DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Efficient and Scalable Kernel Matrix Approximation using Hierarchical Decomposition

Keerthi Gaddameedi

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Efficient and Scalable Kernel Matrix Approximation using Hierarchical Decomposition

Effiziente und Skalierbare Approximation von Kernelmatrizen durch Hierarchische Zerlegung

Author:	Keerthi Gaddameedi
Supervisor:	Prof. Dr. Hans-Joachim Bungartz
Advisor:	Dr. Felix Dietrich, Dr. Tobias Neckl, Severin Reiz, Dr. Daniel Lehmberg
Submission Date:	15-June-2022

I confirm that this is my own work and I have documented all sources and material used.

Munich, 15-June-2022 Keerthi Gaddameedi

Acknowledgments

I would like to thank my supervisor Prof. Hans-Joachim Bungartz and all my advisors Dr. Felix Dietrich, Severin Reiz, Dr. Tobias Neckel, and Dr. Daniel Lemhberg for allowing me to learn and work with these amazingly complex softwares. I am especially grateful for Felix and Severin's immense support, patience, and guidance throughout this thesis. I would like to express gratitude to my former mentors Shripad Joshi and Nils Hoppe for motivating and inspiring me with their knowledge, during my time as a Bachelor's student and a HiWi. I am eternally indebted to my lovely parents who have never cared for anything but my well-being. Last but not least, I would like to thank all my friends at Studentenstadt for their constant involvement in encouraging me.

Abstract

Rapid expansion of data and its availability demands better and efficient ways to process, utilize, visualize and interpret it. Dimensionality reduction algorithms are used to mitigate the curse of dimensionality by extracting useful information and discarding the rest. But most common algorithms follow a linear approach. The intrinsic dimension of the real-world data may not always be in a linear space and hence the traditional approaches fail to produce meaningful results in such cases. Manifold learning algorithms aim to improve on this to also work on non-linear spaces. A manifold is the intrinsic geometric structure of data that is generated with similar processes. datafold is an open-source software that provides data-driven models to find a low-dimensional parametrization of these manifolds in non-linear data such as point cloud data. datafold provides a hierarchy of sub-packages that deal with creation of data, non-linear dimensionality reduction algorithms like diffusion maps and so on. These algorithms basically work with the concept of finding a kernel matrix that captures similarities between data points and performing eigendecompositions on the kernel matrices. The eigendecompositions lead to deduction of the intrinsic dimension of the data. We work with iterative methods for performing eigendecompositions which consist of numerous matrix-vector multiplications.

GOFMM is a portable and high-performance framework that provides implementations of operations such as matrix approximations, matrix-vector and matrix-matrix multiplication using hierarchical algorithms. It contains methods for approximating dense, symmetric and positive-definite matrices by partitioning the matrix into different parts depending on a certain distance metric. The distance metric is applied to partition relevant points from the points far away as they have almost no influence on each other. Then the distant points can be approximated. As mentioned above, eigendecompositions require numerous matrix-vector multiplications which are the most expensive operations in the process taking up to $\mathcal{O}(N^2)$ work. As a solution, we introduce hierarchical algorithms from GOFMM to perform these mat-vec operations with $\mathcal{O}(Nlog(N))$ work. Then the accuracy of eigendecompositions using scipy solver and GOFMM are compared. Multi-core and multi-node scalability experiments are performed on a linux cluster while identifying bottlenecks and analysing performance.

Contents

1. Introduction

1.1. Motivation

Data-driven approaches to solve real-world problems have led to rapid increase in data sizes. The potential of such approaches is limited by the current state of computation power. The space complexity of dense matrices .i.e matrices with mostly non-zero entries, is $\mathcal{O}(N^2)$. Similarly, the time complexity for operations such as *mat-vec* is $\mathcal{O}(N^2)$. Therefore, these operations become computationally expensive when the size of the matrices is large. As a solution to this, we aim to find low-rank approximations of these matrices using hierarchical algorithms. Geometric-oblivious fast multipole method (*GOFMM*) is a novel algorithm for approximating dense symmetric positive definite matrices so that the quadratic space and time complexity reduces to $\mathcal{O}(Nlog(N))$ with a small relative error. GOFMM is *geometry-oblivious*, meaning that it does not require the geometry information or the knowledge of how the data has been generated [1]. It just requires the distribution of data as input. Dense SPD matrices appear in areas such as scientific computing, data analytics and statistical inference. They appear in N-body methods [2], kernel methods in statistical learning [3], LU factorization [4] and so on.

The use of positive definite kernels in estimation and machine learning methods has seen a rapid growth in the past couple of decades. Real-world problems require solving high-dimensional data. Data-driven models assume an intrinsic geometry in the data, referred to as a manifold and it can be used to extract essential information of lower dimension. *datafold* is a python package that provides these data-driven models to find an explicit manifold parametrization for point cloud data [5] by using kernel matrices. Kernels correspond to dot products in a high dimensional feature space and we use them to efficiently solve non-linear cases in machine learning [3].

1.2. Problem Statement

Manifold learning approaches learn the intrinsic geometry of high-dimensional data without the use of predetermined classifications. There are several manifold learning algorithms such as isomap, locally linear embedding, hessian embedding and so on but we focus on diffusion maps. Like PCA, diffusion maps also consist of a kernel matrix computation that describes the relation of data points in the space. The normalized kernel matrix is decomposed to compute the eigen values and vectors. These eigen values and eigen vectors are used to find an embedding with a lower dimension than that of the ambient space. Since matrix-vector multiplications in iterative eigendecompositions is very expensive, especially for huge matrices, hierarchical approaches are applied. The intention of applying hierarchical algorithms on these kernel matrices is to reduce the quadratic run-time down to $\mathcal{O}(Nlog(N))$.

As previously discussed, GOFMM provides hierarchical algorithms for large, dense, symmetric and positive-definite matrices. Let $K \in \mathbb{R}^{NxN}$ be a dense kernel matrix for manifold data. Let it also be symmetric $K = K^T$ and positive-definite $x^TKx \;\forall\; x \in \mathbb{R}^{\mathbb{N}}, x \neq 0$. Our goal is to find an approximation $\widetilde{K}$ such that the matrix-vector multiplications consume only $\mathcal{O}(Nlog(N))$ work. The approximation must also satisfy the following condition

$$\frac{||\widetilde{K} - K||}{||K||} \;\leq\; \epsilon, \quad 0 < \epsilon < 1, \tag{1.1}$$

where ϵ is a user-defined tolerance. `LinearOperator` uses implicitly restarted Lanczos iteration to perform eigendecompositions. The matrix-vector multiplications in every iteration is then performed using hierarchical methods in `GOFMM`, where the dense matrix is first compressed and then evaluated. We want to test the performance of applying hierarchical algorithms from GOFMM to kernel matrices computed from manifold learning data in terms of accuracy and performance.

2. State of the art

2.1. Dimensionality Reduction

Dimensionality reduction is an approach to alleviate the curse of dimensionality. It begins with the assumption that data with high-dimensionality has an intrinsic dimension. Intrinsic dimension is the minimum number of features needed to meaningfully represent the data. Dimensionality reduction is the transformation of high-dimensional data into a representation with the aforementioned intrinsic dimension. Consider an $n \times D$ matrix $\mathbf{X}$ where n is the number of data points and D is the dimensionality of the points. The transformation of the data is an $n \times d$ matrix such that $d \ll D$.

2.1.1. PCA

Principal Component Analysis is a widely known linear dimensionality reduction algorithm. It aims to embed data $\mathbf{X} \in \mathbb{R}^{n \times D}$ in a d-dimensional linear subspace of $\mathbb{R}^D$ along which data has maximum covariance [6]. This is done by finding a linear mapping V that optimizes the cost function $trace(\mathbf{V}^T\mathbf{X}'\mathbf{V})$ where $\mathbf{X}'$ is the sample covariance matrix of $\mathbf{X}$ [7]. This linear mapping V is formed by d eigen vectors or principal components of $\mathbf{X}'$ which are obtained by the following eigendecomposition

$$X'V = \lambda V, \tag{2.1}$$

where λ consists eigen values of $\mathbf{X}'$ which are also the variances of corresponding eigen vectors. We take a subset of principal vectors V' with the largest variance as the new basis. Then, the low-dimensional representation $\mathbf{Y} \in \mathbb{R}^{n \times d}$ is the mapping of X onto the new linear basis V', meaning

$$Y = XV'. \tag{2.2}$$

PCA has a complexity of $\mathcal{O}(d^3)$ [6]. One of the main drawbacks of PCA is that it is a linear approach. Given a data set with non-linear intrinsic dimension, it may fail to produce a meaningful low-dimension representation of the data set.

2.1.2. Diffusion Maps

Diffusion Maps is a non-linear technique of dimensionality reduction. It tries to obtain information about the manifold encoded in the data without any assumptions on the underlying geometry. Alternative to euclidean distance or geodesic distance used in isomaps, we make use of an affinity or similarity matrix obtained using a kernel function that produces positive and symmetric values. Given a dataset $X \in \{x_1, x_2 ... x_n\}$ and a Gaussian kernel function, we can compute the similarity matrix as [8]

$$W_{ij} \quad = \quad w(i,j) \quad = e^{\dfrac{-||x_i - x_j||_2^2}{\sigma^2}}. \tag{2.3}$$

x_i and x_j are a pair of data points and σ is the radius of the neighborhood around point x_i. The affinity matrix is therefore a measure of similarity or connectivity between pairs of data points. We can represent the data as a graph with data points as nodes and the similarity matrix entries as the weighted edges. But since the similarities are collected with respect to local geometry, the distribution of data affects the approximations. Let Q denote the degree of each node, meaning that Q_i is the sum of all affinities associated with the node i. In order to make the influence of density Q explicit, we normalize the weights with a parameter α which consists of values between 0 and 1. If we take $\alpha = 0$, the density has maximal influence on how the underlying geometry is captured and vice versa when $\alpha = 1$. Therefore we normalize the weights with $\alpha = 1$ to obtain new weights [8]

$$W_{ij}^{\alpha} \quad = \quad \frac{W_{ij}}{Q_i^{\alpha} \cdot Q_j^{\alpha}}, \tag{2.4}$$

where $Q_i = \sum_{j=1}^{n} W_{ij}$ is the degree of vertex x_i. Now a Markov chain can be defined as

$$P_{ij} \quad = \quad \frac{W_{ij}^{\alpha}}{Q_i^{\alpha}}, \tag{2.5}$$

where P is the transition matrix whose entries are probabilities of transition from node to another. We obtain transition matrix P^t by performing random walks for t time steps. We then perform eigendecomposition on the transition matrix to compute eigenvalues λ_r and eigenvectors ψ_r of the transition matrix. We obtain the lower dimension by taking only a fraction of the eigenvalues whose values are larger than a threshold. If δ is the predetermined precision factor, the lower dimension $d(t)$ can be defined as max$\{ l : \lambda_l^t < \delta \lambda_1^t \}$ [9]. The mapping of data in a lower dimensional space $\mathbb{R}^{d(t)}$ can then be given as

$$\Psi_t(x) \quad = \quad [\lambda_1^t \psi_1(x), \lambda_2^t \psi_2(x), ... \lambda_{d(t)}^t \psi_{d(t)}(x)]. \tag{2.6}$$

The contents of $\Psi_t(x)$ are called *diffusion coordinates*. The diffusion distance is formulated as [8]

$$D_t^2(x,y) \quad = \quad ||\Psi_t(x) - \Psi_t(y)||_2^2 \quad = \quad \sum_{j=1}^{d(t)} \lambda_j^{2t}(\psi_j(x) - \psi_j(y))^2. \tag{2.7}$$

The diffusion distances are equal to euclidean distances in the lower dimensional space. The value of $D_t(x,y)$ is small if the nodes x and y are highly connected .i.e there are a large number of short paths between them. One of the benefits of diffusion distances is that they are taken over the sum of all paths of length t between x and y and hence are more robust with noise than geodesic distances [9].
To extend the mapping from the given data to unseen data samples, we approximate the new eigenvectors from the old ones without having to compute them again for the new affinity matrix. It is done by Nyström method and is given as [8]

$$\Psi_j(y) \quad = \quad \frac{1}{\lambda_j} \sum_{i=1}^{n} \Psi_i(x) \cdot k(x,y). \tag{2.8}$$

x and y are points from the old and new data sets respectively and $k(y,x)$ is the kernel matrix with affinities between embedded data and the new data. The computational complexity of diffusion maps is $\mathcal{O}(n^3)$ and the most expensive part is the eigendecomposition. Hence, we will take a look at how we can try to mitigate this by using hierarchical matrix approximations.

2.2. Datafold

Point cloud data is a set of unordered points in high-dimensional space. *Time series data* is a series of data points taken in successive equally spaced points in time. datafold is a python package that provides data-driven models for finding a parametrization of manifolds in the aforementioned point cloud data and to identify non-linear dynamical systems from time-series data [5]. The software architecture contains integrated models that have been implemented in a modularized fashion and an API that has been templated from scikit − learn library. The architecture as shown in Figure 2.1 consists of three layers and describes the hierarchy of workflow.

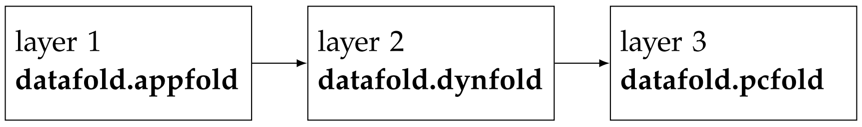


Figure 2.1.: Workflow hierarchy of *datafold*

2.2.1. datafold.pcfold

datafold.pcfold is the lowest layer in the workflow hierarchy that provides data structures and algorithms directly associated with data. The two data structures belonging to the layer are PCManifold and TSCDataFrame.

PCManifold

PCManifold is a class with data structures and algorithms to be applied on point cloud data. It is derived from numpy.ndarray and has a kernel function to compute similarity matrix using different distant metrics and eigen pairs [5]. As shown in Figure 2.2, the classes PCManifoldKernel, TSCManifoldKernel, DmapKernelFixed for the kernel functions is derived from BaseManifoldKernel. These classes further sub-classes for different types of kernel functions. Functions like estimate_cutoff() and estimate_scale() compute the threshold and ϵ for Gaussian radial basis kernel respectively.

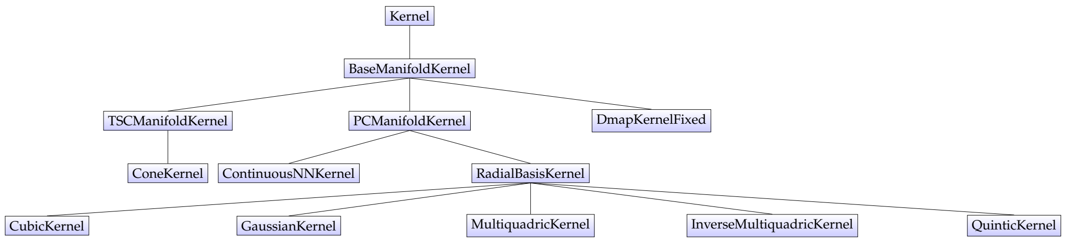


Figure 2.2.: Class inheritance diagram for datafold.pcfold

TSCDataFrame

TSCDataFrame deals with collections of time series data. It is derived from pandas.DataFrame and is used mainly to identify non-linear dynamical systems on the underlying geometry. It consists of classes such as TSCKFoldTime, TSCMetric, TSCScoring for K-fold splits on time values, computing metrics for time series collection, creating scoring function for the metrics respectively. All the aforementioned classes inherit from a base class named TSCCrossValidationSplit. Please refer to the software documentation on the website for datafold as it contains information about every function in detail.

2.2.2. datafold.dynfold

This layer contains models that can be used in analysis tasks and in meta-models in the higher layer of the workflow hierarchy. There are three types of models in this layer. The first type of models are subclasses of TSCTransformMixin and include methods to compute a new representation of data. For example, DiffusionMaps is used to find a lower-dimensional embedding of data, TSCPrincipalComponent for computing principal components of time series data and so on. The second type of models are derived from the class sklearn.base.RegressorMixin. They provide, for example, LaplacianPyramidsInterpolator which is a model for interpolation of function values on manifolds. The third and final type of model contains sub-classes of TSCPredictMixin which inherits from DMDBase. Dynamic mode decomposition algorithms linearly decompose time series data to spatial and temporal components which can be used to predict time series [5].

2.2.3. datafold.appfold

datafold.appfold is the highest layer in the hierarchy workflow. It contains meta-models that capture complex processing pipelines. These meta-models act as a single point of access for other sub-models in the class. They are intended to solve complex data-driven analysis tasks in the machine learning process. They consist of models that are sub-classes of sklearn.pipeline.Pipeline and sklearn.model_selection.GridSearchCV. The former pipeline provides extended dynamic mode decomposition algorithms to predict time series. The latter cross-validation pipeline searches user-defined parameter space that includes parameters over all the sub-models and includes data-splitting schemes for time series data [5].

2.3. Kernel matrix approximations

In this chapter, we define kernel matrices using an example and discuss general matrix approximation methods. We then explain hierarchical matrix representations and compare how these techniques perform in terms of space and run-time complexities.

2.3.1. Kernels

Consider a domain $\mathcal{X}$ with n input points x_i that map to target points y_j in $\mathcal{Y}$, where $i, j \in \{1...n\}$. We want to predict $y \in \mathcal{Y}$ for a new input $x \in \mathcal{X}$ such that it is similar to the training examples. To be able to do this, we need a similarity measure in $\mathcal{X}$ and $\mathcal{Y}$. Kernels are functions that take two inputs and output their similarity. To simplify, let us consider the binary classification problem .i.e $y \in \{\pm 1\}$. A kernel function for $\mathcal{X}$ is defined as

$$k : \mathcal{X} \times \mathcal{X} \to \mathbb{R}, \ \ (x, x') \longmapsto k(x, x') \quad \textit{and} \tag{2.9}$$

$$k(x,x') = \Phi(x)^{\mathrm{T}}\Phi(x'), \tag{2.10}$$

where Φ is the *feature map* of kernel k that maps into a dot product space of input domains called the *feature space* [3]. Thus the kernel or Gram matrix with kernel k and inputs $x_1, x_2..x_n \in \mathcal{X}$ can be defined as

$$\mathrm{K} := k(x_i, x_j)_{ij}. \tag{2.11}$$

A kernel matrix is symmetric positive definite if

$$\mathrm{K} = \mathrm{K}^{\mathrm{T}} \qquad \textit{and} \tag{2.12}$$

$$\sum_{i,j} c_i c_j \mathrm{K}_{ij} \geq 0, \quad \forall c_i \in \mathbb{R}. \tag{2.13}$$

2.3.2. Motivation from N-body problems

N-body problems are associated with predicting an object or an entity's behavior with respect to the interactions with n-1 other objects near or far from it. Examples include celestial body simulations, Coulomb's law, data analysis in geostatistics and so on. Let us consider Coulomb's law where there are N points $x_i \in \mathbb{R}^d$. If x_i is the target point, x_j is the source and w_j is the weight at the source point, the potential at target point can be computed as [10]

$$u_i = u(x_i) = \sum_{j=1}^{N} K(x_i, x_j) w_j. \tag{2.14}$$

This is in the form of matrix-vector multiplication which has a computational complexity of $\mathcal{O}(N^2)$. The approach to reduce these costs is to perform near-far pruning. Since the near-by points have a higher contribution to the potential, it makes sense to separate points by some bounds on the distance. We split the points into two sets $Near_i$ and Far_i, where $Near_i$ consists of points at distance less than a given threshold σ and Far_i contains points further away. The potential then becomes

$$u_i = \sum_{p \in Near_i} K(x_i, x_p) w_p + \sum_{p \in Far_i} K(x_i, x_p) w_p. \tag{2.15}$$

The computations for $Near_i$ are done individually without neglecting any points. But since Far_i do not contribute much to the potential u_i, the idea is to perform low-rank approximations on these points to reduce the number of computations.

2.3.3. Algebraic compression algorithms

In this section we discuss a couple of matrix approximation methods for general matrices.

Reduced Singular Value Decomposition

Consider the matrix-vector product from Equation (2.14),

$$u \;=\; Kw. \tag{2.16}$$

The idea is to compute an approximation of kernel matrices like K by methods like SVD.

The singular value decomposition of matrix A $\in \mathbb{C}^{mxn}$ is given by

$$\mathrm{A} \;=\; \mathrm{U}\Sigma\mathrm{V}^*, \tag{2.17}$$

where $A \in \mathbb{C}^{m\times n}$ is a matrix of rank r, $U \in \mathbb{C}^{m\times n}$ and $V \in \mathbb{C}^{nxn}$ are unitary matrices. $\Sigma \in \mathbb{R}^{m\times n}$ is a non-negative diagonal matrix whose entries are *singular values* or *principal values* of A. Singular values $\{\sigma_1, \sigma_2 ... \sigma_r\}$ are non-negative square roots of eigen values of AA^* and A^*A. SVD can be used to generate a low-rank matrix from a higher rank matrix [11]. It is called the reduced singular value decomposition. To obtain a low-rank approximation $\hat{A}$ of rank p from A of rank r, we sort the rows of Σ in descending order to get $\hat{\Sigma}$. We then set all elements of $\hat{\Sigma}$ to zero except the first p entries .i.e the p largest singular values [12] to obtain the reduced singular value decomposition

$$\hat{A} \;=\; U\hat{\Sigma}V^*. \tag{2.18}$$

This approach thus helps with dimensionality reduction but its still costs $\mathcal{O}(mn^2)$ (m $\geq$ n) to compute the matrix products.

Interpolative Decomposition

Despite the high accuracy provided by SVD, there are certain drawbacks such as $\mathcal{O}(mn^2)$ computational costs. Hence we look at interpolative decomposition method that uses A's own rows and columns to generate a factorization with a simple geometric interpretation. The factorization is given as

$$A \;= A_{col} \cdot B, \tag{2.19}$$

where $A \in \mathbb{C}^{m\times n}$ with rank $k < min(m, n)$ and $A_{col} \in \mathbb{C}^{m\times k}$ is a matrix consisting of a subset of columns from A. $B \in \mathbb{C}^{k\times n}$ consists of a identity matrix of size $k \times k$ and all of its values are less than 2 in absolute value. The cost of computing such a factorization is $\mathcal{O}(mnk)$. Although the costs are lower than that of SVD, there are disadvantages to this method such as loss in accuracy due to norms of B being greater than 1 and the non-uniqueness of the factorization. Interpolative decomposition can be calculated using QR decomposition as [13]

$$A_{col}B = A\Pi = QR = \begin{bmatrix} Q_{left} & Q_{right} \end{bmatrix} \begin{bmatrix} R_{11} & R_{12} \\ 0 & R_{22} \end{bmatrix}. \tag{2.20}$$

$Q \in \mathbb{C}^{m \times n}$ has orthonormal columns and $R \in \mathbb{R}^{n \times n}$ is an upper triangular matrix. Disregarding R_{22} since $||R_{22}|| = \mathcal{O}(\sigma_{k+1})$, Equation (2.20) can be further simplified to

$$A_{col}B = A\Pi = Q_{left}\begin{bmatrix} R_{11} & R_{12} \end{bmatrix}. \tag{2.21}$$

From Equation (2.21) we write

$$A_{col} = Q_{left}R_{11}. \tag{2.22}$$

Combining Equations (2.21) and (2.22), we get

$$Q_{left}R_{11}B = Q_{left}\begin{bmatrix} R_{11} & R_{12} \end{bmatrix}, \tag{2.23}$$

$$R_{11}B = \begin{bmatrix} R_{11} & R_{12} \end{bmatrix} \ \textit{and} \tag{2.24}$$

$$B = \begin{bmatrix} I_{k \times k} & \frac{R_{12}}{R_{11}} \end{bmatrix}. \tag{2.25}$$

A faster randomized procedure is introduced in [14] that applies a structured random matrix $R_{l \times m}$, $l > k$ to each column of A with cost proportional to $\mathcal{O}(mnlog(k) + l^2(m + n))$.

2.3.4. Hierarchical decompositions

The hierarchically low-rank approximation $\widetilde{K}$ of the kernel matrix K is given as [14, 15]

$$\widetilde{K} \ = \ D \ + \ S \ + \ UV, \tag{2.26}$$

where D is a block-diagonal matrix with every block being an $\mathcal{H}$-matrix, S is a sparse matrix and U, V are low rank matrices. The $\mathcal{H}$-matrix $\widetilde{K}$ is to be computed such that

$$\frac{||\widetilde{K} - K||}{||K||} \ \leq \ \epsilon, \tag{2.27}$$

where ϵ is the user defined tolerance and $0 < \epsilon < 1$. If S is zero in Equation (2.26), $\widetilde{K}$ is called hierarchically off-diagonal low-rank approximation and additionally if D is also zero, the approximation is called hierarchically semi-separable. The construction of $\widetilde{K}$ and matrix-vector product both take $\mathcal{O}(NlogN)$ work.

2.4. GOFMM

In this section, we discuss the algorithm and operations introduced in [1], to compute the hierarchical low-rank approximation of dense symmetric positive matrices. The algorithm has a *compression* and an *evaluation* phase.

2.4.1. Compression

The compression algorithm from GOFMM [1] is listed in Algorithm 1.

Algorithm 1 Compression(K)

```
1: HierarchicalPartitioning()
2: NeighborBasedPruning()
3: Skeletonization()
```

Hierarchical partitioning

The first step of compression is to construct the hierarchical structure of the given matrix K using *near-far pruning*. This is done by taking all points as the root node of a binary metric ball tree and splitting the node until *maximum leaf node size m*, using a distance metric. The idea of the distance metric is to separate indices that are near to each other from those that are far away since their contribution is negligible. Recall from Section 2.3.2, we have N points $\{x_1, x_2 ... x_N\}$. We now consider the following three [1] types of distances

Geometric distance The geometric distance is given by

$$d_{ij} = ||x_i - x_j||_2. \tag{2.28}$$

The points are partitioned such that the distance d_{ij} between points belonging to the same partition is minimum. The splitting terminates when the number of leaf nodes is equal to the predetermined maximum m. This distance metric takes $\mathcal{O}(NlogN)$ work.

Kernel distance As we can recall from section 2.3.1, kernel or gram matrix is a set of gram vectors such that $K_{ij} =< \phi_i, \phi_j >$. We compute kernel distance using three entries of the kernel matrix as follows

$$d_{ij} = ||\phi_i - \phi_j||_2 = K_{ii} + K_{jj} - 2K_{ij}. \tag{2.29}$$

Angle distance Both kernel and angle distances require $\mathcal{O}(N^2)$ work without sampling. Angle distance uses the angle between the gram vectors which is the sine distance between the inner product spaces. It is given as

$$d_{ij} = sin^2(\angle \phi_i, \phi_j) = 1 - \frac{K_{ij}^2}{K_{ii}K_{jj}}. \tag{2.30}$$

An approximate centroid is defined using a small sample of gram vectors belonging to the given node of points. A median split is applied with respect to the centroid to all nodes starting from the root node that consists of all points.

Neighbor based pruning

Once we have a tree with hierarchical partitions of the matrix, we perform neighbor-based pruning by computing three lists namely neighbor list $\mathcal{N}(\alpha)$, near interaction list $Near(\alpha)$ and far interaction list $Far(\alpha)$. As shown in Algorithm 2, $\mathcal{N}(\alpha)$ is obtained by exhaustively searching and then merging all neighbors $j \in \alpha$ for each $i \in \alpha$ such that d_{ij} is small. The tree splitting and pruning is repeated until either 10 iterations have finished or until 80% accuracy is attained.

Algorithm 2 $\mathcal{N}(\alpha)$

```
1: for all $i \in \alpha$ do
2:     for all $j \in \alpha$ do
3:         if $d_{ij}$ is small then
4:             $\mathcal{N}(\alpha) = \mathcal{N}(\alpha) \cup j$
5:         end if
6:     end for
7: end for
```

Similarly, $Near(\alpha)$ is obtained in Algorithm 3 by adding *MortonIDs* of the all the indices present in $\mathcal{N}(\alpha)$. Morton Id of a node is a bit array that represents the path from root to the node. The set of near points $Near(\alpha)$ represents the dense blocks that cannot be approximated. Therefore, to stop the list from growing and therefore increasing the cost of computation, the parameter *computation budget* is introduced such

Algorithm 3 $Near(\alpha)$

```
1: for all $i \in \mathcal{N}(\alpha)$ do
2:     $Near(\alpha) = Near(\alpha) \cup MortonId(i)$
3: end for
```

Algorithm 4 $Far(\alpha)$

```
1: for all $\beta \in$ leaf nodes do
2:     if $\alpha \cap Near(\beta) \neq \phi$ then        ▷ there are near interactions between $\alpha$ and $\beta$
3:         $Far(left\_child_\alpha)$
4:         $Far(right\_child_\alpha)$
5:     else
6:         $Far(\alpha) = Far(\alpha) \cup \beta$
7:     end if
8: end for
9: $Far(\alpha) = Far(left\_child_\alpha) \cap Far(right\_child_\alpha)$
10: $Far(left\_child_\alpha) = Far(left\_child_\alpha) \setminus Far(\alpha)$
11: $Far(right\_child_\alpha) = Far(right\_child_\alpha) \setminus Far(\alpha)$
```

that the $|Near(\alpha)| < \frac{N}{m}$.
Finally for $Far(\alpha)$, each leaf node β is traversed to check if $\alpha \cap Near(\beta) = \phi$. As shown in Algorithm 4, if the condition is true, it means that α and β have no near interactions and β can be added to the list of $Far(\alpha)$. If it is false, then the process is recursively continued with the left and right children. The common nodes in left and right child are extracted and added to the parent node $Far(\alpha)$ to expand the size of off-diagonal blocks that can be later approximated.

Skeletonization

The off-diagonal blocks are approximated using interpolative decomposition. A *skeleton* of the off-diagonal block is obtained by taking a subset of columns from the block. The decomposition is similar to Equation (2.19) and is given as

$$K_{I\beta} \quad = \quad K_{I\widetilde{\beta}} \quad \cdot \quad P, \tag{2.31}$$

where β is a leaf node and I is a set complement of β. P is the matrix with interpolation coefficients and $K_{I\widetilde{\beta}}$ is the skeleton with $\widetilde{\beta}$ being a subset of columns of the leaf node. For all non-leaf nodes, skeletons of the left and right children are computed recursively and combined to obtain the decomposition of the parent block.

2.4.2. Evaluation

Matrix-Vector product of kernel matrix K and weights w is performed using the *Near* and *Far* lists to compute the potential u. For all the *Far* nodes, the skeleton basis $K_{\widetilde{\beta\alpha}}$ is

Algorithm 5 *Evaluate*(u) [1]

```
1: $N2S(\alpha)$        ▷ Computes skeleton weights $\widetilde{w}$
2: $S2S(\alpha)$        ▷ Apply skeleton basis $K_{\widetilde{\beta\alpha}}$
3: $S2N(\alpha)$        ▷ Accumulate skeleton potentials $\widetilde{u}$
4: $L2L(\alpha)$        ▷ Accumulate direct matvec for $Near(\beta)$
```

used to approximate the product. In the first step, the skeleton weights $\widetilde{w}$ of each leaf node are computed using a post-order traversal. This is done by computing a product of the corresponding interpolation coefficients with the skeleton weights of the leaf node. The internal nodes are recursively computed using the skeletons of left and right children nodes. In the second step, the skeleton basis $K_{\widetilde{\beta\alpha}}$ is applied to the skeleton weights $\widetilde{w}$. Finally, the skeleton potentials $\widetilde{u}$ are accumulated by preorder traversal of the binary tree. The potentials of $Near(\beta)$ are not approximated and thus direct ***matvec*** is performed on these blocks.

3. Efficient and Scalable Kernel Matrix Approximation using Hierarchical Decomposition

3.1. Implementation

3.1.1. Overview

In this section, we discuss an instance of a LinearOperator with a matvec implementation using GOFMM methods. GOFMM [1] methods are used on python data structures using Simplified Wrapper Interface Generator (SWIG). SWIG generates an interface for the GOFMM C++ methods and generates code for the target language, in our case Python. We can then use the C++ methods in Python scripts without additional compilation. We exploit this to generate manifold learning data using datafold and performing hierarchical decomposition on kernel matrices using GOFMM methods.
The integrated environment has been provided in a docker container. Since the Linux cluster at LRZ does not provide docker support, we convert the docker image to a Charliecloud image [16]. We export the Charliecloud image to the Linux cluster and gather accuracy measurements for datasets such as uniform sampling data, Swiss roll and s-curve. Furthermore, we also perform a pairwise comparison of eigenvectors, fit out-of-sample data and compare the results with the scipy eigen solvers. We then perform strong and weak scaling with multiple cores and also multiple nodes. We collect the run-time information to inspect the scalability of the integrated environment on the Linux cluster.

3.1.2. Charliecloud

We have a docker image containing GOFMM that has been augmented in order to be used with Python data structures. The Linux cluster at LRZ does not provide docker support since it requires admin rights. Hence Charliecloud container technology [16] is used as an alternative.
Containers are lightweight and simple to use. We begin by building a docker image on a local machine using a dockerfile with all the required softwares and dependencies.

The dockerfile contains commands to install all the run-time dependencies followed by installation of GOFMM and datafold. It is also required to set the correct environment variables. Once the environment variables are set, GOFMM is compiled. Then the SWIG interface file is compiled to generate Python versions of GOFMM's C++ methods. The instructions to build the image and the dockerfile can be found in the appendix.

Charliecloud is first installed on the local machine from the github repository and then any docker image can be converted to a Charliecloud image using the command `ch-builder2tar <docker-image> /dir/to/save`. The Charliecloud image is then exported to the Linux cluster and unpacked with the command
`ch-tar2dir <charliecloud-image> /dir/to/unpack`.
Now we can run our code inside the Charliecloud container with the command `ch-run -set-env=./gofmm/ch/environment -w ./gofmm - bash`, where `gofmm` is the image. `-w` mounts the image with read-write permissions while the default is read-only. `-set-env` sets the environment for the container since by default it inherits the environment of the host system. The environment variables are very important since without them there will be compilation, linker and/or execution errors.

3.1.3. LinearOperator

SciPy [17] is an open-source free software with modules for common tasks of Science and Engineering such as linear algebra, solvers, interpolation etc. Our focus in this thesis will be on scipy.sparse package which provides methods for sparse matrices. It contains seven matrix and array classes for different types of representations such as sparse row matrix, column matrix, coordinate format etc. It also accommodates methods to build various kinds of sparse matrices and two submodules csgraph and linalg. The submodule linalg provides an abstract interface named LinearOperator that uses iterative solvers to perform matrix-vector products. This interface consists of methods such as matmat(x), matvec(x), transpose(x) for matrix-matrix multiplication, matrix-vector multiplication and transpose of matrix respectively. A concrete class or subclass of LinearOperator can be built by implementing either one of _matvec or _matmat methods and the properties shape and dtype. Depending on the type of matrices at hand, corresponding matvec methods may also be implemented.
scipy.sparse.linalg also provides methods for computing matrix inverses, norms, decompositions and linear system solvers. The functionality we are interested in is the matrix decomposition. In Table 3.1, we can take a look at various decomposition methods that are present in the module. The method we use to decompose data obtained from datafold is scipy.sparse.linalg.eigsh [17]. This method requires either an ndarray, a sparse matrix or LinearOperator as parameters. It optionally takes k, which is the number of

desired eigen values and eigen vectors. It solves $Ax[i] = \lambda_i x[i]$ and returns two arrays - $\lambda[]$ for eigen values and k vectors $X[:i]$, where i is the column index corresponding to the eigen value. Other optional parameters can be referred at the SciPy documentation website.

scipy.sparse.linalg.eigsh is a wrapper ARPACK functions SSEUPD and DSEUPD which use implicitly restarted Lanczos method to solve the system for eigen values and vectors [18]. Lanczos process belongs to a class called Krylov subspace projection methods. Arnoldi method generalizes Lanczos to deal with non-symmetric matrices [19].

Table 3.1.: Matrix Factorizations in scipy.sparse.linalg.

scipy.sparse.linalg.**eigs**	Finds eigen values and vectors of square matrix
scipy.sparse.linalg.**eigsh**	Finds eigen values and vectors of real symmetric or complex hermitian matrix
scipy.sparse.linalg.**lobpcg**	Locally Optimal Block Preconditioned Conjugate Gradient Method
scipy.sparse.linalg.**svds**	Partial Singular Value Decompositions
scipy.sparse.linalg.**splu**	LU decomposition of sparse square matrix
scipy.sparse.linalg.**spilu**	Incomplete LU decomposition of sparse square matrix
scipy.sparse.linalg.**SuperLU**	LU factorization of a sparse matrix

Implicitly restarted Arnoldi method

Implicitly restarted Arnoldi method is a variation of Arnoldi process. Arnoldi process builds on the *power iteration* method which computes $Ax, Ax^2, Ax^3...$ for an arbitrary vector x, until it converges to the eigen vector of the largest eigen value of matrix A. To overcome the drawbacks of so many unnecessary computations for a single eigen value and its corresponding eigen vector, Arnoldi method aims to save the successive vectors as they contain considerable information that can be further exploited to find new eigen vectors. The saved vectors form a *Krylov* matrix and is given as [20]

$$\mathcal{K}_n = \mathsf{Span}[x, Ax, A^2x...A^{n-1}x]. \tag{3.1}$$

Orthonormal vectors $x_1, x_2, x_3...$ that span a *Krylov* subspace are extracted using *Gram-Schmidt* orthogonalization from each column of *Krylov* matrix. The k-step Arnoldi iteration is given in Algorithm 6 [19]. H is the orthogonal projection of A in the *Krylov*

Algorithm 6 k-step ArnoldiFactorization(A,x)

```
1: $x_1 \leftarrow \frac{x}{||x||}$                                    ▷ Computes first krylov vector $x_1$
2: $w \leftarrow Ax_1$                                                   ▷ Computes new candidate vector
3: $\alpha_1 \leftarrow x_1^H w$
4: $r_1 \leftarrow w - \alpha_1 x_1$
5: $X_1 \leftarrow [x_1]$                                                ▷ Orthonormal basis of krylov subspace
6: $H_1 \leftarrow [\alpha_1]$                                           ▷ Upper Hessenberg matrix
7: for all $j = 1...k-1$ do                      ▷ For k steps, compute orthonormal basis $X$
8:                                      ▷ and the projection of matrix $A$ on the new basis
9:     $\beta_j \leftarrow ||r_j||$ ; $x_{j+1} \leftarrow \frac{r_j}{\beta_j}$
10:    $X_{j+1} \leftarrow [X_j, x_{j+1}]$ ; $\hat{H}_j \leftarrow \left[H_j,\ \beta_j e_j^T\right]^T$
11:    $z \leftarrow Ax_j$
12:    $h \leftarrow X_{j+1}^H z$;  $r_{j+1} \leftarrow z - X_{j+1} h$          ▷ Gram-Schmidt Orthogonalization
13:    $H_{j+1} \leftarrow [\hat{H}_j, h]$
14: end for
```

subspace. It is observed that eigen values of the upper Hessenberg matrix H, namely *Ritz* values converge to eigen values of A. When $r_j = 0$, the Ritz pairs become eigen pairs of A.

One of the drawbacks of Arnoldi process is that the number of iterations taken for convergence is not known prior to the computation of well-approximated Ritz values [19]. This causes the computation of Hessenberg matrix to cost $\mathcal{O}(k^3)$ at the k-th step. To mitigate this problem, the computation of Ritz values are halted when desired accuracy is achieved. Then a polynomial $\psi(A)$ is constructed from the obtained Ritz values and applied to unwanted components [21] as shown in the equations below

$$x_1 \leftarrow \psi(A)x_1 \quad \textit{and} \tag{3.2}$$

$$x_1 \leftarrow \frac{x_1}{||x_1||}. \tag{3.3}$$

This approach is called *explicitly restarted Arnoldi method*. A more efficient approach named *implicitly restarted Arnoldi method* uses implicitly shifted *QR-iteration*. It avoids storage and numerical difficulties associated with the standard approach by compressing the necessary information into a k-dimensional *Krylov* subspace.

Arnoldi factorization of length $m = k + p$ looks like

$$AX_m = X_m H_m + r_m e_m^T. \tag{3.4}$$

The implicit restarting method aims to compress this to length k by using QR steps to apply p shifts resulting in [19]

$$AX_m^+ = X_m^+ H_m^+ + r_m e_m^T Q \tag{3.5}$$

where $V_m^+ = V_m Q$, $H_m^+ = Q^T H_m Q$ and $Q = Q_1 Q_2 ... Q_p$. Q_j is the orthogonal matrix associated with the corresponding shift μ_j. The first $k - 1$ values of $e_m^T Q$ are zero and thus the factorization becomes

$$AX_k^+ = X_k^+ H_k^+ + r_k^+ e_k^T. \tag{3.6}$$

The residual r_m^+ can be used to apply p steps to obtain back the m-step form. A polynomial of degree p, $\prod_1^p (\lambda - \mu_j)$ is obtained from these shifts. The roots of this polynomial are used in the QR process to extract unwanted information from the starting vector.

Implicitly restarted Lanczos method

Consider the Equation 3.4 for Arnoldi factorization. X_m are orthonormal columns and H_m is the upper Hessenberg matrix. If A is a Hermitian matrix, it becomes Lanczos factorization. So Arnoldi is basically a generalization to non-hermitian matrices. For Lanczos method, H_m is a real, symmetric and tridiagonal matrix and the X_m are called Lanczos vectors. The algorithms hence remain the same as the ones described for Arnoldi. The methods scipy.sparse.linalg.eigs uses Arnoldi iteration since it deals with real and symmetric matrices while scipy.sparse.linalg.eigsh invokes implementation of Lanczos methods.

Python script

Everything is put together in a python script as shown in Figure 3.1. Point cloud data is generated using PCManifold class from the pcfold sub-package. DiffusionMaps instance is created with the point cloud data in order to perform the diffusion maps algorithms. A class `FullMatrix` is derived from `LinearOperator` interface which belongs to `scipy.sparse.linalg` package. In short, the kernel matrix from `DiffusionMaps` instance is used by `FullMatrix` to perform eigendecompositions with `GOFMM` algorithms.

`FullMatrix` will provide an implementation of `matvec` method using `GOFMM` methods as shown in Listing 3.1. `FullMatrix` has member variables such as `problem_size`,

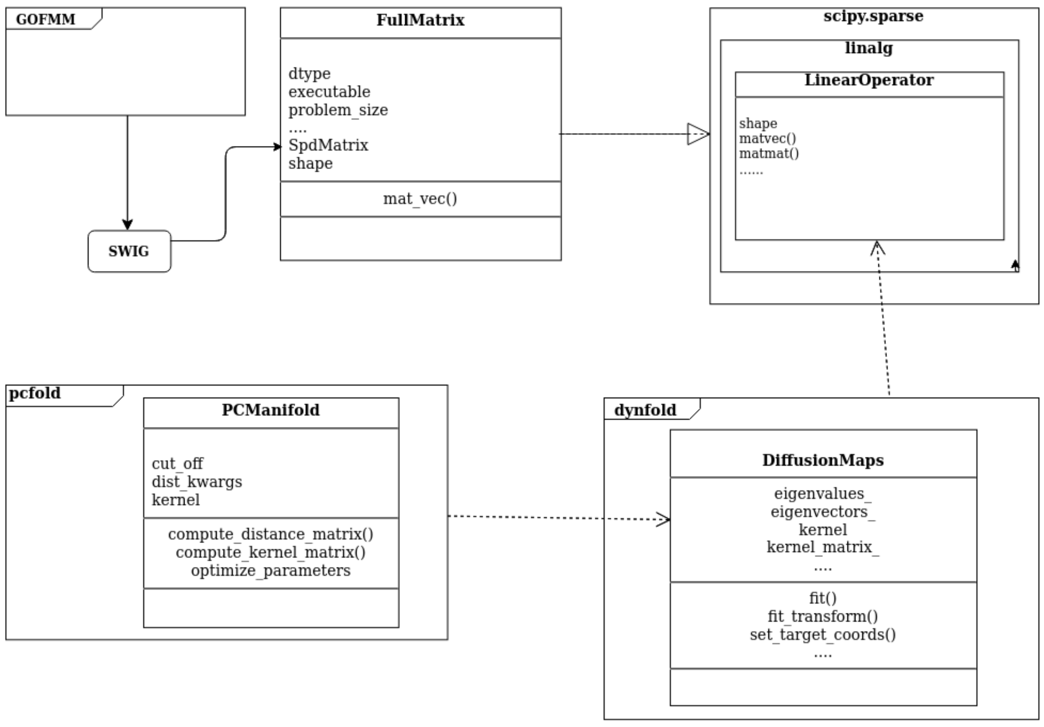


Figure 3.1.: UML diagram of implementation of the python script

`executable` etc. which are necessary to build a `gofmmTree`. The constructor initializes all the member variables and also loads the numpy matrix into the SWIG interface methods. The method `LoadDenseSpdMatrixFromConsole` converts the numpy matrix to an object of type `SPDMATRIX_DENSE`. In `matvec`, an instance of `gofmmTree` is created with the correct parameters. With `self.wData` as a vector consisting of ones, `MultiplyDenseSpdMatrix` is called to perform the compression and evaluation phases of the hierarchical algorithm.

```python
import numpy as np
import scipy.sparse.linalg
from sympy import product
import tools
from scipy.linalg import eig, eigh
from scipy.sparse.linalg import LinearOperator

class FullMatrix( LinearOperator ):
```

```
def __init__( self,
executable,
problem_size,
max_leaf_node_size,
num_of_neighbors,
max_off_diagonal_ranks,
num_rhs,
user_tolerance,
computation_budget,
distance_type,
matrix_type,
kernel_type,
spd_matrix,
weight,
dtype="float32" ):
self.executable = executable
self.problem_size = problem_size
self.max_leaf_node_size = max_leaf_node_size
self.num_of_neighbors = num_of_neighbors
self.max_off_diagonal_ranks = max_off_diagonal_ranks
self.num_rhs = num_rhs
self.user_tolerance = user_tolerance
self.computation_budget = computation_budget
self.distance_type = distance_type
self.matrix_type = matrix_type
self.kernel_type = kernel_type
self.spd_matrix = np.float32( spd_matrix )
self.denseSpd = tools.LoadDenseSpdMatrixFromConsole( self.spd_matrix )
self.weight = np.float32( weight )
self.wData = tools.LoadNumpyMatrixFromConsole( self.weight )
self.lenMul = self.problem_size * self.num_rhs
self.shape = self.spd_matrix.shape
self.dtype = np.dtype( dtype )

def _matvec( self, x ):
gofmmCalculator = tools.GofmmTree( self.executable,
self.problem_size,
self.max_leaf_node_size,
self.num_of_neighbors,
```

```python
self.max_off_diagonal_ranks,
self.num_rhs,
self.user_tolerance,
self.computation_budget,
self.distance_type,
self.matrix_type,
self.kernel_type,
self.denseSpd )

a = x.reshape( self.problem_size,1 )
c = gofmmCalculator.MultiplyDenseSpdMatrix( self.wData, self.lenMul )
spdMatrix_mul = np.resize( c, ( self.problem_size, self.num_rhs ) )
return spdMatrix_mul
```

Listing 3.1: class FullMatrix

Datasets

Datasets are created as shown in the tutorial section of datafold repository [5]. The first example as shown in Listing 3.2 contains samples drawn from a uniform distribution.

```python
import datafold.pcfold as pfold
import numpy as np

random_state = 42
problem_size = 16384

rng = np.random.default_rng( random_state )
data = rng.uniform( low = ( -2, -1 ), high = ( 2, 1 ), size = (
    problem_size, 2 ) )

pcm = pfold.PCManifold( data )
pcm.optimize_parameters()
```

Listing 3.2: Uniform sampling of point cloud data

An instance `rng` of `Generator` is created which is then used to draw uniformly distributed samples from the given interval. The above example is a rectangle with 16384 samples and values ranging from [-2,-1] to (2, 1) as shown in Figure 3.2. Then point cloud representation is created by instantiating `PCManifold` with the data. We further calculate parameters such as `cut_off` and `epsilon` which are the threshold for pairwise distances between points and the kernel bandwidth respectively.

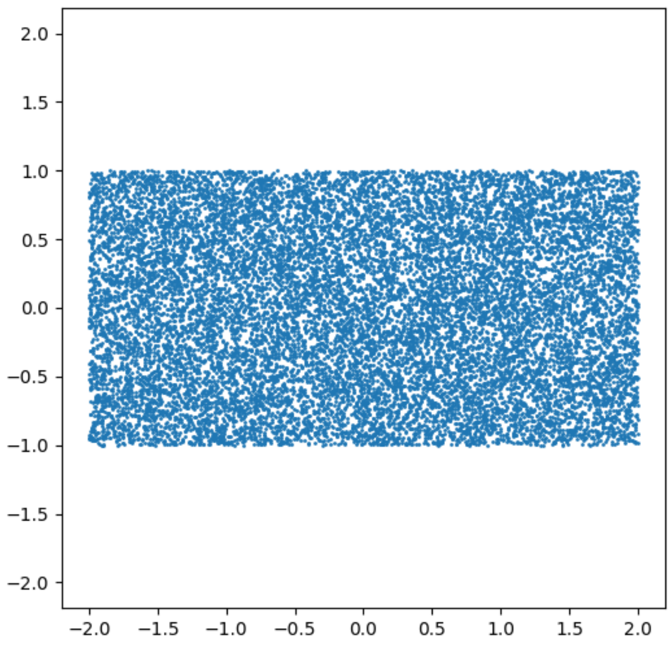


Figure 3.2.: Uniform sampling of size 16384 with values between [-2,-1] and (-2,-1)

Embedding

Once we have the point cloud data, we want to find an emebedding using diffusion maps algorithm. This is done as shown in Listing 3.3 for the uniform sampling data. An instance of `DiffusionMaps` class is created from `dynfold` with a `GaussianKernel`. The embedding is obtained by the method `fit` and the eigenvalues and vectors can be queried from the instance like `dmap.eigenvectors_` and `dmap.eigenvalues_`.

```
dmap = dfold.DiffusionMaps( kernel=pfold.GaussianKernel(
epsilon=pcm.kernel.epsilon),
n_eigenpairs=5,
dist_kwargs=dict(cut_off=pcm.cut_off), )
```

```python
dmap.fit(pcm, store_kernel_matrix=True)
```

Listing 3.3: DiffusionMaps instantiation

The `dmap.kernel_matrix_` is a square matrix of size $N \times N$ where N is the number of samples. Due to its large size, it becomes computationally expensive to compute eigen pairs. Hence we use the stored kernel matrix to perform hierarchical decompositions as in Listing 3.4.

```python
K = dmap.kernel_matrix_
K_sparse = K.copy()
K = K.todense()

kernel_matrix_OP = FullMatrix( executable,
problem_size,
max_leaf_node_size,
num_of_neighbors,
max_off_diagonal_ranks,
num_rhs, user_tolerance,
computation_budget,
distance_type,
matrix_type,
kernel_type,
K, w,
dtype=np.float32 )

n_eigenpairs = 5
solver_kwargs = {
  "k": n_eigenpairs,
  "which": "LM",
  "v0": np.ones(problem_size),
  "tol": 1e-14,
}

eigenvalues, eigenvectors = scipy.sparse.linalg.eigsh(K_sparse,
**solver_kwargs)
eigenvalues_gofmm, eigenvectors_gofmm = scipy.sparse.linalg.eigsh(K,
**solver_kwargs)
```

Listing 3.4: Eigendecomposition with scipy solver matvec and GOFMM matvec evaluations

The stored kernel matrix is converted to a sparse representation and used to perform eigendecompositions using the scipy solver which uses implicitly restarted Arnoldi iteration if the matrix is symmetric and implicitly restarted Lanczos method otherwise. The `FullMatrix` instance is used as a parameter to the scipy solver to make it use the `GOFMM` algorithm. The eigen values and vectors obtained from both the algorithms can then be compared by computing a norm and plotting the eigen vectors. Along with the accuracy measurements, we also explore the scalability of these integrated softwares on CoolMUC-2, CoolMUC-3 by strong and weak scaling experiments on both multiple core and multiple nodes. The run-time measurements are done using `Intel VTune Amplifier` [22]. This tool is used to profile code, obtain system performance, bottlenecks and various other purposes.

3.2. Experiments and Results

The integration of operations from GOFMM and point cloud data from datafold is run on the CoolMUC-2 cluster with 28-way Intel Xeon E5-2690 v3 ("Haswell") based nodes and FDR14 Infiniband interconnect. CoolMUC-2 has 812 nodes with 64GB memory per node. On the other hand, multiple node scaling is conducted on CoolMUC-3 cluster with 64-way Knights Landing 7210-F many-core processors and Intel Omnipath OPA1 interconnect. It has 148 nodes with 64 cores per node and 4 hyper threads per core. Therefore, the accuracy measurements and multi-core scaling experiments with OMP number of threads less than 28 are conducted on the compute node lxlogin1 of CoolMUC-2 and the rest on lxlogin8 of CoolMUC-3.

3.2.1. Accuracy Measurements

The resultant eigenvalues of both scipy solver and the one with GOFMM compression and evaluation are recorded. Then Froebius error norm [23] is calculated as shown in Equation 3.7.

$$||Error_F|| \quad = \quad \left[\sum_{i=1}^{5} abs(x_i)^2 \right]^{\frac{1}{2}}, \tag{3.7}$$

where x_i are the five largest eigen values. The error is then divided by the number of eigen values.

As the first dataset, we consider two-dimensional uniform sampling of different sizes ranging from 2048 to 16384 asshown in Figure 3.2. We work with the same data set for all our experiments. Firstly, the accuracy is measured for varying problem sizes and max_leaf_node_size while the other parameters are fixed as following:

```
executable = "./test_gofmm"
num_of_neighbors = 0
max_leaf_node_size = 0.5*problem_size
max_off_diagonal_ranks = max_leaf_node_size
num_rhs = 1
computation_budget = 0.00
user_tolerance = E-5
distance_type = "kernel"
matrix_type = "dense"
kernel_type = "gaussian"
```

Listing 3.5: Parameter values for *max_leaf_node_size* vs. *user_tolerance*

For simplicity, let us denote problem size as N and maximum leaf node size as m. Therefore we have a matrix of size $N \times N$ and every leaf node in the tree has a maximum leaf node size m. The error norms are computed by sorting eigen values of both algorithms in descending order as shown in Listing 3.6.

```
exact_eigen_values, exact_eigen_vectors = scipy.sparse.linalg.eigsh(
K_sparse, **solver_kwargs)
sorted_exact_eigen_indices = np.argsort(-exact_eigen_values)
sorted_exact_eigen_values = exact_eigen_values[sorted_exact_eigen_indices
    ]
sorted_exact_eigen_vectors = exact_eigen_vectors[:,
sorted_exact_eigen_indices]

approx_eigen_values, approx_eigen_vectors = scipy.sparse.linalg.eigsh(
kernel_matrix_OP,
**solver_kwargs)
sorted_approx_eigen_indices = np.argsort(-approx_eigen_values)
sorted_approx_eigen_values = approx_eigen_values[
sorted_approx_eigen_indices]
sorted_approx_eigen_vectors = approx_eigen_vectors[:,
sorted_approx_eigen_indices]

n_eigenpairs=5
error = np.linalg.norm(sorted_exact_eigen_values -
sorted_approx_eigen_values)/n_eigenpairs
```

Listing 3.6: Sort eigenvalues and their corresponding eigen vectors

It can be observed in Table 3.2 that the errors converge to the order of 10^{-3} and 10^{-4} for m values of 0.5N and 0.25N and seem to decrease for decreasing values of m, which is the expected behavior.

The parameters maximum leaf node size m and maximum off-diagonal rank s are the most important in terms of accuracy [24]. m determines the size of sparse blocks and s determines the maximum approximation rank of the off-diagonal blocks. Therefore, theoretically increasing values of m and s must produce more accurate results. Next we collected accuracies for varying user tolerance *stol* and m values for the problem size 8192 in Table 3.3. *stols* do not seem to have a huge impact on the accuracy. The eigen values for m = 0.0625N and *stol* = E-7 are *NaN* and this could be due to reasons such as overflows.

N \ m	**0.5N**	**0.25N**	**0.125N**	**0.0625N**
16384	8.84E-4	9.87E-4	1.48E-2	1.34E-1
8192	1.15E-3	1.08E-3	1.73E-2	1.50E-2
4096	1.15E-3	8.84E-4	1.28E-2	7.46E-2
2048	1.08E-3	1.00E-3	2.36E-2	1.58E-1
1024	1.41E-3	1.17E-3	1.41E-2	1.70E-1

Table 3.2.: *problem_size* vs. *maximum_leaf_node_size*

stol \ m	**0.5N**	**0.25N**	**0.0625N**
E-3	1.38E-3	2.43E-3	1.08E-1
E-5	1.15E-3	1.08E-3	1.30E-1
E-7	6.91E-3	8.75E-4	nan

Table 3.3.: *user_tolerance* vs. *maximum_leaf_node_size* for problem size **N=8192**

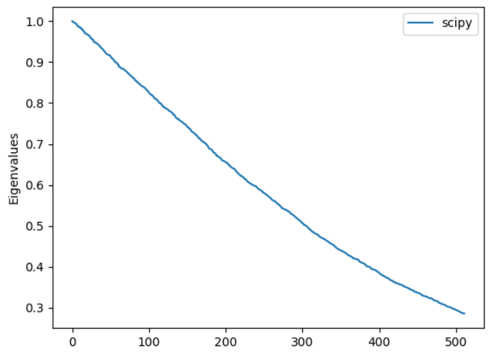
For the above experiments we've maintained $m = s$ and to find out which of the two parameters have a bigger influence on the accuracy, we vary the values of s against m. For problem size 4096, as shown in Table 3.4, the accuracy seems to vary with varying values of s. This behavior in accuracy is consistent with the experiments conducted in [25]. It makes sense that the higher values of s result in higher accuracies because s is the maximum rank of the approximation applied to the off-diagonal blocks. More information is lost when low-rank approximations are performed, thus causing

m \ s	**0.5N**	**0.25N**	**0.125N**
0.5N	1.08E-3	9.73E-4	2.23E-2
0.25N	1.17E-3	1.05E-3	1.54E-2
0.125N	3.83E-3	5.55E-3	1.52E-2

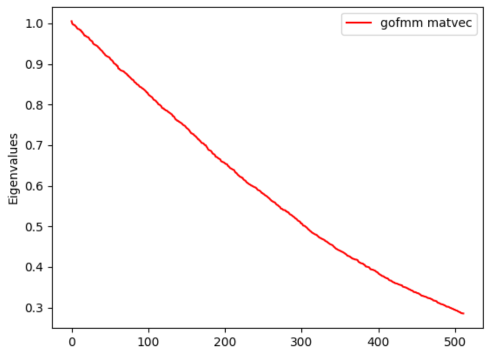
Table 3.4.: *maximum_leaf_node_size* vs. *max_off_diagonal_ranks* for problem size **N=4096**

less accurate results.

Finally, since only the 5 largest values were considered for the error norms, a better illustration required the calculation of more eigen values. Figure 3.3 shows 512 eigen values computed in a problem size of 16384 for both matvec operations. The difference in larger eigen values can be seen at the top of the plot whereas the smaller eigen values almost converge.



(a) scipy solver



(b) GOFMM

Figure 3.3.: 512 Eigen values for N=16384

3.2.2. Multi-core performance

GOFMM employs OpenMP parallelism and hence can be used to speed up our matrix-vector products. We first take a fixed set of parameters and perform two types of scaling with multiple cores on a single node. Before the aforementioned runs, C++ code for the SWIG interface file was restructured. The methods LoadDenseSpdMatrixFromConsole, LoadNumpyMatrixFromConsole load numpy matrices into the C++ data structures SPDMATRIX_DENSE, DATA_s respectively and return them by value. SWIG allocates a new object for these data structures and it is up to the user to deallocate them when it is no longer needed. But since the scope of these variables is inside the function, deallocation is also not possible and thus causes memory leaks. To mitigate this, we declare the data structure variables globally and return a reference to them. For scaling, we collect run time in seconds and also calculate speed-up or efficiency as the ratio of time taken by a single thread versus time taken by multiple threads.

Strong scaling

For strong scaling, we consider the problem sizes **8192** and **16384**. Scaling up to 28 threads was done on CoolMUC − 2 and runs with 56 threads were performed on CoolMUC − 3. CoolMUC − 2 has a peak performance of 1400 TFlop/s and CoolMUC − 3 has 459 TFlop/s. The speed-up bar plot for **8192** can be seen in Figure 3.4 for various OMP_NUM_THREADS starting from 1 to 56. The speed-up increases almost ideally for up to 4 threads. Speed-up for threads 8, 16 and 28 stays constant while for 56 threads, there is a speed-down of 0.42. We suspect this could be due to reasons such as communication and synchronization overhead. Run times for problem size 16384 can be seen in Table 3.5. Time taken by 1 thread exceeds 48 hours of run time while slowly scaling with increasing ranks. But we notice an increase in run time for a higher number of threads, similar to the scaling of problem size 8192.

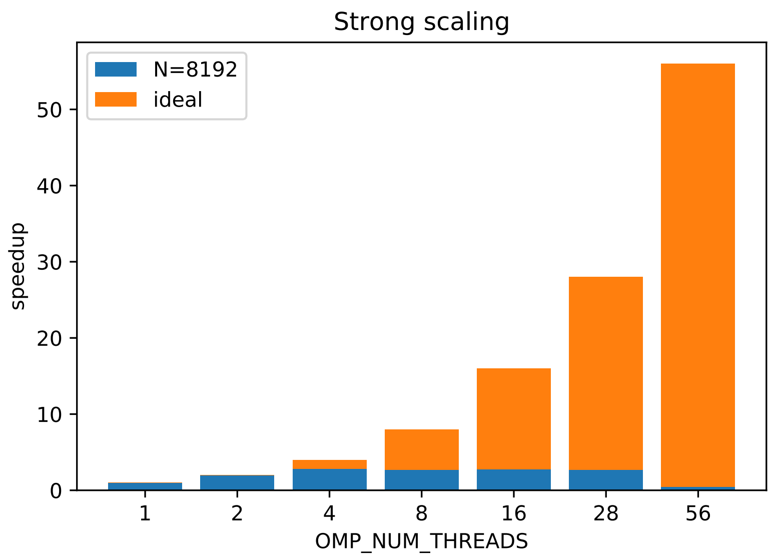


Figure 3.4.: Strong scaling for problem size 8192.

N \ n_t	1	2	4	8	16	28	56
8192	11935	6006	4173	4369	4353	4488	27952
16384	>172k	72924	52454	48774	38460	43106	>172k

Table 3.5.: Run time in seconds for problem sizes **8192** and **16384** with various OMP_NUM_THREADS

Weak scaling

Weak scaling was done with the same sequence of OMP_NUM_THREADS as that of strong scaling. We deal with square matrices and keeping problem size per thread constant would mean that the number of threads increases quadratically. Since we only have a limited number of cores, we have increased the threads linearly as shown in Table 3.6. Load per thread can be given as

$$\frac{problem\_size}{thread} = \frac{N \times N}{1}. \tag{3.8}$$

The load per thread doubles when problem size of the square matrix is doubled while increasing the number of threads linearly as shown in Equation 3.9.

$$\frac{problem\_size}{thread} = \frac{2N \times 2N}{2} = 2 \times \frac{N \times N}{1}. \tag{3.9}$$

We then expect the ideal efficiency to ideally reduce by a fraction of number of omp threads. We observe in Table 3.6 that the efficiency reduces by more than the aforementioned factor. This behavior can also be explained by the reasons mentioned in Section 3.2.2 as we increase the load per thread similar to strong scaling.

N	OMP NUM THREADS	Run time in sec	Efficiency
1024	1	125	1
2048	2	432	0.28
4096	4	605	0.20
8192	8	4369	0.028
16384	16	38460	0.003

Table 3.6.: Weak scaling with multiple cores

Weak scaling with OMP_NUM_THREADS 1, 4 and 16 can be viewed in the Figure 3.5. The line plot describes the ideal behavior while the blue and orange bars are the efficiency of 512 and 1024 problem sizes per thread respectively. Efficiency does not stay constant and rather decreases rapidly and this could be due to load imbalance, granularity and communication overhead.

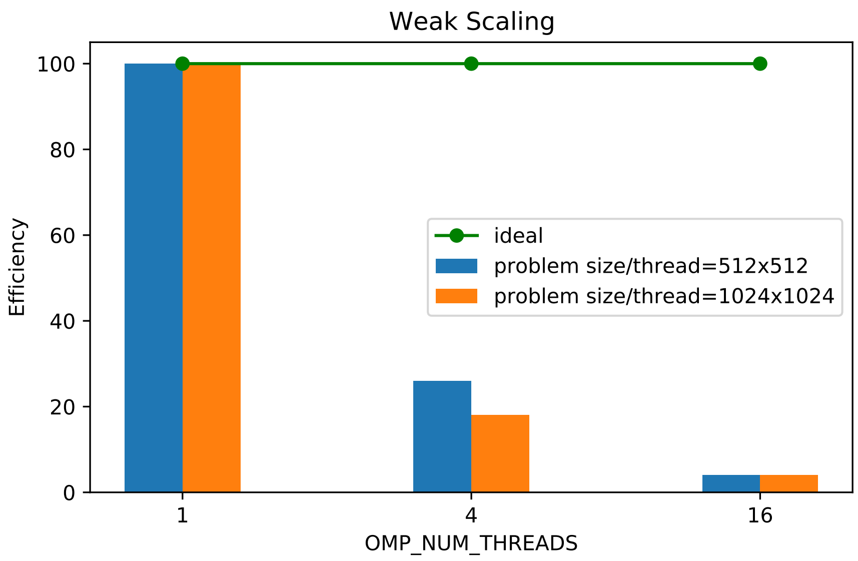


Figure 3.5.: Weak scaling

3.2.3. Multi node performance

GOFMM uses MPI parallelization and hence can be used on larger problem sizes. In order to utilise this feature, we write the kernel matrix computed from the instance of DiffusionMaps as shown in Listing 3.3 into a binary file. We then use the binary file as input to run the mpi executable of gofmm. The decision to not use the python script was made because it is required to write a new SWIG interface for the MPI methods and using mpiexec to run a python script will not work. Therefore it requires distributing the problem in the python script using mpi4py. Given that there is not a lot of support provided for mpi4py [23] with SWIG, it is not known if it could work and also may cause performance drawbacks. Note that using GOFMM without scipy means that we will not be using the iterative method to perform eigendecomposition. Instead, kernel matrices are hierarchically decomposed and evaluated using an arbitrary vector. This means that instead of recording time for several iterations, time-taken to perform one decomposition and one matrix-vector evaluation is recorded for experiments in this section.

Strong scaling

All the multi-node runs were performed on lxlogin8 node on CoolMUC − 3. Firstly, we recorded the influence of OMP threads with MPI. In Table 3.7, the run times for a matrix of size 16384, with 2 MPI ranks and varying OMP number of threads distributed across 2 nodes are shown. The run time for 2 threads is the highest with the least percentage of CPU utilization. Although the CPU utilization increases with the increasing OMP number of threads, there is not a lot of difference in the run times. The top hotspot for 2 OMP threads was the BLAS method for matrix multiplication. For more than 2 OMP threads, locking and synchronization methods consumed the most amount of time.

OMP NUM THREADS	Run time in sec	% CPU utilization
2	57.529	0.8
4	44.199	1.3
8	38.108	2.7
16	37.164	5.5
32	38.085	10.7

Table 3.7.: Run time for problem size 8192 with 2 MPI tasks distributed and different OMP_NUM_THREADS on 2 nodes

Then we fix the OMP number of threads to 8 and vary the number of MPI ranks for the two problem sizes. As shown in Figure 3.6, speed-up for N=8192 increases almost ideally until 4 MPI ranks and behaves similarly to what we observed in Figure 3.4. On the other hand, problem size 16384 scales better than expected for 16 ranks and is due to a better distribution of computational load among the nodes.

MPI tasks / *N*	1	2	4	8	16	32
8192	1	1.48	3.48	3.12	5.26	5.46
16384	1	1.51	5.14	11.499	17.33	21.46

Table 3.8.: Speed up for various number of MPI tasks distributed across 2 nodes

We also performed strong scaling for N=16384 while varying the number of nodes along with the MPI ranks as shown in Figure 3.7. We observed that the problem performs better when distributed across 2 nodes and decreases with the increase in the number of nodes. This is evidently due to load imbalance and reduced CPU utilization.

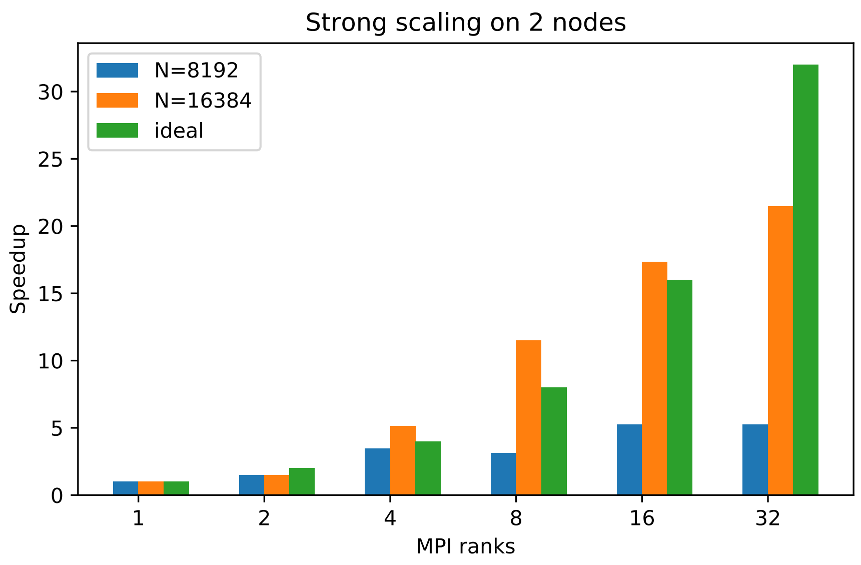


Figure 3.6.: Strong scaling for problem sizes 8192 and 16384 distributed across 2 nodes.

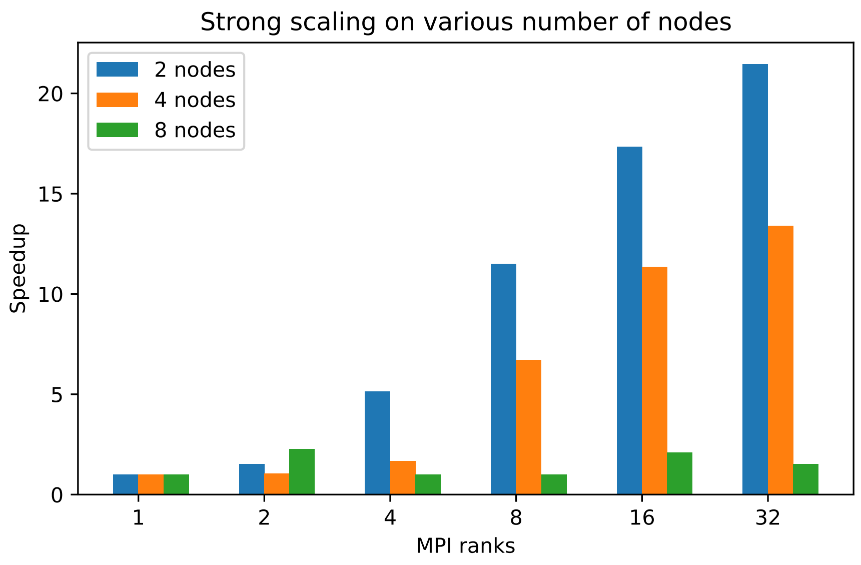


Figure 3.7.: Strong scaling for problem size 16384 on 2,4 and 8 nodes.

Weak scaling

For weak scaling on multiple nodes, we have the same problem as in Section 3.2.2. Since we work with square matrices, we need to increase the number of MPI ranks quadratically to keep the problem size per thread constant. Due to limited resources, we increase them linearly. This means that the problem_size per thread is doubled each time, implying that ideally the run time also doubles. Figure 3.8 shows run times with this type of scaling for 1, 2 and varying nodes. Time taken on 1 node stays constant up to 4 ranks and sharply increases for 8 and 16 ranks. The same trend is observed for scaling on 2 nodes. Even though the load per problem size is increased, the run time is lower than the ideal run time for all three cases. This is similar to the behavior observed in strong scaling. Although the run times were highest in the case of varying nodes, it is still lower than the ideal times.

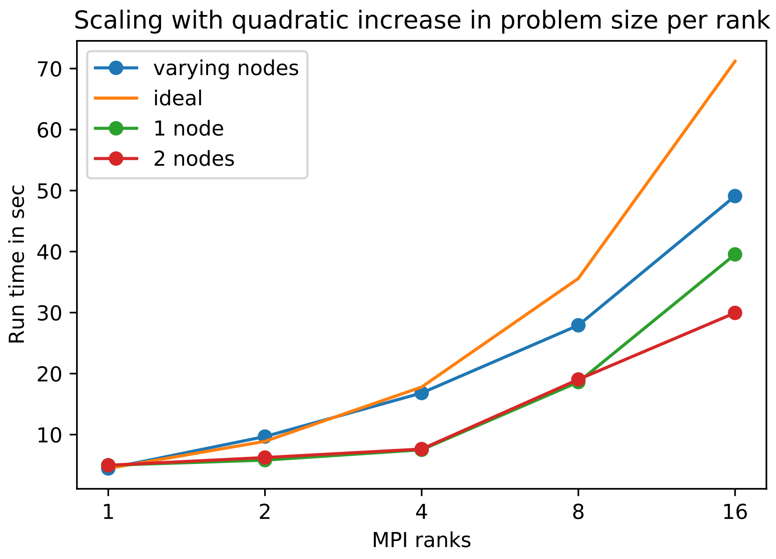
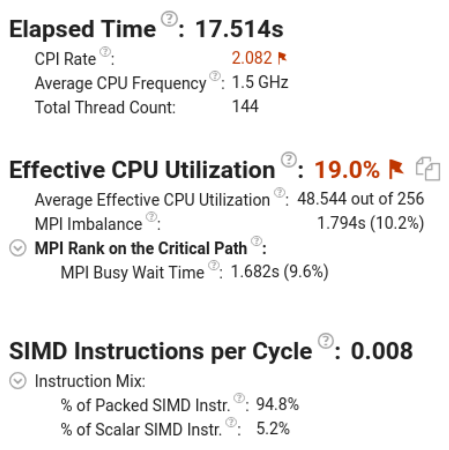


Figure 3.8.: Problem size per rank increases in powers of 2

3.3. Performance analysis

To explain and confirm the observations made in scaling experiments, analysis of performance has been recorded. The following study is done for a problem size N=16384 with 32 ranks and 8 OMP_NUM_THREADS, distributed across 2 nodes. Effective CPU utilization is the highest at almost 20% with 32 ranks as shown in Figure 3.9. Elapsed time is the wall clock time taken for the program to complete executing and is 17.5 sec for 32 ranks and 379.3 sec with 1 rank.

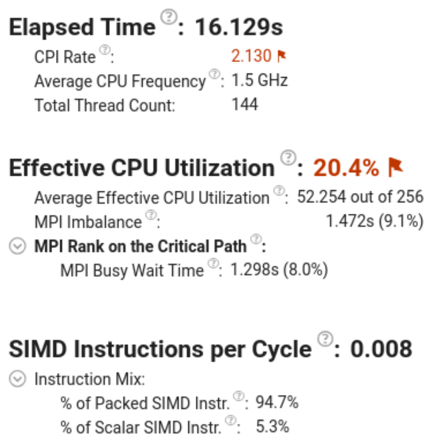


Elapsed Time: 17.514s
CPI Rate: 2.082
Average CPU Frequency: 1.5 GHz
Total Thread Count: 144

Effective CPU Utilization: 19.0%
Average Effective CPU Utilization: 48.544 out of 256
MPI Imbalance: 1.794s (10.2%)
MPI Rank on the Critical Path:
MPI Busy Wait Time: 1.682s (9.6%)

SIMD Instructions per Cycle: 0.008
Instruction Mix:
% of Packed SIMD Instr.: 94.8%
% of Scalar SIMD Instr.: 5.2%

(a) Node mpp3r03c05s01



Elapsed Time: 16.129s
CPI Rate: 2.130
Average CPU Frequency: 1.5 GHz
Total Thread Count: 144

Effective CPU Utilization: 20.4%
Average Effective CPU Utilization: 52.254 out of 256
MPI Imbalance: 1.472s (9.1%)
MPI Rank on the Critical Path:
MPI Busy Wait Time: 1.298s (8.0%)

SIMD Instructions per Cycle: 0.008
Instruction Mix:
% of Packed SIMD Instr.: 94.7%
% of Scalar SIMD Instr.: 5.3%

(b) Node mpp3r03c05s03

Figure 3.9.: Effective CPU utilization and elapsed time

	Node mpp3r03c05s01	Node mpp3r03c05s03
▷**Elapsed Time**	17.514	16.129
▷**CPI rate**	2.082	2.130
▷**Average CPU frequency**	1.5GHz	1.5GHz
▷**Total thread count**	144	144

Table 3.9.: Elapsed time on both the nodes

Low CPU utilization implies that there is low utilization of logical CPU cores which in turn is caused by load imbalance, granularity, synchronization overhead. This could be improved by analyzing the sub-metrics that improve the parallelism. Each node has 256 logical cores and average effective CPU utilization of each can be seen in Table 3.10. The percentage of MPI busy wait time is very high on both nodes causing a lot of CPUs to stay idle for most of the time.

	Node mpp3r03c05s01	**Node mpp3r03c05s03**
▷Effective CPU Utilization	19%	20.4%
▷Average effective CPU utilization out of 256	48.544	52.254
▷MPI Imbalance	1.794	1.472
▷MPI Busy Wait Time	9.6%	8.0%

Table 3.10.: Effective CPU utilization

Figure 3.10 is a histogram that shows a percentage of wall clock time where the CPUs were running simultaneously. Y-axis is the elapsed time and the X-axis is the number of logical CPUs that run simultaneously. The average effective CPU utilization is 12.4 while the target utilization is the total number of logical cores which is 256. The poor indicator below the X-axis implies that the utilization is less than 50% of the maximum. This low thread efficiency can be improved by inspecting the top hot spots in the program execution and finding if they can be reduced or eliminated.

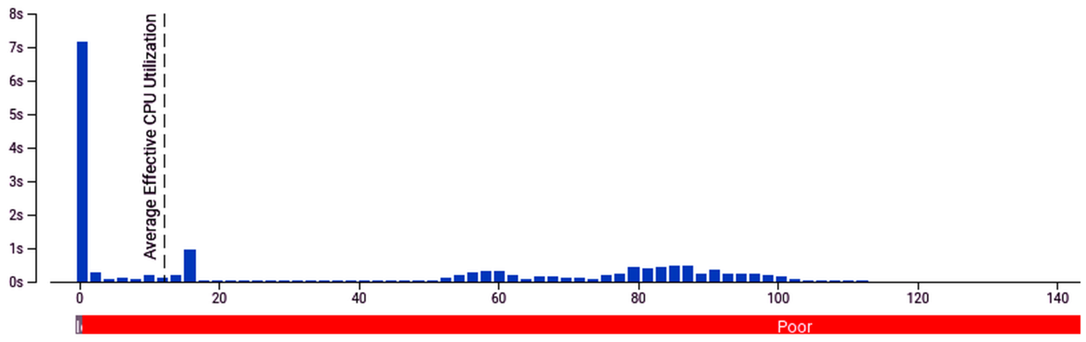


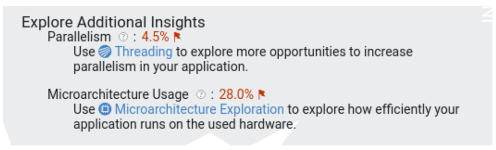
Figure 3.10.: Histogram of average effective utilization of CPUs

Another way to reduce the run time is to take a look at CPU time. Overall CPU time is the total time spent by all the CPUs during the execution. CPU time is split into effective time, spin time and overhead time as shown in Table 3.11. Effective time is the time that the CPUs spend actively executing user code. Spin time as shown in the aforementioned figure is divided into several factors such as lock contention, imbalance and busy-wait time. Overhead time is the time spent for synchronization of threading libraries such as OpenMP and communication between threads. We observe that the overhead is the most time-consuming and that the ratio of effective time versus spin and over time is small. This means that there is a need to improve the parallelism and efficiency of threads.

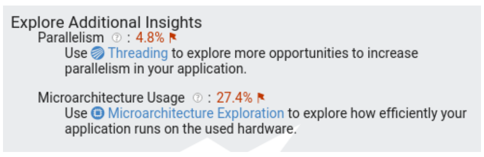
	Node mpp3r03c05s01	**Node mpp3r03c05s03**
▷CPU Time	816.054	809.035
▷Effective Time	203.816	197.487
▷Spin Time	242.079	241.766
▷Imbalance or Serial Spinning	62.069	63.036
▷Lock Contention	114.400	119.093
▷MPI Bust Wait Time	28.701	23.547
▷Other	36.910	36.090
▷Overhead Time	370.159	369.782

Table 3.11.: CPU time in seconds

This is also evident in the percentage of parallelism, as can be seen in Figure 3.11, which is a metric of effective CPU utilization during actual computations. On the other hand, microarchitecture usage indicates how effective our program is on the current microarchitecture and can be impacted by long-latency memory, floating-point or SIMD operations and so on [22].


Explore Additional Insights
Parallelism : 4.5%
Use Threading to explore more opportunities to increase parallelism in your application.
Microarchitecture Usage : 28.0%
Use Microarchitecture Exploration to explore how efficiently your application runs on the used hardware.

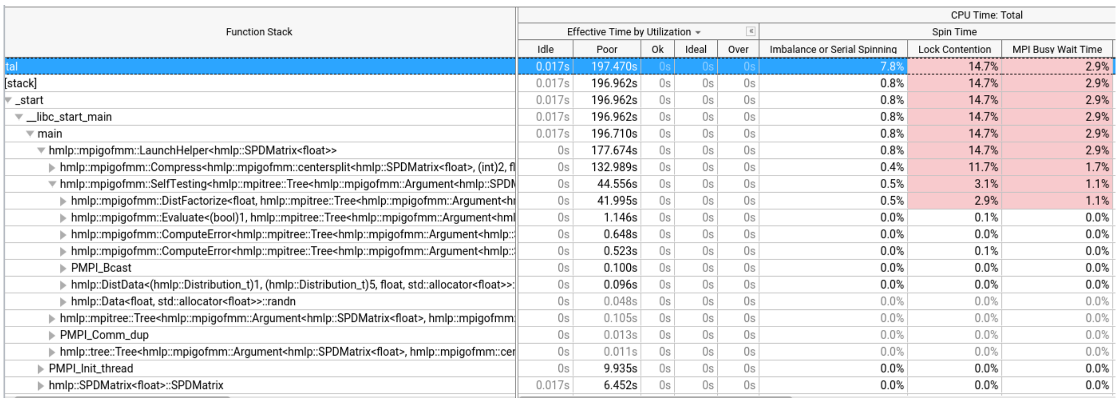
(a) Node mpp3r03c05s01


Explore Additional Insights
Parallelism : 4.8%
Use Threading to explore more opportunities to increase parallelism in your application.
Microarchitecture Usage : 27.4%
Use Microarchitecture Exploration to explore how efficiently your application runs on the used hardware.

(b) Node mpp3r03c05s03

Figure 3.11.: Parallelism

Lastly, we take a look at the most time-consuming methods in Figure 3.12. mpigofmm :: Compress takes 132.9 sec followed by mpigofmm :: SelfTesting with 44.5 sec which contains the multiplication of a compressed matrix with a random vector. Lock contention is also maximum in the compression method which is expected since most of the parallelization happens during this phase. The third highest hot spot is PMPI_Init_thread, which initializes the MPI environment and the fourth highest is hmlp :: SPDMatrix $<$ float $>$:: SPDMatrix, which reads the data from the user and loads it into the data structure.



Function Stack	CPU Time: Total							
	Effective Time by Utilization					Spin Time		
	Idle	Poor	Ok	Ideal	Over	Imbalance or Serial Spinning	Lock Contention	MPI Busy Wait Time
tal	0.017s	197.470s	0s	0s	0s	7.8%	14.7%	2.9%
[stack]	0.017s	196.962s	0s	0s	0s	0.8%	14.7%	2.9%
_start	0.017s	196.962s	0s	0s	0s	0.8%	14.7%	2.9%
__libc_start_main	0.017s	196.962s	0s	0s	0s	0.8%	14.7%	2.9%
main	0.017s	196.710s	0s	0s	0s	0.8%	14.7%	2.9%
hmlp::mpigofmm::LaunchHelper<hmlp::SPDMatrix<float>>	0s	177.674s	0s	0s	0s	0.8%	14.7%	2.9%
hmlp::mpigofmm::Compress<hmlp::mpigofmm::centersplit<hmlp::SPDMatrix<float>, (int)2, fl	0s	132.989s	0s	0s	0s	0.4%	11.7%	1.7%
hmlp::mpigofmm::SelfTesting<hmlp::mpitree::Tree<hmlp::mpigofmm::Argument<hmlp::SPDM	0s	44.556s	0s	0s	0s	0.5%	3.1%	1.1%
hmlp::mpigofmm::DistFactorize<float, hmlp::mpitree::Tree<hmlp::mpigofmm::Argument<hr	0s	41.995s	0s	0s	0s	0.5%	2.9%	1.1%
hmlp::mpigofmm::Evaluate<(bool)1, hmlp::mpitree::Tree<hmlp::mpigofmm::Argument<hml	0s	1.146s	0s	0s	0s	0.0%	0.1%	0.0%
hmlp::mpigofmm::ComputeError<hmlp::mpitree::Tree<hmlp::mpigofmm::Argument<hmlp::	0s	0.648s	0s	0s	0s	0.0%	0.0%	0.0%
hmlp::mpigofmm::ComputeError<hmlp::mpitree::Tree<hmlp::mpigofmm::Argument<hmlp::	0s	0.523s	0s	0s	0s	0.0%	0.1%	0.0%
PMPI_Bcast	0s	0.100s	0s	0s	0s	0.0%	0.0%	0.0%
hmlp::DistData<(hmlp::Distribution_t)1, (hmlp::Distribution_t)5, float, std::allocator<float>>:	0s	0.096s	0s	0s	0s	0.0%	0.0%	0.0%
hmlp::Data<float, std::allocator<float>>::randn	0s	0.048s	0s	0s	0s	0.0%	0.0%	0.0%
hmlp::mpitree::Tree<hmlp::mpigofmm::Argument<hmlp::SPDMatrix<float>, hmlp::mpigofmm:	0s	0.105s	0s	0s	0s	0.0%	0.0%	0.0%
PMPI_Comm_dup	0s	0.013s	0s	0s	0s	0.0%	0.0%	0.0%
hmlp::tree::Tree<hmlp::mpigofmm::Argument<hmlp::SPDMatrix<float>, hmlp::mpigofmm::cer	0s	0.011s	0s	0s	0s	0.0%	0.0%	0.0%
PMPI_Init_thread	0s	9.935s	0s	0s	0s	0.0%	0.0%	0.0%
hmlp::SPDMatrix<float>::SPDMatrix	0.017s	6.452s	0s	0s	0s	0.0%	0.0%	0.0%

Figure 3.12.: Top hot spots during the execution of GOFMM using MPI

4. Conclusion and Future Work

In Section 1, we define the problem and also discuss motivation to solve the problem. Section 2 describes dimensionality reduction techniques such as PCA, DiffusionMaps and a brief discussion on the design of datafold package. It is followed by definition of kernel matrices, detailed explanation of compression algorithms and GOFMM's hierarchical algorithms. Section 3 contains the main part of the thesis that includes implementation of an instance of LinearOperator that uses GOFMM's matrix-vector multiplication methods via SWIG. Then various computational experiments on accuracy, scalability and performance are presented. Finally, this section provides a summary of the thesis and insights into further improvements.

4.1. Conclusion

With ever-growing applications with non-linear high-dimensional data in Machine learning and AI, it becomes more and more difficult to process it efficiently. We utilize an approach called manifold learning to reduce the dimensionality of such non-linear data. We recognize that one of the time consuming modules of the manifold learning techniques is performing eigen decomposition. Depending on the type of matrix, algorithm and the number of eigen pairs, eigen decompositions may have a time complexity of up to $\mathcal{O}(N^3)$. We mainly deal with dense SPD matrices in our thesis which have a space complexity of $\mathcal{O}(N^2)$. Eigen decompositions requires many matrix-vector computations, which have a time complexity of $\mathcal{O}(N^2)$. As the problem size increases, it becomes almost impossible to perform computations on such a large scale. We apply hierarchical algorithms from GOFMM library to perform matrix-vector multiplications since it has been shown to have a time complexity of $\mathcal{O}(NlogN)$.

We used the provided SWIG interface to write a python script that creates an instance of DiffusionMaps and LinearOperator. We provide an implementation for the method matvec in FullMatrix, which is derived from LinearOperator. We then collect the eigen values and vectors computed by scipy.linalg.sparse.eigsh that uses matvec provided in the library and compare them to the values computed by the matvec provided in the GOFMM library. Froebius error norms were gathered for various problem sizes starting from 1024 going up to 16384 and we obtained an order of convergence in 10^{-4}. We observed that when *max_leaf_node_size* decreases, the accuracy also decreases which is what is expected since with increasing rank, the number of data points being approximated decreases. When *max_off_diagonal_ranks* is kept the same as *max_leaf_node_size* while varying *max_leaf_node_size* and *user_tolerance*, the error norms stayed in the same order for all *user_tolerance* values. We also gathered error norms for varying the values of *max_leaf_node_size* and *max_off_diagonal_ranks* and found that as the values of *max_off_diagonal_ranks* decreased, the errors increased and this checks out with the theory provided in [1]. We then plotted 512 eigen values computed using both matvec methods and they are almost equal except for a few first largest and smallest values.

In integrating these two softwares and scaling them on a linux cluster, we made the following observations about scalability:

- Scaling on single node .i.e multi-core was done by varying the number of OMP_NUM_THREADS which means that it is pure OpenMP parallelization. Strong scaling on problem size of 8192 with threads starting from 1 to 56 showed that the OpenMP does not provide enough parallelism. The problem scaled ideally until 4 threads and almost stays constant with increasing threads.
- Problem size 16384 takes more than 48 hours to run with 1 and 56 threads. Run time reduces up to 16 threads and then increases for 28 and 56 threads.
- A comprehensive weak scaling could not be performed as the number of threads should be increased quadratically to keep a constant problem size per thread while the number of cores on the linux clusters we were provided is limited. Hence, the number of threads were increases linearly and expected the run times to double with increasing number of threads. But the efficiency decreases than the expected values and this can be explained by the fact that there is an enormous increase in the problem size per thread and probably could not be handled just by OpenMP parallelization.
- Although, we could not do it in full-scale, we conducted weak scaling with a problem size per thread equal to 1024 using 1, 4, 16 threads and observed that it does not scale well due to factors such as increased overhead time.
- Multi-node scaling could not be performed with the SWIG interface as it does not support mpiexec. Therefore data was written to a binary file and sent as a command line argument to GOFMM.
- Strong scaling on 2 nodes and various OMP_NUM_THREADS confirmed that the percentage of parallelism provided by OpenMP is quite small. On the other hand,

strong scaling on 2 nodes with varying MPI ranks showed that the efficiency is better than the ideal efficiency for a bigger problem size such as N=16384. It is explained by better effective CPU utilization. Scaling performed on varying number of nodes along with MPI ranks confirmed that with the problem sizes at hand, our code performs the best when distributed across 2 nodes.

- A comprehensive performance analysis was performed to identify bottlenecks and top hotspots such as mpigofmm :: compress. It was noted that the overall parallelism could be improved by improving algorithms such that effective CPU time increases and overhead, spin time reduces.
- Similar to the scaling performed on a a single node, the number of MPI ranks were doubled while also doubling the problem size. Performance was better for the case with varying the number of nodes along with MPI ranks while the performance on 2 and 4 nodes remained similar.

Therefore we have results that suggest that hierarchical algorithms from GOFMM not only scale well on the linux clusters available to us but also produce results with considerable accuracy.

4.2. Future Work

Having identified the drawbacks, hot spots and bottlenecks, we can work on the following topics to improve the current state of the software:

- datafold has been used with different solvers such as SLEPc/PETc, scipy and now GOFMM. The next step would be make all these options available to the users such that they just specify which one to use and datafold takes care of the rest. This would include designing the software in way that it makes sense with respect to performance, refactorization, writing a wide variety of tests and so on.
- This thesis mainly focused on uniform sampling of data with a Gaussian kernel. This could be expanded to test different types of data distributions and applications.
- We identified some bottlenecks and factors causing reduced CPU utilization, thread efficiency, parallelization etc. We can now realize solutions to these problems and actualize them to improve run times and scalability.
- Testing scalability on higher number of nodes and larger systems with different micro-architectures while testing the limits in terms of problem sizes and

variations of data. The scalability tests should include strong and weak scaling especially because we were limited by computational resources when it comes to weak scaling.

- GOFMM requires various parameters from users. Therefore we could benefit from a study on hyper-parameters and identifying a set of parameters for specific types of data.

Identifying bottlenecks and improving the methods that slow down the execution as discussed above would further incentivize the use of GOFMM for kernel matrices from datafold. To then unlock full potential, numerous datasets could be used on larger machines to test the functionality of the integration. Providing a SWIG interface with mpi4py to use the MPI methods through a python script would opens up numerous possibilities. Finally, a user-friendly interface to use datafold, with various eigendecomposition methods from libraries such as GOFMM, scipy, etc. in the back-end would improve the usability and accessibility.

Bibliography

[1] C. D. Yu, J. Levitt, S. Reiz, and G. Biros. *Geometry- Oblivious FMM for Compressing Dense SPD Matrices*. In Proceedings of SC17, Denver, CO, USA, 2017.

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

[3] T. Hofmann, B. Schölkopf, and A. J. Smola. *Kernel methods in machine learning*. The annals of statistics (2008), 1171–1220, 2008.

[4] L. Grasedyck, R. Kriemann, and S. Le Borne. *Parallel black box-LU preconditioning for elliptic boundary value problems*. Computing and visualization in science 11, 4 (2008), 273–291, 2008.

[5] D. Lehmberg, F. Dietrich, G. Köster, and H.-J. Bungartz. *datafold: data-driven models for point clouds and time series on manifolds*. Journal of Open Source Software, 5(51), 2283, 2020.

[6] L. Cayton. *Algorithms for manifold learning*. Univ. of California at San Diego Tech. Rep, 12(1-17):1, 2005.

[7] L. van der Maaten, E. Postma, and J. van den Herik. *Dimensionality reduction: A comparative review*. J Mach Learn Res, 10(66-71):13, 2009.

[8] Á. Fernández, A. M. González, J. Díaz, and J. R. Dorronsoro. *Diffusion Maps for Dimensionality Reduction and Visualization of Meteorological Data*. Neuro- computing, 163:25–37, 2015.

[9] R. Coifman and S. Lafon. *Diffusion Maps*. Applied and computational harmonic analysis, 21(1):5–30, 2006.

[10] S. M. Reiz. *Black Box Hierarchical Approximations for SPD Matrices*. Master's Thesis, TUM, 2017.

[11] C. Eckart and G. Young. *The approximation of one matrix by another of lower rank*. Psychometrika, 1 (1936), pp. 211-218, 1936.

[12] G. Golub and W. Kahan. *Calculating the singular values and pseudo-inverse of a matrix*. Journal of the Society for Industrial and Applied Mathematics, Series B: Numer- ical Analysis, 2(2):205–224, 1965.

[13] H. Cheng, G. Zydrunas, M. Per-Gunnar, and V. Rokhlin. *On the compression of low rank matrices*. SIAM Journal on Scientific Computing 26, no. 4 (2005): 1389–1404, 2005.

[14] M. Bebendorf. *Hierarchical matrices*. Springer Publishing Company, Incorporated, 1st edition, 2008.

[15] W. Hackbusch. *Hierarchical Matrices: Algorithms and Analysis*. Springer-Verlag Berlin Heidelberg, 2015.

[16] R. Priedhorsky and T. Randles. *Charliecloud: unprivileged containers for user-defined software stacks in HPC*. Association for Computing Machinery, 2017.

[17] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, et al. *SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python*. Nature Methods, 2020.

[18] *ARPACK Software*. http://www.caam.rice.edu/software/ARPACK/.

[19] D. C. Sorensen. *Implicitly restarted Arnoldi/Lanczos methods for large scale eigenvalue calculations*. SIAM J. Matrix Anal. Appl., 13, pp. 357–385, 1992.

[20] R. B. Lehoucq, D. C. Sorensen, and C. Yang. *ARPACK USERS GUIDE: Solution of Large Scale Eigenvalue Problems by Implicitly Restarted Arnoldi Methods*. SIAM, Philadelphia, PA, 1998.

[21] Y. Saad. *Chebyshev acceleration techniques for solving nonsymmetric eigenvalue problems*. Mathematics of Computation, 42(166), 567-588, 1984.

[22] *Intel® VTune™ Profiler User Guide*. Intel Corporation, 2020.

[23] N. J. Higham. *Accuracy and Stability of Numerical Algorithms*. Philadelphia: Soc. Industrial and Appl. Math, 1996.

[24] C. D. Yu, S. Reiz, and G. Biros. *Distributed O(N) Linear Solver for Dense Symmetric Hierarchical Semi-Separable Matrices*. IEEE 13th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC), 2019.

[25] T. Ge. *Efficient integration of a novel hierarchical matrix format (GOFMM) for solving kernel matrices from autoregression problems*. Master's Thesis, TUM, 2021.

Appendix

A. Detailed descriptions

The following section has instructions on how to install the integrated software on a local machine or the Linux cluster at LRZ. Please note that Linux cluster and CoolMUC-2, CoolMUC-3 are used interchangeably in the following section. It also provides a python script that illustrates the computation of kernel matrices from datafold and eigen decomposition of these matrices.

A.1. Installation

Docker has been used to package and deploy the software on the local machine. But since there is no docker support on the Linux cluster, we use Charliecloud container technology [16]. The installation can be done as shown below:

```
$ git clone https://gitlab.lrz.de/ge25duq/gofmm_datafold.git
$ cd gofmm_datafold
$ rm -rf gofmm
$ module load charliecloud
$ ch-tar2dir gofmm.tar.gz .
# IMPORTANT : The last line may not work if there is no git-lfs installed.
    Then a normal download of the tarball should be done.
```

Since git-lfs is not installed on the cluster, the files may not be fetched correctly. Hence alternatively, the charliecloud container can be downloaded as shown below:

```
$ rm -rf gofmm.tar.gz
$ wget https://www5.in.tum.de/~reiz/keerthi/gofmm.tar.gz
$ module load charliecloud
$ ch-tar2dir gofmm.tar.gz .
```

A.1.1. Creating Charliecloud image

Charliecloud image is created from the docker image on the local machine and has to be transferred to the Linux cluster. Once the repository is downloaded, the docker image can be built as shown below:

```
$ cd gofmm_datafold/docker4datafold/dockerTest_gcc/files_4_jupyterlab/
$ docker build -f ../dockerfiles/Dockerfile_test-gcc_gofmm-python-
    jupyterlab:shared   --tag=gofmm:latest . | tee test-gcc:gofmm-python-
    jupyterlab-shared.build.log
```

The docker image gofmm : latest can then be converted to a charliecloud image by installing charliecloud in the local machine and then executing A.1.1 the following command that creates a tarball in the current directory.

```
$ ch-builder2tar gofmm .
```

A.2. Compilation and Execution

Once the tarball is extracted, compilation can be done as follows:

```
$ ch-run --set-env=./gofmm/ch/environment -w ./gofmm -- bash
$ cd workspace/gofmm/build
$ source ../set_env_mpi.sh && cmake ..
$ make
$ make install
$ ./compile_swig_mpigofmm.sh   #compile swig interface files
```

It is important to give correct paths to all the necessary libraries in the environment file set_env_mpi.sh, otherwise the code will fail to work. Now the python script can be executed on the cluster using the following commands:

```
$ salloc
$ module load charliecloud
$ ch-run --set-env=./gofmm/ch/environment -w ./gofmm -- python3 /
    workspace/gofmm/test_linoperator.py
```

Performance analysis is done using Intel VTune Amplifier on lxlogin8 using the following commands:

```
$ salloc --nodes=2 --nodes-per-task=2
$ export OMP_NUM_THREADS=8
$ mpiexec -n 4 amplxe-cl -collect hotspots -r results ./test_mpigofmm
    16384 4096 0 4096 1 E-7 0.01 kernel dense kernel16k.bin
```

Intel parallel studio must be available on lxlogin8 for the above commands to work. After the execution is completed, 2 results folders will be available. These folders contain performance analysis information that can be viewed in a csv format or with the help of a graphical user interface. The GUI can be invoked with the command shown below:

```
$ amplxe-gui results.<node_id>/results.<node_id>.amplxe
```

With the help of the GUI, we can obtain a comprehensive analysis of the bottlenecks, parallelization and microarchitecture usage and so on.