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

\[
\tilde{K} = D + S + UV,
\]

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 \(\tilde{K}\) requires \(O(N \log N)\) 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 \(K_{i,j} = K(x_i, x_j)\), where \(x_i\) and \(x_j\) are points in \(D\) dimensions and \(K\) is a kernel function, for example a Gaussian with bandwidth \(h\): \(\exp(-1/2h^{-2}\|x_i - x_j\|^2)\). For such problems, \(H\)-matrix methods (like \(N\)-body methods) use clustering methods (typically trees) to define compressible blocks of \(K\) and appropriately permute it. However, such an 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 \S 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 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 H-matrix multiplication algorithm that can lead up to $2x$ improvement over a synchronous scheme ([II-B]) by avoiding expensive alltoally 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 VI-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 $O(1)$ time arbitrary matrix entries, which may not be possible in general. We can guarantee that $K$ 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 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 come close is STRUMPACK [13], [14], [15], which constructs an approximation $\tilde{K}$ 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 $O(N^2)$ work for compressing a dense SPD matrix, and $O(N)$ 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 H-matrix approximation $\tilde{K}$ such that

$$u = Kw \approx \tilde{K}w, \quad w \in \mathbb{R}^{N \times r},$$

(2)

can be computed with $O(N \log N)$ work and $O(N/p \log N)$ 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 VI.1 and VI.2 in Appendix VI-C.

### 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 $\tilde{K}$ recursively using a binary tree such that

$$\tilde{K}_{\alpha \alpha} = \begin{bmatrix} K_{ll} & 0 \\ 0 & \tilde{K}_{rr} \end{bmatrix} + \begin{bmatrix} 0 & S_{lr} \\ S_{rl} & 0 \end{bmatrix} + \begin{bmatrix} 0 & UV_{rl} \\ VU_{rl} & 0 \end{bmatrix},$$

(3)

where $l$ and $r$ are left and right children of treenode $\alpha$. Each node $\alpha$ contains a set of column indices (or row indices due to symmetry) and the two children evenly split this set such that $\alpha = l \cup r$. For example, task $\text{Split}(\alpha)$ 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 ($UV_{rl}$) and sparse ($S_{lr}$) 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 $O(1)$ using raw entries of a given SPD matrix $K$. For example, the Gram-$l^p$ 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.

**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

\begin{table}[h]
\centering
\begin{tabular}{|c|c|}
\hline
Task & Operations \\
\hline
\text{Split}(\alpha) & \begin{align*}
    p &= \text{argmax}\{(d_{ij} \mid i \in \alpha)\} \\
    q &= \text{argmax}\{(d_{ij} \mid j \in \alpha)\} \\
    \{l, r\} &= \text{medianSplit}(\{d_{ij} \mid i \in \alpha\})
\end{align*} \\
\hline
\text{ID}(\alpha) & \begin{align*}
    \text{if } \text{isleaf}(\alpha) & \text{ then } \text{Near}(\alpha) = \{\text{MortonID}(i) \mid i \in \text{Near}(\alpha)\} \\
    \text{else } \text{Near}(\alpha) &= \text{SampleFrom}((\text{Near}(\alpha)) \setminus \text{Near}(\alpha)); \\
    (\tilde{a}, P_{ij}) &= \text{GEPS}3(K_{ij}, K_{ij}^{-1}) \\
    \text{Far}(\alpha) &= \{\text{Near}(\alpha) \cap \text{Far}(\alpha)\}
\end{align*} \\
\hline
\text{FindNear}(\beta, \alpha) & \begin{align*}
    \text{if } \beta \text{ is parent of any node in } \text{Near}(\alpha) & \text{ then } \text{Far}(\beta) = \text{Far}(\beta) \cup \text{Far}(\alpha) \\
    \text{else if } \text{MortonID}(\beta) > \text{MortonID}(\alpha) & \text{ then } \text{Far}(\beta) = \text{Far}(\beta) \cup \text{Far}(\alpha)
\end{align*} \\
\hline
\text{MergeFar}(\alpha) & \begin{align*}
    \text{if } \text{isleaf}(\alpha) & \text{ then } \text{Far}(\alpha) = \text{Far}(\alpha) \cup \text{Far}(\alpha) \\
    \text{else } & \text{Far}(\alpha) = \text{MergeFar}(\text{Far}(\alpha))
\end{align*}
\hline
\end{tabular}
\caption{Tasks invoked in the compression phase of GOFMM.}
\end{table}

\(^1\)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.
k-nearest neighbors \( N \) by an approximate randomized spatial tree using the geometry-oblivious distance of \( K \).

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

\[
K_{I\alpha} \approx K_{I\tilde{\alpha}}P_{\tilde{\alpha}\beta}, \quad \text{or} \quad K_{I[\tilde{\alpha}]} \approx K_{I\tilde{\alpha}}P_{\tilde{\alpha}[\tilde{\alpha}]}
\]

if node \( \alpha \) has children 1 and \( r \). Skeleton \( \tilde{\alpha} \) is computed by rank-revealing QR factorization (GEQP3) to find columns \( \tilde{\alpha} \) 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 \( O(N^2) \) 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 \( \tilde{\alpha} \) has been determined, \( P_{\tilde{\alpha}\alpha} \) and \( P_{\tilde{\alpha}[\tilde{\alpha}]r} \) are computed using TRSM using the upper triangular matrix computed from GEQP3. For non-leaf nodes we greedily select their skeletons \( \tilde{\alpha} \) from the node’s children’s skeletons \( 1 \cup \tilde{\alpha} \). As a result, coefficient matrices have the following recursive relation (known as the **telescoping** relation)

\[
P_{\tilde{\alpha}\alpha} = P_{\tilde{\alpha}[\tilde{\alpha}]r} \begin{bmatrix} P_{11} \\ P_{rr} \end{bmatrix}.
\]

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 \( P_{\tilde{\alpha}\alpha}K_{\tilde{\alpha}\beta}P_{\beta}\beta \).

**Evaluation:** The four tasks in Table II compute the matvec of \( K \). (See Appendix VI-C, and Algorithm VI.2 for more details.) Task L2L computes direct matvecs (\( S_{12} \) and \( S_{11} \) in Equation 3) stored in the near interaction lists \( \text{Near}(\alpha) \). These lists are defined using the nearest-neighbor information \( N \) (computed by task \( \text{FindNear}(\alpha) \) in Table I). Tasks N2S, S2S, and S2N compute all approximate matvecs (\( U_{ij}V_{ij} \) and \( U_{i}V_{i} \)) in far interaction lists \( \text{Far}(\beta, \alpha) \) constructed with \( \text{FindFar}(\beta, \alpha) \), and \( \text{MergeFar}(\alpha) \) \[\text{II-B}]. We discuss how these lists are constructed in \[\text{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 VI-B for a discussion.

**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 VI-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

### Table II

<table>
<thead>
<tr>
<th>Task</th>
<th>Operations</th>
<th>FLOPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>N2S(( \alpha ))</td>
<td>if ( \alpha \text{is leaf} ) then ( W_{\alpha\alpha} = P_{\alpha\alpha}W_{\alpha\alpha} ) else ( W_{\alpha\alpha} = P_{\alpha\alpha}[\tilde{\alpha}] )</td>
<td>( 2m_{\alpha\alpha} )</td>
</tr>
<tr>
<td>S2S(( \beta ))</td>
<td>( u_{\beta} = \sum_{x \in \text{Far}(\beta, \alpha)} K_{\beta\alpha} W_{\alpha\beta} )</td>
<td>( 2s_{\alpha\beta}r_{\alpha\beta} )</td>
</tr>
<tr>
<td>S2N(( \beta ))</td>
<td>if ( \alpha \text{is leaf} ) then ( W_{\beta\beta} = P_{\beta\beta}[\tilde{\beta}] ) else ( W_{\beta\beta} = P_{\tilde{\beta}\beta}[\tilde{\beta}] )</td>
<td>( 2s_{\beta\beta}r_{\beta\beta} )</td>
</tr>
<tr>
<td>L2L(( \beta ))</td>
<td>( W_{\beta\beta} = \sum_{x \in \text{Near}(\beta)} K_{\beta\alpha} W_{\alpha\beta} )</td>
<td>( 2m_{\beta\beta}r_{\beta\beta} )</td>
</tr>
</tbody>
</table>

**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.
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}\text{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{c}] using "geometry-oblivious" distance, where c can be left (l) or right child (r). Indices 1 are kept by processes with rank(α) < \frac{1}{2}\text{size}(α), and r are kept by the other group of processes. l and \bar{r} are exchanged between "partner" processes (with rank dest). We continue using recursion 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 St and S\bar{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 α is only computed on rank 0 for distributed tree nodes above level-log(p). This requires gathering right child’s skeleton β 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 ∈ \mathcal{N}[*\{α\}], we add MortonID(i) (the same as the node MortonID of its belonged leaf node) to Near(α) such that

\[ \text{Near}(α) = \{\text{MortonID}(i) | i ∈ \mathcal{N}[*\{α\}]\}. \] (6)

Notice that |Near(α)| determines the number of direct entry evaluations (sparsity of Sx and S\bar{r}). 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)

```python
for each node \(\alpha\) and \(\beta\) \(\in\) List\(=\)Near/Far do
  
  if \(\alpha\) \(\neq\) \(\beta\) then 
    
    // Symmetrizing
    
    if size\(=\)Near then send \(\alpha\) to \(\beta\), \(\beta\) to \(\alpha\)
  
  if \(\alpha\) \(\neq\) \(\beta\) then 
    
    // Redistribution
    
    if size\(=\)Near then receive \(\beta\) from \(\alpha\)
  
end
```

### Task DistN2S(\(\alpha\)) : N2S(\(\alpha\))

```python
if size\(=\)Near then return N2S(\(\alpha\))
else 
  
  // Above level-log(p) of the tree
  
  case rank\(=\)0 then 
    
    Recv\(=\)Near(\(\bar{w}_0\)) : N2S(\(\alpha\))
  
  case rank\(=\)\(\frac{1}{2}\)size\(=\)Near then 
    
    Send\(=\)Near(\(\bar{w}_0\))
  
end
```

### Task DistS2N(\(\alpha\)) : S2N(\(\alpha\))

```python
if size\(=\)Near then return S2N(\(\alpha\))
else 
  
  // Above level-log(p) of the tree
  
  case rank\(=\)0 then 
    
    [AnyOrder] Send\(=\)Near(\(\bar{w}_0\))
  
  case rank\(=\)\(\frac{1}{2}\)size\(=\)Near then 
    
    Recv\(=\)Near(\(\bar{w}_0\))
  
end
```

### Task DistL2L(\(\alpha\))

```python
Task DistL2L(\(\alpha\))
```

### Task DistS2S(\(\alpha\))

```python
Task DistS2S(\(\alpha\))
```

### Table IV: All-to-all communication for symmetrizing and redistributing indices and data from [IDS] to [LET] order.

<table>
<thead>
<tr>
<th>Task</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DistEvaluate</td>
<td>Evaluate weights in the evaluation phase.</td>
</tr>
<tr>
<td>DistL2L</td>
<td>Local essential tree (LET): Reduce sends from leaf (\alpha) to (\alpha) to all other (\alpha)</td>
</tr>
<tr>
<td>DistS2S</td>
<td>Local essential tree (LET): Reduce sends from leaf (\alpha) to (\alpha) to all other (\alpha)</td>
</tr>
</tbody>
</table>

### Algorithm II.2 u[IDS,\(\ast\)]=DistEvaluate(w[IDS,\(\ast\)])

1. DistEvaluate(Near)
2. [Anyorder] for each leaf \(\alpha\) do DistL2L(\(\alpha\))
3. [Postorder] for each node \(\alpha\) do DistN2S(\(\alpha\))
4. DistEvaluate(Far)
5. [AnyOrder] for each node \(\alpha\) do DistS2S(\(\alpha\))
6. [Preorder] for each node \(\alpha\) do DistS2N(\(\alpha\))

### Evaluation (Algorithm II.2)

Given weights \(w[IDS] \in \mathbb{R}^{N \times r}\) (distributed from CYC), matvec of \(K\) involves four steps: an anyorder 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 \(\bar{w}\) must be redistributed from [IDS] to [LET] order before the execution of all DistL2L and DistS2S tasks. Function DistSymmetric() in Table IV 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, \(Near(\alpha) = \{1, p\}\) contains tree nodes from both green and yellow processes. To evaluate submatrices \(K_{i1}\) and \(K_{1p}\), we must redistribute indices carried by \(1\) and \(p\). To compute matvec \(K_{i1}\bar{w}_p\) and \(K_{1p}\bar{w}_1\), we must redistribute weights \(w_p\) and \(w_1\). Similarly, \(Far(\theta) = \{\pi, u\}\) contains tree nodes from all other processes (since \(u\) is distributed among blue and yellow processes). As a result, we must also redistribute indices \(\pi\), \(\bar{u}\), skeleton weights \(\bar{w}_\pi\), and \(\bar{w}_u\). In Table IV, function IDS2LET() redistributes indices and skeletons, and function DistSymmetric() redistributes weights and skeleton weights.
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 \((p - 1)^2\) \(I\)send and \(I\)recv tasks handled by the runtime system. To prevent blocking any worker (core), asynchronous messages are received by periodic polling using \(I\)probe. 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.

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 \(D\)ist\(N2S\) tasks, and the rightmost tree denotes the downward traversal of \(D\)ist\(S2N\) tasks. In the middle, the four circle nodes denote \(D\)ist\(S2S\) 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 \(D\)ist\(S2S\). Tasks \(D\)ist\(L2L\) are implemented similarly.

In Table V, task \(D\)ist\(S2S\) computes the reduction of the far interaction list over \(p\) processes. For example, node \(\beta\) in the middle of Figure 2 has three incoming edges carrying skeleton weights \(\bar{w}\) from process blue, green, and yellow (itself). Waiting for all messages to arrive inherently imposes a global synchronization barrier. To resolve this bottleneck, a \(D\)ist\(S2S\) 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 \(\text{Far}(\beta)\) into \(p\) sublists \(\text{Far}(\beta)[0:p-1]\) during the symmetrization. By splitting task

\[\text{Task MySend(TAR,TAG,List=[Near/Far]) : SendTask}\]

\[/*/ Isend(data_sent_to,TAR,TAG) is handled
by SendTask during after packing. */\]

\[\text{Override RecvTask::Pack(data_sent_to)}:\]

\[\text{foreach node } \alpha \text{ } E\text{List}_\text{received_from}[TAR] \text{ } do\]

\[\text{data_sent_to} \cap (\text{List=[Near]} ?) \text{ } \bar{w}_\alpha \text{ } \bar{w}_\alpha = \text{data_received_from}[\beta] \]

\[\text{Task MyRecv(SRC,TAG,List=[Near/Far]) : RecvTask}\]

\[/*/ Recv(data_received_from) is handled
by Listener::Listen() before unpacking. */\]

\[\text{Override RecvTask::Unpack(data_received_from)}:\]

\[\text{foreach node } \alpha \text{ } I\text{sent}_\text{to}(SRC,TAG) \text{ } do\]

\[\text{Irecv} \text{ } \text{Ibarrier(request)} \text{ } \text{only once}\]

\[\text{if} \text{ } \text{Irecv} \text{ } \text{request} \text{ } \text{then return}\]

\[\text{Function AsyncRedistribute(List=[Near/Far])} : \]

\[\text{foreach rank } \rho \text{ } \text{do msg_tasklist.insert(pair(SRC,TAG))}\]

\[\text{foreach rank } \rho \text{ } \text{do MySend(p,AUTOTAG,List)}\]

\[\text{foreach rank } \rho \text{ } \text{do MyRecv(p,AUTOTAG,List)}\]

\[\text{TABLE VI} \text{ } \text{Tasks and functions for asynchronous communication and redistribution: List sent_to and List received_from are inherited from IDS2LET(List) in Table IV.}\]

\[\text{DistS2S} \text{ accordingly, the data required by each subtask only}\]

\[\text{depends on one message to arrive. To address the possible}\]

\[\text{concurrent write on factor } i, \text{ we use a mutex (or an atomic}\]

\[\text{update) to resolve the race condition.}\]

\[\text{Message tasks: To handle asynchronous message passing,}\]

\[\text{we introduce message tasks and listeners in Table VI. Function}\]

\[\text{AsyncRedistribute()} \text{ simultaneously creates a pair of}\]

\[\text{SendTask} \text{ (with } I\text{send}) \text{ and } RecvTask \text{ (with } Recv).}\]

\[\text{By overriding SendTask::Pack(), child class MySend}\]

\[\text{can generate Read-After-Write (RAW) dependencies from}\]

\[\text{other normal tasks during the dependency analysis phase.}\]

\[\text{Similarly, child class MyRecv can result in RAW dependencies}\]

\[\text{by overriding RecvTask::Unpack(). During the execution}\]

\[\text{epoch, the former is dispatched to normal workers while all}\]

\[\text{incoming dependencies are satisfied (i.e. SendTask is treated}\]

\[\text{as normal tasks), but the latter can only be picked up and}\]

\[\text{consumed by listeners.}\]

\[\text{Listener: To implement non-blocking periodic polling, each}\]

\[\text{normal worker (thread) can be promoted to consume}\]

\[\text{RecvTask tasks. Function Listener::Listen()} \text{ describes}\]

\[\text{how listeners perform non-blocking periodic polling}\]

\[\text{to consume both } RecvTask \text{ and normal tasks. A listener}\]

\[\text{constantly probes for messages and check if their [SRC,TAG]}\]

\[\text{match any RecvTask task in the list. If so, messages will}\]

\[\text{be received and unpacked by the listener. Otherwise, listeners}\]

\[\text{behave the same as normal workers, trying to dispatch a}\]

\[\text{normal task from its own ready queue or steal one from}\]

\[\text{others. Finally the task (either a } SendTask \text{ or }\]

\[\text{another } RecvTask) \text{ will be executed to release other}\]

\[\text{dependent tasks. Notice that increasing the number of listeners}\]

\[\text{may potentially reduce the time of receiving and}\]

\[\text{unpacking messages (given sufficient}\]

\[\text{bandwidth). Listeners can still consume normal tasks even}\]

\[\text{during the symmetrization. By splitting task } I\text{send}\]

\[\text{from process blue, green, and yellow}\]

\[\text{there is no incoming message. As a result, we typically use}\]

\[\text{sender with computation without reducing the maximum}\]

\[\text{parallelism obtained from several traversals, Alltoallv in}\]

\[\text{Algorithm II.2 must be broken down into at least}\]

\[\text{\((p - 1)^2\)}\]
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 $O(1)$ work. Let $D = \log(N/m)$ 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, $\kappa$ 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 $s/m = O(1)$. Parallel tree partition and skeletonization take

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

(7)
time and $O(\kappa n + s^2(n + \log p))$ space.

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

$$ |\text{Near}(\alpha)| = O(b(N/m)), \quad |\text{Far}(\alpha)| = O(Db(N/m)) $$

(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

$$ t_sp + t_w Dbn^2 $$

(9)
with additional storage of size $Dbn^2$ 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 $O(2snN)$. 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 $2.5 \times$ speedup comparing to synchronous evaluation in §V.

$^3$Given that the neighbors we compute are unsymmetric, the near interaction list can be $2 \times$ 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 ($\$\text{SCRATCH}$ 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 $O(1)$ 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-$\ell^2$, Gram-angle, and geometric-$\ell^2$. For compression, we set parameters
$m$ (leaf node size), $s$ (maximum rank), $\tau$ (accuracy tolerance used to adaptively select the rank), and $k$ (number of neighbors used for $S$ and for sampling). The relative error $e_2$ 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.](image)

![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.](image)

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 HQRKP [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 50%—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.
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 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 $2\times$ 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 $\sim$100k MPI-GOFMM achieves significant speedups ScaLAPACK for $\sim$2k right-hand sides for five digits of accuracy.

### 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.

### 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 $\epsilon_2$, the wall-clock time (in seconds) for compression and multiplication, and G$\ell$O$\ell$S$\ell$ 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 it 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 $O(N \log N)$ 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 $7 \times$ 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 \( K02 \), \( C02 \), 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 \( \mathcal{F} \) requires a full (or partial) feed-forward step on the whole mini batch. Submatrix \( K_{ij} \) is evaluated as a \( \mathcal{F}(i,:)^T \mathcal{F}(j,:) \) 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 to 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 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


virtual function SendIndices(ids,dest,comm)
virtual function RecvIndices(ids,src,comm)
virtual operator T K(i,j)
virtual function RequestIndices(ids[1:p])
virtual function SendIndices(ids,dest,comm)
virtual function RecvIndices(ids,src,comm)
virtual function BcastIndices(ids,root,comm)
virtual function RequestIndices(ids[1:p])
virtual function SendIndices(ids,dest,comm)
virtual function RecvIndices(ids,src,comm)
virtual function BcastIndices(ids,root,comm)
virtual function RequestIndices(ids[1:p])
virtual function SendIndices(ids,dest,comm)
virtual function RecvIndices(ids,src,comm)
virtual function BcastIndices(ids,root,comm)
virtual function RequestIndices(ids[1:p])
virtual function SendIndices(ids,dest,comm)
virtual function RecvIndices(ids,src,comm)
virtual function BcastIndices(ids,root,comm)
virtual function RequestIndices(ids[1:p])
### 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.

<table>
<thead>
<tr>
<th>class Task</th>
</tr>
</thead>
<tbody>
<tr>
<td>deque&lt;Task*&gt; in, out;</td>
</tr>
<tr>
<td>Lock lock;</td>
</tr>
<tr>
<td>TaskStatus status=ALLOCATED;</td>
</tr>
<tr>
<td>volatile int n_dependencies_remaining=0;</td>
</tr>
<tr>
<td>string name;</td>
</tr>
<tr>
<td>float cost = 1.0;</td>
</tr>
<tr>
<td>virtual void DependencyAnalysis()=0;</td>
</tr>
<tr>
<td>virtual void Execute(Worker *worker)=0;</td>
</tr>
</tbody>
</table>

```cpp
class DistNode : Node |
this->node = 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) |
```

<table>
<thead>
<tr>
<th>class DistS2N&lt;DistNode,T&gt; : Task</th>
</tr>
</thead>
<tbody>
<tr>
<td>DistNode *node;</td>
</tr>
<tr>
<td>function DistS2N(DistNode *node)</td>
</tr>
<tr>
<td>this-&gt;node = node</td>
</tr>
<tr>
<td>this-&gt;name = string(&quot;DistS2N&quot;)</td>
</tr>
<tr>
<td>this-&gt;cost = estimated from flops and mops</td>
</tr>
<tr>
<td>DependencyAnalysis()</td>
</tr>
<tr>
<td>function DependencyAnalysis()</td>
</tr>
<tr>
<td>node-&gt;DependenOnChildren(this)</td>
</tr>
<tr>
<td>function Execute(Worker* worker)</td>
</tr>
</tbody>
</table>

For example, in Table X task DistIDTask will read skeletons $\alpha$, $\beta$ and write $\alpha$. This can be annotated by DistNode::DependOnChildren(), which encodes activities on node $\alpha$ and its children $\beta$ and $\gamma$. Similarly, task DistS2N will read $\beta$, and write $\alpha$, $\beta$, $\gamma$, 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 Read/Write sets inherited from class ReadWrite for each node.

Following [25], we present class ReadWrite in Table XII 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 $\tau$ in Table X). This is done by calling function DependencyAdd(src,task), which insert task $\tau$ to the out queue of task $src$ and insert task $src$ to the in queue of task $\tau$. 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 DistS2N.

### TABLE XII Definition of classes ReadWrite and DistNode. A shared memory tree node has type Node, which includes 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 lchild to access the children.

```cpp
class ReadWrite |
deqe<Task*> read, write; |
function DependencyAnalysis([R/W/RW],Task* $\tau$) |
if activity is R of RW then |
read.push_back($\tau$) |
foreach src : write do DependencyAdd(src,$\tau$) |
if activity is W of RW then |
foreach src : read do DependencyAdd(src,$\tau$) |
DependencyCleanUp() |
write.push_back($\tau$) |
function DependencyCleanUp() |
read.clear() |
write.clear() |
```

```cpp
class DistNode : Node |
DistNode *child; |
Comm comm; |
function DependOnChildren(Task* $\tau$) |
if CommSize(comm)<2 then |
lchild->DependencyAnalysis(R,$\tau$) |
rchild->DependencyAnalysis(R,$\tau$) |
else child->DependencyAnalysis(R,$\tau$) |
function DependOnParent(Task* $\tau$) |
 function DependOnParent(Task* $\tau$) |
if CommSize(comm)<2 then |
lchild->DependencyAnalysis(RW,$\tau$) |
rchild->DependencyAnalysis(RW,$\tau$) |
else child->DependencyAnalysis(RW,$\tau$) |
```

C. GOFMM and MPI-GOFMM Comparison
Algorithm VI.1 Side-by-side comparison between GOFMM (left) and MPI-GOFMM compression (right)

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

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

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

In Algorithm VI.1 and Algorithm VI.2, we summarize 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 VI.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 VI.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 \( \tilde{w} \) 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.
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 §VI-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 $O(1)$ time. Note that we do not require all the entries, only the ability to evaluate them. The $K(i,j)$ API is defined in VI-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 $\epsilon_2$ 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 -gopenmp 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 slurm, 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.