** will also be reordered with their **keys**.

[5]Registers are the fastest memory in modern CPUs, GPUs and accelerators. Arithmetic logic units (ALUs) can directly perform instructions on registers. High-performance CPU and GPU `GEMM` implementations keep the output matrix in the registers to maximize the reuse rate of these low-latency memory units.

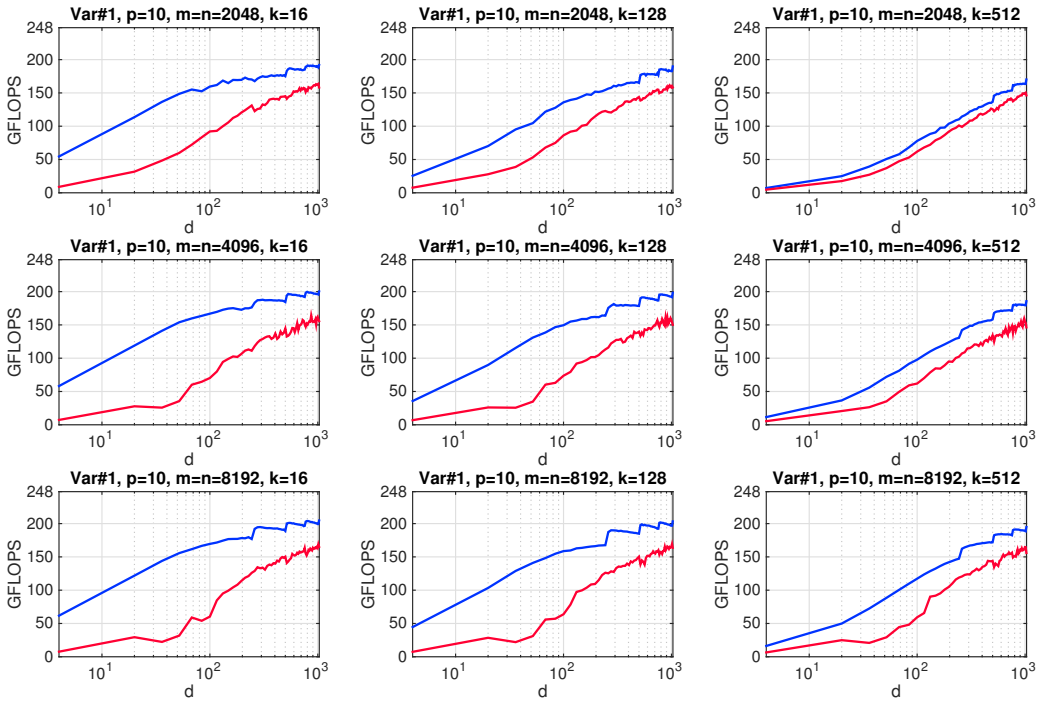

Figure 4.3: 10-core floating point efficiency comparison: m ($|\alpha|$) and n ($|\beta|$) from top to bottom are 2048, 4096 and 8192. κ from left to right are 16, 128, and 512. The X-axis is the dimension size d from 4 to 1028, and the Y-axis is `GFLOPS` where the theoretical peak performance is 248 `GFLOPS` (4 `double` per `AVX256` unit $\times$ 2 `FLOPS` per `FMA` instruction $\times$ 3.1 Ghz $\times$ 10 cores).

10-core efficiency overview: In Figure 4.3, we plot the floating point efficiency (in `GFLOPS`) as a function of the problem size m (number of points in α), n (number of points in β), κ, and d (Notice the logarithmic scale for the horizontal coordinate). Performance (`GFLOPS`) increases with problem size m, n, and dimension d but degrades with κ. This is because larger m, n, and d provide higher parallelism, but neighbor search only contributes to the runtime without performing any floating point operations. Fixing m, n, and d, performance degrades when κ is getting larger. This reflects that fact that the reuse rate of the heap increases and the performance of `GEMM` has been affected. Especially in low d and large κ case, `GFLOPS` does not capture the efficiency very well, since the runtime is dominated by heap selections,

which do not involve any floating point operation. To better observe the efficiency in this memory bound problem, IPC (Instructions per cycle) that taking all floating point, non-floating point and memory operations into account should better reveal the performance.

4.3 Setup

The source code of GOFMM can be found in the Github repository

$$\texttt{https://github.com/ChenhanYu/hmlp}. \tag{4.6}$$

GOFMM is implemented in C++ and CUDA, employing OpenMP for shared-memory parallelism, MPI (MPI_THREAD_MULTIPLE is required) for distributed-memory parallelism. The only dependencies are BLAS/LAPACK, OpenMP, and MPI.

Hardware: We conducted our experiments on the "Maverick", "Stampede2", "Lonestar5" system at Texas Advanced Computing Center, and the "Piz Daint" system at Swiss National Supercomputing Centre. Each Maverick CPU node has two 10-core, Intel Xeon E5-2680 v2 "Ivy Bridge" processors, which have two AVX256 units. Each Stampede2 CPU node has two 24-core, Intel Xeon Platinum 8160 "Skylake" processors, which have two AVX512 units. Each Stampede KNL node has one 68-core, Intel Xeon Phi 7250 "KNL" processors, which also have two AVX512 units Each Lonestar CPU node has two 12-core, Intel Xeon E5-2690 v3 "Haswell" processors, which have two AVX256 units. Each Piz Daint GPU node has one 12-core, Intel Xeon E5-2650 v3 "Haswell" CPU processors and one NVIDIA Tesla P100 ("Pascal") GPU.

The theoretical peak performance per Skylake node is roughly 4.3 TFLOPS in single precision. We estimate this metric according to the base frequency (1.4Ghz), AVX512 vector length (16 floats), FMA throughput per core/cycle (dual issue, i.e., 4 flops), and the number of physical cores per node (48 cores divided into two sockets).

Similarly, we can estimate the theoretical peak performance for Ivy Bridge, KNL, Haswell, and Pascal node as well.

Matrices: We generated 22 matrices emulating different problems. **K02** is a 2D regularized inverse Laplacian squared, resembling the Hessian operator of a PDE-constrained optimization problem. The Laplacian is discretized using a 5-stencil finite-difference scheme with Dirichlet boundary conditions on a regular grid. **K03** has the same setup with the oscillatory Helmholtz operator and 10 points per wavelength. **K04**–**K10** are kernel matrices in six dimensions (Gaussians with different bandwidths, narrow and wide; Laplacian Green's function, polynomial, and cosine-similarity). **K12**–**K14** are 2D advection-diffusion operators on a regular grid with highly variable coefficients. **K15,K16** are 2D pseudo-spectral advection-diffusion-reaction operators with variable coefficients. **K17** is a 3D pseudo-spectral operator with variable coefficients. **K18** is the inverse squared Laplacian in 3D with variable coefficients. **G01**–**G05** are the inverse Laplacian of the **powersim**, **poli_large**, **rgg_n_2_16_s0**, **denormal**, and **conf6_0-8x8-30** graphs from UFL (`http://yifanhu.net/GALLERY/GRAPHS/search.html`).

Matrices **K02**–**K03**, **K12**–**K14**, and **K18** resemble inverse covariance matrices and Hessian operators from optimization and uncertainty quantification problems. **K04**–**K10** resemble classical kernel/Green function matrices but in high dimensions. **K15**–**K17** resemble pseudo-spectral operators. **G01**–**G05** ($N = 15838$, 15575, 65536, 89400, 49152) are graphs for which we do not have geometric information. For **K02**–**K18**, we use $N = 65536$ if not specified.

Also, we use kernel matrices from real-world machine learning and synthetic datasets: **COVTYPE** (100K, 54D, cartographic variables), **HIGGS** (500K, 28D, physics) [50], **SUSY** (5M, 18D, high energy physics)[6], **MNIST** (60K, 780D, digit recognition) [18], and **NORMAL** (1–64M, 6D, synthetic point cloud). For these datasets, we use a Gaussian kernel with bandwidth h.

[6] `https://archive.ics.uci.edu/ml/datasets/SUSY`

H02 is (roughly) a 100k-by-100k matrix, which is the Gauss-Newton Hessian operator of a three-layer fully connected neural network with a ReLU nonlinearity with topology $784 \times 100 \times 200 \times 10$ with batch size 10k (**MNIST** dataset). H03 is (roughly) a 266k-by-266k matrix, which is the Gauss-Newton operator of a three-layer fully connected neural network with a ReLU nonlinearity with topology $784 \times 256 \times 256 \times 1$ and batch size 60k (the **MNIST** dataset). **C01** and **C02** are shifted empirical covariance matrices by sampling 33k points in 100k dimensions (**C01**) and 66k points in 500k dimensions (**C02**). In both **C01** and **C02**, the points are generated by a normal distribution with a covariance that is a hierarchical matrix, so in some sense, **C01** and **C02** can be considered as synthetic algorithm verification cases. All matrices but **H03**, **C01**, and **C02** use a combination of stored matrices and evaluation. The cost of evaluation for on entry of **H03** is linear to the batch size (60k); the cost of one entry of **C01** and **C02** is linear to the sample size.

`GOFMM` supports both double and single precision. All experiments with matrices **K02**–**K18** and **G01**–**G05** are in single precision. The results for **COVTYPE, HIGGS, MNIST, SUSY** are in double precision. In the Github repository, we provide a `MATLAB` script to generate **K02**–**K18**. For real-world datasets and graphs, we provide the link to their original sources.

Parameter selection and accuracy metrics: We typically control m (leaf node size), s (maximum rank), τ (adaptive tolerance), κ (number of neighbors), **budget** (the percentage of direct evaluations and for switching between `HSS` and `FMM`). For a full list of controllable parameters and inputs, please refer to Table 2.1. We use m =256–512; on average this gives good overall time. The adaptive tolerance τ reflects the error of the subsampled block and may not correspond to the output error ϵ_2. Depending on the problem, τ may underestimate the rank. Similarly, this may occur in `HODLR`, `STRUMPACK`, and `ASKIT`. We use τ between 1E-2 and 1E-7, $s = m$, $k = 32$ and 3% budget. To enforce an `HSS` approximation, we use 0% budget. The

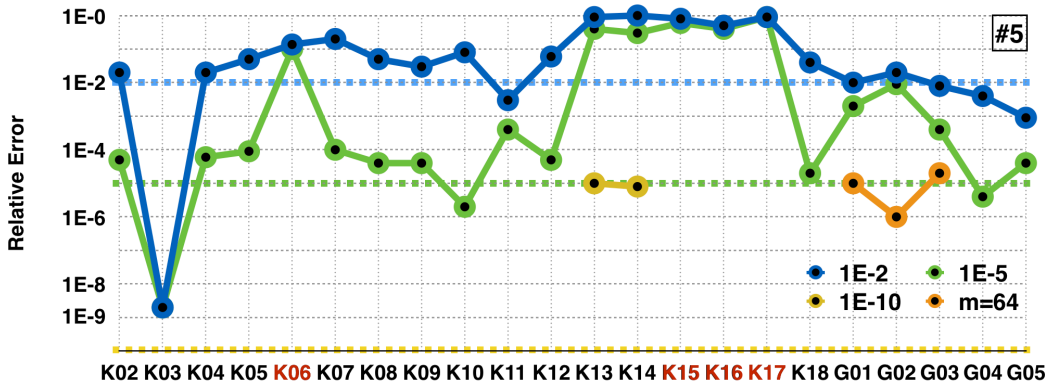

Figure 4.4: Relative error ϵ_2 (y-axis, the smaller the better) on all matrices (x-axis) using angle distance. Blue bars use τ1E$-$2 and 1% budget (except for **K6**, **K15**, **K16**, **K17**, other matrices take 0.8s to compress and 0.1 to evaluate in average). Green bars use τ1E$-$5 and 3% budget (in average, compression takes 1s and evaluation takes 0.2s). Red labels denotes matrices that do not compress. **K13** and **K14** have hierarchical low-rank structure, but the adaptive `ID` underestimates the rank. **K13** and **K14** can reach high accuracy (yellow plots) with τ1E$-$10 and 3% budget (1.0s in compression and 0.2s in evaluation).

Gaussian bandwidth values are taken from [54] and produce optimal learning rates.

Evaluation metrics: We use relative error ϵ_2 defined as the following

$$\epsilon_2 = \frac{\|\widetilde{K}w - Kw\|_F}{\|Kw\|_F}, \text{ where } w \in \mathbb{R}^{N\times r} \tag{4.7}$$

to evaluate the accuracy of our approximation. This metric requires $\mathcal{O}(rN^2)$ work; to reduce the computational effort, we instead sample 100 rows of K. In all tables, we use "Comp" and "Eval" to refer to the compression and evaluation time in seconds, and "GFs" to the floating point operation efficiency `GFLOPS` per node.

4.4 Accuracy and Robustness

We examine the accuracy and robustness of `GOFMM` (up to single precision) with 24 different SPD matrices in Figure 4.4. Given $m = 512$, $s = 512$ and $r = 512$, we

report relative error ϵ_2 on matrices **K02-18** and **G01-G05** using the **Gram angle** distance with two tolerances: 1E−2 (in blue) and 1E−5 (in green). Throughout, except for **K06**, **K15**–**K17** (high rank), **K13**, **K14** (underestimating the rank), and **G01**–**G03** (requiring smaller leaf size m), other matrices can usually achieve high accuracy with tolerance 1E−5 (taking 0.9s in compression and 0.2s in evaluation on a Haswell compute node).

Our adaptive `ID` underestimates the rank of **K13** and **K14** such that ϵ_2 is high. By imposing a smaller tolerance 1E−10 (yellow plots), both matrices reach 1E−5 (1s in compression and 0.2s in evaluation). **K6**, **K15**–**K17** have high ranks in the off-diagonal blocks; thus they cannot be compressed with $s = 512$ and 3% budget. **G01**–**G03** requires direct evaluation in the off-diagonal blocks to reach high accuracy. When we reduce the leaf node size from 512 to 64, we can still reach 1E−5 (orange plots). However, decreasing leaf size to 64 results in a longer wall-clock time ($0.8s$ in evaluation), because small m hurts performance.

Overall, we can observe that `GOFMM` can quite robustly discover low-rank plus sparse structure from different SPD matrices. We now investigate how increasing the cost (either with higher rank or more direct evaluations) can improve accuracy.

Comparison between `FMM` and `HSS`: In Figure 4.5 we show that even with more direct evaluations, `FMM` can be faster than `HSS` for achieving the same accuracy. For `HSS` the relative error (blue plots) in the leftmost plot plateaus at 5E−4. Further increasing rank from 256 to 512 (or even 1,024) results in $O(s^3)$ work (green bars). Using a combination of low-rank ($s = 64$) and 3% direct evaluation, FMM can achieve higher accuracy with little increment in the evaluation time (compression time remains the same). Similarly, in the rightmost plot, we can observe that by using $s = 512$ and 3% budget we achieve better accuracy than the `HSS` approximation ($s = 2048$) in less time.

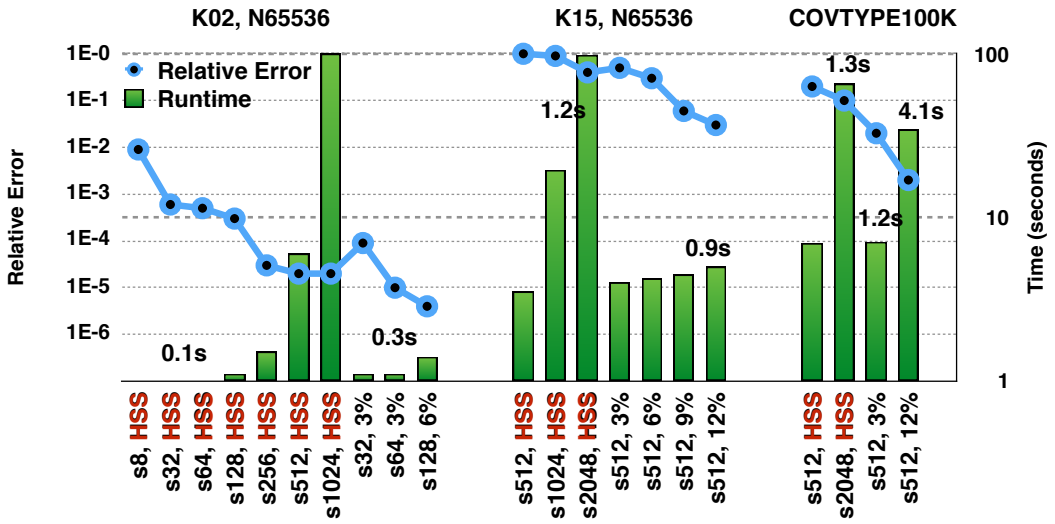

Figure 4.5: Comparison between HSS and FMM in wall-clock time (seconds, green bars, right y-axis) and accuracy (ϵ_2, blue plots, left y-axis). We use **K02**, **K15** ($m512$) and **COVTYPE** ($m800$) datasets. The fixed rank and budget are labeled on x-axis. The green bar is the total wall-clock time including compression and evaluation on 512 right hand sides. For some experiments, we also provide wall-clock time for evaluation to contrast the trade-off of using high rank and high budget.

4.5 Strong and Weak Scaling

Shared-memory strong scaling: In Figure 4.6 we use a 24-core Haswell and a 68-core KNL to perform strong scaling experiments. Each set of experiments contains 6 bars including three different parallel schemes on both **compression** phase (Algorithm 3.1) and **evaluation** phase (Algorithm 3.2). Strong scaling measures the parallel efficiency maintainability by fixing the problem size and increasing the number of cores (or threads).

The blue dots indicate the absolute efficiency (ratio to the peak) of our **evaluation** using dynamic scheduling. **COVTYPE** requires 12% budget with average rank 487 to achieve 2E−3. This compute-bound problem can reach 65% peak performance on Haswell and 33% on KNL. However, **K02** only requires 3% budget with average rank 35 to achieve 5E−5. As a result, this memory-bound problem does

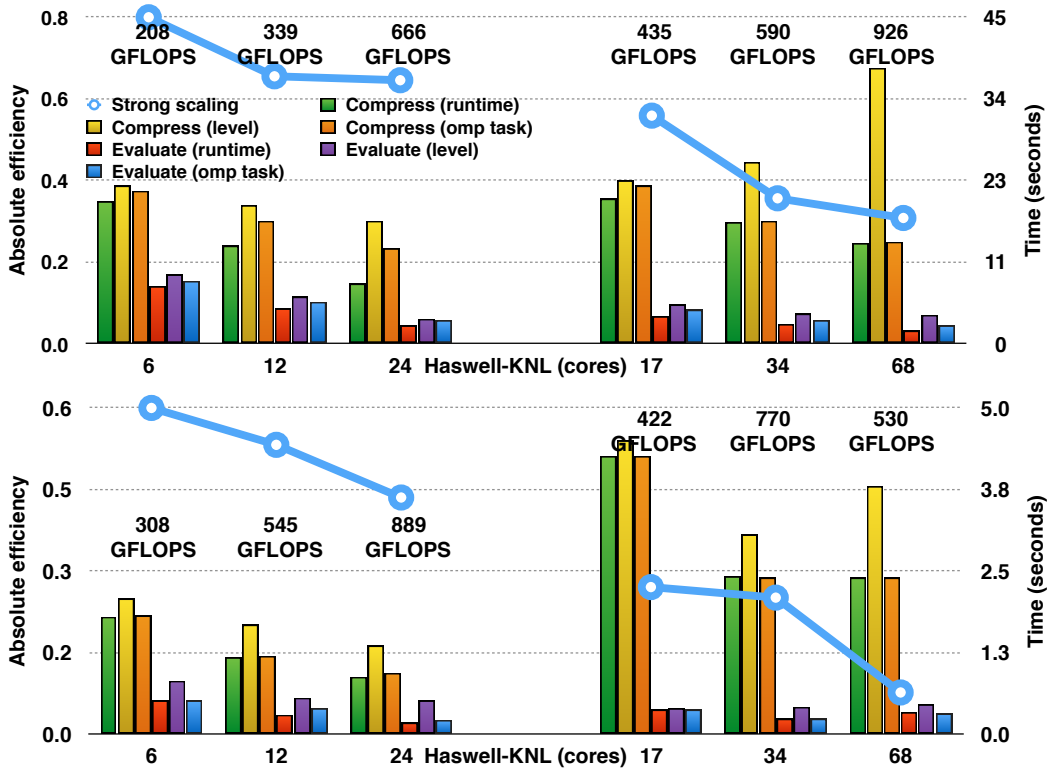

Figure 4.6: Strong scaling on a single Intel Haswell and KNL node (y-axis, time in seconds on the right, absolute efficiency to the peak GFLOPS on the left). We use $s = 512$, $\tau = 1E-5$ and $r = 512$. Experiments above use **COVTYPE** to create a Gaussian kernel matrix with $m = 800$ and 12% budget ($h = 0.1$), achieving $\epsilon_2 = 2\text{E}{-}3$ with average rank 487. Experiments below use **K02** with $m = 512$ and 3% budget, achieving $\epsilon_2 = 5\text{E}{-}5$ but only with average rank 35. We increase the number of cores up to 24 Haswell cores and 68 KNL cores. Each set of experiments contains compression time and evaluation time on three different parallel schemes: wall-clock time, level-by-level and omp tasks.

not scale (46% and 8%[7]) very well. In the rightmost two figures, we can even observe slow down from 34-core to 68-core. This is because the task bounds the wall-clock

[7]The average rank of **K02** is too small. Except for `L2L` tasks, other tasks can only reach about 5% of the peak during the evaluation. We suspect that MKL' `SGEMM` uses a 30×16 micro-kernel to perform a $30 \times 256 \times 16$ rank-k update each time. For an $m \times k \times n$ `SGEMM` to be efficient, m and n usually need to be at least four times of the micro-kernel size in each way. In the evaluation phase of **K02**, many `SGEMM`s have $m < 30$. Still the micro-kernel must compute $2 \times 30 \times 256 \times 16$ FLOPS. These sparse FLOPS are not counted in our experiments.

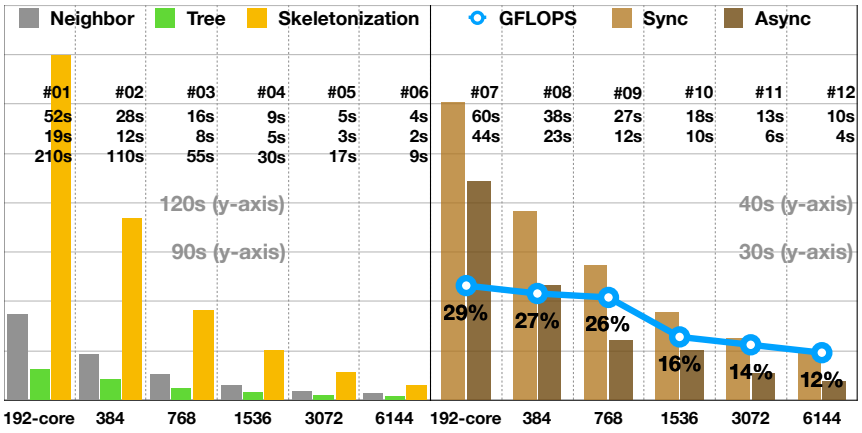

Figure 4.7: **Strong scaling (runtime in seconds as a function of the number of cores)** of `GOFMM` applied to a 5M-by-5M Gaussian kernel matrix generated by dataset SUSY. **Left:** Compression time and break down into three phases: neighbor search, tree creation, and skeletonization. **Right:** Matrix-matrix multiplication of kernel matrix with 5M-by-512 matrix for both synchronous kernel (light brown) and asynchronous version (dark brown). (6,144 Skylake cores correspond to 128 Stampede-2 nodes.) As mentioned, the y-axis denotes time in seconds, but the scales of the two figures are different. We annotate the scale in the middle of each figure. For the multiplication phase, we further provide absolute efficiency measured as the ratio between achieved `GFLOPS` and theoretical peak (≈4.3 `TFLOPS`).

time in the critical path; thus, increasing the number of cores does not help.

Throughout, we can observe that the wall-clock time for compression is less than the level-by-level and `omp task` traversals. While the work of `SKEL` is bounded by $2s^3$, parallel `GEQP3` in the level-by-level traversal does not scale (especially on KNL). On the other hand, task-based implementations can execute `COEF` out-of-order to maintain the parallelism. Our wall-clock time is better than `omp task` since we use the cost-estimate model for scheduling.

Distributed-memory strong scaling: In Figure 4.7 (#1, #2, #3, #4, #5, #6, #7, #8, #9, #10, #11, and #12), we use the high-energy physics 18-D point dataset **SUSY** in order to evaluate matrix entries using a Gaussian kernel. Using this matrix, we perform strong scaling experiments using up to 6,144 Skylake cores

(128 compute nodes, using one MPI process per node and 48 OpenMP threads).

The compression phase on the left is memory-bound; thus, we only report runtime of different stages in seconds (y-axis). The bar charts reveal the scaling trend. We also report the raw values (in seconds) below the labels. The multiplication phase with 512 right-hand sides is compute-bound. For this phase, we report the absolute floating point arithmetic efficiency computed as the ratio of the achieved GFLOPS over the theoretical peak (4.3 TFLOPS per node Section 4.3). We do not report GFLOPS performance for the compression phase since it is mostly memory-bound due a pivoted QR factorization during the most time-consuming phase—skeletonization. The pivoted QR can be done either with GEQP3 (default) or with the more recent HQRRP [59] (roughly 1.5× faster than GEQP3 but still achieving less than 10% peak performance).

Comparing #1 to #6, we also observe that the neighbor search efficiency (gray bars) degrades 59%—mainly due to the Alltoallv redistribution on neighbor candidates from [IDS] to [CYC] partitioning Section 3.2. Overall tree partition, interaction lists, and symmetrization (yellow bars) degrade by 69%. Building symmetry interaction lists requires several Alltoallv() calls and is expensive. Skeletonization (yellow bars) is the most scalable part of the compression phase despite the fact that its GFLOPS is not as high as the multiplication phase. Skeletonization only involves point-to-point communication within each tree node, which can be overlapped by other tasks such as importance sampling, GEQP3, TRSM, or caching. As a result, the performance only degrades by 23%. Overall, the compression efficiency degrades by 41%, while increasing core numbers from 192 to 6,144 in our strong scaling results.

Distributed-memory weak scaling: In Figure 4.8 (experiments #13, #14, #15, #16, #17, #18, #19, #20, #21, #22, #23, #24, #25, #26, #27, and #28), we perform weak scaling experiments on synthetic Gaussian kernel matrices with up to 128 MPI processes (6,144 cores). The GOFMM parameters are adjusted to keep the

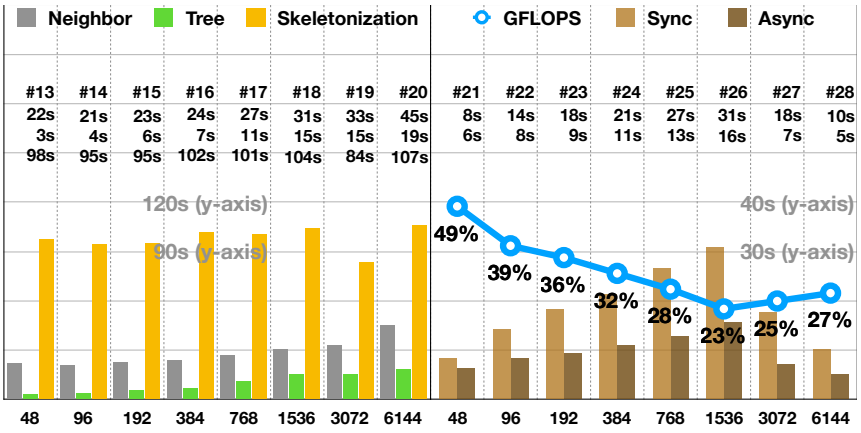

Figure 4.8: **Weak scaling (runtime in seconds vs cores)** of `GOFMM` applied to Gaussian kernel matrices generated synthetically with point clouds in 6-D. **Left:** Compression time and break down into three phases: neighbor search, tree creation, and skeletonization. **Right:** Matrix-matrix multiplication of kernel matrix with N-by-512 matrix for both synchronous kernel (light brown) and asynchronous version (dark brown). We also report the exact timings in the embedded table, the first row is the experiment index, the second row is the time for the synchronous mode (using `alltoallv()` and the third row is asynchronous mode using the scheme described in §3.2, Algorithm 4.1). The grain size is 524k points per MPI process. The largest problem size involves the multiplication of the 67M-by-67M kernel matrix by the 67M-by-512 matrix of random vectors. We report results for this problem in #20 and #28. We also report the absolute efficiency (as opposed to the performance in 48 cores) of the asynchronous matrix-matrix multiplication case (i.e., observed `FLOPS` over peak `FLOPS`)

accuracy fixed to about 1E-2. Here we also keep the bandwidth fixed. An alternative would be to scale the bandwidth based on some theoretical regression criterion, but since is rather complex and depends on the application, we have opted for keeping the bandwidth fixed.

One potential artifact is that $|\texttt{Near}(\alpha)|$ (the length of the near interaction list) may double in the worst case after the symmetry interaction list is built. As a result, the multiplication time may not reveal the weak scaling efficiency, because the actual computed `FLOPS` do not scale linearly with N even when the budget is controlled. Another potentially biasing factor is that the Gaussian kernel bandwidth

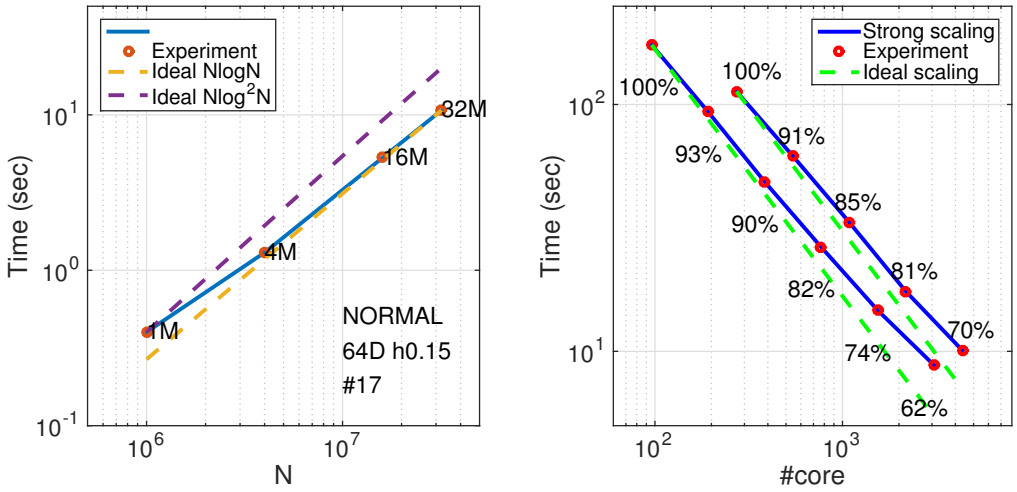

Figure 4.9: $\mathcal{O}(N \log N)$ verification (left) and strong scaling (right): We uses **NORMAL** with $m = 512$, $\kappa = 128$, and $s = 256$ to examine the log-linear complexity. The blue lines are experimental factorization time, and yellow lines are theoretical (ideal) time. The strong scaling experiment uses **NORMAL1M** with $\kappa = 128$, $m = s = 2048$. We increase the number of nodes up to 128 Haswell nodes (3,072 cores) and 64 KNL nodes (4,352 cores). The green lines represent the ideal scaling.

is fixed for all experiments. As a result, as we add more points the support of the kernel becomes narrower, which makes the matrix easier to compress. To somewhat compensate for this potentially positive bias, we provide the absolute efficiency (in blue) (as opposed to normalizing to the performance observed with 48 cores).

From #13 to #20, efficiency (gray bars) degrades 51% in neighbor search, 81% in tree partition and building the symmetry interaction lists. We observe that the synchronous multiplication algorithm doesn't scale that well and this is due to the `Alltoallv()`, which scales badly even with 32 `MPI` processes (#17). In contrast, our asynchronous multiplication algorithm that avoids `Alltoallv()` achieves better performance—it is 2× faster for most of our experiments. For example, for run #26–#27 `Alltoallv()` (which is part of the synchronous multiplication) requires 12.2, and 6.1 seconds respectively, which is almost as much as the time required

for the asynchronous scheme. Skeletonization only loses 8% in efficiency; since it dominates the compression runtime, the compression phase achieves 72% efficiency in our weak scaling experiments. Observe that runtimes of #19, #27 and #28 reduce—somewhat unexpectedly. One possible explanation is that the matrix is more compressible and that the communication patterns more local. Overall, our method is able to maintain roughly 20% of peak performance (for the specific matrix) in the weak scaling experiments.

Factorization strong scaling: In Figure 4.9, we use 128 Haswell nodes (3,072 cores) and 64 KNL nodes (4,352 cores) and increase N from 1M to 32M. We can observe that our implementation is very close to the theoretical $N \log N$ scaling (yellow) but lower than the $N \log^2 N$ scaling (purple). In the right figure, we fix the data set (**NORMAL 1M**) and increase the number of cores. The green line is the ideal scaling (100%), and our implementation reaches 62% efficiency on 3,072 Haswell cores and 70% on 4,352 KNL cores. This relatively small problem (1M) cannot fully exploit all computing resources; thus, we can see the degradation while N is small or when the number of cores is large (~ 230 points per core for 64 KNL nodes).

4.6 Kernel Ridge Regression

Following [54], we conduct non-linear classification experiments on dataset **COVTYPE**, **SUSY**, and **MNIST** using kernel ridge regression with a Gaussian kernel. Given N training data points $\{x_i\}_{i=1}^N \in \mathbb{R}^d$ and labels $u \in \mathbb{R}^N$, the method predicts the label of an unseen data point x as

$$\sum_{i=1}^{N} \mathcal{K}(x, x_i) w_i, \tag{4.8}$$

#	M	T_{train}	#iter	Train (%)	Test (%)	ρ
COVTYPE						$(\lambda I + \tilde{K})w = u.$
#29	I	231	100	99%	96%	4e-3
#30	HSS	62	0	99%	96%	2e-11
COVTYPE						$(\lambda I + \tilde{K} + S)w = u.$
#31	I	1725	100	99%	97%	3e-3
#32	HSS	950	43	99%	97%	1e-3
SUSY						$(\lambda I + \tilde{K})w = u.$
#33	I	3056	49	80%	79%	1e-3
#34	HSS	870	0	80%	79%	1e-14
SUSY						$(\lambda I + \tilde{K} + S)w = u.$
#35	I	11519	47	80%	79%	1e-3
#36	HSS	3419	7	80%	79%	1e-3
MNIST						$(\lambda I + \tilde{K})w = u.$
#37	I	185	5	100%	100%	1e-9
#38	HSS	36	0	100%	100%	2e-14

Table 4.2: Classification results using 640 cores: we use either I or the HSS factorization of K as preconditioners. The number if GMRES iterations is denoted by #iter. When it is 0, it means immediate convergence because the solver is highly accurate. The GMRES solver will terminate while reaching 100 iteration steps or $\rho < 1e-3$.

where the weights $w \in \mathbb{R}^N$ should be chosen to minimize the following objective

$$\frac{1}{2}\|u - Kw\|_2^2 + \lambda\|w\|_2^2. \tag{4.9}$$

In the context of binary classification tasks, we simply convert the real value output w_i to binary labels $+1$ or -1 depending on `sign(`w_i`)`. Alternatively, one can perform a kernel logistic regression instead. The objective function above is convex and has a closed form solution $w = (\lambda I + K)^{-1}u$, which can be solved either directly with $\mathcal{O}(N^3)$ work or iteratively with $\mathcal{O}(N^2)$ work per iteration. GOFMM can approximately solve the system with a preconditioned iterative solver (GMRES) by taking advantage of the $\mathcal{H}$-matrix factorization as a preconditioner.

In Table 4.2, the training time T_{train} is presented in pairs to show both the runtime and the iteration steps of the GMRES solver. Four experiments are conducted

for each dataset, combining different kernel matrix approximations ($\lambda I + \tilde{K}$ or $\lambda I + \tilde{K} + S$) and different preconditioners (no preconditioning or `HSS` factorization). We present both training and testing (10K samples) accuracy, and we present the normalized residual ρ upon termination of `GMRES`.

Approximating the kernel matrix using $\tilde{K}$, we reach 96% accuracy in **COVTYPE**, 79% accuracy in **SUSY** and 100% accuracy in **MNIST**. Using the nearest-neighbor-based separation criterion which creates a nonzero S, we can reach a higher accuracy (97%) in **COVTYPE**. In #30, #34 and #38, we use $(\lambda I + \tilde{K})^{-1}u$ as an initial guess, which allows `GMRES` to converge immediately to high precision by only applying the preconditioner. The training time only involves the factorization time and the solving time, which are both deterministic. Without preconditioners, `GMRES` takes 3× to 5× longer to converge to the desired accuracy. During the cross-validation, time spent on training may be much longer depending on the combination h and λ.

In runs #32 and #36, we show that the $\mathcal{H}$-matrix solver also works well as a preconditioner when we include the nearest-neighbors S in the `MATVEC` approximation. Comparing #31 to #32, we can find that #32 with the $\mathcal{H}$-matrix solver only takes 43 iterations to converge to 1e-3. On the other hand, #31 reaches the maximum iteration limit, just converging to 3e-3. Involving nearest-neighbors S can better approximate K in some cases. For example, #32 can reach a higher classification accuracy than #30. Overall our $\mathcal{H}$-matrix solver provides a 2× to 3× speedup.

Kernel ridge regression discussion: Overfitting is a known problem for the ridge regression task, resulting in numerical stability issue and huge fluctuation for predicting rarely seen data (or unseen). Other than increasing the regularization parameter λ (smoothing), kernel ridge regression usually employs iterative gradient decent base methods to solve the optimization problem approximately with an early termination. Although here we solve the inverse problem directly, which seems to ignore the overfitting problem, the fact is that we never compute the true inverse of

$\lambda I + K$. Instead we compute the approximate inverse $(\lambda I + \widetilde{K})^{-1}$. Thus, similar to the gradient-based methods we can also avoid the overfitting problem by controlling λ and the accuracy of $\widetilde{K}$ with s and the level restriction scheme.

4.7 Summary

We demonstrate the robustness and effectiveness of our geometry-oblivious FMM, the scalability of our H-matrix algorithms, and the accuracy and efficiency while applying our methods to kernel ridge regression task. We also demonstrate we design our systems and algorithms hand-by-hand to help exploit the *out-of-order* parallelism and perform architecture-dependent optimization. Overall, these implementation details and results show how this dissertation advances the field of high-performance $\mathcal{H}$-matrix methods.

Chapter 5

Related Work

The literature on $\mathcal{H}$-matrix and fast multipole methods is vast. Our discussion is brief and limited to the most closely related work. Recall that an $\mathcal{H}$-matrix approximation permutes and decomposes a matrix K such that

$$\widetilde{K}_{\alpha\alpha} = \begin{bmatrix} \widetilde{K}_{\mathtt{ll}} & 0 \\ 0 & \widetilde{K}_{\mathtt{rr}} \end{bmatrix} + \begin{bmatrix} 0 & S_{\mathtt{lr}} \\ S_{\mathtt{rl}} & 0 \end{bmatrix} + \begin{bmatrix} 0 & U_{\mathtt{l}} V_{\mathtt{r}} \\ U_{\mathtt{r}} V_{\mathtt{l}} & 0 \end{bmatrix}. \tag{5.1}$$

We recap the three challenges of efficiently constructing $\mathcal{H}$-matrix approximations and their fast matrix operations. We also review related work in task scheduling and high-performance linear algebra algorithms.

5.1 Matrix Permutation

Recall that before constructing any $\mathcal{H}$-matrix, one must first properly permute the matrix to expose the low-rank and sparse structure, because the $\mathcal{H}$-matrix structure is *not invariant* to row and column permutation. With points (coordinates), permutation of kernel matrices resembles data clustering. For examples, `ASKIT` [78] uses metric ball trees [63], and `MEKA` [68] uses κ-means [43] to perform panel clustering. When K

is sparse, the method of choice uses graph-partitioning. This does not scale to dense matrices because practical graph partitioning algorithms scale at least linearly with the number of edges and thus the construction cost would be at least $\mathcal{O}(N^2)$ [1, 45].

5.2 Low-rank Decomposition

The second step to constructing an $\mathcal{H}$-matrix approximation is to compute the low-rank decomposition for the off-diagonal blocks. The most popular approach for compressing arbitrary matrices is a global low-rank approximation using randomized linear algebra. In our $\mathcal{H}$-matrix decomposition Equation 5.1, this is equivalent to setting D and S to zero and constructing only U and V. Examples include the `CUR` [52] factorization, the Nystrom approximation [75], the adaptive cross approximation (`ACA`) [10], and randomized rank-revealing factorizations [57, 36]. These techniques can also be used for $\mathcal{H}$-matrix approximations when D is not zero. Instead of applying these methods directly to K, we can apply them to the off-diagonal blocks of K.

`FMM`-specific techniques that are a mix between analytic and algebraic methods include kernel-independent methods [60, 79] and the black-box `FMM` [25]. Constructing both U and V accurately and with optimal complexity is hard. The most robust algorithms require $\mathcal{O}(N^2)$ complexity or higher (randomized methods and leverage-score sampling) since they require one to "touch" all the entries of the matrix (or block) to be approximated.

5.3 Sparse Correction

The last challenge is to identify the sparse pattern S in the off-diagonal blocks. Excluding these high-rank sparse entries from low-rank approximations can effectively improve the accuracy. With points (coordinates), the sparsity of kernel matrices can be identified with nearest neighbors. For examples, `ASKIT` uses approximate κ-nearest

neighbor search [78] to introduce sparse correction in the off-diagonal blocks. For SPD matrices, `GOFMM` introduces geometry-oblivious κ-nearest neighbor search, which can also help identify the sparsity.

The most robust method to decompose a sparse matrix from a dense matrix is **robust PCA** (`RPCA`) [13]. Given tolerance ϵ, `RPCA` aims to decompose a dense matrix into a rank-s singular value decomposition (`SVD`) with a sparse correction with $\mathcal{O}(nnz)$ non-zeros. The process reduces s and nnz iteratively by gradually shifting entries with the highest error to the sparse correction term. `RPCA` is robust but expensive. Each iteration takes $\mathcal{O}(sN^2)$ work due to the truncated `SVD`, and the process typically requires more than ten iterations to converge.

5.4 Factorization

Factorization methods based on hierarchical decomposition have been studied for kernel matrices from points in two or three dimensions [5, 4, 9, 35], but less so in high dimensions with a few exceptions [46, 84, 83]. In the machine learning domain, most of the studies remain on refining low-rank approximation, but there are a few exceptions that use one-level clusters [68] or multi-resolution factorization [46] to improve the accuracy.

5.5 $\mathcal{H}$-matrix Classification

Treecodes and fast multipole methods originally were developed for N-body problems and integral equations. Algebraic variants led the way to the abstraction of $\mathcal{H}$-matrix methods and the application to the factorization of sparse systems arising from the discretization of elliptic PDEs [34, 9, 3, 31, 37, 76].

Let us briefly summarize the $\mathcal{H}$-matrix classification. Recall the decomposition Equation 5.1. If S is zero, the approximation is called a hierarchically off-diagonal low

Method	Matrix Interface	Low-rank	Permutation	S
FMM [19]	$\mathcal{K}(x_i, x_j)$	Analytic	Octree	Yes
KIFMM [79]	$\mathcal{K}(x_i, x_j)$	Equivalent	Octree	Yes
BBFMM [25]	$\mathcal{K}(x_i, x_j)$	Equivalent	Octree	Yes
HODLR [4]	K_{ij}	Algebraic	None	No
STRUMPACK [65]	K_{ij}	Algebraic	None	No
ASKIT [56]	$\mathcal{K}(x_i, x_j)$	Algebraic	Tree	Yes
MLPACK [22]	$\mathcal{K}(x_i, x_j)$	Equivalent	Tree	Yes
GOFMM	K_{ij}	Algebraic	Tree	Yes

Table 5.1: We summarize the main features of different $\mathcal{H}$-matrix methods/codes for dense matrices. "**Matrix Interface**" indicates whether the method requires a kernel function and points—indicated by $\mathcal{K}(x_i, x_j)$—or it just requires kernel entries—indicated by K_{ij}. "**Low-rank**" indicates the method used for the off-diagonal low-rank approximations: "Analytic" indicates kernel function-dependent analytic expansions; "Equivalent" indicates the use of equivalent points (restricted to low d problems); "Algebraic" indicates an algebraic method. "**Permutation**" indicates the permutation scheme used for dense matrices: "Octree" indicates that the scheme does not generalize to high dimensions; "None" indicates that the input lexicographic order is used; and "Tree" indicates geometric partitioning that scales to high dimensions. S indicates whether a sparse correction (FMM or $\mathcal{H}^2$) is supported.

rank (HODLR) scheme. In addition to S being zero, if the $\mathcal{H}$-matrix decomposition of D is used to construct UV, then we have a hierarchically semi-separable (HSS) scheme. If S is not zero we have a generic $\mathcal{H}$-matrix, but if the UV terms are constructed in a nested way then we have an $\mathcal{H}^2$-matrix or an FMM matrix depending on more technical details.

HSS and HODLR matrices lead to very efficient approximation algorithms for K^{-1}. However, $\mathcal{H}^2$ and FMM compression schemes better control the maximum rank of the U and V matrices than HODLR and HSS schemes. For the latter, the rank of U and V can grow with N [16], and the complexity bounds are no longer valid. Recently, there have been algorithms to effectively compress FMM and $\mathcal{H}^2$-matrices [21, 80].

One of the most scalable methods is STRUMPACK [26, 65, 58], which constructs an HSS approximation of a square matrix (not necessarily SPD) and then uses it to construct an approximate factorization. For dense matrices, STRUMPACK uses the

lexicographic ordering. If no fast matrix-vector multiplication is available, `STRUMPACK` requires $\mathcal{O}(N^2)$ work for compressing a dense SPD matrix, and $\mathcal{O}(N)$ work for the `MATVEC` and `SOLVE`.

5.6 Task-Based Parallelism

The proposed tree-based algorithms are parallelized by a task-based runtime system where the scheduler can statically or dynamically assign tasks to heterogeneous processors while all dependencies are satisfied. Early work on *out-of-order* scheduling is done on dense linear algebra domain [15]. Later the runtime system is generalized to exploit DAG-based task parallelism. [7, 69] are examples that provide a full set of API (or programming language) for general purposes. Early works discussing parallel operations for hierarchical matrices on shared memory system include bulk synchronous parallelization [48] and DAG-based task parallelism [49]. In this work, task dependencies are detected by a runtime data flow analysis.

5.7 Distributed-Memory Parallelism

Distributed hierarchical semi-separable (`HSS`) factorization and operations were discussed in [73, 42]. `STRUMPACK` [26, 65, 58] is the most scalable software that creates an `HSS` approximation. To our best knowledge, our previous work [84] and [83] are the most scalable software that factorizes `HODLR` and `p-HSS` matrices generated from `ASKIT`. With an `HSS` matrix, parallel `ULV` [70, 17] decomposition (generalized Schur decomposition) can be applied in $\mathcal{O}(N)$ work in both factorization and `SOLVE`. With Sherman-Morrison-Woodbury (SMW) formula, [84] can factorize `HODLR` matrices with $\mathcal{O}(\frac{N}{p}\log^2 N)$ time in parallel, and [83] can factorize `p-HSS` matrices with $\mathcal{O}(\frac{N}{p}\log N)$ time in parallel. For both [84, 83], a `SOLVE` takes $\mathcal{O}(\frac{N}{p}\log N)$ time in parallel.

5.8 N-body Computation Primitives

The computation occurs on each tree node in our tree-based algorithms include several different N-body operations. For examples, we use κ-nearest neighbors to decide the pattern of S in Equation 1.1. For kernel matrices, kernel summation (dense matrix-vector multiplication on submatrix) is required while multiplying both UV and S terms. These N-body computation primitives are usually implemented using high-performance linear algebra and math libraries. We redesign these primitives based on the `BLIS` (Basic Linear Algebra Instantiation Software) [72, 85] framework to expose more for optimization and modulization. The idea of fusing kernel evaluations, reduction, and selection into linear algebra subprograms can be found in [81]. Similar ideas to accelerate Strassen-like fast matrix-multiplication are explored in [41, 40].

Chapter 6

Conclusion

In this dissertation, we presented new contributions to the science and engineering high-performance $\mathcal{H}$-matrix algorithms. $\mathcal{H}$-matrix algorithms combine both low-rank and sparse matrix approximation and make use of a tree structure to approximate submatrices with different resolutions. As a result, $\mathcal{H}$-matrix methods are more robust and able to achieve higher accuracy than solely using either low-rank or sparse matrix approximation. The price that comes with these sophisticated methods is reflected in the domain-specific knowledge (geometry, physics, and analytical properties of the problem) required up front and the challenge in parallelization.

By observing and summarizing the common challenge in constructing and parallelizing $\mathcal{H}$-matrix methods, this dissertation demonstrates the principal components that systematically exploit the $\mathcal{H}$-matrix structure in a purely algebraic manner. The most significant innovation of this work is introducing geometry-oblivious distances that guide a hierarchical matrix partition and define neighbors on matrix entries without knowing any geometry, physics or analytical properties of the underlying problem. As a result, we can generalize $\mathcal{H}$-matrix methods to arbitrary SPD matrices such as graph Laplacian, Schur complement, and Hessian matrices.

We analyze and demonstrate the full data dependencies of our algorithms.

We understand the methods and the underlying computing architectures so well such that we can develop systems that automatically describe and dispatch dependent tasks to fully exploit the parallelism. As a result, our methods can scale on shared- and distributed-memory heterogeneous systems and achieve high absolute efficiency.

6.1 Contributions

We reiterate the contributions of this dissertation and corresponding conference and journal publications [82, 83, 84, 81, 56, 53, 54][1] to the field of computer science and computational science.

- A result from reproducing kernel Hilbert space theory is that any SPD matrix corresponds to a Gram matrix of vectors in some, unknown Gram (or feature) space [39]. Based on this result, the matrix entries are inner products, which we use to define distances. These distances allow us to design an efficient, purely algebraic $\mathcal{H}$-matrix method (`GOFMM`) for approximating arbitrary SPD matrices. To our knowledge, `GOFMM` is the first method that extends $\mathcal{H}$-matrix ideas to arbitrary matrices.

[1] Several my prior publications contribute to this document. In [82] (Chenhan D. Yu, James Levitt, Severin Reiz, and George Biros. Geometry-oblivious FMM for compressing dense SPD matrices.), I introduced `GOFMM` and $\mathcal{H}$-matrix schemes that can be used to compress arbitrary SPD matrices. In [84] (Chenhan D. Yu, William B. March, Bo Xiao, and George Biros. `INV-ASKIT`: a parallel fast direct solver for kernel matrices.), I introduced `INV-ASKIT` an $\mathcal{O}(N \log^2 N)$ direct solver for kernel matrices. In [83] (Chenhan D. Yu, William B March, and George Biros. An N log N parallel fast direct solver for kernel matrices.), I introduced an $\mathcal{O}(N \log N)$ hybrid direct-iterative solver for kernel matrices. In [81] (Chenhan D. Yu, Jianyu Huang, Woody Austin, Bo Xiao, and George Biros. Performance optimization for the κ-nearest neighbors kernel on x86 architectures.), I introduced an efficient algorithm for exhausted κ-nearest neighbor search on modern CPUs. By fusing neighbor search with GEMM in the assembly level, the proposed method is free from extra working space and suffer from smaller memory latency. In [55] (William B. March, Bo Xiao, Chenhan D. Yu, and George Biros. An algebraic parallel treecode in arbitrary dimensions.), In introduced an efficient algorithm for kernel summation on modern GPUs. This further shows that fusing additional operations with GEMM does not compromise its cache behavior, and such optimization can be applied to other N-body primitives. In [54] (William B. March, Bo Xiao, Sameer Tharakan, Chenhan D. Yu, and George Biros. Robust treecode approximation for kernel machines.), I conducted experiments for kernel ridge regression using `ASKIT` and apply efficient neighbor search and kernel summation kernels to reduce the training and inference time. In [53] (William B. March, Bo Xiao, Sameer Tharakan, Chenhan D. Yu, and George Biros. A kernel-independent FMM in general dimensions.), I introduce a hybrid algorithm that efficiently schedule kernel summation tasks to available CPUs and GPUs. In [56] (William B. March, Bo Xiao, Chenhan D. Yu, and George Biros. `ASKIT`: An efficient, parallel library for high-dimensional kernel summations.), I design and conduct experiments to empirically analyze the performance of `ASKIT`.

- We present parallel algorithms that construct an $\mathcal{H}$-matrix approximation (and its factorization) of a SPD matrix K in $\mathcal{O}(N \log N)$ work. The only required input to our algorithm is a routine that returns submatrix K_{IJ}, for arbitrary row and column index sets I and J. Once K is compressed and factorized, `MATVEC` of $\widetilde{K}$ and `SOLVE` of $\lambda I + \widetilde{K}$ can be applied in $\mathcal{O}(N \log N)$ time.

- Despite our method only requiring raw matrix values as input, the framework we present can use geometry information to improve its accuracy and performance if provided. For example, `GOFMM` has generic support to kernel matrices. While data points are provided, highly optimized N-body operators are exploited to boost the performance.

- We abstract key algorithmic components of `GOFMM` that generalizes to other tree-based methods. Algorithms are described as several tree traversals. Computation and communication on each tree node are tasks that follow the dependencies encoded (or analyzed at runtime) by the traversals. As a result, `GOFMM` can exploit *out-of-order* parallelism by task scheduling in distributed-memory, shared-memory heterogeneous architectures. We found that scheduling significantly improves the performance when compared to level-by-level tree traversals and synchronous message passing.

- We conduct extensive experiments to demonstrate the feasibility of the proposed approach. We test our code on 22 different matrices related to machine learning, stencil PDEs, spectral PDEs, inverse problems, and graph Laplacian operators. We perform numerical experiments on Intel Skylake, Haswell, and KNL, Qualcomm ARM, and NVIDIA Pascal architectures. Finally, we compare with three state-of-the-art codes: `HODLR`, `STRUMPACK`, and `ASKIT`.

This dissertation also contributes to the reproducibility and open source community. For details, please refer to the reproducible artifact in [82] and the source code

provided with the artifact.

6.2 Future Work

Possible directions that may further build and extend the work in this dissertation are suggested in this section.

- **Indefinite and unsymmetric extension:** Although the proposed methods can be applied to indefinite or symmetric matrices if the pairwise Euclidean distances are provided, geometry-oblivious distances do not generalize to these cases. One possible solution is to treat an unsymmetric matrix as the off-diagonal block of an SPD matrix. In this case, defining distance metrics requires estimating the unknown diagonal values of the SPD matrix.

- **Improving stability:** Although the proposed methods can guarantee symmetry, the positive definiteness of the $\mathcal{H}$-matrix approximation may be compromised when the relative error is large. As a result, Cholesky factorization may fail to decompose the $\mathcal{H}$-matrix we generate even with a positive regularization. How to fix this stability issue without affecting the asymptotic complexity of the algorithms requires more analytical studies.

- **$\mathcal{H}^2$-matrix factorization:** The $\mathcal{H}$-matrix factorization we presented in this dissertation assumes that the off-diagonal blocks are purely low-rank. Taking the advantage of this property, the factorization algorithm can be parallelized effectively. However, this factorization does not generalize to $\mathcal{H}^2$-matrices, which contain sparse correction in the off-diagonal blocks. The main issue is that the sparse fill-in generated during the factorization may compromise the low-rank structure. As a result, the factorization and solve will not scale. At this moment, we are actively looking for approximating schemes that can effectively control the fill-in and accuracy.

- **Extending use cases:** `GOFMM` requires the ability to evaluate arbitrary (and not contiguous) matrix entries, an option that may not be always available. If K is only available through matrix-free interfaces, these assumptions may not be satisfied. To support more use cases, we plan to introduce several matrix-free interfaces in the future.

6.3 Closing Remarks

Due to the magical $\mathcal{O}(N)$ and $\mathcal{O}(N \log N)$ complexity, $\mathcal{H}$-matrix methods and `FMM` exhibit irresistible power ever since they were introduced. It is easy to observe the complexity saving, but the detail and the machinery under the hood can be involved. Here, we summarize two insights from my science of high-performance algorithms for $\mathcal{H}$-matrices as the final takeaway.

To systematically build a class of $\mathcal{H}$-matrix methods, one must anatomize and understand all involving computations. We conclude $\mathcal{H}$-matrix methods with three common challenges and propose a class of geometry-oblivious techniques for a purely algebraic generalization. As a result, the complexity of `GOFMM` can be decomposed into contributions from low-rank, sparse factors, and tree traversals. With such design, `GOFMM` can also be benefited from the development of high-performance linear algebra libraries and optimized primitives.

To systematically exploit the performance and parallelism of $\mathcal{H}$-matrix methods, we build a runtime system hand-in-hand with `GOFMM` to help resolve the data dependencies. While describing algorithms in terms of tree traversals, we only annotate the reading/writing/communication activities, leaving dependency analysis to our system, which typically better discovers and resolves complicate and dynamic dependencies than human. The runtime system also manages the software and hardware mapping, which systematically resolves thread binding, computation communication overlapping, nested parallelism, and scheduling issues.

Literature and research on $\mathcal{H}$-matrix methods are vast, and this dissertation focuses its scope on the algebraic variants and their high-performance algorithms. Numerous related work inspires this dissertation, and I genuinely hope that the insights and the great details presented in this document can advance the field and inspire other future work.

Appendix A

Error Analysis

In Section 2.3, we have illustrated how interpolative decomposition (`ID`) is used to construct the low-rank factors with nested basis. To control the approximation error ϵ, we must decide the rank s (number of skeletons) of `ID`. To be specific, given a relative error bound ϵ we want to determine rank s in different off-diagonal blocks. We start by reviewing the error bound of the classic `ID` and later taking sampling into account. See more details in [56].

Interpolative decomposition (`ID`) error bound: Recall that the `ID` of submatrix $K_{I\beta}$ contains a pair of skeleton $\widetilde{\beta}$ and coefficient matrix $P_{\widetilde{\beta}\beta}$ such that

$$K_{I\beta} \approx K_{I\widetilde{\beta}} P_{\widetilde{\beta}\beta}. \tag{A.1}$$

The rank of $K_{I\widetilde{\beta}}$ and $P_{\widetilde{\beta}\beta}$ is $s = |\widetilde{\beta}|$. With the least square solution computed by a rank-revealing QR (`GEQP3`), it can be shown [20, 33] that

$$\frac{\|K_{I\beta} - K_{I\widetilde{\beta}}(K^{\dagger}_{I\widetilde{\beta}} K_{I\beta})\|}{\|K_{I\beta}\|} \leq \sqrt{1 + |\beta||\widetilde{\beta}|(|\beta| - |\widetilde{\beta}|)}\frac{\sigma_{\mathrm{s+1}}(K_{I\beta})}{\sigma_1(K_{I\beta})}, \tag{A.2}$$

where $\sigma_{\mathrm{s+1}}$ is the $(s+1)^{\mathrm{st}}$ largest singular value of $K_{I\beta}$. Observe that the accuracy

is mainly determined by the ratio between the $(s+1)^{\text{st}}$ ant the 1^{st} singular value. The faster the ratio decades, the faster the accuracy increases with rank $s = |\widetilde{\beta}|$.

Adaptive rank skeletonization: As a result, given an user-defined relative error tolerance ϵ, the required rank $s = |\widetilde{\beta}|$ can be selected according to Equation A.2. If we knew the singular values of the off-diagonal block $K_{I\beta}$, then we could control the approximation error by choosing s such that

$$\sqrt{1 + |\beta||\widetilde{\beta}|(|\beta| - |\widetilde{\beta}|)}\frac{\sigma_{\text{s}+1}(K_{I\beta})}{\sigma_1(K_{I\beta})} \leq \epsilon \tag{A.3}$$

for every $K_{I\beta}$. While this method can be effective, *s selected with this scheme is typically too large.* Notice that the relative errors of different off-diagonal blocks contribute differently to the overall relative error of our $\mathcal{H}$-matrix approximation. As a result, the criteria above should be relaxed according to the weighted contribution of $K_{I\beta}$ over the whole matrix K.

Recall that the full matrix-vector multiplication of K is contributed by several terms as $\sum_\beta K_{I\beta} w_\beta$. As a result, the approximation error can be expanded as

$$\frac{\sum_\beta \|K_{I\beta} w_\beta - K_{I\widetilde{\beta}}(K^\dagger_{I\widetilde{\beta}} K_{I\beta}) w_\beta\|}{\|K_{I:} w\|} \approx \sum_\beta \frac{\sigma_{s+1}(K_{I\beta})}{\sigma_1(K_{I\beta})} \mathcal{O}(\frac{\sigma_1(K_{I\beta})}{\sigma_1(K_{I:})}). \tag{A.4}$$

Observe that the relative error is sensitive to both $\frac{\sigma_{s+1}(K_{I\beta})}{\sigma_1(K_{I\beta})}$ (the spectrum decay ratio of $K_{I\beta}$) and $\frac{\sigma_1(K_{I\beta})}{\sigma_1(K_{I:})}$ (the weighted contribution over $K_{I:}$). This observation allows us to relax the adaptive rank selection when the weighted contribution of $K_{I\beta}$ is insignificant (while $\sigma_1(K_{I\beta})$ is relatively small).

Random and greedy skeletonization: With the observation in mind, we describe an alternative method in [56] to estimate the singular values of the off-diagonal block $K_{I\beta}$ and use them to choose an approximation rank. Instead of using

the relative targeting error ϵ, we specify the absolute error tolerance τ such that

$$\sqrt{1+|\beta||\widetilde{\beta}|(|\beta|-|\widetilde{\beta}|)}\sigma_{s+1}(K_{I\beta}) \leq \tau. \tag{A.5}$$

This alternative method assumes that $\sigma_1(K_{I\beta})$ is small (with a small relative contribution over $K_{I:}$). In `ASKIT` and `GOFMM`, this assumption holds when the metric tree effectively permutes the matrix to block diagonal dominance.

In practice, we need to estimate the singular values of $K_{I\beta}$ using our subsampled approximation and the diagonal elements in the upper triangular factor in the QR factorization computed for skeletonization [33]. To be specific, `ASKIT` and `GOFMM` compute the `ID` of $K_{I'[\widetilde{\mathtt{l}}\widetilde{\mathtt{r}}]}$ instead of $K_{I\beta}$ by introducing randomness to reduce the complexity. Here row samples I' are selected using neighbors as importance samples, and columns are greedily selected from children's skeletons ($\widetilde{\mathtt{l}}$ and $\widetilde{\mathtt{r}}$). Let R_{ii} be the ith diagonal value of the upper triangular matrix computed by `GEQP3`. We approximate the ith singular value of $K_{I\beta}$ according to [56] as

$$\sigma_i(K_{I\beta}) \approx \widetilde{\sigma}_i(K_{I\beta}) = R_{ii}\sqrt{\frac{|\beta|}{|\widetilde{\mathtt{l}}|+|\widetilde{\mathtt{r}}|}}\sqrt{\frac{|I|}{|I'|}}. \tag{A.6}$$

The scaling factor is derived from the uniform sampling case, where we assume I' is uniformly sampled from I. We then select rank $s = |\widetilde{\beta}|$ by the following criteria

$$s = \arg\min\{\mathrm{i} : \widetilde{\sigma}_{\mathrm{i}}(\mathrm{K}_{\mathrm{I}\beta}) < \tau\}. \tag{A.7}$$

In this way, we can account for submatrices which make a small contribution to the whole matrix-vector multiplication while still retaining the flexibility of the adaptive rank algorithm.

Error bound: Following [56], `GOFMM` (with **budget** $b = 0$) and `ASKIT` (with

number of neighbors $\kappa = 1$) have a similar error bound

$$\frac{\|\widetilde{K}w - Kw\|}{\|Kw\|} \leq (c_{\text{sample}} + c_{\texttt{ID}}) \log(\frac{N}{m}) \frac{\max_\beta\{\sigma_{s+1}(K_{I\beta})\}\|w\|}{\|Kw\|}, \tag{A.8}$$

where c_{sample} captures the random uniform sampling, $c_{\texttt{ID}}$ captures the error introduced by the interpolative decomposition, and the maximum is taken over all off-diagonal blocks $K_{I\beta}$. To be specific,

$$c_{\text{sample}} = 2 + \sqrt{1 + \frac{6(N-m)}{l}}, \tag{A.9}$$

$$c_{\texttt{ID}} = \sqrt{1 + ms(m-1)} + \log(\frac{N}{m})\sqrt{1 + 2s^3}, \tag{A.10}$$

where m is the leaf node size and l is the sample size. The factor $\log(\frac{N}{m})$ is from the accumulation of the `ID` error up the levels of the tree. The term

$$\frac{\max_\beta\{\sigma_{s+1}(K_{I\beta})\}}{\|Kw\|} \approx \mathcal{O}(\frac{\max_\beta\{\sigma_{s+1}(K_{I\beta})\}}{\sigma_1(K)}) \tag{A.11}$$

is the key to the approximation error `ASKIT` and `GOFMM`. Recall that $\frac{\sigma_{s+1}(K_{I\beta})}{\sigma_1(I)}$ is the spectrum decay ratio, and our adaptive skeletonization process described in Equation A.4 is adjusted proportionally according to $\sigma_{s+1}(K_{I\beta})$.

The discussion above only accounts for the case of pure hierarchical low-rank matrix approximation (when **budget** $b = 0$). The `FMM` case ($b > 0$) is more involved. In summary, `GOFMM` and `ASKIT` can better compress K if the hierarchical tree partition can effectively permute the matrix such that the spectrum of the off-diagonal blocks decays much faster than the on-diagonal block. Off-diagonal blocks with substantial contribution are expected to be captured by the nearest neighbors. As a result, these submatrices will be evaluated directly without approximation.

Bibliography

[1] Emmanuel Agullo, Eric Darve, Luc Giraud, and Yuval Harness. *Nearly optimal fast preconditioning of symmetric positive definite matrices*. PhD thesis, Inria Bordeaux Sud-Ouest, 2016.

[2] Dror Aiger, Efi Kokiopoulou, and Ehud Rivlin. Random grids: Fast approximate nearest neighbors and range searching for image search. In *Computer Vision (ICCV), 2013 IEEE International Conference on*, pages 3471–3478. IEEE, 2013.

[3] Sivaram Ambikasaran. *Fast algorithms for dense numerical linear algebra and applications*. PhD thesis, Stanford University, 2013.

[4] Sivaram Ambikasaran and Eric Darve. An $\mathcal{O}(n \log n)$ fast direct solver for partial hierarchically semi-separable matrices. *Journal of Scientific Computing*, 57(3):477–501, 2013.

[5] Sivaram Ambikasaran, Daniel Foreman-Mackey, Leslie Greengard, David W Hogg, and Michael O'Neil. Fast direct methods for Gaussian processes and the analysis of NASA Kepler mission data. *arXiv preprint arXiv:1403.6015*, 2014.

[6] A. Andoni and P. Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *Communications of the ACM*, 51(1):117, 2008.

[7] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. Starpu: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198, 2011.

[8] Sudipto Banerjee and Anindya Roy. *Linear algebra and matrix analysis for statistics*. Crc Press, 2014.

[9] Mario Bebendorf. *Hierarchical matrices*. Springer, 2008.

[10] Mario Bebendorf and Sergej Rjasanow. Adaptive low-rank approximation of collocation matrices. *Computing*, 70(1):1–24, 2003.

[11] Michele Benzi, Gene H Golub, and Jörg Liesen. Numerical solution of saddle point problems. *Acta numerica*, 14(1):1–137, 2005.

[12] Nicolas Bourbaki. *Algebra II: Chapters 4-7*. Springer Science & Business Media, 2013.

[13] Thierry Bouwmans and El Hadi Zahzah. Robust pca via principal component pursuit: A review for a comparative evaluation in video surveillance. *Computer Vision and Image Understanding*, 122:22–34, 2014.

[14] Nicola Cancedda, Eric Gaussier, Cyril Goutte, and Jean Michel Renders. Word sequence kernels. *Journal of Machine Learning Research*, 3:1059–1082, March 2003.

[15] Ernie Chan, Enrique S Quintana-Orti, Gregorio Quintana-Orti, and Robert Van De Geijn. Supermatrix out-of-order scheduling of matrix operations for smp and multi-core architectures. In *Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, pages 116–125. ACM, 2007.

[16] S. Chandrasekaran, P. Dewilde, M. Gu, and N. Somasunderam. On the mumerical rank of the off-diagonal blocks of Schur complements of discretized elliptic PDEs. *SIAM Journal on Matrix Analysis and Applications*, 31(5):2261–2290, 2010.

[17] Shiv Chandrasekaran, Ming Gu, and Timothy Pals. A fast ulv decomposition solver for hierarchically semiseparable representations. *SIAM Journal on Matrix Analysis and Applications*, 28(3):603–622, 2006.

[18] Chih-Chung Chang and Chih-Jen Lin. Libsvm: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 2(3):27, 2011.

[19] H. Cheng, Leslie Greengard, and Vladimir Rokhlin. A fast adaptive multipole algorithm in three dimensions. *Journal of Computational Physics*, 155:468–498, 1999.

[20] Hongwei Cheng, Zydrunas Gimbutas, Per-Gunnar Martinsson, and Vladimir Rokhlin. On the compression of low rank matrices. *SIAM Journal on Scientific Computing*, 26(4):1389–1404, 2005.

[21] P. Coulier, H. Pouransari, and E. Darve. The inverse fast multipole method: using a fast approximate direct solver as a preconditioner for dense linear systems. *ArXiv e-prints*, 2016.

[22] Ryan R. Curtin, James R. Cline, Neil P. Slagle, William B. March, P. Ram, Nishant A. Mehta, and Alexander G. Gray. MLPACK: A scalable C++ machine learning library. *Journal of Machine Learning Research*, 14:801–805, 2013.

[23] Bo Dai, Bo Xie, Niao He, Yingyu Liang, Anant Raj, Maria-Florina F Balcan, and Le Song. Scalable kernel methods via doubly stochastic gradients. In *Advances in Neural Information Processing Systems*, pages 3041–3049, 2014.

[24] S. Dasgupta and Y. Freund. Random projection trees and low dimensional manifolds. In *Proceedings of the 40th annual ACM symposium on Theory of computing*, pages 537–546. ACM, 2008.

[25] William Fong and Eric Darve. The black-box fast multipole method. *Journal of Computational Physics*, 228(23):8712–8725, 2009.

[26] Pieter Ghysels, Xiaoye S. Li, François-Henry Rouet, Samuel Williams, and Artem Napov. An efficient multicore implementation of a novel HSS-structured multifrontal solver using randomized sampling. *SIAM Journal on Scientific Computing*, 38(5):S358–S384, 2016.

[27] John R Gilbert, Cleve Moler, and Robert Schreiber. Sparse matrices in matlab: Design and implementation. *SIAM Journal on Matrix Analysis and Applications*, 13(1):333–356, 1992.

[28] Alex Gittens and Michael Mahoney. Revisiting the Nystrom method for improved large-scale machine learning. In *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, pages 567–575, 2013.

[29] Lars Grasedyck, Ronald Kriemann, and Sabine Le Borne. Parallel black box-lu preconditioning for elliptic boundary value problems. *Computing and visualization in science*, 11(4):273–291, 2008.

[30] A.G. Gray and A.W. Moore. N-body problems in statistical learning. *Advances in neural information processing systems*, pages 521–527, 2001.

[31] Leslie Greengard, Denis Gueyffier, Per-Gunnar Martinsson, and Vladimir Rokhlin. Fast direct solvers for integral equations in complex three-dimensional domains. *Acta Numerica*, 18(1):243–275, 2009.

[32] Greengard, L. Fast algorithms for classical physics. *Science*, 265(5174):909–914, 1994.

[33] Ming Gu and Stanley C Eisenstat. Efficient algorithms for computing a strong rank-revealing qr factorization. *SIAM Journal on Scientific Computing*, 17(4):848–869, 1996.

[34] Wolfgang Hackbusch. *Hierarchical matrices: Algorithms and analysis*. Springer Series in Computational Mathematics 49. Springer-Verlag Berlin Heidelberg, 1 edition, 2015.

[35] Wolfgang Hackbusch, Boris N Khoromskij, and Eugene E Tyrtyshnikov. Approximate iterations for structured matrices. *Numerische Mathematik*, 109(3):365–383, 2008.

[36] N. Halko, P.-G. Martinsson, and J.A. Tropp. Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions. *SIAM Review*, 53:217–288, 2011.

[37] Kenneth L Ho and Leslie Greengard. A fast direct solver for structured linear systems by recursive skeletonization. *SIAM Journal on Scientific Computing*, 34(5):A2507–A2532, 2012.

[38] Torsten Hoefler, Christian Siebert, and Andrew Lumsdaine. Scalable communication protocols for dynamic sparse data exchange. *ACM Sigplan Notices*, 45(5):159–168, 2010.

[39] Thomas Hofmann, Bernhard Schölkopf, and Alexander J Smola. Kernel methods in machine learning. *The annals of statistics*, pages 1171–1220, 2008.

[40] Jianyu Huang, Leslie Rice, Devin A Matthews, and Robert A van de Geijn. Generating families of practical fast matrix multiplication algorithms. *arXiv preprint arXiv:1611.01120*, 2016.

[41] Jianyu Huang, Tyler M Smith, Greg M Henry, and Robert A van de Geijn. Strassen's algorithm reloaded. In *Proceedings of the International Conference*

for High Performance Computing, Networking, Storage and Analysis, page 59. IEEE Press, 2016.

[42] Mohammad Izadi Khaleghabadi et al. Parallel H-matrix arithmetic on distributed-memory systems. 2012.

[43] Anil K Jain. Data clustering: 50 years beyond k-means. *Pattern recognition letters*, 31(8):651–666, 2010.

[44] P.W. Jones, A. Osipov, and V. Rokhlin. Randomized approximate nearest neighbors algorithm. *Proceedings of the National Academy of Sciences*, 108(38):15679–15686, 2011.

[45] George Karypis and Vipin Kumar. Multilevel k-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed computing*, 48(1):96–129, 1998.

[46] Risi Kondor, Nedelina Teneva, and Vikas Garg. Multiresolution matrix factorization. In *Proceedings of ICML14*, pages 1620–1628, 2014.

[47] Risi Imre Kondor and John Lafferty. Diffusion kernels on graphs and other discrete input spaces. In *ICML*, volume 2, pages 315–322, 2002.

[48] Ronald Kriemann. *Parallele algorithmen für H-matrizen*. Dissertation, Universität Kiel, 2004.

[49] Ronald Kriemann. H-LU factorization on many-core systems. Preprint, Max Planck Institute for Mathematics in the Sciences, 2014.

[50] M. Lichman. UCI machine learning repository, 2013.

[51] Zhiyun Lu et al. How to scale up kernel methods to be as good as deep neural nets. *arXiv preprint arXiv:1411.4000*, 2014.

[52] M.W. Mahoney and P. Drineas. Cur matrix decompositions for improved data analysis. *Proceedings of the National Academy of Sciences*, 106(3):697, 2009.

[53] William B. March, Bo Xiao, Sameer Tharakan, Chenhan D. Yu, and George Biros. A kernel-independent FMM in general dimensions. In *Proceedings of SC15*, The SCxy Conference series, Austin, Texas, November 2015. ACM/IEEE.

[54] William B. March, Bo Xiao, Sameer Tharakan, Chenhan D. Yu, and George Biros. Robust treecode approximation for kernel machines. In *Proceedings of the 21st ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 1–10, Sydney, Australia, August 2015.

[55] William B. March, Bo Xiao, Chenhan Yu, and George Biros. An algebraic parallel treecode in arbitrary dimensions. In *Proceedings of IPDPS 2015*, 29th IEEE International Parallel and Distributed Computing Symposium, May 2015.

[56] William B. March, Bo Xiao, Chenhan D. Yu, and George Biros. Askit: An efficient, parallel library for high-dimensional kernel summations. *SIAM Journal on Scientific Computing*, 38(5):S720–S749, 2016.

[57] P.-G. Martinsson, V. Rokhlin, and M. Tygert. A randomized algorithm for the decomposition of matrices. *Applied and Computational Harmonic Analysis*, 2010.

[58] Per-Gunnar Martinsson. Compressing rank-structured matrices via randomized sampling. *SIAM Journal on Scientific Computing*, 38(4):A1959–A1986, 2016.

[59] Per-Gunnar Martinsson, Gregorio Quintana Qrti, Nathan Heavner, and Robert van de Geijn. Householder QR factorization with randomization for column pivoting (HQRRP). *SIAM Journal on Scientific Computing*, 39(2):C96–C115, 2017.

[60] Per-Gunnar Martinsson and Vladimir Rokhlin. An accelerated kernel-independent fast multipole method in one dimension. *SIAM Journal on Scientific Computing*, 29(3):1160–1178, 2007.

[61] D.M. Mount and S. Arya. ANN: A library for approximate nearest neighbor searching. In *CGC 2nd Annual Fall Workshop on Computational Geometry*, 1997. `www.cs.umd.edu/~mount/ANN/`.

[62] K. R. Muske and J. W. Howse. A Lagrangian method for simultaneous nonlinear model predictive control. In L.T. Biegler, O. Ghattas, M. Heinkenschloss, and B. van Bloemen Waanders, editors, *Large-Scale PDE-constrained Optimization: State-of-the-Art*, Lecture Notes in Computational Science and Engineering. Springer-Verlag, 2001.

[63] Stephen Malvern Omohundro. *Five balltree construction algorithms*. International Computer Science Institute Berkeley, 1989.

[64] Jack Poulson, Bryan Marker, Robert A Van de Geijn, Jeff R Hammond, and Nichols A Romero. Elemental: A new framework for distributed memory dense matrix computations. *ACM Transactions on Mathematical Software (TOMS)*, 39(2):13, 2013.

[65] François-Henry Rouet, Xiaoye S. Li, Pieter Ghysels, and Artem Napov. A distributed-memory package for dense hierarchically semi-separable matrix computations using randomization. *ACM Transactions in Mathematical Software*, 42(4):27:1–27:35, June 2016.

[66] H. Samet. *Foundations of multidimensional and metric data structures*. Morgan Kaufmann, 2006.

[67] Bernhard Schölkopf and Alexander J Smola. *Learning with kernels: support vector machines, regularization, optimization, and beyond*. MIT press, 2002.

[68] Si Si, Cho-Jui Hsieh, and Inderjit S Dhillon. Memory efficient kernel approximation. In *ICML*, pages 701–709, 2014.

[69] Elliott Slaughter, Wonchan Lee, Sean Treichler, Michael Bauer, and Alex Aiken. Regent: A high-productivity programming language for hpc with logical regions. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 81. ACM, 2015.

[70] Gilbert W Stewart. Updating a rank-revealing ulv decomposition. *SIAM Journal on Matrix Analysis and Applications*, 14(2):494–499, 1993.

[71] Haluk Topcuoglu, Salim Hariri, and Min-you Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE transactions on parallel and distributed systems*, 13(3):260–274, 2002.

[72] Field G Van Zee and Robert A Van De Geijn. Blis: A framework for rapidly instantiating blas functionality. *ACM Transactions on Mathematical Software (TOMS)*, 41(3):14, 2015.

[73] Shen Wang, Xiaoye S Li, Jianlin Xia, Yingchong Situ, and Maarten V De Hoop. Efficient scalable algorithms for solving dense linear systems with hierarchically semiseparable structures. *SIAM Journal on Scientific Computing*, 35(6):C519–C544, 2013.

[74] R. Weber, H.J. Schek, and S. Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *Proceedings of the International Conference on Very Large Data Bases*, pages 194–205. IEEE, 1998.

[75] Christopher Williams and Matthias Seeger. Using the Nyström method to speed up kernel machines. In *Proceedings of the 14th Annual Conference on Neural*

Information Processing Systems, number EPFL-CONF-161322, pages 682–688, 2001.

[76] Jianlin Xia, Shivkumar Chandrasekaran, Ming Gu, and Xiaoye S Li. Fast algorithms for hierarchically semiseparable matrices. *Numerical Linear Algebra with Applications*, 17(6):953–976, 2010.

[77] Bo Xiao. *Parallel algorithms for the generalized n-body problem in high dimensions and their applications for bayesian inference and image analysis*. PhD thesis, Georgia Institute of Technology, Atlanta, GA, USA, 8 2014.

[78] Bo Xiao and George Biros. Parallel algorithms for nearest neighbor search problems in high dimensions. *SIAM Journal on Scientific Computing*, 38(5):S667–S699, 2016.

[79] Lexing Ying, George Biros, and Denis Zorin. A kernel-independent adaptive fast multipole method in two and three dimensions. *Journal of Computational Physics*, 196(2):591–626, 2004.

[80] Rio Yokota, Huda Ibeid, and David Keyes. Fast multipole method as a matrix-free hierarchical low-rank approximation. *Computing Research Repository*, abs/1602.02244, 2016.

[81] Chenhan D. Yu, Jianyu Huang, Woody Austin, Bo Xiao, and George Biros. Performance optimization for the k-nearest neighbors kernel on x86 architectures. In *Proceedings of SC15*, pages 7:1–7:12. ACM, 2015.

[82] Chenhan D Yu, James Levitt, Severin Reiz, and George Biros. Geometry-oblivious FMM for compressing dense SPD matrices. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 53. ACM, 2017.

[83] Chenhan D Yu, William B March, and George Biros. An n log n parallel fast direct solver for kernel matrices. In *Parallel and Distributed Processing Symposium (IPDPS), 2017 IEEE International*, pages 886–896. IEEE, 2017.

[84] Chenhan D. Yu, William B. March, Bo Xiao, and George Biros. INV-ASKIT: a parallel fast direct solver for kernel matrices. In *Proceedings of the IPDPS16*, Chicago, USA, May 2016.

[85] Field G Van Zee, Tyler M Smith, Bryan Marker, Tze Meng Low, Robert A Geijn, Francisco D Igual, Mikhail Smelyanskiy, Xianyi Zhang, Michael Kistler, Vernon Austel, et al. The blis framework: Experiments in portability. *ACM Transactions on Mathematical Software (TOMS)*, 42(2):12, 2016.

Vita

Chen-Han Yu was born in Taipei, Taiwan on 16 March 1987, the son of Shih-Kuang Yu and Chao-Jung Lee. He received Bachelor and Master of Science degrees in Mathematics from National Taiwan University in January 2010 and June 2012. Starting from August, he served his duty in Republic of China National Army at Kinmen island. He began his graduate studies at the University of Texas at Austin in September 2013.

Permanent E-mail Address: `chenhan@utexas.edu`

This dissertation was typeset with LaTeX 2_ε[1] by the author.

[1] LaTeX 2_ε is an extension of LaTeX. LaTeX is a collection of macros for TeX. TeX is a trademark of the American Mathematical Society. The macros used in formatting this dissertation were written by Dinesh Das, Department of Computer Sciences, The University of Texas at Austin, and extended by Bert Kay, James A. Bednar, and Ayman El-Khashab.