Distributed-Memory Hierarchical Compression of Dense SPD Matrices

Chenhan D. Yu*, Severin Reiz†, and George Biros‡
*Department of Computer Science
† Technical University of Munich, Germany
*‡ Institute for Computational Engineering and Science
The University of Texas at Austin, Austin, Texas, USA
*chenhan@utexas.edu, †s.reiz@tum.de, ‡gbiros@acm.org

***Abstract*—We present a distributed memory algorithm for the hierarchical compression of symmetric positive definite (SPD) matrices. Our method is based on GOFMM, an algorithm that appeared in doi:10.1145/3126908.3126921. For many SPD matrices, GOFMM enables compression and approximate matrix-vector multiplication that for many matrices can reach $N \log N$ time—as opposed to N^2 required for a dense matrix. But GOFMM supports only shared memory parallelism. In this paper, we use the message passing interface (MPI) and extend the ideas of GOFMM to the distributed memory setting. We also propose and implement an asynchronous algorithm for faster multiplication. We present different usage scenarios on a selection of SPD matrices that are related to graphs, neural-networks, and covariance operators. We present results on the Texas Advanced Computing Center's "Stampede 2" system. We also compare with the STRUMPACK software package, which, to our knowledge, is the only other available software that can compress arbitrary SPD matrices in parallel. In our largest run, we were able to compress a 67M-by-67M matrix in less than three minutes and perform a multiplication with 512 vectors within 5 seconds on 6,144 Intel "Skylake" cores.**

I. Introduction

If $K \in \mathbb{R}^{N \times N}$ is a dense SPD matrix it requires $\mathcal{O}(N^2)$ storage and work for a matrix-vector multiplication (hereby ***"matvec"***). GOFMM [1] (reviewed in §II) constructs a matrix $\widetilde{K}$ (hereby ***"compression"***) using $\mathcal{O}(N \log N)$ entries from K so that $\|\widetilde{K} - K\| \leq \epsilon \|K\|$ (for a user-defined $\epsilon > 0$) and a ***matvec*** with $\widetilde{K}$ requires $\mathcal{O}(N \log N)$ work. The constants in the compression and multiplication complexity estimates depend on ϵ. (Thus, the compression may not offer benefit over a dense ***matvec***.) The only required input to GOFMM is a routine that returns a submatrix K_{IJ}, for arbitrary row and column index sets I and J. GOFMM belongs to the class of hierarchical matrix ($\mathcal{H}$-matrix in brief) approximation methods [2]. Roughly speaking, we say that a matrix $\widetilde{K}$ is ***hierarchically low-rank*** [2], [3], if

$$\widetilde{K} = D + S + UV, \tag{1}$$

where D is ***block-diagonal*** with ***every block being hierarchically low-rank***, U and V are ***low rank***, and S is ***sparse***. A ***matvec*** with $\widetilde{K}$ requires $\mathcal{O}(N \log N)$ work. The constant in the complexity estimate depends on the rank of U and V. It is critical to observe that this hierarchical low-rank structure is not invariant to row and column permutations. GOFMM appropriately permutes K ***using only entries from*** K, before constructing the matrices U, V, D, and S.

Background and significance: Dense SPD matrices appear in Cholesky and LU factorization [4], in Schur complement matrices [5], in Hessian operators in optimization in simulation [6] and machine learning [7], in kernel methods for statistical learning [8], [9], and in N-body methods and integral equations [10], [2]. In many applications, the entries of the input matrix K are given by $K_{ij} = \mathcal{K}(x_i, x_j)$, where x_i and x_j are points in D dimensions and $\mathcal{K}$ is a ***kernel function***, for example a Gaussian with bandwidth h: $\exp(-1/2h^{-2}\|x_i - x_j\|)$. For such problems, $\mathcal{H}$-matrix methods (like N-body methods) use clustering methods (typically trees) to define compressible blocks of K and appropriately permute it. However, such approach is not possible for arbitrary SPD matrices whose entries are not defined by kernel functions or for which we don't have points. Examples include SPD matrices related to graphs of social networks, protein/gene networks, fMRI data, microarray data, Hessian operators (e.g., from neural networks), maximum likelihood empirical covariance matrices, and kernel methods for word sequences and graphs [11], [12]. Furthermore, once a hierarchical approximation of K is built, we can construct an approximate inverse for K [2]. Given the large number of applications and the increasing data size, it is imperative that we develop fast, distributed-memory algorithms for algebraic operations such as matrix multiplication for generic dense SPD matrices.

Contributions: We introduce MPI-GOFMM for the compression of arbitrary SPD matrices. Our method requires only the ability to evaluate arbitrary entries from K. MPI-GOFMM is the only distributed memory method that permutes the matrix and allows for sparse direct interactions (S matrix in Equation 1) to improve compression. The scheme is described in §II-B and §III.

The novel algorithms in our work are summarized below:

- We introduce parallel algorithms for matrix-row nearest neighbors, permutations, and sampling for compressing off-diagonal blocks. We also extend the local essential tree idea (which require points) to the fully algebraic case.
- MPI-GOFMM exploits the parallelism of tree-based algorithms by runtime dependency analysis. MPI-GOFMM uses

SC18, November 11-16, 2018, Dallas, Texas, USA
978-1-5386-8384-2/18/$31.00

a task model that annotates reading/writing/communication activities and provides a runtime system that enables dynamic dependencies, communication overlapping, and nested parallelism. Experimentally, it outperforms static scheduling and OpenMP-based dynamic scheduling.

- We provide an asynchronous $\mathcal{H}$-matrix multiplication algorithm that can lead up to $2\times$ improvement over a synchronous scheme (§II-B) by avoiding expensive Alltoallv operations and enabling high FLOP intensity.
- We describe and implement interfaces for: (1) in memory evaluation of K, (2) out-of-core memory evaluation of K. We also define an interface that allows the user to redistribute any data related to the evaluation of K_{ij} (§IV, §V, and Appendix VII-A).
- We test the algorithm on PDE-constrained optimization matrices, kernel matrices, inverse graph Laplacian matrices, neural-network Hessians, and covariance matrices. We provide weak and strong scaling results, compare the synchronous and asynchronous version, and compare with STRUMPACK (which doesn't reorder the matrix) and ScaLAPACK, (the baseline N^2-distributed GEMM with no compression) (§V).

Limitations: Like GOFMM, we cannot simultaneously guarantee both approximation accuracy and work complexity. We require the ability to evaluate in $\mathcal{O}(1)$ time arbitrary matrix entries, which may not be possible in general. We can guarantee that $\widetilde{K}$ is symmetric but not its positivity. The compression phase is quite expensive, mainly due to a memory-bound pivoted QR but also due to dependencies and synchronizations.

Related work: The literature on $\mathcal{H}$-matrix methods is vast. For a review, see [1]. To our knowledge, there is no distributed-memory algorithm that (1) supports permutations to expose low-rank structure and (2) supports sparse direct evaluations (FMM-like interactions). The only algorithm and software that comes close is STRUMPACK [13], [14], [15], which constructs an approximation $\widetilde{K}$ without the sparse term S. (As we will see in §V this creates approximation difficulties for certain matrices). For dense matrices, STRUMPACK uses the lexicographic ordering. If no fast ***matvec*** of K is available, STRUMPACK requires $\mathcal{O}(N^2)$ work for compressing a dense SPD matrix, and $\mathcal{O}(N)$ work for the ***matvec***. Regarding distributed-memory parallelism, there is a lot of work for kernel matrices for which we have points. For a review of the state-of-the-art, see [16].

II. Methods

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

$$u = Kw \approx \widetilde{K}w, \quad \text{for} \quad w \in \mathbb{R}^{N\times r}, \tag{2}$$

can be computed with $\mathcal{O}(N \log N)$ work and $\mathcal{O}(N/p \log N)$ time where K, u, and w are also distributed on p processes. First, we review the key components of GOFMM in §II-A. Second, we discuss how it can be parallelized in §II-B. Third, we discuss the complexity of our algorithms in §III. For a side-by-side comparison between GOFMM (shared memory algorithms) and MPI-GOFMM (distributed memory algorithms), see Algorithm VII.1 and VII.2 in Appendix VII-C.

Task	Operations
Split(α)	p = argmax ($\{d_{ic} \vert i \in \alpha\}$) q = argmax($\{d_{ip} \vert i \in \alpha\}$) $[\mathtt{l},\mathtt{r}]$ = medianSplit($\{d_{ip} - d_{iq} \vert i \in \alpha\}$)
ID(α)	**if** α isleaf **then** J = α **else** J = $[\widetilde{\mathtt{lr}}]$ I=SampleFrom($\mathcal{N}(\alpha) \setminus \alpha$) $[\widetilde{\alpha}, P_{\widetilde{\alpha}J}]$=[GEQP3($K_{\mathtt{IJ}}$), $K_{I\widetilde{\alpha}}^{\dagger} K_{IJ}$]
FindNear(α)	**for each node** α isleaf **then** Near(α) $\cap$ = {MortonID(i) $\vert$ i $\in \mathcal{N}(\alpha)$}
FindFar(β,α)	**if** β is parent of any node in Near(α) **then** FindFar(l,α); FindFar(r,α); **else if** MortonID(β) > MortonID(α) **then** Far(α) $\cap$ = {β}
MergeFar(α)	**if** α isleaf **then return** FindFar(root,α) Far(α) = Far(l) $\cap$ Far(r) Far(l)\ = Far(α) Far(r)\ = Far(α)

TABLE I *Tasks invoked in the* ***compression*** *phase of* GOFMM.

A. Geometric-Oblivious FMM Review

Following [1], GOFMM comprises two phases: ***compression*** and ***evaluation***. In the ***compression*** phase, an SPD matrix K is compressed to $\widetilde{K}$ recursively using a binary tree such that

$$\widetilde{K}_{\alpha\alpha} = \begin{bmatrix} \widetilde{K}_{\mathtt{ll}} & 0 \\ 0 & \widetilde{K}_{\mathtt{rr}} \end{bmatrix} + \begin{bmatrix} 0 & S_{\mathtt{lr}} \\ S_{\mathtt{rl}} & 0 \end{bmatrix} + \begin{bmatrix} 0 & UV_{\mathtt{lr}} \\ UV_{\mathtt{rl}} & 0 \end{bmatrix}, \tag{3}$$

where l and r are **left** and **right** children of treenode α. Each node α contains a set of column indices (or row indices due to symmetry) and the two children evenly split this set such that $\alpha = \mathtt{l} \cup \mathtt{r}$. For example, task Split(α) in Table I uses an approximate ball split that only requires pairwise distances d_{ij}. In GOFMM, if a partition is computed by an induced distance metric[1] (not restricted to Euclidean distance) on matrix entries, then we are likely to find low-rank ($UV_{\mathtt{lr}}$) and sparse ($S_{\mathtt{lr}}$) structures in off-diagonal blocks.

Geometry-oblivious distance metric: While distances are typically defined with domain-specific geometry information (e.g. Euclidean distance embedded in ordering K), GOFMM uses a class of *"geometry-oblivious"* distances that can be defined and computed in $\mathcal{O}(1)$ using raw entries of a given SPD matrix K. For example, the Gram-ℓ^2 distance $d_{ij} = K_{ii} - 2K_{ij} + K_{jj}$ and the Gram-angle distance (the angle between underlying gram vectors using the cosine similarity) $d_{ij} = 1 - K_{ij}^2/(K_{ii}K_{jj})$ can be computed using only matrix entries. Throughout the paper, we use the second definition (Gram-angle) to compute pairwise distances.

Sparse corrections and neighbor search: GOFMM performs hierarchical clustering of the matrix indices using a binary tree and index distance d_{ij}. Ideally, indices in different clusters are separated by large distances (Far-clusters) that correspond to off-diagonal blocks, which in turn, can be low-rank approximated. However, a balanced spatial tree is prone to misclassification on the cluster boundaries. Indices around the boundaries exhibit high sensitivity, i.e., they cannot be approximated accurately using a low-rank decomposition. These interactions (pairs of indices) are discovered by neighbor search, forming the sparse correction S in Equation 1 and Equation 3. We find

[1] As we already mentioned, the off-diagonal low-rank property is not invariant to matrix permutation, and it strongly depends on the ordering of the columns (and rows). Hence, a ordered distance metric is essential for matrix partition.

Task	Operations	FLOPS
N2S(α)	**if** α isleaf **then** $\widetilde{w}_\alpha = P_{\widetilde{\alpha}\alpha} w_\alpha$ **else** $\widetilde{w}_\alpha = P_{\widetilde{\alpha}[\widetilde{1}\widetilde{r}]}[\widetilde{w}_1; \widetilde{w}_r]$	$2msr$ $2s^2r$
S2S(β)	$\widetilde{u}_\beta = \sum_{\alpha \in \text{Far}(\beta)} K_{\widetilde{\beta}\widetilde{\alpha}} \widetilde{w}_\alpha$	$2s^2r\lvert\text{Far}(\beta)\rvert$
S2N(β)	**if** α isleaf **then** $u_\beta = P_{\beta\widetilde{\beta}} \widetilde{u}_\beta$ **else** $[\widetilde{u}_1; \widetilde{u}_r] += P_{[\widetilde{1}\widetilde{r}]\widetilde{\beta}} \widetilde{u}_\beta$	$2msr$ $2s^2r$
L2L(β)	$u_\beta += \sum_{\alpha \in \text{Near}(\beta)} K_{\beta\alpha} w_\alpha$	$2m^2r\lvert\text{Near}(\beta)\rvert$

TABLE II *Tasks N2S (nodes to skeletons), S2S (skeletons to skeletons), S2N (skeletons to nodes), and L2L (leaves to leaves) occur in GOFMM* ***evaluation*** *phase.*

k-nearest neighbors $\mathcal{N}$ by an approximate randomized spatial tree using the geometry-oblivious distance of K.

Low-rank approximation: GOFMM compresses the off-diagonal blocks of node α with a nested interpolative decomposition (task ID in Table I) [17]. A subset of s indices $\widetilde{\alpha} \subset \alpha$ is selected to construct a rank-s approximation such that

$$K_{I\alpha} \approx K_{I\widetilde{\alpha}} P_{\widetilde{\alpha}\beta}, \text{ or } K_{I[\widetilde{1}\widetilde{r}]} \approx K_{I\widetilde{\alpha}} P_{\widetilde{\alpha}[\widetilde{1}\widetilde{r}]} \tag{4}$$

if node α has children l and r. Skeleton $\widetilde{\alpha}$ is computed by rank-revealing QR factorization (GEQP3) to find columns $\widetilde{\alpha}$ that can capture the range of $K_{I\alpha}$. Here I is an index set that contains all off-diagonal entries. GOFMM uses nearest neighbor-based importance sampling to avoid the $\mathcal{O}(N^2)$ compression cost of $K_{I\alpha}$ using standard algorithms (say QR). Using neighbors as row samples, as opposed to uniform sampled rows, produces more accurate skeletonizations [1], [16].

Once the index set $\widetilde{\alpha}$ has been determined, $P_{\widetilde{\alpha}\alpha}$ and $P_{\widetilde{\alpha}[\widetilde{1}\widetilde{r}]}$ are computed using TRSM using the upper triangular matrix computed from GEQP3. For non-leaf nodes we greedily select their skeletons $\widetilde{\alpha}$ from the node's children's skeletons $\widetilde{1} \cup \widetilde{r}$. As a result, coefficient matrices have the following recursive relation (known as the *telescoping* relation)

$$P_{\widetilde{\alpha}\alpha} = P_{\widetilde{\alpha}[\widetilde{1}\widetilde{r}]} \begin{bmatrix} P_{\widetilde{1}\mathtt{l}} & \\ & P_{\widetilde{r}\mathtt{r}} \end{bmatrix}. \tag{5}$$

All skeletons (GEQP3) and coefficient matrices (TRSM) can be computed with a postorder traversal, and the K blocks $K_{\alpha\beta}$ can be approximated by $P_{\alpha\widetilde{\alpha}} K_{\widetilde{\alpha}\widetilde{\beta}} P_{\widetilde{\beta}\beta}$.

Evaluation: The four tasks in Table II compute the ***matvec*** of $\widetilde{K}$. (See Appendix VII-C, and Algorithm VII.2 for more details.) Task L2L computes direct ***matvec***s ($S_{\mathtt{l}\mathtt{r}}$ and $S_{\mathtt{r}\mathtt{l}}$ in Equation 3) stored in the near interaction lists $\text{Near}(\alpha)$. These lists are defined using the nearest-neighbor information $\mathcal{N}$ (computed by task FindNear(α) in Table I). Tasks N2S, S2S, and S2N compute all approximate ***matvec***s ($U_{\mathtt{l}}V_{\mathtt{r}}$ and $U_{\mathtt{r}}V_{\mathtt{l}}$) in far interaction lists $\text{Far}(\alpha)$ constructed with FindFar(β,α), and MergeFar(α) §II-B. We discuss how these lists are constructed in §II-B. The total interactions are computed using a postorder traversal with N2S, an any-order traversal with S2S, and a preorder traversal with S2N.

B. Distributed-Memory Parallelism

MPI-GOFMM is parallelized using MPI+OpenMP, following the algorithms for distributed-memory treecodes described in [18] and the task-based runtime scheduler design in [1]. The latter will be generalized to perform out-of-order execution in the distributed memory setting. We briefly describe the system below. See Appendix VII-B for a discussion.

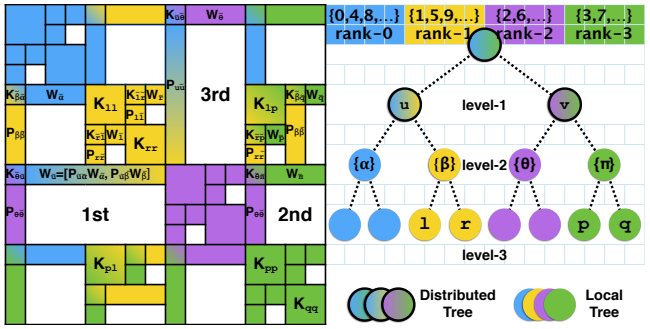

Fig. 1 *A 4-processes distributed metric tree (right), and its algebraic FMM compression (left): each color represents an MPI process. Nodes and factors (with a single color) are local. Mixed-colored nodes and factors above level-*$\log(p)$ *are distributed.*

Runtime dependency analysis: To help exploit the parallelism of tree-based algorithms in different granularities, MPI-GOFMM employs a self-contained runtime system. *Lazy symbolic tree traversals* are performed ahead of execution to annotate reading/writing/communication workloads of each task (i.e., a workload defined at each tree node visit) without performing any actual computation. Then, the runtime constructs a direct acyclic graph (DAG) by transforming into data dependencies between tasks. Tasks satisfying all dependencies are dispatched to computing resources according to EFT (Earliest Finish Time) policy in an *out-of-order* manner. As a result, the system can systematically handle thread binding, communication overlapping, and nested parallelism.

Data distribution: Given p processes, MPI-GOFMM partitions the matrix indices using a complete binary tree (see Figure 1). Any computation involving tree nodes and low-rank factors (depicted with different colors in Figure 1) above the $\log p$ level of the tree requires communication.

Since we do not control the data distribution of K, we only repartition the ownership of the N matrix 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 (see Appendix VII-A) assuming the user provides an array with fixed-size memory per index. This is conceptually equivalent to redistributing points, although the data structure can contain anything. MPI-GOFMM only invokes an opaque (i.e. black-box) callback function $K(i,j)$ without accessing the content in the data structure directly.

To describe our algorithms, we follow the notation inspired by libElemental [19]. We use four possible index partitioning (with possible replications) among MPI processes: [*] assume all processes have a copy of all indices; [CYC] is the 1D cyclic partitioning, which we assume to be the input partitioning; [IDS] (distributed tree partition) is the repartitioning (permutation) for MPI-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

Task DistSplit(α) : Split(α)	
if size(α) $= 1$ **then return** [l,r]=Split(α)	
else /** Above level-log(p) of the tree */	
case rank(α) $< \frac{1}{2}$size(α) :	**case** rank(α) $\geq \frac{1}{2}$size(α) :
comm(l) $=$ CommSplit(α)	comm(r) $=$ CommSplit(α)
[l,$\bar{\text{l}}$]=Split(α)	[r,$\bar{\text{r}}$]=Split(α)
dest $=$ rank(α) $+ \frac{1}{2}$size(α)	dest $=$ rank(α) $- \frac{1}{2}$size(α)
Sendrecv($\bar{\text{r}}$, $\bar{\text{l}}$, dest)	Sendrecv($\bar{\text{l}}$, $\bar{\text{r}}$, dest)
Task DistID(α) : ID(α)	
if size(α) $= 1$ **then return** [$\tilde{\alpha}$,$P_{\tilde{\alpha}J}$]=ID(α)	
else /** Above level-log(p) of the tree */	
case rank(α) $= 0$:	**case** rank(α) $= \frac{1}{2}$size(α) :
Recv($\tilde{\text{r}}$); ID(α)	Send($\tilde{\text{r}}$)
Task DistMergeFar(α) : MergeFar(α)	
if size(α) $= 1$ **then return** MergeFar(α)	
else /** Above level-log(p) of the tree */	
case rank(α) $= 0$:	**case** rank(α) $= \frac{1}{2}$size(α) :
Sendrecv(Far(l),Far(r))	Sendrecv(Far(r),Far(l))
Far(α) $=$ Far(l) $\cap$ Far(r)	Far(α) $=$ Far(l) $\cap$ Far(r)
Far(l)$\backslash$ $=$ Far(α)	Far(r)$\backslash$ $=$ Far(α)

TABLE III *Tasks invoked in GOFMM* ***compression*** *phase.*

multiple MPI processes. In the following, we overload the "=" operator to also denote repartitioning. In most cases we implement this using several Alltoallv operations.

Tree partition: DistSplit(α) in Table III is the first distributed-memory task we invoke in the **compression** phase (Algorithm II.1), partitioning the metric tree shown in Figure 1. Given the initial order [CYC], DistSplit(α) redistribute indices to [IDS] order in a preorder traversal. Different from the shared memory task Split in Table II, DistSplit further redistributes indices for distributed nodes above level-log p.

Each distributed tree node above level-log(p) has a unique MPI sub-communicator for collectives. A node α is assigned with communicator comm(α), where size(α) and rank(α) denote the local MPI rank and size in comm(α). *While different tasks may be assigned according to* rank(α)*, we describe our distributed algorithms side-by-side as shown in Table III. The left column shows algorithms for processes with* rank(α) $< \frac{1}{2}$size(α)*, and the right column represents the other group of processes. We use this algorithm format in the remainder of this section.*

While splitting a node α, processes are divided into two subgroups comm(l) and comm(r) according to rank(α). Task Split(α) 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). Indices l are kept by processes with rank(α) $< \frac{1}{2}$size(α), and r are kept by the other group of processes. $\bar{\text{l}}$ and $\bar{\text{r}}$ are exchanged between "partner" processes (with rank dest). 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., number of indices owned by α less than parameter m).

All nearest-neighbor: We compute κ-nearest neighbors for each pivot in parallel (using the same distance metric as DistSplit(α)) in Algorithm II.1. Neighbors will be used for sampling and to find sparse corrections S_{l} and S_{r}. Neighbors $\mathcal{N}$ are approximated iteratively using randomized

Algorithm II.1 DistCompress(K)

1: [Preorder] **foreach node** α **do** DistSplit(α)
2: $\mathcal{N}$[*,CYC] $=$ AllNearestNeighbors(K)
3: $\mathcal{N}$[*,IDS] $= \mathcal{N}$[*,CYC]
4: [Postorder] **foreach node** α **do** DistID(α)
5: [Anyorder] **foreach leaf** α **do** FindNear(α)
6: Symmetrize(Near)
7: IDS2LET(Near)
8: [Postorder] **foreach** α **do** DistMergeFar(α)
9: Symmetrize(Far)
10: IDS2LET(Far)

trees (see [20] for a summary and an implementation for the geometric case). In each iteration, we create a randomized metric tree by randomly assign p and q in task Split(α), where the leaf node size is 4κ (larger than κ to reduce the number of tree iterations). Exhaustive neighbor search is performed on all indices that belong to the same leaf node (by which generate κ candidates). Neighbor candidates are redistributed from [IDS] back to [CYC] distribution. Finally, we merge existing neighbors with the new candidates, trim the list down to κ neighbors again, and iterate until we converge.

Algebraic FMM compression (Algorithm II.1): With a metric tree and neighbors $\mathcal{N}$, we first redistribute $\mathcal{N}$ from [CYC] to [IDS]. As a result, neighbors of a local node α can be accessed as $\mathcal{N}$[*,α]. Neighbors are used to defining near interaction lists in task FindNear(α) (see Table I); neighbors are also used as importance samples in task DistID in Table III to improve the quality of the low-rank compression.

Task DistID compresses the off-diagonal blocks of K using a nested ID described in Equation 4 in a postorder traversal. Different from its shared memory version ID defined in Table II, skeleton $\tilde{\alpha}$ is only computed on rank 0 for distributed tree nodes above level-log p. This requires gathering right child's skeleton $\tilde{\text{r}}$ from its local sibling.

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 to [LET] distribution that satisfies all local essential data dependencies required by the interaction lists.

List Near(α) is constructed by task FindNear (see Table I) in an anyorder traversal. It is defined on leaf nodes and contains only leaf nodes. For each neighbor i $\in \mathcal{N}$[*,α], we add MortonID(i) (the same as the node MortonID of its belonged leaf node) to Near(α) such that

$$\text{Near}(\alpha) \cap = \{\text{MortonID(i)} \mid \text{i} \in \mathcal{N}[*,\alpha]\}. \quad (6)$$

Notice that |Near(α)| determines the number of direct entry evaluations (sparsity of S_{lr} and S_{rl}). To control the complexity, we use an user-defined parameter **budget** (b in brief) introduced in [1] such that |Near(α)| $< b(N/m)$. When exceeding the budget (depending on how neighbors are distributed over leaf nodes), we use a majority voting to shrink

Function Symmetrize(List=[Near/Far])
foreach node α **and** $\beta \in$ List(α) **do**
pairs_sent_to[owner(β)] $\cap =$ pair(β, α)
Alltoallv(pairs_sent_to,pairs_received_from)
foreach rank p **and** pair$(\beta, \alpha) \in$ pairs_received_from[p] **do**
List(β)[p] $\cap = \alpha$
Function IDS2LET(List=[Near/Far])
foreach α **and** $\beta \in$ List(α) **do**
List_sent_to[owner(β)] $\cap =$(List=Near) **?** β **:** $\widetilde{\beta}$
Alltoallv(List_sent_to,List_received_from)
Function Redistribute(List=[Near/Far])
foreach rank p **and** $\alpha \in$ List_received_from[p] **do**
data_sent_to[p] $\cap =$ (List=Near) **?** w_α **:** $\widetilde{w}_\alpha$
Alltoallv(data_sent_to,data_received_from)
foreach rank p **and node** $\beta \in$ data_received_from[p] **do**
(List=Near) **?** w_β **:** $\widetilde{w}_\beta$ = data_received_from[p][β]

TABLE IV *All-to-all communication for symmetrizing and redistributing indices and data from* `[IDS]` *to* `[LET]` *order.*

Task DistN2S(α) : N2S(α)	
if size$(\alpha) = 1$ **then return** N2S(α)	
else /** Above level-log(p) of the tree */	
case rank$(\alpha) = 0$ **:**	**case** rank$(\alpha) = \frac{1}{2}$size(α) **:**
Recv$(\widetilde{w}_\mathtt{r})$; N2S$(\alpha)$	Send$(\widetilde{w}_\mathtt{r})$
Task DistS2N(α) : S2N(α)	
if size$(\alpha) = 1$ **then return** S2N(α)	
else /** Above level-log(p) of the tree */	
case rank$(\alpha) = 0$ **:**	**case** rank$(\alpha) = \frac{1}{2}$size(α) **:**
$[\widetilde{u}'_\mathtt{l}; \widetilde{u}'_\mathtt{r}] = P_{[\widetilde{\mathtt{l}}\widetilde{\mathtt{r}}]\widetilde{\alpha}} \widetilde{u}_\alpha$	NOP
Send$(\widetilde{u}'_\mathtt{r})$; $\widetilde{u}_\mathtt{l} += \widetilde{u}'_\mathtt{l}$	Recv$(\widetilde{u}'_\mathtt{r})$; $\widetilde{u}_\mathtt{r} += \widetilde{u}'_\mathtt{r}$
Task DistS2S(β)	
foreach rank p **and node** $\alpha \in$ Far(β)[p] **do** $\widetilde{u}_\beta += K_{\widetilde{\beta}\widetilde{\alpha}} \widetilde{w}_\alpha$	
Task DistL2L(β)	
foreach rank p **and node** $\alpha \in$ Near(β)[p] **do** $u_\beta += K_{\beta\alpha} w_\alpha$	

TABLE V *Tasks involved in* `MPI-GOFMM` ***evaluation*** *phase.*

Algorithm II.2 u[IDS,$\star$]=DistEvaluate(w[IDS,$\star$])

1: Redistribute(Near)
2: [Anyorder] **for each leaf** α **do** DistL2L(α)
3: [Postorder] **for each node** α **do** DistN2S(α)
4: Redistribute(Far)
5: [AnyOrder] **for each node** α **do** DistS2S(α)
6: [Preorder] **for each node** α **do** DistS2N(α)

the list. This allows us to control the percentage of direct entry evaluation in a finer granularity than solely depending on the neighbor distribution.

List Far(α) is constructed in Algorithm II.1 with a postorder traversal of task DistMergeFar. For nodes below level-log p, task MergeFar described in Table I is called to construct the list locally. For leaf nodes, FindFar(β, α) is called to partially traverses the tree in preorder. Throughout the recursion, argument α does not change, but β changes according a preorder ordering of the tree nodes. The recursion continues to the left and right children (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$(\alpha)\}$. 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, and the lower part will be symmetrized later.

For an internal node α, interaction list Far(α) is merged from Far(l) and Far(r). The merging process involves point-to-point communication while visiting the distributed tree nodes in the postorder traversal. Far(α) is computed as a set intersection ($\cap$) between Far(l) and Far(r) for internal nodes. While α is distributed, Far(l) is owned by rank 0 in comm(α), and Far(l) is stored on rank $\frac{1}{2}$size(α). These two ranks exchange their interaction lists, performing set interaction and update their Far(l) and Far(r). Far(α) is stored on rank 0, which handles all computation for α during the evaluation.

Symmetrizing: Given that κ-nearest neighbors are not necessarily symmetric, Near(α) can also be non-symmetric (i.e. $S_{\mathtt{lr}} \neq S_{\mathtt{rl}}^T$). In a distributed-memory environment, we invoke function Symmetrize(Near) (see Table IV) to ensure that if $\beta \in$ Near(α), then $\alpha \in$ Near(β). For each node α, we must send a pair (α, β) to the rank that owns β (denoted as owner(β)). This is done in two stages. We first pack all pairs into p messages by traversing all interaction lists, and then invoke Alltoallv to exchange and symmetrize the lists. Similarly, we invoke Symmetrize(Far) in Algorithm II.1 after the far interaction lists were built. Notice that this process not only symmetrizes but also splits a list into p sublists according to their sources. With these sublists, we can describe the point-to-point data dependencies and replace later replace synchronous Alltoallv operators with several asynchronous Isend and Irecv.

Local essential tree (LET): Recall that the tree nodes (including indices and factors) in Figure 1 are distributed among p processes. While interaction lists may contain tree nodes from other processes, we must properly redistribute the indices and factors from the tree order [IDS] to the local essential tree order [LET] before the ***evaluation*** phase. For example, Near(l) $= \{$l,p$\}$ contains tree nodes from both green and yellow processes. To evaluate submatrices $K_{\mathtt{lp}}$ and $K_{\mathtt{pl}}$, we must redistribute indices carried by l and p. To compute ***matvec***s $K_{\mathtt{lp}} w_\mathtt{p}$ and $K_{\mathtt{pl}} w_\mathtt{l}$, we must redistribute weights $w_\mathtt{p}$ and $w_\mathtt{l}$ as well. Similarly, Far$(\theta) = \{\pi, \mathtt{u}\}$ contains tree nodes from all other processes (since node u is distributed among blue and yellow processes). As a result, we must also redistribute indices $\widetilde{p}i$, $\widetilde{\mathtt{u}}$, skeleton weights $\widetilde{w}_\pi$, and $\widetilde{w}_\mathtt{u}$. In Table IV, function IDS2LET() redistributes indices and skeletons, and function Redistribute() redistributes weights and skeleton weights.

Evaluation (Algorithm II.2): Given weights w[IDS] $\in \mathbb{R}^{N \times r}$ (redistributed from CYC), ***matvec*** of $\widetilde{K}$ involves four steps: an anyoder traversal of task DistL2L, a postorder traversal of task DistN2S, an anyorder traversal of task DistS2S, and a preorder traversal of task DistS2N. Different from the shared memory tasks listed in Table II, tasks in Table V typically require communication. In DistN2S and DistS2N tasks, computation on distributed tree nodes above level-log(p) requires point-to-point communication using the local communicator. Nevertheless weights w and skeleton weights $\widetilde{w}$ must be redistributed from [IDS] to [LET] order before the execution of all DistL2L and DistS2S tasks. Function Redistribute() in Table IV

requires `Alltoallv` communication, which scales poorly on distributed systems and also prevents out-of-order execution between four steps.

Asynchronous message passing: To overlap communication with computation without reducing the maximum parallelism obtained from several traversals, `Alltoallv` in Algorithm II.2 must be broken down into at least[2] $(p-1)^2$ `Isend` and `Irecv` tasks handled by the runtime system. To prevent blocking any worker (core), asynchronous messages are received by periodic polling using `Iprobe`. This problem is known as the **dynamic sparse data exchange** (DSDE) [21]. To prevent any rank from early termination before all messages received, we modify the nonblocking-consensus solution of DSDE to fit into our client-server base runtime system. We use a distributed dependency graph (Figure 2) to explain this distributed mechanism.

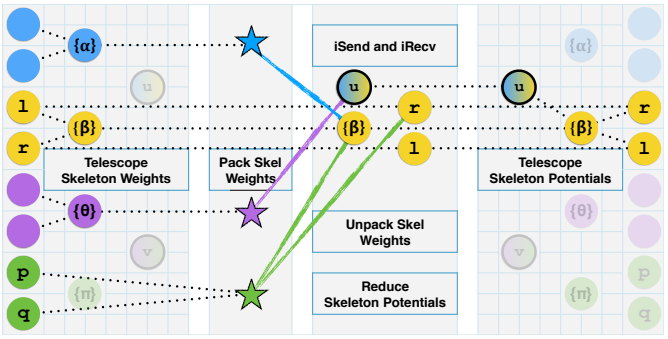

Fig. 2 *Distributed dependency graph for process yellow during evaluation (without `DistL2L`). The whole graph has four stages: an upward pass to compute skeleton weights, packing skeleton weights to $p-1$ messages according to the far interaction list, exchanging and unpacking messages, and finally a downward pass to compute the aggregate far contribution.*

Edges and nodes in Figure 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 `DistS2S` tasks, and the three star-shape tasks represent the incoming messages from process blue, purple and green. In the following, we focus on the case of `DistS2S`. Tasks `DistL2L` are implemented similarly.

In Table V, task `DistS2S` computes the reduction of the far interaction list over p processes. For example, node β in the middle of Figure 2 has three incoming edges carrying skeleton weights $\widetilde{w}$ from process blue, green, and yellow (itself). Waiting for all messages to arrive inherently imposes a global synchronization barrier. To resolve this bottleneck, a `DistS2S` task must also be broken down into at least p subtasks. Each subtask only depends on one incoming message and contributes partly to the final lump sum. Recall that we have split interaction list `Far`(β) into p sublists `Far`(β)`[0:p-1]` during the symmetrization. By splitting task

[2]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.

```
Task MySend(TAR,TAG,List=[Near/Far]) : SendTask
/** Isend(data_sent_to,TAR,TAG) is handled
    by SendTask during after packing. */
Override RecvTask::Pack(data_sent_to) :
    foreach node α ∈List_recevied_from[TAR] do
        data_sent_to ∩= (List=Near) ? w_α : w̃_α
Task MyRecv(SRC,TAG,List=[Near/Far]) : RecvTask
/** Recv(data_received_from) is handled
    by Listener::Listen() before unpacking. */
Override RecvTask::Unpack(data_received_from) :
    foreach node α ∈List_sent_to[SRC] do
        (List=Near) ? w_β : w̃_β = data_received_from[β]
Function Listener::Listen(terminaton_signal)
while in the runtime epoch do
    [SRC,TAR]=Iprobe(ANY_SRC_TAG)
    if task=msg_tasklist.find[SRC](TAG) then
        Recv(data_received_from,SRC,TAG)
        task.Unpack(data_received_from)
    else StealAndConsumeFromOthers()
    if termination_signal then
        Ibarrier(&request) only once
        if Test(request) then return
Function AsyncRedistribute(List=[Near/Far])
foreach rank p do msg_tasklist.insert(pair(SRC,TAG))
foreach rank p do MySend(p,AUTOTAG,List)
foreach rank p do MyRecv(p,AUTOTAG,List)
```

TABLE VI *Tasks and functions for asynchronous communication and redistribution: `List_sent_to` and `List_received_from` are inherited from `IDS2LET(List)` in Table IV.*

`DistS2S` accordingly, the data required by each subtask only depends on one message to arrive. To address the possible concurrent write on factor $\widetilde{u}$, we use a mutex (or an atomic update) to resolve the race condition.

Message tasks: To handle asynchronous message passing, we introduce message tasks and listeners in Table VI. Function `AsyncRedistribution()` simultaneously creates a pair of `SendTask` (with `Isend`) and `RecvTask` (with `Recv`). By overriding `SendTask::Pack()`, child class `MySend` can generate **Read-After-Write (RAW)** dependencies from other normal tasks during the dependency analysis phase. Similarly, child class `MyRecv` can result in **RAW** dependencies by overriding `RecvTask::Unpack()`. During the execution epoch, the former is dispatched to normal workers while all incoming dependencies are satisfied (i.e. `SendTask` is treated as normal tasks), but the latter can only be picked up and consumed by listeners.

Listener: To implement non-blocking periodic polling, each normal worker (thread) can be promoted to consume `RecvTask` tasks. Function `Listener::Listen()` describes how listeners perform non-blocking periodic polling to consume both `RecvTask` and normal tasks. A listener constantly probes for messages and check if their `[SRC,TAG]` match any `RecvTask` task in the list. If so, messages will be received and unpacked by the listener. Otherwise, listeners behave the same as normal workers, trying to dispatch a normal task from its own ready queue or steal one from others. Finally the task (either a `RecvTask` or normal one) will 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 normal tasks even there is no incoming message. As a result, we typically use

several listeners (half of the total workers) as default.

Listeners terminate while reaching a global consensus on variable `termination_signal`. This signal is set by normal 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 asynchronous distributed barrier (`Ibarrier`) and periodically test for global consensus. This guarantees all incoming message will be received and handled before global termination.

III. Complexity Estimates

The complexity analysis of `MPI-GOFMM` is based on the work for high dimensional kernel matrices presented in [16], [22], [23]. One minor difference is that the symmetry is not enforced in those works—whereas we do so in `MPI-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 $\mathcal{D} = \log(N/m)$ be the tree depth, $n = N/p$ 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, s be the maximum rank of the skeletonization, and b be the "budget" parameter. For simplicity, let $m \geq s$ and $s/m = \mathcal{O}(1)$. Parallel tree partition and skeletonization take

$$(t_s + t_w)\log^2 p \log N + t_w \kappa n \log p + s^3\left(\frac{n}{m} + \log p\right) \quad (7)$$

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

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

$$|\texttt{Near}(\alpha)| = \mathcal{O}(b(N/m)),\ |\texttt{Far}(\alpha)| = \mathcal{O}(\mathcal{D}b(N/m)) \quad (8)$$

in the worst case[3]. 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

$$t_s p + t_w \mathcal{D} b n^2 \quad (9)$$

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

We now compare asynchronous and synchronous evaluation. Algorithm II.2 involves four `Alltoallv` calls with complexity shown in (9). For simplicity, we assume the total direct evaluation computed in tasks `DistL2L` and `DistS2S` are bounded as bN^2. All low-rank approximations including `DistN2S` and `DistS2N` 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 `Alltoallv`, the main advantage of asynchronous evaluation is to overlap communication and resolve the diminishing parallelism in `DistN2S`. With sufficient budget, communication can be hidden. We observe up to 2.5× speedup comparing to synchronous evaluation in §V.

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

IV. Experimental Setup

Implementation and hardware: `MPI-GOFMM` is implemented in `C++` with `MPI` (`MPI_THREAD_MULTIPLE` is required) and `OpenMP`. `MPI-GOFMM`'s only dependencies are multi-threaded `BLAS/LAPACK` and `MPI` (we used the Intel MPI library). We conducted our experiments on the TACC's "Stampede 2" system. Each node has two 24-core, Intel Xeon Platinum 8160 "Skylake" processors, which have two AVX-512 units. Thus, theoretical peak per 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). For the out-of-core experiments, the input data is striped (distributed over 30 stripes with 1MB stripe size) and stored on mounted disks (`$SCRATCH` partition on "Stampede 2"). All runs are performed in single precision.

Test matrices: **K01** is a forward 2D Poisson operator and **K02** is a 2D regularized inverse Laplacian squared, resembling the Hessian operator of a PDE-constrained optimization problem. **K04**, **K07** are kernel matrices using 6D points with Gaussian and Laplacian kernel functions. **K11** is an inverse squared 1D variable coefficient Poisson problem operator, and **K12** is a 2D advection-diffusion operator on a regular equidistant grid in lexicographic numbering. **G03** is a graph Laplacian of size $N = 89400$ called "denormal" from UFL. **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 an 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.

For the weak and strong scaling experiments, we use a kernel matrix from a high-energy physics point set (SUSY) and a synthetic dataset (normally distributed points in 6D). In these experiments, since we wish to isolate the performance of `MPI-GOFMM`, we loaded points from disk and used them *only* for the kernel evaluation. We did not use them for repartitioning or sampling. In this case, every K_{ij} evaluation costs $\mathcal{O}(1)$ cost, the evaluation of a Gaussian kernel using points x_i and x_j.

Parameters and approximation error: Like `GOFMM`, our algorithm allows different *distance metrics*: Gram-ℓ^2, Gram-angle, and geometric-ℓ^2. For ***compression***, we set parameters

m (leaf node size), s (maximum rank), τ (accuracy tolerance used to adaptively select the rank), and κ (number of neighbors used for S and for sampling). The relative error ϵ_2 is computed by a Frobenius norm towards a (sampled) exact ***matvec***.

V. Empirical Results

We conduct a number of experiments (#1–#46) to illustrate the scalability, absolute efficiency (as the ratio of the theoretical peak in `GFLOPS`) and the capability of `MPI-GOFMM` to deal with large-scale SPD matrices. We discuss how our *out-of-order* task-based scheduling can improve the performance by overlapping computation with communication. In all, we report results from 45 experiments organized as follows: scaling experiments (Figure 3 and Figure 4); comparisons with `ScaLAPACK` and `STRUMPACK` (Table VII); out-of-core experiments (Table VIII). All experiments are indexed using `#experiment_id` in the figures and tables.

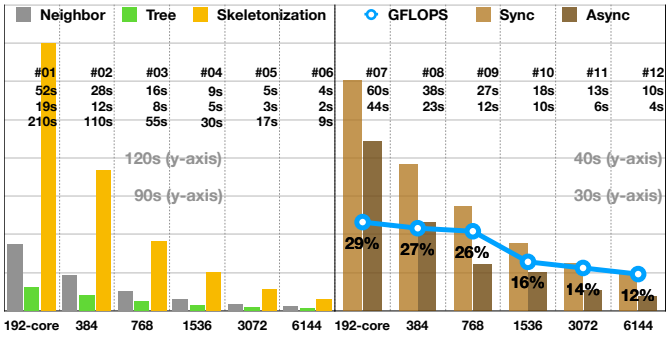

Fig. 3 ***Strong scaling (runtime in seconds as a function of the number of cores)*** *of `MPI-GOFMM` applied to a 5M-by-5M Gaussian kernel matrix generated by dataset SUSY.* ***Left:*** *Compression time and break down into three phases: neighbor search, tree creation, and skeletonization.* ***Right:*** *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`).*

Strong scaling: In Figure 3 (#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 §IV). We do not report `GFLOPS` performance for the compression phase since it is mostly memory-bound due to 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` [24] (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 §II-B. Overall tree partition, interaction lists, and symmetrization (yellow bars) degrade by 69%. Building symmetry interaction lists require several `Alltoallv` steps and are 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.

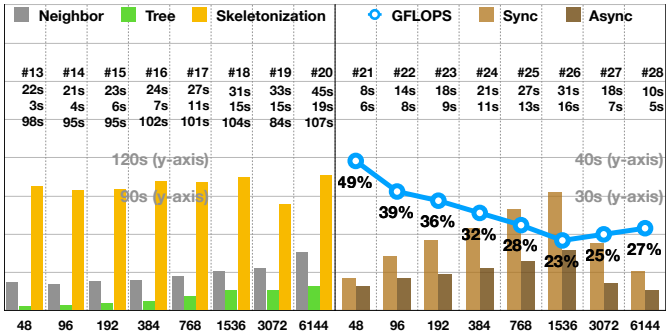

Fig. 4 ***Weak scaling (runtime in seconds vs cores)*** *of `MPI-GOFMM` applied to Gaussian kernel matrices generated synthetically with point clouds in 6-D.* ***Left:*** *Compression time and break down into three phases: neighbor search, tree creation, and skeletonization.* ***Right:*** *Matrix-matrix 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()`; the third row is the asynchronous mode using schemes described in §II-B, Table VI). 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`).*

Weak scaling: In Figure 4 (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 `MPI-GOFMM` parameters are adjusted to keep the 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 $|\texttt{Near}(\alpha)|$ may double in the worst case after the symmetry interaction list is built. As a result, the multiplication time

#	case	Sca LAPACK	STRUMPACK ϵ_2	STRUMPACK Compress	STRUMPACK Mult	MPI-GOFMM ϵ_2	MPI-GOFMM Compress	MPI-GOFMM Mult
29	K04	4.1	2E−1	224.6	7.5	2E−5	1.7	0.09
30	K07	4.0	4E−2	23.2	1.3	1E−4	1.5	0.04
31	K07	4.0	2E−2	15.9	3.3	1E−4	1.5	0.04
32	K11	4.1	1E−2	93.6	2.2	8E−6	1.5	0.04
33	K12	4.1	1E−1	222.6	4.3	7E−5	1.6	0.05
34	G03	2.8	1E−1	**33.6**	2.1	7E−5	**1.4**	0.04
35	H02	4.0	9E−2	**19.6**	5.3	7E−4	**11.7**	0.57

TABLE VII *Comparison between ScaLAPACK, STRUMPACK, and MPI-GOFMM for several different matrices. All results are on four Stampede-2 nodes. All the matrices are roughly 100K-by-100K. We report the time to do a dense matrix-matrix multiplication with ScaLAPACK (no approximation), and then the accuracy, compression time and multiplication time for STRUMPACK and MPI-GOFMM respectively. For STRUMPACK we used the lexicographic ordering for K04; for K07 we used both lexicographic (run # 30) and geometry-based permuted (run # 31). The permutation doesn't help that much, and we think this is because STRUMPACK does not include sparse corrections. Notice that with the exception of run #34, MPI-GOFMM (compression + evaluation) is faster than ScaLAPACK.*

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 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 does not scale that well and this is due to the Alltoallv(), which scales poorly 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 the 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 can maintain roughly 20% of peak performance (for the specific matrix) in the weak scaling experiments.

Comparison with other software: In Table VII, we compare MPI-GOFMM to ScaLAPACK and STRUMPACK. We compare with ScaLAPACK to benchmark MPI-GOFMM against a dense GEMM. The combined compression and evaluation should be faster than the GEMM. Indeed, for matrices of size ∼100k MPI-GOFMM achieves significant speedups ScaLAPACK for ∼2k right-hand sides for five digits of accuracy.

#	Cache	Nested	$\%N^2$	ϵ_2	Compress	Mult	GFLOPS
K01, N=262,144							
36	-	-	5.0%	2E-1	639.6	1005.6	2
37	x	-	5.0%	1E-1	1477.0	1.0	1108
K02, N=147,456							
38	-	-	1.0%	5E-5	3.9	0.4	324
39	x	-	1.0%	4E-5	4.0	0.3	434
C01, N=100,000							
40	-	-	$< 0.1\%$	5E-6	119.4	1.7	1762
41	-	x	$< 0.1\%$	4E-6	74.3	1.7	2998
42	x	x	$< 0.1\%$	6E-6	72.7	0.1	1018
C02, N=500,000							
43	-	-	$< 0.1\%$	3E-6	2359.6	21.1	1238
44	-	x	$< 0.1\%$	5E-6	1623.3	20.7	1828
45	x	x	$< 0.1\%$	4E-6	1790.0	0.4	1532
H03, N=266,496							
46	x	x	2.0%	1E-3	551.2	3.8	973

TABLE VIII *Out-of-core experiments.* All experiments were done on four Stampede-2 nodes (192 Skylake cores) for different matrices. We report timings for matrix-matrix multiplication with a N-by-512 random matrix. We report compression rate ($\%N^2$), that is the percentage of matrix samples required for compression and multiplication as a percentage of N^2 (dense exact matrix cost). We also report the relative error of our approximation ϵ_2, the wall-clock time (in seconds) for compression and multiplication, and GFLOPS **per node. For each matrix, we also consider three different setups defined by combinations of "caching" and "nesting". By "caching" we refer to storing (in RAM) an entry K_{ij} every time is computed/loaded from the disk. For the cases in which computation is required to obtain K_{ij} (cases **C01**, **C02**, and **H03**), nesting indicates that further parallelism was exploited in the K_{ij} evaluation.***

STRUMPACK [13] constructs an HSS representation in $\mathcal{O}(N \log N)$ work using a randomized ID explained in [15]. We present a comparison with STRUMPACK because besides MPI-GOFMM it is the only distributed-memory library that can compress dense matrices. STRUMPACK is more general than MPI-GOFMM: it supports sparse matrices and matrix-free peeling. For dense matrices, STRUMPACK differs from MPI-GOFMM in three main ways: (1) it does not permute the input matrix; (2) it is an HSS and not an FMM, i.e., it does not include sparse corrections; and (3) it compresses off-diagonal blocks differently. Although we allowed ranks up to 6400 on off-diagonals in STRUMPACK, it was still hard to achieve the target accuracy. It seems that sparse corrections (which MPI-GOFMM is using) are needed for some instances. Let us remark that STRUMPACK has not been designed for this problem, the main design and optimization have been targeted towards the factorization of sparse matrices. Additionally, in runs #29-30 we applied geometric permutation, a memory-intensive operation that is done beforehand and cannot be performed by STRUMPACK directly. In the #34 run (**H02** matrix related to neural networks), the multiplication is only 7× faster than ScaLAPACK, and the compression is quite expensive. In this case, using MPI-GOFMM would be advantageous only if we had even more right-hand sides or if we want to compute an approximate factorization of **H02**.

Out-of-core (OOC) experiments: For matrices that cannot be fully loaded into memory or efficiently computed on the fly, our method currently support two specific OOC formats implemented using Linux's mmap() (memory mapped) functionality. In Table VIII, we report results on dense matrices

mapped using mmap in all MPI processes. Notice that we tested MPI I/O, but it did not perform as well as mmap() due to the random access pattern. Dense matrices are fully mapped by all processes. Matrices **C01**, **C02**, and **H03**, require storing data, and the evaluating entry K_{ij} is expensive. In particular, these matrices can be formed by writing $K = I + JJ^T$. We only store J and we compute the matrix entry on the fly.

Further, to reduce I/O costs or/and redundant computation during the multiplication phase, we can optionally cache all the "visited" entries of K, assuming we have sufficient memory. Caching shifts $K(i, j)$ evaluation overhead to the compression phase, which partially overlaps with skeletonization. As a result, the one-time compression cost will be higher, but multiplication can be much faster. Other than caching, for **C01**, **C02**, and **H03**, we also allow exploiting nested parallelism in evaluating an entry K_{ij} or submatrix K_{IJ}. In all three cases, K_{ij} can be computed by GEMM_TN. To fully exploit the parallelism, we use the approach in [25] to systematically generate subtasks and temporary buffers for the reduction.

For **K02** (which is only 81GB), mmap performs almost as well as in-memory access (compare #29 to #34). As a result, caching in #39 does not significantly reduce the matrix multiplication time. The effect of caching stands out for **K01** (256GB) in #37, which delivers $1,000\times$ speedup comparing to #36. Notice that **K01** (256GB) is quite large and does not fit in the memory of a single node. As a result, the compression time reflects the I/O costs (mostly latency). We choose to cache all required submatrices (about 5% of the whole matrix) in #37, which shifts multiplication time to the compression phase. Notice that the compression time of #37 is lower than the sum of compression and multiplication cost in #36. This is because the caching process is partially overlapped with the skeletonization phase, which further provides higher parallelism to be exploited.

In the evaluation of K_{ij} for **C01** and **C02**, most of the time spent on evaluating K_{ij} (using GEMM_TN) and not on reading J. This can be observed from the achieved performance in terms of GFLOPS per node. #41 achieves higher GFLOPS than #40 because of the extra nested parallelism provided by the GEMM_TN subtasks. A 512-by-512 submatrix evaluation in #40 is encapsulated as a $512 \times 33K \times 512$ GEMM task. It takes roughly 0.2 seconds on a Skylake core running in near its peak performance. This huge task results in a longer critical path and increases the difficulty of scheduling. (The worst case scheduling scenario is more likely to happen with highly heterogeneous tasks). #41 splits the huge tasks into smaller $512 \times 512 \times 512$ subtasks, trading extra buffer space with parallelism. As a result, we can observe roughly $1.5\times$ performance gain in #41. With caching, #42 has much shorter evaluation time, but its GFLOPS drop without these computational intensive evaluations. For a larger case, matrix **C02** (132GB, $10\times$ bigger), we observe $12\times$ to $13\times$ longer multiplication time in #43 and #44. The extra runtime is contributed to the high mmap latency. For square submatrix evaluation, mmap takes roughly 10% of the evaluation time in #40 and #41, but its portion increases to 20% in #43 and #44. The mmap (I/O) latency may dominate the runtime subcolumn evaluations in the compression phase (mainly used in tree partition). As a result, we observe more performance degradation in the compression phase.

Finally, we present our results on compressing a Gauss-Newton Hessian related to a three-layer full connected perceptron. Each column of $\mathcal{J}$ requires a full (or partial) feed-forward step on the whole mini batch. Submatrix K_{IJ} is evaluated as a $\mathcal{J}(:, \mathtt{I})^T \mathcal{J}(:, \mathtt{J})$ if the whole Jacobian matrix is stored. Different from #35 where the whole system is stored, here #46 in Table VIII we only store the full Jacobian matrix and compute the entries on the fly just like covariance matrices. Although the compression cost is higher, this allows us to approximate a larger problem. In the evaluation phase with all required submatrices cached, our method is able to get 3-digit of accuracy in 3.8s using 4 MPI processes and reaching 23% of peak performance.

VI. Conclusions

MPI-GOFMM is quite capable of approximating a wide range of dense SPD matrices. Although we have not discussed heterogeneous architectures, we remark that the original GOFMM was ported on KNL, GPUs, and ARM architectures. Similar extensions for MPI-GOFMM should be relatively straightforward. However, the GOFMM acceleration was done only in the evaluation phase, not the compression [1].

In all, the compression phase remains a challenge. The pivoted QR eats up cycles, and the dependencies create synchronization bottlenecks. Perhaps one ought to consider alternative methods that have worse complexity on a sequential machine but scale better and can easier utilize heterogeneous architectures. Nevertheless, assuming that the evaluation of K_{ij} is fast, compression can be still quite efficient. For example, it only takes 150 seconds to compress a 67M-by-67M matrix. The I/O performance using mmap is somewhat disappointing and unpredictable—especially in light that MPI I/O was even slower. We believe that for practitioners is important to provide the OOC capability since access to a large number of nodes may be difficult.

We evaluated our algorithms on several datasets, including covariance matrices and neural-network Hessians. We believe we are the first to find that the Hessian of multi-layer fully connected feed-forward networks is hierarchically compressible. There is much more to be said and done about this, but this is beyond the scope of this paper.

Fast H-matrix factorization algorithms are our ultimate goal. Here we focused on building an $\mathcal{H}$-matrix approximation of an arbitrary SPD matrix. Our future work will deal with integrating distributed fast factorization algorithms into MPI-GOFMM to enable fast solvers and preconditioners.

Acknowledgment

This material is based upon work supported by AFOSR grants FA9550-12-10484; by NSF grant CCF-1725743; by the Technische Universität München;and by a gift from Qualcomm Foundation. Computing time on the Texas Advanced Computing Centers was provided by an allocation from TACC.

REFERENCES

[1] C. D. Yu, J. Levitt, S. Reiz, and G. Biros, "Geometry-oblivious FMM for compressing dense SPD matrices," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2017, p. 53.

[2] W. Hackbusch, *Hierarchical Matrices: Algorithms and Analysis*, 1st ed., ser. Springer Series in Computational Mathematics 49. Springer-Verlag Berlin Heidelberg, 2015.

[3] M. Bebendorf, *Hierarchical Matrices*. Springer, 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*, vol. 11, no. 4, pp. 273–291, 2008.

[5] M. Benzi, G. H. Golub, and J. Liesen, "Numerical solution of saddle point problems," *Acta numerica*, vol. 14, no. 1, pp. 1–137, 2005.

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

[7] R. H. Byrd, G. M. Chin, W. Neveitt, and J. Nocedal, "On the use of Stochastic Hessian Information in Optimization methods for Machine Learning," *SIAM Journal on Optimization*, vol. 21, no. 3, pp. 977–995, 2011.

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

[9] A. Gray and A. Moore, "N-body problems in statistical learning," *Advances in neural information processing systems*, pp. 521–527, 2001.

[10] Greengard, L., "Fast Algorithms For Classical Physics," *Science*, vol. 265, no. 5174, pp. 909–914, 1994.

[11] N. Cancedda, E. Gaussier, C. Goutte, and J. M. Renders, "Word sequence kernels," *Journal of Machine Learning Research*, vol. 3, pp. 1059–1082, Mar. 2003. [Online]. Available: http://dl.acm.org/citation.cfm?id=944919.944963

[12] R. I. Kondor and J. Lafferty, "Diffusion kernels on graphs and other discrete input spaces," in *ICML*, vol. 2, 2002, pp. 315–322.

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

[14] F.-H. Rouet, X. S. Li, P. Ghysels, and A. Napov, "A distributed-memory package for dense hierarchically semi-separable matrix computations using randomization," *ACM Transactions in Mathematical Software*, vol. 42, no. 4, pp. 27:1–27:35, Jun. 2016. [Online]. Available: http://doi.acm.org/10.1145/2930660

[15] P.-G. Martinsson, "Compressing rank-structured matrices via randomized sampling," *SIAM Journal on Scientific Computing*, vol. 38, no. 4, pp. A1959–A1986, 2016.

[16] W. B. March, B. Xiao, C. D. Yu, and G. Biros, "ASKIT: An Efficient, Parallel Library for High-Dimensional Kernel Summations," *SIAM Journal on Scientific Computing*, vol. 38, no. 5, pp. S720–S749, 2016. [Online]. Available: http://dx.doi.org/10.1137/15M1026468

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

[18] W. B. March, B. Xiao, C. Yu, and G. Biros, "An algebraic parallel treecode in arbitrary dimensions," in *Proceedings of IPDPS 2015*, ser. 29th IEEE International Parallel and Distributed Computing Symposium, Hyderabad, India, May 2015. [Online]. Available: http://dx.doi.org/10.1109/IPDPS.2015.86

[19] J. Poulson, B. Marker, R. A. Van de Geijn, J. R. Hammond, and N. A. Romero, "Elemental: A new framework for distributed memory dense matrix computations," *ACM Transactions on Mathematical Software (TOMS)*, vol. 39, no. 2, p. 13, 2013.

[20] B. Xiao and G. Biros, "Parallel algorithms for nearest neighbor search problems in high dimensions," *SIAM Journal on Scientific Computing*, vol. 38, no. 5, pp. S667–S699, 2016.

[21] T. Hoefler, C. Siebert, and A. Lumsdaine, "Scalable communication protocols for dynamic sparse data exchange," *ACM Sigplan Notices*, vol. 45, no. 5, pp. 159–168, 2010.

[22] W. B. March, B. Xiao, S. Tharakan, C. D. Yu, and G. Biros, "A kernel-independent FMM in general dimensions," in *Proceedings of SC15*, ser. The SCxy Conference series. Austin, Texas: ACM/IEEE, November 2015. [Online]. Available: http://dx.doi.org/10.1145/2807591.2807647

[23] ——, "Robust treecode approximation for kernel machines," in *Proceedings of the 21st ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, Sydney, Australia, August 2015, pp. 1–10. [Online]. Available: http://dx.doi.org/10.1145/2783258.2783272

[24] P.-G. Martinsson, G. Quintana Orti, N. Heavner, and R. van de Geijn, "Householder QR factorization with randomization for column pivoting (HQRRP)," *SIAM Journal on Scientific Computing*, vol. 39, no. 2, pp. C96–C115, 2017.

[25] E. Chan, F. G. Van Zee, P. Bientinesi, E. S. Quintana-Orti, G. Quintana-Orti, and R. van de Geijn, "SuperMatrix: a multithreaded runtime scheduling system for algorithms-by-blocks," in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, ser. PPoPP '08. New York, NY, USA: ACM, 2008, pp. 123–132. [Online]. Available: http://doi.acm.org/10.1145/1345206.1345227

```
class KernelMatrix<T,TPARAM> : VirtualMatrix<T>
virtual operator T K(i,j) return 𝒦(𝒳(i),𝒳(j))
virtual function SendIndices(ids,dest,comm)
   Send(ids,dest,comm)
   Send(𝒳' = 𝒳(ids),dest,comm)
virtual function RecvIndices(ids,src,comm)
   Recv(ids,src,comm)
   Recv(𝒳',src,comm)
   𝒳.Insert(ids,𝒳')
virtual function BcastIndices(ids,root,comm)
   Bcast(ids,root,comm)
   Bcast(𝒳' = 𝒳(ids),root,comm)
   𝒳.Insert(ids,𝒳')
virtual function RequestIndices(ids[1:p])
   Alltoallv(ids[1:p])
   for i=1:p then 𝒳'[i]= 𝒳(ids[i])
   Alltoallv(𝒳'[1:p])
   for i=1:p then 𝒳.Insert(ids[i],𝒳'[i])
```

TABLE IX *MPI-GOFMM API to define the evaluation of K_{ij}: Five virtual methods define the API for MPI-GOFMM. These functions need to be implemented by the user. Only the first one is required. The remaining four are optional to support data repartitioning (all four must be implemented by the user if repartitioning is to take place). The implementation above is an example of how these functions would be implemented for the case of a Kernel Matrix.*

VII. Appendix

Here we give details on the MPI-GOFMM API, the runtime systems, the distributed memory algorithms, and the differences between GOFMM and MPI-GOFMM.

A. The matrix definition API

The MPI-GOFMM API requires a user-defined function K(i,j) that returns entry K_{ij} given matrix indices i and j. One complication is that the user-provided evaluation K_{ij} may require MPI communication. We assume that such evaluation requires two memory blocks, one for index i and one for j. We further assume that the evaluation of K_{ij} can take place in a single MPI process. However, the memory blocks for i and j may reside in different MPI processes and thus evaluating K_{ij} may require MPI communication. This communication can add up and slow down the overall computation—sometimes to unacceptable levels. For this reason, we provide an (optional) API so that the data can be repartitioned (redistributed) to be favorable to the MPI-GOFMM algorithm. This API consists of four methods: SendIndices(), RecvIndices(), BcastIndices(), RequestIndices(). These allow MPI-GOFMM guided, user-implemented, data repartitioning that are invoked by MPI-GOFMM in order to minimize the communication costs associated with evaluating K_{ij}.

The only required method is the user-defined K(i,j), which inherits the abstract base class VirtualMatrix<T>. Operator T K(i,j), defined as a pure virtual function, is instantiated to return entry K_{ij} with type T. In Table IX, we give an example in which K is a kernel matrix and K(i,j) is defined by $\mathcal{K}(x_i, x_j)$, where the points x_i and x_j have type TPARAM. For a dense column-major matrix (defined in-memory or using mmap from the disk), K(i,j) is simply instantiated to return K[i+j*N].

The remaining four methods are used to redistribute data required to evaluate K_{ij}. Again, taking a kernel matrix as an example, we can repartition the point dataset in order to minimize communication. Functions SendIndices(), RecvIndices() use only point-to-point communication, whereas BcastIndices(), RequestIndices() involve collective communication. Except for RequestIndices, all other three methods can take arbitrary communicators. RequestIndices uses the same communicator that was assigned to invoke MPI-GOFMM. These methods are called automatically during the compression phase and provided all required arguments for users to perform the communication. If there is no communication required, these methods will still be called but return immediately.

For the kernel matrix example, we assume that the points $\mathcal{X}$ are distributed in the [CYC] (1D cyclic) order. That is, MPI process q only holds

$$\mathcal{X}\,[\mathrm{CYC}] = \{x_i | \mathrm{mod(i,p)} = \mathrm{q}\}. \tag{10}$$

During the repartitioning, we insert x_j to $\mathcal{X}$ for each received $x_j \notin \mathcal{X}$[CYC]. In MPI-GOFMM, index matching is implemented with a hashmap that records the offset to the dense matrix $\mathcal{X}$. For an unseen x_j, index j is first inserted into the hashmap as a pair of j and $\mathcal{X}$.col(). Then x_j is inserted from the back of $\mathcal{X}$.

In practice, SendIndices and RecvIndices appear in tasks DistSplit and DistID where indices and skeletons are exchanged between two partner ranks (q and $\mathrm{mod(q + p/2, p)}$) in the same communicator. BcastIndices is used in task DistSplit to broadcast the centroid and the two farthest points (indices). Finally, RequestIndices is used in Algorithm II.1 to handle all-to-all repartitioning introduced by the neighbors and interaction lists.

B. Dependency Analysis

MPI-GOFMM performs a runtime dependency analysis to exploit dynamic data dependencies. In Table X, we take tasks DistID and DistS2N as examples to illustrate how MPI-GOFMM instantiates tasks and performs dependency analysis. A normal task of MPI-GOFMM must inherit the abstract base class Task. For example, we instantiate DistID with parameter type DistNode. There are three member functions must be instantiated: the constructor, DependencyAnalysis(), and Execute(). The first two functions are called during the symbolic traversal (in this case, a post-order traversal) to provide the parameter (tree node), estimate the execution cost (according to its flops and mops), and perform dependency analysis. The last function Execute(), performing the actual computation, is invoked by a worker (thread) assigned by the runtime system when all the dependencies are satisfied.

Most of the MPI-GOFMM tasks take tree nodes as parameters. As a result, we choose to annotate the read/write activities in the granularity of tree nodes. Tree nodes (both shared memory Node and distributed memory DistNode) defined in Table **??** inherit class ReadWrite (defined in Table **??**) to record the **R**ead/**W**rite activities. Each task must annotate these activities on each node by calling function ReadWrite::DependencyAnalysis() during the symbolic execution (a symbolic tree traversal in MPI-GOFMM).

```
class Task
deque<Task*> in, out;
Lock lock;
TaskStatus status=ALLOCATED;
volatile int n_dependencies_remaining=0;
string name;
float cost = 1.0;
virtual void DependencyAnalysis()=0;
virtual void Execute(Worker *worker)=0;
class DistID<DistNode,T> : Task
DistNode *node;
function DistID(DistNode *node)
   this->node = node
   this->name = string("DistID")
   this->cost = estimated from flops and mops
   DependencyAnalysis()
function DependencyAnalysis()
   node->DependenOnChildren(this)
function Execute(Worker* worker)
   DistID<DistNode,T>(node) defined in Table III
class DistS2N<DistNode,T> : Task
DistNode *node;
function DistS2N(DistNode *node)
   this->node = node
   this->name = string("DistS2N")
   this->cost = estimated from flops and mops
   DependencyAnalysis()
function DependencyAnalysis()
   node->DependenOnParent(this)
function Execute(Worker* worker)
   DistS2N<DistNode,T>(node) defined in Table V
```

TABLE X *The pseudo code of task `DistID` and `DistS2N`. Tasks `DistID` are instantiated and created during the post-order symbolic traversal of Algorithm II.1. Tasks `DistS2N` are instantiated and created during the preorder symbolic traversal of Algorithm II.2.*

For example, in Table X task `DistIDTask` will read skeletons $\widetilde{\mathtt{l}}$, $\widetilde{\mathtt{r}}$ and write $\widetilde{\alpha}$. This can be annotated by `DistNode::DependOnChildren()`, which encodes activities on node α and its children `l` and `r`. Similarly, task `DistS2N` will read $\widetilde{w}_\alpha$ and write $\widetilde{w}_\mathtt{l}$, $\widetilde{w}_\mathtt{r}$, and the activities can be annotated by `DistNode::DependOnParent()` during the preorder symbolic traversal in Algorithm II.2. With these records, dependencies between tasks can be resolved by tracking the **R**ead/**W**rite sets (inherited from class `ReadWrite`) for each node.

Following [25], we present class `ReadWrite` in Table **??** to perform runtime dependency analysis. When a task performs a read operation on a tree node, the pointer of the task is inserted to the `read` queue of this tree node (the `read` queue is inherited from class `ReadWrite`). Then for every task that previously modified (or *"wrote"*) the tree node and still in its `write` queue, we add a **Read-After-Write (RAW)** dependency (a direct edge in the dependency graph) with the new task (named as `tar` in Table X). This is done by calling function `DependencyAdd(src,tar)`, which insert task `tar` to the `out` queue of task `src` and insert task `src` to the `in` queue of task `tar`. Similarly, tasks that write the node create **Write-After-Read (WAR)** dependencies for every task in the `read` queue. After all **WAR** dependencies are generated, both `read` and `write` queues are flushed, and the last write task is inserted to the write queue. Independent tasks (with no incoming edges after the dependency analysis) are detected and directly enqueued to one of the ready queue by function `Task::TryEnqueue()`. These independent tasks are the source nodes of the dependency graph.

```
class Node : ReadWrite
Node* parent, lchild, rchild;
function DependOnChildren(Task* tar)
   if lchild then lchild->DependencyAnalysis(R,tar)
   if rchild then rchild->DependencyAnalysis(R,tar)
   this->DependencyAnalysis(RW,tar)
   tar->TryEnqueue()
function DependOnParent(Task* tar)
   this->DependencyAnalysis(R,tar)
   if lchild then lchild->DependencyAnalysis(RW,tar)
   if rchild then rchild->DependencyAnalysis(RW,tar)
   tar->TryEnqueue()
class DistNode : Node
DistNode *child;
Comm comm;
function DependOnChildren(Task* tar)
   if CommSize(comm)<2 then
      lchild->DependencyAnalysis(R,tar)
      rchild->DependencyAnalysis(R,tar)
   else child->DependencyAnalysis(R,tar)
   this->DependencyAnalysis(RW,tar)
   tar->TryEnqueue()
function DependOnParent(Task* tar)
   this->DependencyAnalysis(RW,tar)
   if CommSize(comm)<2 then
      lchild->DependencyAnalysis(RW,tar)
      rchild->DependencyAnalysis(RW,tar)
   else child->DependencyAnalysis(RW,tar)
   tar->TryEnqueue()
```

TABLE XI *Definition of classes `Node` and `DistNode`. A shared memory tree node has type `Node`, which inherits class `ReadWrite` to perform dependency analysis. Class `Node` also carries pointers `parent`, `lchild`, and `rchild` to access factors or indices carried by the parent and children. A distributed memory tree node `DistNode` inherits `Node` and further carries a local communicator and a unique pointer `child` to access the children.*

```
class ReadWrite
deque<Task*> read, write;
function DependencyAnalysis([R/W/RW],Task* tar)
   if activity is R of RW then
      read.push_back(tar)
      foreach src : write do DependencyAdd(src,tar)
   if activity is W of RW then
      foreach src : read do DependencyAdd(src,tar)
      DependencyCleanUp()
      write.push_back(tar)
function DependencyCleanUp()
   read.clear()
   write.clear()
function DependencyAdd(Task* src, Task* tar)
   src->lock.Acquire()
   src->out.push_back(tar)
   src->lock.Release()
   tar->lock.Acquire()
   tar->in.push_back(src)
   if src->status!=DONE then
      tar->n_dependencies_remaining ++
   tar->lock.Release()
```

TABLE XII *Definition of class `ReadWrite`: Each `MPI-GOFMM` tree node inherits class `ReadWrite` to track all **R**ead/**W**rite activities. Member function `DependencyAnalysis()` shows how we encode read/write activities to data dependencies between tasks. Member function `DependencyCleanUp()` flushes all activities such that the object can perform another round of analysis.*

C. `GOFMM` and `MPI-GOFMM` Comparison

In Algorithm VII.1 and Algorithm VII.2, we summarize

Algorithm VII.1 Side-by-side comparison between GOFMM (left) and MPI-GOFMM ***compression*** (right)

1: [Preorder] **for each node** α **do** Split(α)	11: [Preorder] **foreach node** α **do** DistSplit(α)
2: $\mathcal{N}$ = AllNearestNeighbors(K)	12: $\mathcal{N}$[*,CYC] = DistAllNearestNeighbors(K)
3: NOP	13: $\mathcal{N}$[*,IDS] = $\mathcal{N}$[*,CYC]
4: [Postorder] **foreach node** α **do** ID(α)	14: [Postorder] **foreach node** α **do** DistID(α)
5: [Anyorder] **foreach leaf** α **do** FindNear(α)	15: [Anyorder] **foreach leaf** α **do** FindNear(α)
6: **foreach** α **and** $\beta \in$ Near(α) **do** Near(β)$\cap = \alpha$	16: Symmetrize(Near)
7: NOP	17: IDS2LET(Near)
8: [Postorder] **foreach** α **do** MergeFar(α)	18: [Postorder] **foreach** α **do** DistMergeFar(α)
9: **foreach** α **and** $\beta \in$ Far(α) **do** Far(β)$\cap = \alpha$	19: Symmetrize(Far)
10: NOP	20: IDS2LET(Near)

Algorithm VII.2 Side-by-side comparison between GOFMM (left) and MPI-GOFMM ***evaluation*** (right)

1: NOP	7: (Async)Redistribute(Near)
2: [Anyorder] **foreach leaf** α **do** L2L(α)	8: [Anyorder] **foreach leaf** α **do** DistL2L(α)
3: [Postorder] **foreach node** α **do** N2S(α)	9: [Postorder] **foreach node** α **do** DistN2S(α)
4: NOP	10: (Async)Redistribute(far)
5: [AnyOrder] **foreach node** α **do** S2S(α)	11: [AnyOrder] **foreach node** α **do** DistS2S(α)
6: [Preorder] **foreach node** α **do** S2N(α)	12: [Preorder] **foreach node** α **do** DistS2N(α)

compression and ***evaluation*** phase of GOFMM (left) and MPI-GOFMM (right) to highlight their differences. All components and tasks can be found in Table II, Table I, Table V, Table III, Table IV, and Table VI.

Traversals: MPI-GOFMM uses a complete binary tree data structure to cluster the row and column index sets of K. If we're using p MPI processes, the tree from root to level $\log p - 1$ is shared among the p processes the remaining p subtrees are local in each MPI process. Every MPI process locally stores information for $\log p$ tree nodes from the distributed part of the tree in vector<DistNode*>. The local tree nodes (belong to the local subtree) are stored as vector<Node*> in breadth-first traversal order. A top-down traversal will first visit all $\log p$ distributed tree nodes and then visit all the local nodes in the lexicographic order. A bottom-up traversal reverses this operation. A local node has both lchild and rchild pointers to it children DistNode inherits Node and stores a local MPI communicator.

Compression: In Algorithm VII.1, we state the original shared memory version of GOFMM on the left and the distributed memory version on the right. Except for three additional redistributing steps at line 13, line 17, and line 20, the rest of the execution flows are almost the same. The tree is traversed in the same order, but the involving tasks are different. For example, instead of calling the shared memory splitter Split in Table I, MPI-GOFMM calls the distributed memory splitter DistSplit in Table III. Most of the changes happen in the distributed tree nodes above level-$\log p$. The operations performed on the local tree nodes below level-$\log p$ remain the same. This observation can also be applied to other distributed tasks including DistID and DistMergeFar.

The main difference between the compression phase of GOFMM and MPI-GOFMM is the all-to-all repartitioning performed at line 13, line 17, and line 20. While neighbors $\mathcal{N}$ are computed and stored in parallel, an additional step at line 13 is required to repartition the neighbor pairs from 1D cyclic [CYC] to the MPI-GOFMM tree order [IDS]. This is because the neighbors are approximated with a tree-based algorithm. Also, notice that interaction lists of MPI-GOFMM can contain remote tree nodes stored on other processes. As a result, additional Alltoallv operations are required to symmetrize both near and far interaction lists using function Symmetrize(). Essential matrix indices and corresponding factors are redistributed by function IDS2LET() that constructs the local essential tree (the [LET] order).

Evaluation: The side-by-side comparison of the evaluation phase is presented in Algorithm VII.2. The main difference is the all-to-all redistribution performed at line 7 and line 10. These two steps redistribute weights w and skeleton weights $\widetilde{w}$ from the MPI-GOFMM tree order [IDS] to the local essential order [LET]. Different from shared memory tasks N2S and S2N, distributed memory tasks DistN2S and DistS2N handle additional communication using the local communicator for distributed tree nodes above level-$\log p$. Different from tasks S2S and L2L, tasks DistS2S and DistL2L are split into p subtasks to facilitate asynchronous communication (AsyncRedistribute()). While each subtask only depends on one message from a remote process, the whole evaluation phase is free from global synchronization barriers.

VIII. Artifact

A. Abstract

The artifact description comprises the source code, datasets, and installation instruction on a GitHub repository that can be used to reproduce results for this SC'18 paper. We also provide all hardware and software configuration in §VIII-B.

B. Description

`MPI-GOFMM` approximates arbitrary SPD matrices using a distributed memory algebraic `FMM` with geometry-oblivious techniques. The only requirement is the capability to evaluate any matrix element K_{ij} in the distributed environment in nearly $\mathcal{O}(1)$ time. Note that we do not require all the entries, only the ability to evaluate them. The $K(i,j)$ API is defined in VII-A.

- **Program:** `MPI-GOFMM` is developed in `C++14` employing a self-contained runtime system (using `OpenMP` threads) for shared-memory parallelism and `MPI` for distributed-memory parallelism (`MPI_THREAD_MULTIPLE` is required).
- **Hardware:** We conducted experiments on TACC (Texas Advanced Computing Center) Stampede2 (48 cores on two sockets, i.e. 24 cores/socket, 2.1GHz (1.4-3.7GHz), Intel Xeon Platinum 8160 "Skylake").
- **Compilation:** `MPI-GOFMM` and all software (including `GOFMM`, `ScaLapack`, and `STRUMPACK`) are compiled with `intel-17 -O3`. All `BLAS/LAPACK` routines use Intel MKL (`-mkl=parallel`). `OpenMP` uses `OMP_PROC_BIND=spread`.
- **Datasets:** We use datasets and the `MATLAB` scripts (`datasets/spdmatrices.m`) provided by `GOFMM` to generate **K01**, **K02**, **K04**, **K07**, **K11**, **K12**, and **G03** matrices. Neural-network related matrices and covariance matrices can be can be generated using `python numpy` scripts. (`/path_to_repo/datasets/*.(m/py)`, provided in the repository). We provide the sources in §IV (i.e. URLs) for all real-world datasets we use in the experiments (e.g. **MNIST60K** and **SUSY5M**).
- **Output:** The executable of `MPI-GOFMM` reports execution time, total `FLOPS` of the compression and evaluation phases, and accuracy ϵ_2 of the first 10 entries and averaged over 100 sampled entries.
- **Experiment workflow:** `git clone` repository; generate datasets with the Matlab/Python scripts; and run test scripts.

How delivered: https://github.com/ChenhanYu/hmlp. To reproduce results see `/artifact/sc18gofmm`. The software comprises code, build, and evaluation instructions, and is provided under GPL-3.0 license.

Hardware dependencies: To reproduce timings one should use the TACC Stampede2 system. Notice that we report absolute `GFLOPS` (Giga Floating Points Operations per Second) and the ratio to the theoretical peak performance in the paper. To approximately reproduce the same results on a different environment, one should seek a platform that has similar capabilities. The theoretical peak in single precision is 4,300 `GFLOPS` per `MPI` node.

Software dependencies: Compilation of `MPI-GOFMM` requires `MPI-3.0`, a generic parallel `C/C++` compilers that support `c++14` features (`-std=c++14`) and `OpenMP` (`-fopenmp` for GNU or `-qopenmp` for Intel compilers). `MPI-GOFMM` also requires the full functionality of `BLAS` and `LAPACK` routines.

Linux and Unix: If environment variables `CC`, `CXX`, `MKLROOT` (or `OPENBLASROOT`) are all set up properly, then users only have to set `USE_MPI=true` in the `set_env.sh` before the compilation. Otherwise, you must export the proper values for all variables: the *required* region of `set_env.sh` first. Compiler vendor is detected by environment variables `CC` and `CXX` and `cmake` is used for compilation.

```
export USE_MPI = true
export CC      = ${CC}  % icc  or gcc
export CXX     = ${CXX} % icpc or g++
```

`cmake` will automatically look for proper `MPI` support. Upon failure in auto detection, users must set up the related flags

```
${MPI_CXX_INCLUDE_FLAGS}
${MPI_CXX_COMPILE_FLAGS}
${MPI_CXX_LIBRARIES}
${MPI_LINK_FLAGS}
```

manually in `CMakeLists.txt`.

C. Installation

Given the repository URL, you should be able to clone the `release` branch. The first step is to

```
source set_env.sh
```

to set up targeting architecture, compilers, environment variables, `BLAS/LAPACK` library. If no error occurs, use the following instructions for standard `cmake` compilation.

```
mkdir build
cd build
cmake ..
make install
```

Execution: Once the compilation (`make install`) is completed, directory `/build/bin` should contain at least the following files and directories:

```
artifact_sc18gofmm.x % executable
/datasets            % MATLAB/python
sbatch_artifact_sc18gofmm_default.sh
run_artifact_sc18gofmm_default.sh
```

Executing `run_artifact_sc18gofmm_default.sh` will invoke the executable (`artifact_sc18gofmm.x`) on a small 5K-by-5K matrix. Notice that if you need to submit a task using `slrum`, then use the `sbatch` file we provide. By default, we use TACC setting, and the executable is run by `ibrun tacc_affinity`. To execute with custom settings, change the corresponding variables in the script and edit other options in the bash script appropriately. To reproduce other experiments in this paper, you must first generate datasets using the MATLAB/python scripts we provide in `/datasets`.