Distributed O(N) Linear Solver for Dense Symmetric Hierarchical Semi-Separable Matrices

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

We present a distributed memory algorithm for the approximate hierarchical factorization of symmetric positive definite (SPD) matrices. Our method is based on the distributed memory GOFMM, an algorithm that appeared in SC18 (doi:10.1109/SC.2018.00018). GOFMM constructs a hierarchical matrix approximation of an arbitrary SPD matrix that compresses the matrix by creating low-rank approximations of the off-diagonal blocks. GOFMM method has no guarantees of success for arbitrary SPD matrices. (This is similar to the SVD; not every matrix admits a good low-rank approximation.) But for many SPD matrices, GOFMM does enable compression that results in fast matrix-vector multiplication that can reach $N \log N$ time—as opposed to N^2 required for a dense matrix. GOFMM supports shared and distributed memory parallelism. In this paper, we build an approximate "ULV" factorization based on the Hierarchically Semi-Separable (HSS) compression of the GOFMM. This factorization requires $\mathcal{O}(N)$ work (given the compressed matrix) and $\mathcal{O}(N/p) + \mathcal{O}(\log p)$ time on p MPI processes (assuming a hypercube topology). The previous state-of-the-art required $\mathcal{O}(N \log N)$ work. We present the factorization algorithm, discuss its complexity, and present weak and strong scaling results for the "factorization" and "solve" phases of our algorithm. We also discuss the performance of the inexact ULV factorization as a preconditioner for a few exemplary large dense linear systems. In our largest run, we were able to factorize a 67M-by-67M matrix in less than one second; and solve a system with 64 right-hand sides in less than one-tenth of a second. This run was on 6,144 Intel "Skylake" cores on the SKX partition of the Stampede2 system at the Texas Advanced Computing Center.

I. Introduction

Let $K \in \mathbb{R}^{N \times N}$ be a dense SPD matrix. The work for an exact factorization of K scales as $\mathcal{O}(N^3)$ and therefore it is not practical when N is large. (For example, factorizing a dense matrix with $N = 67M$ would require over three days on a one-exaflop machine.) To enable the solution of very large systems we resort to approximation. Here we propose an algorithm for the approximate factorization of K.

Our method is based on GOFMM [23], [24], which constructs a hierarchical matrix factorization for K, using only entries in K. (We review GOFMM in §II). In particular, GOFMM constructs a matrix $\widetilde{K}$ (hereby ***"compresses"***) using $\mathcal{O}(N \log N)$ entries from K such that $\|\widetilde{K} - K\| \leq \epsilon\|K\|$ (for a user-defined tolerance $\epsilon > 0$) and a ***matvec*** with $\widetilde{K}$ requires as low as $\mathcal{O}(N)$ work. The constants in the compression and multiplication complexity estimates depend on ϵ. K itself requires $\mathcal{O}(N^2)$ storage, which is not scalable. So we assume that either K is of modest size so that it can fit on a distributed memory system or there is a way to compute the entries K_{ij} of K in sublinear time, typically $\mathcal{O}(1)$ or $\mathcal{O}(\log N)$. From a software engineering point of view, 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 [12]. Roughly speaking, we say that a matrix $\widetilde{K}$ is ***hierarchically low-rank*** [4], [12], 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. There are many variants on this structure. For example, when the matrices U and V are nested and $S = 0$ then we refer to HSS matrices. If U and V are not nested we refer to HODLR matrices. If $S \neq 0$, then $\widetilde{K}$ is an FMM matrix. (Note that there is no standard agreement on the nomenclature.) GOFMM supports both HSS and FMM variants. For non experts, in HSS, every off-diagonal block is approximated by a low-rank approximation; in FMM, certain off-diagonal blocks remain dense.

Background and significance: Dense SPD matrices appear in Schur complement matrices [5], in Hessian operators, in optimization, in simulation [20] and machine learning [6], in kernel methods for statistical learning [10], [14], and in N-body methods and integral equations [11], [12]. 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 kernel

matrix with bandwidth h:

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

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 an approach is not possible for arbitrary SPD matrices whose entries are not defined by kernel functions or for which we do not have points. Examples include SPD matrices related to graphs of social networks, protein/gene networks, fMRI data, microarray data, Hessian operators, maximum likelihood empirical covariance matrices [1], [17], and kernel methods for word sequences and graphs [7], [15].

Contributions: Based on GOFMM [24] (which we summarize in §§II-A), we introduce an approximate factorization for dense matrices. As we mentioned, GOFMM supports both FMM and HSS approximations of K. The FMM, however, is much harder to factorize. Our ULV factorization is based on the HSS version of GOFMM. When K is better approximated by FMM, we use the HSS variant as a preconditioner. (When the user selects FMM for the approximation of K, GOFMM builds the HSS variant so it can be used as preconditioner.) Our contributions are summarized below.

- We introduce a parallel algorithm approximate factorization based on the ULV scheme proposed in [8]. Our algorithm supports both distributed and shared memory, and is based on the HSS factorization of GOFMM. (See §II-B for the ULV algorithm and §II-C for its parallelization.)
- We present weak and strong scaling results for several matrices on up to 6,144 cores on the Stampede2 system on the Texas Advanced Computing center.
- We make the code publicly available in [25].

The algorithm has several parameters and presents several opportunities for tuning. For example, the leaf size in the compression, the maximum rank, and the number of nearest neighbors that are used to permute the matrix. We discuss more in §II-E.

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. What about the SPD properties of K? Do they extend to $\widetilde{K}$? GOFMM does guarantee symmetry but it cannot guarantee positivity if the approximation error is large. Standard eigenvalue perturbation theory can be used to relate the approximation error to the minimum eigenvalue and the distance from being singular. In practice, however, we do not have access to the minimum eigenvalue. As a result, the factorization may suffer from instabilities when the approximation error is large. To partially address this issue, we use a pivoted LU factorization for the block factorization, as opposed to a Cholesky one.

Related work: There is a very rich literature in exact and approximate dense matrix factorization for solving linear systems. Here we only review algorithms for distributed memory architectures. For arbitrary matrices the state-of-the-art is STRUMPACK [9], [19], [21]. It uses OpenMP and MPI based on a static 2D block cyclic data decomposition and, like us, uses a HSS compression. Unlike GOFMM, STRUMPACK does not permute K; such permutation is critical in order to improve the "approximability" of K by a HSS or FMM matrix. In [23], [24] we compared GOFMM with STRUMPACK and in most cases GOFMM was significantly faster. However, STRUMPACK also supports sparse matrices (with appropriate reordering), whereas GOFMM does not. Another library that supports low-rank approximations for the Cholesky factorization is the HiCMA library [2], [3], which is essentially a one-level block-Cholesky factorization where the off-diagonal blocks are low-rank approximated. Unlike GOFMM, HiCMA is not hierarchical so diagonal blocks are not further approximated. This limits HiCMA's scalability. In [22], we presented a similar $\mathcal{O}(N \log N)$ algorithm for approximate factorization of kernel matrices. However, our $\widetilde{K}$ approximation in [22] does not preserve the symmetry of K, whereas GOFMM does. Here, we exploit this to construct an ULV factorization that has $\mathcal{O}(N)$ linear complexity (once the matrix has been compressed). Furthermore, [22] is applicable only on kernel matrices whereas GOFMM can be applied to arbitrary matrices.

II. Methods

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

$$u = Kw \approx \widetilde{K}w, \;\; w \approx \texttt{solve}(\widetilde{K}, u) \text{ for } \; w \in \mathbb{R}^{N \times r}, \quad (3)$$

can be computed with $\mathcal{O}(N \log N)$ work and $\mathcal{O}(N/p \log N) + \mathcal{O}(\log p)$ time where K, u, and w are also distributed on p-process. First, we review the key components of GOFMM in §II-A. Second, we discuss how we can use the hierarchical semi-separable (HSS) portion of GOFMM to construct a fast direct solver in §II-B. Third, we derive the complexity of our algorithms in §II-D. Finally, we discuss the tuning and selection of hyper-parameter in our method.

A. *GOFMM: Geometry-Oblivious Fast Multipole Method*

Following [23], 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

$$\widetilde{K}_{\alpha\alpha} = \begin{bmatrix} \widetilde{K}_{\texttt{ll}} & 0 \\ 0 & \widetilde{K}_{\texttt{rr}} \end{bmatrix} + \begin{bmatrix} 0 & S_{\texttt{lr}} \\ S_{\texttt{rl}} & 0 \end{bmatrix} + \begin{bmatrix} 0 & UV_{\texttt{lr}} \\ UV_{\texttt{rl}} & 0 \end{bmatrix}, \quad (4)$$

where `l` and `r` are the **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 $\alpha = \texttt{l} \cup \texttt{r}$. The off-diagonal low-rank property is not invariant to matrix permutations. An ordered distance metric between rows and columns is essential for matrix partition. GOFMM uses either geometry based (for kernel matrices) or the Gram distance, which is computed using matrix entries. 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 metric (Gram-angle) to compute pairwise distances. Then `GOFMM` performs hierarchical clustering of the matrix indices using a binary tree based on pairwise distance d_{ij} between indices. 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. The low-rank approximations are computed by randomized linear algebra, in particular using a sampled nested interpolative decomposition [13]. Due to space limitations we cannot provide more details, nor describe the `FMM` variant of the algorithm. The complete mathematical description of the algorithm can be found in [23].

B. Fast Solver

The approximate factorization and linear solver are constructed from the hierarchical semi-separable (`HSS`) portion of the `GOFMM` approximation. If the `FMM` variant is used in $\widetilde{K}$, the sparse correction S is dropped such that the off-diagonal blocks are purely low-rank:

$$\widetilde{K}_{\alpha\alpha} = \begin{bmatrix} \widetilde{K}_{\mathtt{ll}} & 0 \\ 0 & \widetilde{K}_{\mathtt{rr}} \end{bmatrix} + \begin{bmatrix} 0 & P^T_{\widetilde{\mathtt{l}}\mathtt{l}} K_{\widetilde{\mathtt{l}}\widetilde{\mathtt{r}}} P_{\widetilde{\mathtt{r}}\mathtt{r}} \\ P^T_{\widetilde{\mathtt{r}}\mathtt{r}} K_{\widetilde{\mathtt{r}}\widetilde{\mathtt{l}}} P_{\widetilde{\mathtt{l}}\mathtt{l}} & 0 \end{bmatrix}. \quad (5)$$

For a two-level `HSS` matrix $\widetilde{K}$ that contains two inner nodes $\{\alpha, \beta\}$ and four leaf nodes $\{\mathtt{l}, \mathtt{r}, \mathtt{t}, \mathtt{b}\}$, $\widetilde{K}$ has the following structure (where $\times$ denotes symmetric part):

$$\widetilde{K} = \begin{bmatrix} \begin{bmatrix} K_{\mathtt{ll}} & \times \\ P^T_{\widetilde{\mathtt{r}}\mathtt{r}} K_{\widetilde{\mathtt{r}}\widetilde{\mathtt{l}}} P_{\widetilde{\mathtt{l}}\mathtt{l}} & K_{\mathtt{rr}} \end{bmatrix} & \times \\ A_{\beta\alpha} & \begin{bmatrix} K_{\mathtt{tt}} & \times \\ P^T_{\widetilde{\mathtt{b}}\mathtt{b}} K_{\widetilde{\mathtt{b}}\widetilde{\mathtt{t}}} P_{\widetilde{\mathtt{t}}\mathtt{t}} & K_{\mathtt{bb}} \end{bmatrix} \end{bmatrix}, \quad (6)$$

where the off-diagonal block $A_{\beta\alpha}$ can be written in terms of a two-way expansion using the children of node β and α

$$A_{\beta\alpha} = \begin{bmatrix} P^T_{\widetilde{\mathtt{t}}\mathtt{t}} & \\ & P^T_{\widetilde{\mathtt{b}}\mathtt{b}} \end{bmatrix} P^T_{\widetilde{\beta}[\widetilde{\mathtt{t}}\widetilde{\mathtt{b}}]} K_{\widetilde{\beta}\widetilde{\alpha}} P_{\widetilde{\alpha}[\widetilde{\mathtt{l}}\widetilde{\mathtt{r}}]} \begin{bmatrix} P_{\widetilde{\mathtt{l}}\mathtt{l}} & \\ & P_{\widetilde{\mathtt{r}}\mathtt{r}} \end{bmatrix}. \quad (7)$$

In summary, an `HSS` matrix is typically stored as a binary tree. Each inner node α owns factor $P_{\widetilde{\alpha}[\widetilde{\mathtt{l}}\widetilde{\mathtt{r}}]}$ and $K_{\widetilde{\mathtt{l}}\widetilde{\mathtt{r}}}$, where $\mathtt{l}$ and $\mathtt{r}$ are the two child nodes of α. Each leaf node α owns factor $P_{\widetilde{\alpha}\alpha}$ and a diagonal block $K_{\alpha\alpha}$. In the rest of the discussion, we use $P(\alpha)$ to denote factor $P_{\widetilde{\alpha}\alpha}$ (leaf nodes) or $P_{\widetilde{\alpha}[\widetilde{\mathtt{l}}\widetilde{\mathtt{r}}]}$ (inner nodes). Similarly, we use $Q(\alpha)$ to denote the Q factor (the full orthonormal basis computed from the full QR-factorization of $P(\alpha)^T$) owned by node α. The sub-index will be used to denote the matrix partitioning in the partial LU factorization.

ULV factorization: [8] describes how an `HSS` matrix can be factorized and solved fast by exploiting the nested basis (i.e. the two-way expansion in off-diagonal blocks). The key idea of ULV is to factorize the diagonal blocks recursively from the left- and right-hand side, which results in a post-order traversal of the tree due to the Read-After-Write data dependency of the parent node. We present the full factorization and solve in Algorithm II.1 and Algorithm II.2 for symmetric matrices (not necessary SPD). We discuss their distributed memory version in Algorithm II.3 and Algorithm II.4.

In Algorithm II.1, the first step is to factorize the leaf nodes by invoking `GEQRF` (QR factorization without pivoting) and extract the orthogonal basis from factor $P(\alpha)^T$. We then call `ORGQR` to generate the orthonormal basis $Q_{[\mathtt{C},\mathtt{F}]}(\alpha)$. $Q_{\mathtt{C}}(\alpha)$ spans the **column space** of $P(\alpha)^T$ and $Q_{\mathtt{F}}(\alpha)$ spans the **null space** of $P(\alpha)^T$ (i.e. the null space of the off-diagonal block). Factorizing $Q_{[\mathtt{C},\mathtt{F}]}(\alpha)$ out from both sides is equivalent to a similar transformation $Q^T K(\alpha, \alpha) Q$. In the parent view of $\mathtt{l}$ and $\mathtt{r}$, the similar transformation on $\widetilde{K}_{\alpha\alpha}$ in Equation 5 (on-diagonal block in Equation 6) results in

$$\begin{bmatrix} Q(\mathtt{l}) & \\ & Q(\mathtt{r}) \end{bmatrix}^T \left[\begin{array}{cc} K_{\mathtt{ll}} & \times \\ \hline P^T_{\widetilde{\mathtt{r}}\mathtt{r}} K_{\widetilde{\mathtt{r}}\widetilde{\mathtt{l}}} P_{\widetilde{\mathtt{l}}\mathtt{l}} & K_{\mathtt{rr}} \end{array}\right] \begin{bmatrix} \times & \\ & \times \end{bmatrix} \quad (8)$$

If we further repartition the blocked 2-by-2 matrix above to separate the matrix indices in the column space of the off-diagonal block from the null space, then we get the following

$$\left[\begin{array}{cccc} Z(\mathtt{l})_{\mathtt{CC}} & Z(\mathtt{l})_{\mathtt{CF}} & \Sigma(\alpha)^T & 0 \\ Z(\mathtt{l})_{\mathtt{FC}} & Z(\mathtt{l})_{\mathtt{FF}} & 0 & 0 \\ \hline \Sigma(\alpha) & 0 & Z(\mathtt{r})_{\mathtt{CC}} & Z(\mathtt{r})_{\mathtt{CF}} \\ 0 & 0 & Z(\mathtt{r})_{\mathtt{FC}} & Z(\mathtt{r})_{\mathtt{FF}} \end{array}\right], \quad (9)$$

where $\Sigma(\alpha) = Q_{\mathtt{C}}(\mathtt{r}) P^T_{\widetilde{\mathtt{r}}\mathtt{r}} K_{\widetilde{\mathtt{r}}\widetilde{\mathtt{l}}} P_{\widetilde{\mathtt{l}}\mathtt{l}} Q_{\mathtt{C}}(\mathtt{l})^T = R(\mathtt{r}) K_{\widetilde{\mathtt{r}}\widetilde{\mathtt{l}}} R(\mathtt{l})^T$. Observe that the three quarters of the off-diagonal blocks have been zeroed automatically. The `partialLU` function (at line 11, Algorithm II.1) factorizes $Z_{\mathtt{FF}}$, $Z_{\mathtt{FC}}$, and $Z_{\mathtt{CF}}$ from the both sides such that the partial L and U factors can be decomposed from Equation 9 as presented in Equation 10.

$$\left[\begin{array}{cccc} Z(\mathtt{l})_{\mathtt{CC}} & Z(\mathtt{l})_{\mathtt{CF}} & \Sigma(\alpha)^T & 0 \\ Z(\mathtt{l})_{\mathtt{FC}} & Z(\mathtt{l})_{\mathtt{FF}} & 0 & 0 \\ \hline \Sigma(\alpha) & 0 & Z(\mathtt{r})_{\mathtt{CC}} & Z(\mathtt{r})_{\mathtt{CF}} \\ 0 & 0 & Z(\mathtt{r})_{\mathtt{FC}} & Z(\mathtt{r})_{\mathtt{FF}} \end{array}\right] = \left[\begin{array}{cccc} 0 & & & \\ L_{\mathtt{FC}} & L_{\mathtt{CC}} & & \\ \hline 0 & 0 & 0 & \\ 0 & 0 & L_{\mathtt{FC}} & L_{\mathtt{CC}} \end{array}\right] \left[\begin{array}{cccc} Z(\mathtt{l})_{\mathtt{CC}} & & \Sigma(\alpha)^T & \\ & I & & \\ \hline \Sigma(\alpha) & & Z(\mathtt{r})_{\mathtt{CC}} & \\ & & & I \end{array}\right] \left[\begin{array}{cccc} 0 & U_{\mathtt{CF}} & 0 & 0 \\ & U_{\mathtt{CC}} & 0 & 0 \\ \hline & & 0 & U_{\mathtt{CF}} \\ & & & U_{\mathtt{CC}} \end{array}\right] \quad (10)$$

Observe that $Z_{\mathtt{FF}}$, $Z_{\mathtt{FC}}$, and $Z_{\mathtt{CF}}$ have all been factorized. $Z_{\mathtt{CC}}$ is the Schur complement of the partial factorization of Z. The remaining system in the middle of Equation 10 is now reducible and will be factorized when the algorithm visits the parent node.

Different from factorizing the leaf node, factors $V(\alpha)$ and $Z(\alpha)$ of an inner node α are computed using the factors of the two children. While $V(\alpha)$ represents the low-rank factor in the off-diagonal block of an `HSS` matrix, it can be computed from its children as shown at line 6, Algorithm II.1. For example, consider the off-diagonal block $A_{\beta\alpha}$ in Equation 6. After the similar transformation that we apply from Equation 8, the odd-

Algorithm II.1 $\texttt{factorize}(\alpha)$

1: $[\mathtt{C}, \mathtt{F}] = [\{\widetilde{\alpha}\}, \{\alpha \setminus \widetilde{\alpha}\}]$
2: **if** $\texttt{isLeaf}(\alpha)$ **then**
3: $\quad V(\alpha) = P(\alpha)$, $Z(\alpha) = K(\alpha, \alpha)$
4: **else**
5: $\quad \Sigma = R(\mathtt{r}) K_{\widetilde{\mathtt{r}}\widetilde{\mathtt{l}}} R(\mathtt{l})^T$
6: $\quad V(\alpha) = P(\alpha) \begin{bmatrix} R(\mathtt{l}) & \\ & R(\mathtt{r}) \end{bmatrix}$
7: $\quad Z(\alpha) = \begin{bmatrix} Z(\mathtt{l})_{\mathtt{CC}} & \Sigma^T \\ \Sigma & Z_{\mathtt{CC}}(\mathtt{r}) \end{bmatrix}$
8: **end if**
9: $[Q_{\mathtt{C}}(\alpha), R(\alpha)] = \mathtt{GEQRF}(V(\alpha)^T)$
10: $[Q_{[\mathtt{C},\mathtt{F}]}(\alpha), R(\alpha)] = \mathtt{ORGQR}(Q_{\mathtt{C}}(\alpha))$
11: $Z_{[\mathtt{CC},\mathtt{CF};\mathtt{FC},\mathtt{FF}]}(\alpha) = \texttt{partialLU}(Q^T_{[\mathtt{C},\mathtt{F}]}(\alpha) Z(\alpha) Q_{[\mathtt{C},\mathtt{F}]}(\alpha))$

diagonal block becomes

$$\left[\begin{array}{cc} R(\mathtt{t})^T & \\ 0 & 0 \\ \hline & R(\mathtt{b})^T \\ 0 & 0 \end{array}\right] P^T_{\widetilde{\beta}[\widetilde{\mathtt{t}}\widetilde{\mathtt{b}}]} K_{\widetilde{\beta}\widetilde{\alpha}} P_{\widetilde{\alpha}[\widetilde{\mathtt{l}}\widetilde{\mathtt{r}}]} \begin{bmatrix} R(\mathtt{l}) & 0 & & 0 \\ & 0 & R(\mathtt{r}) & 0 \end{bmatrix}. \tag{11}$$

Observe that the factors above also have the columns and rows representing the null space zero-out after applying the orthornomal transformation from the two sides. Specifically, $Q(\mathtt{l})$ and $Q(\mathtt{r})$ are applied from the right such that only the upper triangular matrices $R(\mathtt{l})$ and $R(\mathtt{r})$ were left. On the other hand, the diagonal block $Z(\alpha)$ is reduced from Equation 10. The rest of the factorization steps are the same except that the computation of factor $V(\alpha)$ is skipped while reaching the root of the tree. In this case, $Q(\alpha)$ is identity and the partial factorization becomes a full LU factorization since there is no null space left.

Algorithm II.2 $\texttt{solve}(\alpha, \texttt{forward/backward})$

1: **if** $\texttt{forward}$ **then**
2: $\quad$ **if** $\texttt{!isLeaf}(\alpha)$ **then** $B(\alpha) = [B_{\mathtt{C}}(\mathtt{l}); B_{\mathtt{C}}(\mathtt{r})]$
3: $\quad B_{[\mathtt{C};\mathtt{F}]}(\alpha) = Q_{[\mathtt{C},\mathtt{F}]}(\alpha)^T B_{[\mathtt{C};\mathtt{F}]}(\alpha)$
4: $\quad B_{[\mathtt{C};\mathtt{F}]}(\alpha) = \texttt{partialLUForward}(Z(\alpha), B_{[\mathtt{C};\mathtt{F}]}(\alpha))$
5: **else**
6: $\quad B_{[\mathtt{C};\mathtt{F}]}(\alpha) = \texttt{partialLUBackward}(Z(\alpha), B_{[\mathtt{C};\mathtt{F}]}(\alpha))$
7: $\quad B_{[\mathtt{C};\mathtt{F}]}(\alpha) = Q_{[\mathtt{C},\mathtt{F}]}(\alpha) B_{[\mathtt{C};\mathtt{F}]}(\alpha)$
8: $\quad$ **if** $\texttt{!isLeaf}(\alpha)$ **then** $[B_{\mathtt{C}}(\mathtt{l}); B_{\mathtt{C}}(\mathtt{r})] = B(\alpha)$
9: **end if**

ULV solve: Mathematically, ULV factorization applies similar transformation and LU factorization alternatively. As a result, the linear solve also contains two phases: a forward phase that resembles a post-order traversal of the tree (bottom-up) and a backward phase (pre-order traversal, top-down). Algorithm II.2 summarizes both traversals. Note that the concatenation between $B_{\mathtt{C}}(\mathtt{l})$, $B_{\mathtt{C}}(\mathtt{r})$ and orthonormal transformation $Q^T B$ are the only differences compared to a normal forward substitution. Similarly, in the backward phase we need to apply orthonormal transformation QB and split the intermediate results $[B_{\mathtt{C}}(\mathtt{l}); B_{\mathtt{C}}(\mathtt{r})]$.

C. Parallelism

We implemented our distributed ULV factorization (Algorithm II.3) and solver (Algorithm II.4) using GOFMM's distributed tree infrastructure. GOFMM maps the whole tree to p distributed processes (see Figure 1 for an example where $p = 4$ in four different colors). At depth-$log(p)$, p subtrees are distributed over p-process without replication. We say the portion of the tree below depth-$log(p)$ is local to each process (with a single color in the figure). Tree nodes above depth-$log(p)$ are distributed (filled with the gradient of colors).

Each distributed tree node above depth-$\log(p)$ has a unique MPI sub-communicator for collectives. A node α is assigned with communicator $\texttt{comm}(\alpha)$, where $\texttt{size}(\alpha)$ and $\texttt{rank}(\alpha)$ denote the local MPI rank and size in $\texttt{comm}(\alpha)$. While splitting a node α, processes are divided into two subgroups $\texttt{comm}(\mathtt{l})$ and $\texttt{comm}(\mathtt{r})$ according to $\texttt{rank}(\alpha)$. Processes with $\texttt{rank}(\alpha) < (1/2)\texttt{size}(\alpha)$ will be assigned to the $\texttt{comm}(\mathtt{l})$ otherwise $\texttt{comm}(\mathtt{r})$. Processes in $\texttt{comm}(\mathtt{l})$ and $\texttt{comm}(\mathtt{r})$ store the factors of node $\mathtt{l}$ and $\mathtt{r}$. In our distributed algorithm, factors of a distributed node $\mathtt{l}$ are stored on $\texttt{rank}(\mathtt{l}) = 0$, and factors of $\mathtt{r}$ are stored on $\texttt{rank}(\mathtt{r}) = 0$. In the view of their parent's communicator $\texttt{comm}(\alpha)$, these two MPI processes represent $\texttt{rank}(\alpha) = 0$ (the left master) and $\texttt{rank}(\alpha) = (1/2)\texttt{size}(\alpha)$ (the right master).

Factorization: In Algorithm II.3, we first deal with the base case at line 2. While $\texttt{size}(\alpha)$ is one (the sub-communicator only has one MPI process), node α is the root of the local tree. We call our shared memory factorization (Algorithm II.1) to decompose α. Otherwise, we must send factors $R(\mathtt{r})$ and $Z(\mathtt{r})$ from the right master ($\texttt{rank}(\alpha) = (1/2)\texttt{size}(\alpha)$) to the left master. Once received $R(\mathtt{l})$ and $R(\mathtt{r})$, the left master ($\texttt{rank}(\alpha) = 0$) can proceed to invoke the shared memory factorization. Overall, the whole distributed factorization involves a post-order traversal of the local tree and a post-order traversal of the distributed tree.

Solve: Our distributed ULV solver Algorithm II.4 has a similar structure where only the left and right master participate in the computation. Line 2 calls the shared memory solver (Algorithm II.2) to deal with the base case that contains only one MPI process. While node α is distributed, point-to-point communication is required between the left and right master. During the forward pass, the left master only needs to receive $B_{\mathtt{C}}(\mathtt{r})$ from the right master before calling the shared memory forward solve. This is because all other factors have been received previously the factorization step. During the backward pass, the left master will first invoke a shared memory backward solve and send $R_{\mathtt{C}}(\mathtt{r})$ back to the right master. Overall the whole distributed solve involves a post-order traversal of the local and distributed tree, and a preorder traversal of the distributed and local tree.

D. Complexity Analysis

Let m be the leaf node size, $\mathcal{D} = \log(N/m)$ be the tree depth, N/p be the number of indices owned per MPI process, t_s be the latency, t_w be the reciprocal of the bandwidth, and s

Algorithm II.3 `distFactorize`(α)

1: **if** `size`(α) `is one` **then**
2: `return factorize`(α)
3: **end if**
4: **if** `rank`(α) `is 0` **then**
5: `RECV`(R(`r`),`comm`(α))
6: `RECV`(Z(`r`),`comm`(α))
7: `factorize`(α)
8: **else if** `rank`(α) `is (1/2)size`(α) **then**
9: `SEND`(R(`r`),`comm`(α))
10: `SEND`(Z(`r`),`comm`(α))
11: **end if**

Algorithm II.4 `distSolve`(α,`forward/backward`)

1: **if** `size`(α) `is one` **then**
2: `return solve`(α,`forward/backward`)
3: **end if**
4: **if** `rank`(α) `is left master` **then**
5: `RECV`($B_{\mathtt{C}}$(`r`),`comm`(α)) **if** `forward`
6: `solve`(α,`forward/backward`)
7: `SEND`($B_{\mathtt{C}}$(`r`),`comm`(α)) **if** `backward`
8: **else if** `rank`(α) `is the right master` **then**
9: `SEND`($B_{\mathtt{C}}$(`c`),`comm`(α)) **if** `forward`
10: `RECV`($B_{\mathtt{C}}$(`c`),`comm`(α)) **if** `backward`
11: **end if**

be the rank of the off-diagonal blocks. For simplicity, let leaf node size m be equal to the rank s of the off-diagonal blocks.

With the assumption above, distributed factorization Algorithm II.3 takes $\mathcal{O}(s^3)$ work per node due to the QR and partial LU factorization. Given a tree with $2(N/m)-1$ nodes, the total work is $\mathcal{O}(s^3(N/m))$ for the factorization. While only the left and right master participate in the factorization above depth-$\log p$, the parallel time complexity with p-process is

$$\mathcal{O}\left((t_s+t_w s^2)\log p + s^3\left(\frac{N}{mp}\right)+s^3\log p\right). \quad (12)$$

The first portion represents the communication cost of $R(\mathtt{r})$ and $Z(\mathtt{r})$. Each factor has message size s^2 and it takes one `SEND` and `RECV`. As a result, the latency is constant per tree level. The second portion represents the `flops` per process. Each process factorizes a local subtree with $2(N/(pm))-1$ nodes. The distributed tree above depth-$\log p$ is factorized level-by-level due to the dependencies, hence $\mathcal{O}(s^3\log p)$ time.

Similarly distributed solve Algorithm II.4 takes $\mathcal{O}(s^2b)$ work per node where b denotes the number of right-hand sides. Given a tree with $2(N/m)-1$ nodes, the total work (including both forward and backward) is $\mathcal{O}(s^2b(N/m))$ for the solve. While only the left and right master participate in the factorization above depth-$\log p$, the parallel time complexity with p-process is

$$\mathcal{O}\left((t_s+t_w sb)\log p + s^2b\left(\frac{N}{pm}\right)+s^2b\log p\right). \quad (13)$$

The first portion represents the communication cost of $B_{\mathtt{C}}(\mathtt{r})$. The message size is sb per level and it only takes a single `SEND` and `RECV`. As a result, the latency is constant per tree level. The second portion represents the `flops` per process. The distributed tree above depth-$\log p$ is solved level-by-level due to the dependencies, hence $\mathcal{O}(s^2b\log p)$ time.

E. Parameter Turning

`GOFMM` has several parameters and algorithmic choices, e.g. `FMM` vs. `HSS`; or adaptive rank selection vs. preset rank selection (for the off-diagonal blocks). The most important parameters are the leaf node size m, and the maximum rank of the off-diagonal block s. These two parameters control the asymptotic behavior of the total work to a great extent (§II-D); both m and s must be small constants and must not scale with the problem size N. For m this is not an issue. But for s it highly depends on the generating process for the matrix. For example for certain matrices constant s is only possible when the `FMM` variant is used. A secondary side effect of s and m is also their impact on `FLOPS`. They affect the size of `BLAS` and `LAPACK` functions on each node; hence m and s must be sufficiently large to achieve high `FLOPS` efficiency. Of course, rank s directly affects the accuracy of the approximation and m determine the block size of the sparsity (also accuracy related). We tune these two parameters using a 2D grid-search in log-2 scale and scan values from 64 to 2,048.

While the actual rank of each off-diagonal block is unknown, `GOFMM` employs an adaptive interpolative decomposition (ID) [13] to select the rank based on the desired accuracy of the low-rank approximation. However, the accuracy of the off-diagonal block (see [18] for a proof of the upper bound of the nested adaptive ID with random sampling) does not translate directly to the accuracy of the full matrix due to the randomized sampling. To properly select tolerance for the ID, we manually select and adjust the maximum rank.

Finally, there are a few parameters we need to tune depending on the applications. Here we take kernel ridge regression with 2-norm regularization as an example. The objective function we want to minimize is

$$\min_w \|u-Kw\|_2^2+\frac{\lambda}{2}\|w\|_2^2, \quad (14)$$

where u denotes the labels, and λ denotes the regularization. The optimization above is equivalent to solving $u=(K+\lambda I)w$ given λ. When K is a Gaussian kernel matrix, we further need to decide the bandwidth h that achieve the smallest objective. To tune both λ and h, we typically employ line search and ten-fold cross-validation.

III. Empirical Results

We conduct a number of experiments (#1–#36) to demonstrate the efficiency, scalability of the direct solver. First, we discuss the implementation and hardware. Then we report the scaling experiments (Figure 2 and Figure 3). Finally, we test the `HSS` factorization as a preconditioner for the `FMM` approximation for $\tilde{K}$. We report the results in Table I.

Implementation and hardware: Our solver is implemented in `C++` with `MPI` (`MPI_THREAD_MULTIPLE` is required) and `OpenMP`. The implementation is available in `GOFMM`, and its only dependencies are multi-threaded `BLAS/LAPACK`

and MPI (we used the Intel MPI library in the following experiments). 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). Except for kernel matrices generated by the real world dataset COVTYPE [16], all runs are performed in single precision and we use one MPI rank and 48 OpenMP threads per node.

SPD matrices: We conduct our experiments on several different SPD and Gaussian kernel matrices. For the kernel matrices we use random point clouds (normally distributed from 500M to 16M points in 6D) and the COVTYPE UCI dataset (100K, 54D). Notice that for kernel matrices, the higher the intrinsic dimension of the dataset the harder is for the matrix to compress and factorize. We also use three other non-kernel matrices with size 65,536. **K02** is a 2D regularized inverse Laplacian squared, resembling the Hessian operator of a PDE-constrained optimization problem. The Laplacian is discretized using a 5-stencil finite-difference scheme with Dirichlet boundary conditions on a regular grid. **K13** is 2D advection-diffusion operators on a regular grid with highly variable coefficients. Both resemble inverse covariance matrices and Hessian operators from optimization and uncertainty quantification problems. **K15** is 2D pseudo-spectral advection-diffusion-reaction operators with variable coefficients.

Strong scaling: In Figure 2 (#1, #2, #3, #4, #5, #6, #7, #8, #9, #10, #11, and #12), we use a 2^{24}-by-2^{24} Gaussian kernel matrix generated with a synthetic 6-D point dataset. 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). First, K is compressed by GOFMM. Then we report timings for the factorization phase on the left; and the solving phase (with 64 right-hand sides) on the right. The x-axis denotes the number of cores, and the y-axis denotes the runtime in seconds. The bar charts reveal the scaling trend, and the blue dotted lines denote the absolute floating point arithmetic efficiency computed as the ratio of the achieved GFLOPS over the theoretical peak (4.3 TFLOPS per node).

The factorization phase achieves a higher floating efficiency due to the $O(s^3)$ QR and LU factorization computed on each node. The solving phase is more memory bound, since there is only $P(s^2b)$ FLOPS per node (with number of right-hand sides $b = 64$). The efficiency degradation is expected as an artifact of our distributed algorithm. Recall that the factorization and solving only happen on the left-master of the local communicator, and the efficiency achieved by these distributed tree nodes degrades exponentially. As a result, the deeper the distributed tree (that is toward the right side of the x-axis in the figure) is, the faster the efficiency decades. Overall, our implementation is able to achieve 15.3% and 7.3% of peak using 6,144 CPU cores in the strong scaling experiments.

#	Matrix	Spy	Pred	MATVEC$_\epsilon$	SOLVE$_\epsilon$	SOLVE$_t$	#Iter
25	COV	0%	True	1E-2	4E-11	0.43	0
26	COV	1%	True	7E-3	8E-4	0.43	940
27	COV	1%	False	7E-3	6E-2	-	2,532
28	K02	0%	True	1E-2	1E-6	0.13	0
29	K02	1%	True	8E-3	3E-5	0.13	0
30	K02	1%	False	8E-3	4E-4	-	1
31	K13	0%	True	5E-5	9E-7	0.13	0
32	K13	1%	True	4E-5	1E-6	0.13	0
33	K13	1%	False	4E-5	2E-5	-	1
34	K15	0%	True	5E-1	1E-6	0.05	0
35	K15	1%	True	1E-1	2E-4	0.05	3
36	K15	1%	False	1E-1	8E-4	-	3

TABLE I ***Preconditioner experiments using precondition conjugate gradient (PCG).*** *All experiments were done on four Stampede-2 nodes (192 Skylake cores) for different matrices. Throughout the experiments we control the percentage of the sparse correction in GOFMM and the type of the preconditioner (identity or HSS). The results are reported base on a mixed stopping criteria. The PCG execution terminates while either reaching 1E-3 in relative error or exceeding 20-minute in the execution time.*

Week scaling: In Figure 3 (experiments #13, #14, #15, #16, #17, #18, #19, #20, #21, #22, #23, and #24, we perform weak scaling experiments (2^{17} per process) on synthetic Gaussian kernel matrices with up to 128 MPI processes (6,144 cores). To ensure the flops required by the computation scales linear with the number cores, we fix the rank of the HSS matrix generated by GOFMM to 256.

Comparing with the strong scaling results, we can observe that weak scaling results usually achieve lower efficiency using the same amount of cores. For example, #13 in the weak scaling experiment achieves only 20% of peak but #1 in the strong scaling experiment achieves almost 30%. This is because #1 has 64× more available tasks to schedule due to a taller tree. As a result, #13 suffers less from the effect of the diminishing parallelism (according to the Amdahl's law). The discrepancy between strong and weak scaling in the solving phase is smaller due to the fact that there are 2× more tasks than the factorization phase and our setup with $b = 64$ is more memory bound. Notice that the efficiency typically drops dramatically from 768-core (16 compute nodes) to 1,536-core (32 compute nodes). We suspect that this is because the leaf switch size of Stampede2 fat-tree topology is 28-node. As a result, the cost of the additional hop is introduced while scaling from 16- to 32-node.

Preconditioner: We show that our HSS solver can be used as a preconditioner for other more sophisticated hierarchical matrices such as FMM matrices. To be specific, we use GOFMM to construct algebraic FMM approximation of different SPD matrices and extract their HSS structures to construct the preconditioners. We show that the HSS preconditioners can effectively improve the convergent rate.

In Table I, we report the convergence results (#25–#33) of precondition conjugate gradient (PCG) method, an iterative linear solver for symmetric matrices, on four different SPD matrices. For each matrix, we conduct three experiments that

vary the percentage of the sparse correction in the off-diagonal blocks, the preconditioner (identity of HSS) and observe the convergent behavior. Throughout the experiments we use a mixed stopping criteria that will terminate the PCG execution either each 1E-3 in relative error or exceeding 20-minute in the execution time.

Experiments #25, #28, #31 and #34 do not introduce any sparse correction in the approximation. As a result, our HSS solver is able to solve the system directly without any iteration. While increasing the sparse correction to 1%, our HSS solver can no longer solve the systems directly. Take #26 as an example: From column $\texttt{MATVEC}_{\epsilon}$, we can observe that increasing sparse correction in the off-diagonal blocks (the amount of direct evaluation) can improve the approximation accuracy. However, PCG takes 940 iterations to converge even with the help of our HSS preconditioner. Without the preconditioner (using an identity matrix as a preconditioner), we can observe that the iterative solver terminated at 6E-2 (2,532 iterations) and did not converge within 20 minutes. Despite that matrices **K02**, **K13**, and **K15** do not have no difficulty to converge without a preconditioner. Still our HSS preconditioner further improves the convergence rate in #29, #32, and #35.

IV. Conclusions

We presented an $\mathcal{O}(N)$ parallel algorithm for approximate factorization of SPD matrices. To our knowledge, this is the only publicly available library with this capability. Still this complexity does not come with some hefty "fine print". To improve stability we use GETRF that messes up with the symmetry and makes the factorization slower. Moreover, we could switch to POTRF if we know that the approximation will be stable; this however, has not been implemented yet. Finally, we did not consider GPU implementation. The latter is quite critical as most upcoming architectures will package most of the bandwidth and compute power on multi-GPU systems. This is future work.

Acknowledgments

This material is based upon work supported by AFOSR grant FOSR-FA9550-17-1-0190; by NSF grants CCF-CCF-1725743 and CCF-1817048; by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Applied Mathematics program under Award Number DE- SC0019393; by NIH grant R01NS042645-14; and the Deutsche Forschungsgemeinschaft (DFG) through the TUM International Graduate School of Science and Engineering (IGSSE) and SPPEXA. Any opinions, findings, and conclusions or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the AFOSR, the DOE, the NIH, or the NSF. Computing time on the Texas Advanced Computing Centers Stampede system was provided by an allocation from TACC and the NSF.

References

[1] S. Abdulah, H. Ltaief, Y. Sun, M. G. Genton, and D. E. Keyes, *Exageostat: A high performance unified software for geostatistics on manycore systems*, IEEE Transactions on Parallel and Distributed Systems, 29 (2018), pp. 2771–2784.

[2] ———, *Parallel approximation of the maximum likelihood estimation for the prediction of large-scale geostatistics simulations*, in 2018 IEEE International Conference on Cluster Computing (CLUSTER), IEEE, 2018, pp. 98–108.

[3] K. Akbudak, H. Ltaief, A. Mikhalev, A. Charara, A. Esposito, and D. Keyes, *Exploiting data sparsity for large-scale matrix computations*, in European Conference on Parallel Processing, Springer, 2018, pp. 721–734.

[4] M. Bebendorf, *Hierarchical matrices*, Springer, 2008.

[5] M. Benzi, G. H. Golub, and J. Liesen, *Numerical solution of saddle point problems*, Acta numerica, 14 (2005), pp. 1–137.

[6] 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, 21 (2011), pp. 977–995.

[7] N. Cancedda, E. Gaussier, C. Goutte, and J. M. Renders, *Word sequence kernels*, Journal of Machine Learning Research, 3 (2003), pp. 1059–1082.

[8] S. Chandrasekaran, M. Gu, and T. Pals, *A fast ulv decomposition solver for hierarchically semiseparable representations*, SIAM Journal on Matrix Analysis and Applications, 28 (2006), pp. 603–622.

[9] 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, 38 (2016), pp. S358–S384.

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

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

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

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

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

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

[16] M. Lichman, *UCI machine learning repository*, 2013.

[17] H. Ltaief, A. Charara, D. Gratadour, N. Doucet, B. Hadri, E. Gendron, S. Feki, and D. Keyes, *Real-time massively distributed multi-object adaptive optics simulations for the european extremely large telescope*, in 2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS), IEEE, 2018, pp. 75–84.

[18] 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, 38 (2016), pp. S720–S749.

[19] P.-G. Martinsson, *Compressing rank-structured matrices via randomized sampling*, SIAM Journal on Scientific Computing, 38 (2016), pp. A1959–A1986.

[20] 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, L. Biegler, O. Ghattas, M. Heinkenschloss, and B. van Bloemen Waanders, eds., Lecture Notes in Computational Science and Engineering, Springer-Verlag, 2001.

[21] 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, 42 (2016), pp. 27:1–27:35.

[22] C. Yu, W. March, and G. Biros, *An $n \log n$ parallel fast direct solver for kernel matrices*, in 31th IEEE International Parallel & Distributed Processing Symposium (IEEE IPDPS 2017), Orlando, USA, May 2017.

[23] 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.

[24] C. D. Yu, S. Reiz, and G. Biros, *Distributed-memory hierarchical compression of dense SPD matrices*, in Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis, SC '18, Piscataway, NJ, USA, 2018, IEEE Press, pp. 15:1–15:15.

[25] C. D. Yu, S. Riesz, and J. Levitt, GOFMM *home page*. 2018.

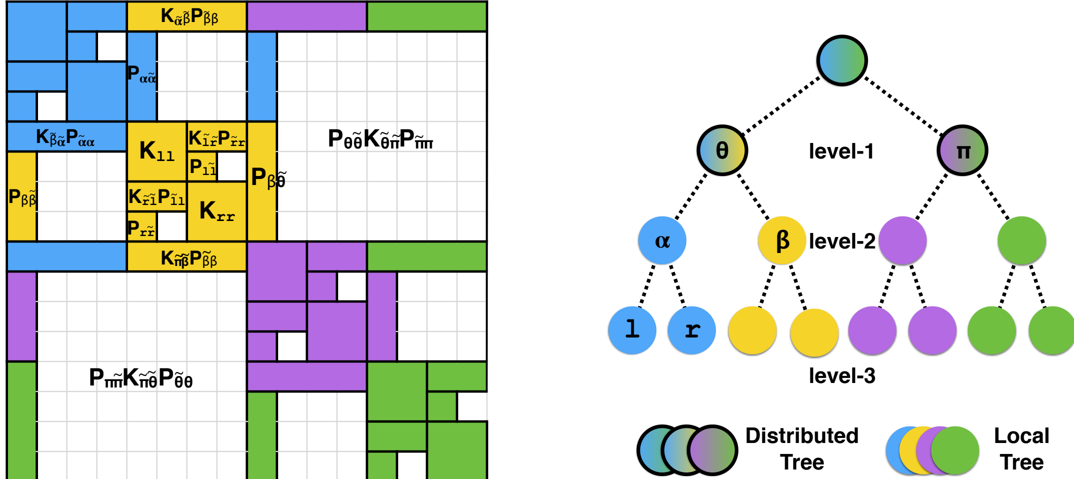

Fig. 1 ***Distributed HSS matrix stored in a complete binary tree.*** *The four subtrees (at depth-*$\log 4$*) in four single colors on the right represent the four hierarchical diagonal blocks on the left. Each process owns all the factors in the same color in the diagonal block, which represents the local subtree. Tree nodes above depth-*$\log 4$ *are distributed. That is, the factors of the off-diagonal blocks are stored on multiple processes. To be specific, the low-rank factors* $P_{\theta\tilde{\theta}}K_{\tilde{\theta}\tilde{\pi}}P_{\tilde{\pi}\pi}$ *are stored on four different processes, where each process owns the factors in the same color.*

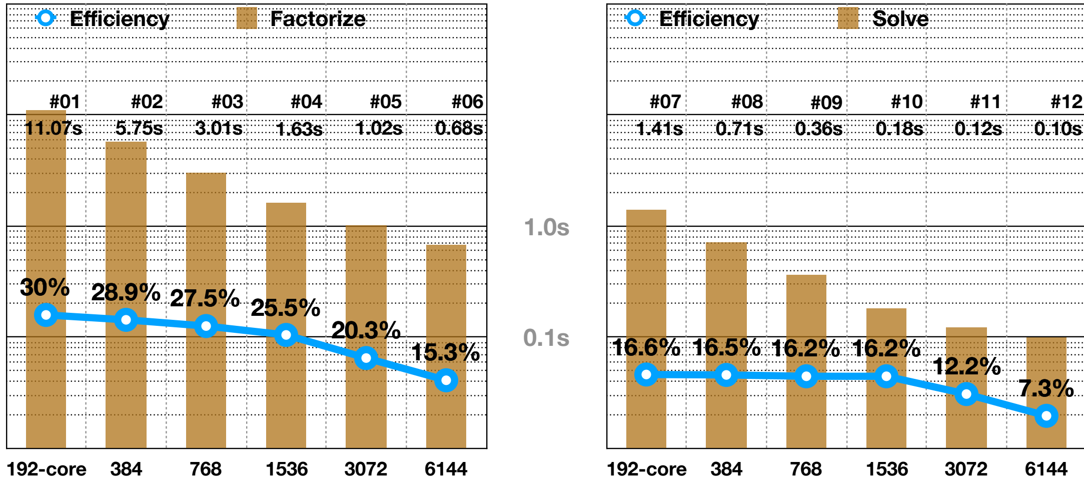

Fig. 2 ***Strong scaling (runtime in seconds as a function of the number of cores)*** *of our ULV factorization and solver applied to a 16M-by-16M Gaussian kernel matrix generated by a 6D random point cloud.* ***Left:*** *Factorization time and the floating point operation efficiency (as the ratio between achieved* `GFLOPS` *and theoretical peak (≈4.3* `TFLOPS`*).* ***Right:*** *Solving time (64 right-hand sides) and the floating point operation efficiency. The x-axis denotes number of cores (6,144 Skylake cores correspond to 128 Stampede-2 nodes), the y-axis denotes time in seconds in log scale. We annotate the scale in the middle of each figure.*

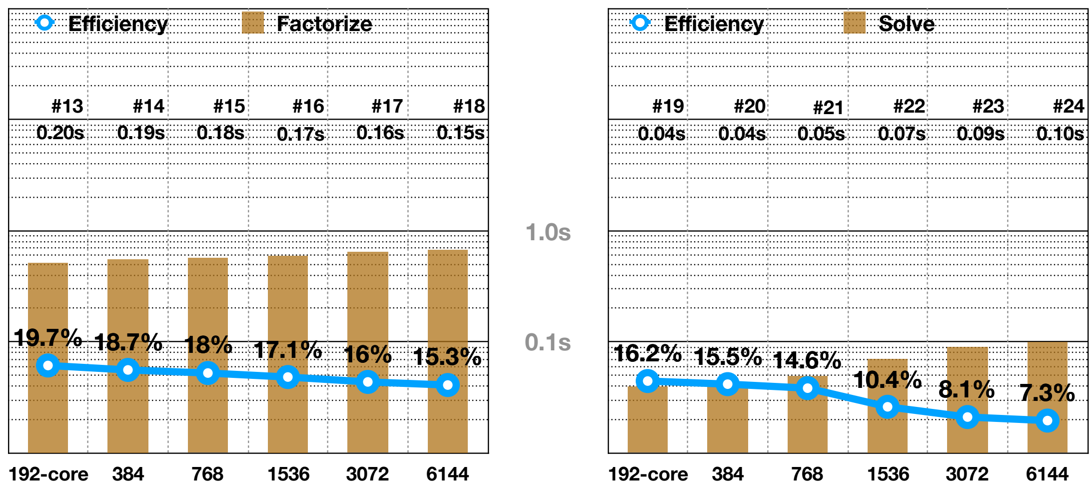

Fig. 3 ***Weak scaling (runtime in seconds vs. cores)*** *of our ULV factorization and solver applied to Gaussian kernel matrices generated synthetically with point clouds in 6-D. The configuration is the same as Figure 2. The grain size is 524k points per MPI process. The largest problem size involves factorizing of the 67M-by-67M kernel matrix and solving right-hand sides with a 67M-by-64 matrix of random vectors.*