Preprint ANL/MCS-P5112-0314

DATA STRUCTURE AND ALGORITHMS FOR RECURSIVELY LOW-RANK COMPRESSED MATRICES

JIE CHEN*

Abstract. We present a data structure and several operations it supports for recursively low-rank compressed matrices; that is, the diagonal blocks of the matrix are recursively partitioned, and the off-diagonal blocks in each partition level admit a low-rank approximation. Such a compression is embraced in many linear- or near-linear-time methods for kernel matrices, including the fast multipole method, the framework of hierarchical matrices, and several other variants. For this compressed representation, we develop a principled data structure that enables the design of matrix algorithms by using tree traversals and that facilitates computer implementation, especially in the parallel setting. We present three basic operations supported by the data structure—matrix-vector multiplication, matrix inversion, and determinant calculation—all of which have a strictly linear cost. These operations consequently enable the solution of other matrix problems with practical significance, such as the solution of a linear system and the computation of the diagonal of a matrix inverse. We show comprehensive experiments with various matrices and kernels to demonstrate the favorable numerical behavior and computational scaling of the algorithms.

Key words. Compressed matrix, kernel matrix, data structure, matrix-vector multiplication, inversion, determinant, linear system

AMS subject classifications. 65F05, 65F10, 65F30, 65F40

1. Introduction.

Kernel matrices arise from an extensive array of applications in science and engineering, ranging from the solving of differential/integral equations, N-body simulations, and electronic structures, to statistical covariance modeling, scattered data interpolation, and kernel machine learning. Informally, an $n \times n$ kernel matrix Φ is defined based on a set of n points $\{\boldsymbol{x}_i\}$ and a two-argument kernel function $\phi(\cdot,\cdot)$, where the (i,j) element of Φ is $\phi(\boldsymbol{x}_i,\boldsymbol{x}_j)$. It was found that the kernel ϕ in many practical situations is numerically degenerate, in the sense that ϕ can be (approximately) expressed as

$$\phi(\boldsymbol{x},\boldsymbol{y}) = \sum_{i=1}^{r} \psi_i(\boldsymbol{x})\varphi_i(\boldsymbol{y})$$

by using two bases $\{\psi_i\}$ and $\{\varphi_i\}$ with a finite number r of summation terms, when $\boldsymbol{x}$ and $\boldsymbol{y}$ are reasonably distant. It suffices for the approximation, if not exact equality, to be accurate up to machine precision from a numerical perspective although in practice, lower precision is also acceptable. Hence, when the set of points $\{\boldsymbol{x}_i\}$ is ordered in a manner such that the points listed in the front are geometrically separate from those listed at the end, then the off-diagonal blocks of the corresponding matrix are numerically low rank, with the rank governed by r. One can further reorder the points corresponding to each diagonal block of the matrix, and this reordering is recursive. The result is a permuted kernel matrix that forms a hierarchical structure, where in each level the off-diagonal blocks are low rank.

Motivated by this observation, we consider a fundamental data structure for storing such a matrix and design supportive algorithms for performing basic matrix operations. Specifically, we abstract the hierarchical structure and encode the contents

*Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439. Email: jiechen@mcs.anl.gov

(e.g., the low-rank decomposition) in a tree data structure. The supported matrix operations are then all in the form of tree traversals, a different (but closely related) perspective from that of traditional dense matrix computations. In what follows, we use the notation A, an $n \times n$ complex matrix, to denote the matrix. In most of the cases, one instantiates A as an (approximated) kernel matrix, although it is also possible that A does not relate to any kernel but only conforms to the structure.

Most important, the data structure requires a storage that is linear in n, and the matrix operations it supports also have a strictly linear time cost. The matrix operations we present in this paper include matrix-vector multiplication, matrix inversion, and determinant calculation. (More supported operations will be described in subsequent papers.) These basic operations support straightforward extensions to other matrix calculations that have a highly practical significance, such as solving a linear system of equations and computing the diagonal of the inverse of a matrix [5, 4, 21]. Hence, this data structure overcomes the fundamental scalability barrier of general dense matrix computations (that typically require an $O(n^2)$ memory and $O(n^2)$ to $O(n^3)$ time cost) and proves useful in many applications when the matrix becomes large.

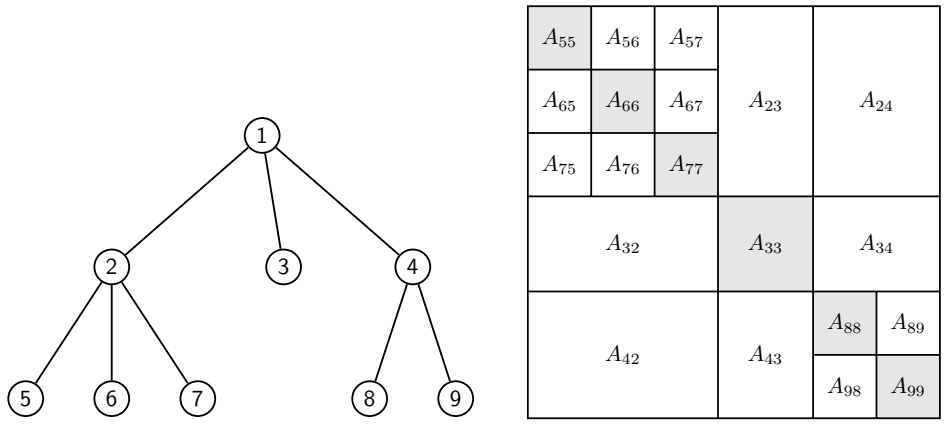
After presenting the technical details in Sections 2 to 6, we discuss in Section 7 the fine distinctions and connections between this work and existing work, including the fast multipole method (FMM) [13, 28, 11], tree code methods [3, 18], interpolative decomposition [10, 24, 17], hierarchical matrices (H matrix) and H^2 matrices [14, 15, 7], and hierarchically semiseparable matrices (HSS matrices) [9, 8, 26]. Readers familiar with the literature may find it more entertaining to read Section 7 immediately after Section 2, before proceeding with the technicalities in Sections 3 to 6. This work is by no means a completely novel invention in view of the prosperous literature on fast methods for kernel matrices, all of which aim at a linear or near-linear computational cost. However, a distinguishing feature of this work is a new matrix inversion algorithm (and an extension to the computation of determinant) tied to the proposed tree data structure, which differs from other tree structures considered in the literature in one way or another. The computed inverse can be used to solve a linear system with an accuracy matching that of a traditional dense direct solver. Furthermore, including inversion, all proposed algorithms are designed to maximally follow the computational flow of the FMM method, for which reason they have a good potential to scale to and beyond $O(10^5)$ processor cores on high-performance computers, as has already been demonstrated for the FMM case [19].

Numerical experiments gauging the numerical errors for medium-sized matrices are presented in Section 8. Empirically quantifying the errors for large-scale matrices is highly difficult: the "ground truths" must be computed in a complexity higher than linear, for which experiments are not affordable. We did, however, conduct comprehensive tests for cases as general as possible, including real versus complex matrices, well-conditioned versus ill-conditioned matrices, Hermitian versus non-Hermitian matrices, positive-definite versus indefinite matrices, and translational-invariant kernels versus general kernels. The purpose is to showcase the wide applicability of the proposed data structure as a framework for designing matrix algorithms, whose numerical stability should not be traded for a fast execution. Nevertheless, we demonstrate timing results to confirm the linear scaling of all the algorithms. Concluding remarks pointing to future development are given in Section 9.

2. Recursively low-rank compressed matrix. The matrix of interest entails a multilevel structure: it is block partitioned, and the main-diagonal blocks are re-

cursively subpartitioned. In addition, all the off-diagonal blocks have a rank that is bounded by a (relatively small) constant. Intricacy exists, however, in the relationship between the off-diagonal blocks across different levels.

Consider an arbitrary rooted tree (as in graph theory), an example of which is shown in Figure 2.1(a). The tree is not necessarily a full tree (all nonleaf nodes have the same number of children) or a complete tree (every level is completely filled); however, every nonleaf node must have more than one child.



(a) Partitioning tree T (b) Recursively low-rank compressed matrix A

FIG. 2.1. *A tree and the matrix it represents.*

This tree corresponds to a multilevel structure of a square matrix A: the root represents all the row (column) indices of the matrix; and the children of every nonleaf node i collectively represent a partitioning of the set of indices i contains. Hence, we denote by A_{ij} a matrix block whose row indices and column indices are represented by nodes i and j, respectively. Note that i and j are not arbitrary; they are either equal or siblings (that is, they share the same parent). When $i \neq j$, the block A_{ij} is located off diagonal and is not subpartitioned. On the other hand, when $i = j$ and i is not a leaf node, A_{ii} is subpartitioned. The partitioning need not be balanced. Figure 2.1(b) shows a recursively block-partitioned matrix corresponding to the tree on the left.

For every off-diagonal block A_{ij}, assume that it admits a factorization[1]

$$A_{ij} = U_i \Sigma_{ij} V_j^*, \tag{2.1}$$

where U_i and V_j have r columns and Σ_{ij} has a size $r \times r$. The factorization only implies that A_{ij} has a rank no greater than r; the actual rank can be strictly smaller. The value r is the same for all off-diagonal blocks on all levels. Conceptually, r is small as in "low" rank; however, we impose no constraints on the magnitude of r. In Figure 2.1(b), all the unshaded blocks are off-diagonal blocks, and they admit such a factorization.

The aforementioned intricacy lies in a same-subspace requirement for the U_i and V_j factors across levels. Specifically, for any pair of parent i and child k, there exist

[1]In this paper, a star appearing in the superscript means the transpose of a real matrix or conjugate transpose of a complex matrix, whereas a star appearing in the subscript means general and unspecified indices.

$r \times r$ matrices W_{ki} and Z_{ki} such that

$$U_i(I_k, :) = U_k W_{ki} \quad \text{and} \quad V_i(I_k, :) = V_k Z_{ki}, \tag{2.2}$$

where I_k denotes the set of indices a node k contains. Then, if k is a descendant of i through the path $(i, k_1, k_2, \ldots, k_t, k)$, we have

$$U_i(I_k, :) = U_k W_{kk_t} \cdots W_{k_2 k_1} W_{k_1 i}, \tag{2.3}$$

and similarly for $V_i(I_k, :)$. This means that for a leaf node k, the $U_i(I_k, :)$'s for all i along the path from k to the root span the same subspace, if the change-of-basis matrices $W_{k_s k_t}$ are nonsingular for all consecutive pairs $k_s k_t$ along the path; and similarly for the $V_i(I_k, :)$'s.

We are now ready to present the precise definition of the matrix we consider in this paper.

DEFINITION 2.1. *A rooted tree T is called a* partitioning tree *of a set I of indices if*

1. *no nodes have exactly one child;*
2. *the root contains I;*
3. *the children of a nonleaf node i constitutes a partitioning of the set of indices i contains.*

DEFINITION 2.2. *For every partitioning tree T of a set of indices $\{1, \ldots, n\}$ and for a constant r, there defines the structure of a* recursively low-rank compressed matrix $A \in \mathbb{C}^{n \times n}$ *such that*

1. *for every node i, A_{ii} is defined as $A(I_i, I_i)$, where I_i denotes the collection of indices i contains;*
2. *for every pair of siblings i and j in the tree, A_{ij} is defined as $A(I_i, I_j)$;*
3. *every such matrix block A_{ij} admits a factorization* (2.1) *where $\Sigma_{ij} \in \mathbb{C}^{r \times r}$;*
4. *for every U_i (resp. V_j) in* (2.1)*, if i (resp. j) has a child k, there exists $W_{ki} \in \mathbb{C}^{r \times r}$ (resp. Z_{kj}) such that* (2.2) *is satisfied.*

Note that a node of the partitioning tree T contains a not necessarily consecutive sequence of indices. For illustration purpose, however, the blocks of the matrix A in Figure 2.1(b) are all drawn to contain consecutive rows and columns. The matrix reordering is always implicit.

2.1. Data structure. The matrix A in Definition 2.2 is naturally stored with the tree T. The essential elements are

1. A_{ii} for all leaf nodes i,
2. U_i and V_i for all leaf nodes i,
3. Σ_{ij} for all pairs of siblings i and j, and
4. W_{ki} and Z_{ki} for all pairs of parent i and child k.

Clearly, if i is not a leaf node, U_i and V_i are not stored. The reason is that U_i can be built based on (2.3) by using U_k for all k that are leaf descendants of i.

In computer implementation, a tree is usually represented as a linked list or an array, where each element represents a tree node. The parent-to-child links in the tree are implicit in these data structures. For example, when the number of children of a node is dynamic, the parent node usually has only one pointer pointing to one of the children, whereas the children are connected by using a linked list. Hence, the essential elements of the matrix are preferably stored with the tree nodes rather than with the tree edges. Naturally, we store A_{ii}, U_i and V_i in a leaf node i. For Σ_{ij}, we store it in the parent node l of i and j. Hence, if a nonleaf node l has s children,

the number of Σ_{ij}'s stored in l is $s(s-1)$. We store each W_{ki} in the child node k. Figure 2.2 illustrates the storage for the example matrix in Figure 2.1.

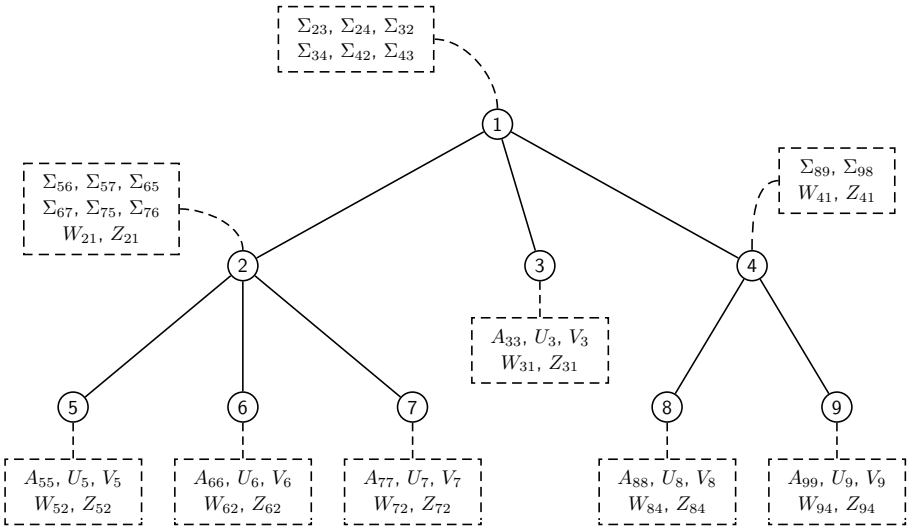


FIG. 2.2. *Data stored in the tree of Figure 2.1.*

With this scheme, the required storage is

$$\underbrace{\sum_{i \text{ leaf}} n_i^2}_{\text{for } A_{ii}} + \underbrace{2nr}_{\text{for } U_i, V_i} + \underbrace{\sum_{l} s_l(s_l-1)r^2}_{\text{for } \Sigma_{ij}} + \underbrace{2(m-1)r^2}_{\text{for } W_{ki}, Z_{ki}},$$

where n_i is the number of indices a node i contains (that is, $n_i = |I_i|$), s_l is the number of children a node l has, and m is the total number of nodes in the tree. In all subsequent cost analyses, we assume that the tree is a full and complete s-ary tree and that the indices are always partitioned in balance such that each leaf node contains n_0 indices. This assumption simplifies the analysis and we see that the storage cost can be simplified to

$$O(n \cdot (r + n_0 + s^2 r^2 / n_0)),$$

which is linear in n, assuming that r, n_0, and s are constants.

Another useful angle for analyzing the storage requirement (and the time requirement of the algorithms presented later) is to cast the cost as a function of the tree size, that is, the number of tree nodes. In the balanced case, the number of tree nodes is $O(n \cdot s/n_0)$. Then, any cost proportional to the tree size is linear in n.

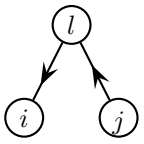
2.2. Augmenting the data structure. We sometimes need to store additional information in the tree, thus augmenting the data structure. In every matrix algorithm we discuss later, augmentation is necessary. In such cases, it is crucial to verify the additional cost. A cost linear in n is desirable. For example, if the storage added to each tree node is constant, then the total additional storage is also $O(n)$. We assert here that for all the operations studied in this paper, augmenting the tree data structure maintains the linear storage cost.

3. Matrix-vector multiplication. We now start to present the matrix operations that the data structure supports. The first one is computing matrix-vector products.

The multiplication of Ab conceptually consists of two parts: multiplying $A_{ii}b_i$ for all leaf nodes i and multiplying $A_{ij}b_j$ for all pairs of sibling nodes i and j. The first part is straightforward. We can interpret the second part

$$A_{ij}b_j = U_i \Sigma_{ij} V_j^* b_j \tag{3.1}$$

as a wedge visit of the tree



that is, visiting the nodes in the order $j \to l \to i$, by reading the subscripts of (3.1) from right to left. In reality, U_i may not be explicitly stored if i is not a leaf node, and similarly for V_j. Hence, the wedge visit must be extended to the leaf descendants of i and j. Specifically, for every leaf descendant k of i, we write the I_k block of the vector $U_i \Sigma_{ij} V_j^* b_j$ as

$$U_i(I_k,:)\Sigma_{ij}V_j^*b_j = U_i(I_k,:)\Sigma_{ij}\left(\sum_p V_j(I_p,:)^* b_p\right),$$

where p ranges from all leaf descendants of j. Denote by $Ch(\cdot)$ the set of children of a node, and let the path connecting i and k be $(i, k_1, k_2, \ldots, k_t, k)$. Then, using the change-of-basis matrices, we further expand the above quantity as

$$(U_k W_{kk_t} \cdots W_{k_2k_1} W_{k_1 i})\, \Sigma_{ij} \left(\sum_{p_1 \in Ch(j)} Z_{p_1 j}^* \sum_{p_2 \in Ch(p_1)} Z_{p_2 p_1}^* \cdots \sum_{p \in Ch(p_s)} Z_{p p_s}^* V_p^* b_p \right). \tag{3.2}$$

Here, for simplicity, we assume that all the leaf descendants p are on the same tree level. The situation that the p's lie on different levels necessarily makes the expression too complicated but shows no additional insights. The use of this expression is that it is suggestive on the calculation steps, if one reads it from right to left: For all leaf descendants p of j, we multiply $V_p^* b_p$, followed by a change of basis to obtain $Z_{pp_s}^* V_p^* b_p$. We sum this result over all the p's that share the same parent p_s and move the calculation one level above. We continuously move up until reaching the node j, where the expression inside the larger pair of parentheses in (3.2) has been computed. Then we perform the wedge visit, premultiply Σ_{ij}, cross over the parent of j, and reach node i. From node i, we descend to node k level by level, in each of which we premultiply a change-of-basis matrix W_{**}. When we land on k, we premultiply U_k and complete the calculation of (3.2).

3.1. Formal algorithm. Based on the preceding discussion, we now describe the full algorithm for computing $y = Ab$, where y is partitioned in the same manner as b.

First, we augment the tree data structure by adding the storage of two length-r vectors, c_i and d_i, in each node i. The additional storage maintains the linear complexity of the data structure.

Next, the algorithm consists of two passes, an upward pass and a downward pass. Conceptually, connecting the two passes are wedge visits, which in reality we put in the upward pass. In this pass, we start with each leaf i by setting $c_i = V_i^* b_i$; we also initialize $y_i = A_{ii} b_i$. Each time when we move up one level of the tree, say, when the child level is j and the parent level is i, we perform a change of basis on c_j and accumulate the results to c_i; that is, $c_i = \sum_{j \in Ch(i)} Z_{ji}^* c_j$. Meanwhile, we perform a wedge visit and compute $d_k = \Sigma_{ki} c_i$ for all siblings k of i. This wedge visit is performed on all levels, every time a c_i is computed. We continuously move up across levels until reaching the root.

In the downward pass, for every node i in the same level, we premultiply the change-of-basis matrix W_{ji} to d_i and accumulate it to d_j for all children j of i. We descend level by level, until finally reaching the leaves. On a leaf node i, we premultiply U_i to d_i and accumulate the result to y_i. This concludes the calculation of the overall vector y.

The upward pass and the downward pass are, in fact, a postorder and a preorder tree traversal, respectively. Algorithm 1 presents the traversals by using recursions for ease of understanding. One can rewrite them in an iterative fashion, if the depth of the recursion poses a problem for the limited stack size of a computer program.

Algorithm 1 Computing $y = Ab$

```
1: Initialize c_i ← 0, d_i ← 0 for each node i of the tree
2: UPWARD(root)
3: DOWNWARD(root)

4: subroutine UPWARD(i)
5:     if i is leaf then
6:         c_i ← V_i^* b_i;   y_i ← A_ii b_i
7:     else
8:         for all children j of i do
9:             UPWARD(j)
10:            c_i ← c_i + Z_ji^* c_j, if i is not root
11:        end for
12:    end if
13:    if i is not root then
14:        for all siblings k of i do d_k ← d_k + Σ_ki c_i end for
15:    end if
16: end subroutine

17: subroutine DOWNWARD(i)
18:    if i is leaf then y_i ← y_i + U_i d_i and return end if
19:    for all children j of i do
20:        d_j ← d_j + W_ji d_i, if i is not root
21:        DOWNWARD(j)
22:    end for
23: end subroutine
```

3.2. Time cost. Because the algorithm consists of two tree traversals, where the time cost of the computation when visiting each node can be bounded by a constant

$$O(\underbrace{n_0^2}_{\text{for } y_i} + \underbrace{n_0 r}_{\text{for } c_i, y_i} + \underbrace{r^2}_{\text{for } c_i, d_k, d_j}),$$

we conclude that the overall time cost is linear in the tree size, that is, linear in n.

4. Matrix inversion. In this section, we consider computing $\tilde{A} = A^{-1}$. Through construction, we will see that $\tilde{A}$ is also a recursively low-rank compressed matrix, having exactly the same structure as A, with the "rank" r unchanged. Hence, we add a tilde to all the elements of A (i.e., A_{**}, U_*, V_*, Σ_{**}, W_{**}, and Z_{**}) to denote the elements of $\tilde{A}$. Then, the core of the matrix inversion is to construct $\tilde{A}_{**}$, $\tilde{U}_*$, $\tilde{V}_*$, $\tilde{\Sigma}_{**}$, $\tilde{W}_{**}$, and $\tilde{Z}_{**}$. A direct consequence is that solving the linear system $A^{-1}b$ amounts to computing the matrix-vector product $\tilde{A}b$, which can be carried out by using Algorithm 1. We note that using $\tilde{A}$ as a preconditioner in an iterative solver can improve the accuracy of the linear system solution; more details are given in Section 8.

To facilitate the derivation of the algorithm, we first show in Figure 4.1 a schematic arrangement for the tree node indices. This arrangement is used almost throughout the section. We let i be some node of current interest. Two of its children are denoted as j and j'; but it may have more children. The node j further has two children k and l; and again, j may have more than two children. The nodes j', k, and l might be leaves or might span subtrees.

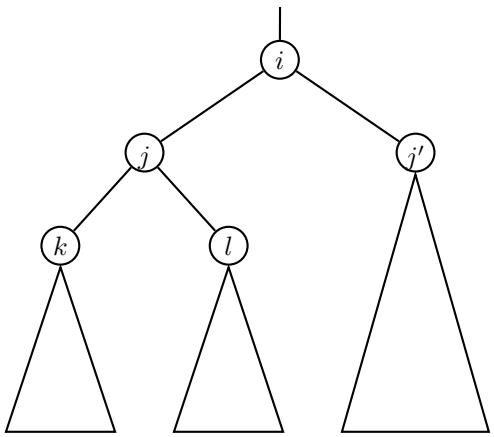


FIG. 4.1. *Schematic tree node arrangements for the discussions in Section 4. A subtree rooted at i is displayed. Each nonleaf node may have more than two children, but only two are drawn.*

4.1. $\tilde{A}$ is recursively low-rank compressed. To understand that $\tilde{A}$ and A have the same structure, we first present the following result.

PROPOSITION 4.1. *Let j and j' denote two children of a node i and let all A_{jj}'s be invertible. Define block matrix Λ and block-diagonal matrix Ξ where*

$$(j, j') \text{ block of } \Lambda = \Sigma_{jj'} \text{ if } j \neq j'; \text{ otherwise} = 0,$$
$$(j, j) \text{ block of } \Xi = V_j^* A_{jj}^{-1} U_j,$$

and a matrix $H = I + \Lambda\Xi$. If H is invertible, then A_{ii} is invertible with

$$(j,j') \text{ block of } A_{ii}^{-1} = \begin{cases} A_{jj}^{-1} - A_{jj}^{-1} U_j D_{jj} V_j^* A_{jj}^{-1} & j = j' \\ -A_{jj}^{-1} U_j D_{jj'} V_{j'}^* A_{j'j'}^{-1} & j \neq j', \end{cases} \tag{4.1}$$

where $D = H^{-1}\Lambda$ is a block matrix having the same block structure as H and Λ.

Proof. We write

$$A_{ii} = \begin{bmatrix} A_{jj} & & \\ & \ddots & \\ & & A_{j'j'} \end{bmatrix} + \begin{bmatrix} U_j & & \\ & \ddots & \\ & & U_{j'} \end{bmatrix} \Lambda \begin{bmatrix} V_j^* & & \\ & \ddots & \\ & & V_{j'}^* \end{bmatrix},$$

where Λ is the block matrix defined in the proposition. Then, applying the Sherman-Morrison-Woodbury formula we obtain

$$A_{ii}^{-1} = \begin{bmatrix} A_{jj}^{-1} & & \\ & \ddots & \\ & & A_{j'j'}^{-1} \end{bmatrix} - \begin{bmatrix} A_{jj}^{-1} U_j & & \\ & \ddots & \\ & & A_{j'j'}^{-1} U_{j'} \end{bmatrix} D \begin{bmatrix} V_j^* A_{jj}^{-1} & & \\ & \ddots & \\ & & V_{j'}^* A_{j'j'}^{-1} \end{bmatrix},$$

where D is the block matrix defined in the proposition. □

Proposition 4.1 is used for an induction proof of the claim that $\tilde{A}$ is a recursively low-rank compressed matrix. The essential idea is that if A_{jj}^{-1} has been constructed for all children j of i, then we can construct A_{ii}^{-1} to preserve the compressed structure. In particular, the (j,j') blocks of A_{ii}^{-1} can be naturally defined when $j \neq j'$, and the (j,j) blocks of A_{ii}^{-1} are updated from A_{jj}^{-1}.

Noting that Proposition 4.1 can be applied one level lower, that is,

$$(k,l) \text{ block of } A_{jj}^{-1} = \begin{cases} A_{kk}^{-1} - A_{kk}^{-1} U_k D_{kk} V_k^* A_{kk}^{-1} & k = l \\ -A_{kk}^{-1} U_k D_{kl} V_l^* A_{ll}^{-1} & k \neq l, \end{cases}$$

we multiply A_{jj}^{-1} with U_j and obtain

$$k \text{ block of } A_{jj}^{-1} U_j = A_{kk}^{-1} U_k \left[W_{kj} + \sum_{l \in Ch(j)} (-D_{kl})(V_l^* A_{ll}^{-1} U_l) W_{lj} \right].$$

Similarly, we can obtain the formula for the k block of $(A_{jj}^{-1})^* V_j$. Thus, if we assume that for all children j of a node i, A_{jj} is a recursively low-rank compressed matrix with

$$\tilde{U}_k = A_{kk}^{-1} U_k, \quad \tilde{V}_k = A_{kk}^{-1} V_k, \quad \forall\ k \in Ch(j)$$
$$\tilde{\Sigma}_{kl} = -D_{kl}, \quad \forall\ k,l \in Ch(j),\ k \neq l,$$

and if we define $\tilde{\Theta}_l = V_l^* A_{ll}^{-1} U_l$ for all $l \in Ch(j)$, then for one level higher we have

$$\tilde{U}_j = A_{jj}^{-1} U_j, \quad \tilde{W}_{kj} = W_{kj} + \sum_{l \in Ch(j)} \tilde{\Sigma}_{kl} \tilde{\Theta}_l W_{lj}, \quad \forall\ k \in Ch(j),\ j \in Ch(i)$$
$$\tilde{V}_j = (A_{jj}^{-1})^* V_j, \quad \tilde{Z}_{kj} = Z_{kj} + \sum_{l \in Ch(j)} \tilde{\Sigma}_{lk}^* \tilde{\Theta}_l^* Z_{lj}, \quad \forall\ k \in Ch(j),\ j \in Ch(i)$$
$$\tilde{\Sigma}_{jj'} = -D_{jj'}, \quad \forall\ j,j' \in Ch(i),\ j \neq j'.$$

Furthermore, $\tilde{\Theta}_j$ can be computed from $\tilde{\Theta}_k$ for all $k \in Ch(j)$:

$$\tilde{\Theta}_j = V_j^* A_{jj}^{-1} U_j = \sum_{k \in Ch(j)} Z_{kj}^* V_k^* A_{kk}^{-1} U_k \tilde{W}_{kj} = \sum_{k \in Ch(j)} Z_{kj}^* \tilde{\Theta}_k \tilde{W}_{kj}.$$

So far, we have seen how $\tilde{U}_j$, $\tilde{V}_j$, $\tilde{W}_{kj}$, $\tilde{Z}_{kj}$, and $\tilde{\Sigma}_{jj'}$ are defined by induction. The $\tilde{U}_*$, $\tilde{V}_*$, $\tilde{W}_{**}$, and $\tilde{Z}_{**}$'s in lower levels need not change. It remains to modify the $\tilde{\Sigma}_{**}$'s in lower levels to ensure that the recursively low-rank compressed structure of A_{ii}^{-1} is correct. For this, we return to (4.1), in particular, the first case $j = j'$. This case indicates that for any pair of sibling nodes k and l that are descendants (not necessarily children) of j, the computed $\tilde{\Sigma}_{kl}$ for A_{jj}^{-1} need only add a correction term in order that $\tilde{\Sigma}_{kl}$ becomes correct for A_{ii}^{-1}. Specifically, we write

$$(k,l) \text{ block of } A_{ii}^{-1} = [(k,l) \text{ block of } A_{jj}^{-1}] + \tilde{U}_j(I_k,:)(-D_{jj})\tilde{V}_j(I_l,:)^*.$$

Let $(j, j_1, \ldots, j_s)$ be the path connecting j and the parent j_s of k and l. If we define $\tilde{\Sigma}_{jj} = -D_{jj}$, then the above equality is expanded as

$$(k,l) \text{ block of } A_{ii}^{-1} = [(k,l) \text{ block of } A_{jj}^{-1}] + \tilde{U}_k \tilde{W}_{kj_s} \cdots \tilde{W}_{j_1 j} \tilde{\Sigma}_{jj} \tilde{Z}_{j_1 j}^* \cdots \tilde{Z}_{l j_s}^* \tilde{V}_l^*.$$

Hence, the correction update of $\tilde{\Sigma}_{kl}$ is simply

$$\tilde{\Sigma}_{kl} \leftarrow \tilde{\Sigma}_{kl} + \tilde{W}_{kj_s} \cdots \tilde{W}_{j_1 j} \tilde{\Sigma}_{jj} \tilde{Z}_{j_1 j}^* \cdots \tilde{Z}_{l j_s}^*. \tag{4.2}$$

This formula updates $\tilde{\Sigma}_{kl}$ for all siblings k and l that are descendants of j, including the case $k = l$.

Clearly, in the base case, for each leaf node k, we initially have $\tilde{A}_{kk} = A_{kk}^{-1}$. Then, we in effect have completed the induction of the proof that $\tilde{A}$ is a recursively low-rank compressed matrix. The proof, a constructive one, in addition defines all the elements $\tilde{A}_{**}$, $\tilde{U}_*$, $\tilde{V}_*$, $\tilde{\Sigma}_{**}$, $\tilde{W}_{**}$ of $\tilde{A}$ and gives the explicit computation formulas for them.

4.2. Improved computation: part 1. We could have used the formulas in the preceding subsection to straightforwardly compute $\tilde{A}$ in a recursive manner. One pitfall of doing so, however, is that the calculation may cause severe numerical instability. To see this, consider the base case where $\tilde{A}_{kk}$ is obtained initially as the inverse of A_{kk}. When A_{kk} is ill-conditioned, the numerical error of $\tilde{A}_{kk}$ is nonnegligible and will accumulate and amplify across levels.

To encourage a better numerical behavior, we utilize the previously vacant placeholders Σ_{ii}. Recall that in the definition of a recursively low-rank compressed matrix, Σ_{ij} is defined only for a pair of sibling nodes i and j. We now in addition define Σ_{ii} for all nodes i. The content of Σ_{ii} is arbitrary, as long as it fulfills a preconditioning goal. Let

$$A_{ii} = B_{ii} + U_i \Sigma_{ii} V_i^*. \tag{4.3}$$

Conceptually speaking, Σ_{ii} is some $r \times r$ matrix such that B_{ii} is better conditioned than A_{ii}. In practice, when one constructs the whole matrix A (as we do in Section 6), one should consider additionally defining the Σ_{ii}'s if A will be inverted later. For now, the specific content of Σ_{ii} is irrelevant, and we assume only that it exists with (4.3).

The following result is parallel with Proposition 4.1. In the proposition, the notations Λ, Ξ, H, and D are redefined. All subsequent discussions refer to the new notations.

PROPOSITION 4.2. *Let j and j' denote two children of a node i, and let all B_{jj}'s be invertible. Define block matrix Λ and block-diagonal matrix Ξ where*

$$(j,j') \text{ block of } \Lambda = \Sigma_{jj'} - W_{ji}\Sigma_{ii}Z_{j'i}^*,$$
$$(j,j) \text{ block of } \Xi = V_j^* B_{jj}^{-1} U_j,$$

and a matrix $H = I + \Lambda\Xi$. If H is invertible, then B_{ii} is invertible with

$$(j,j') \text{ block of } B_{ii}^{-1} = \begin{cases} B_{jj}^{-1} - B_{jj}^{-1}U_j D_{jj} V_j^* B_{jj}^{-1} & j = j' \\ -B_{jj}^{-1}U_j D_{jj'} V_{j'}^* B_{j'j'}^{-1} & j \neq j', \end{cases} \tag{4.4}$$

where $D = H^{-1}\Lambda$ is a block matrix having the same block structure as H and Λ.

Proof. We write A_{ii} in the following two forms, which naturally equate:

$$B_{ii} + U_i\Sigma_{ii}V_i^* \quad \text{and} \quad \begin{bmatrix} B_{jj} & & \\ & \ddots & \\ & & B_{j'j'} \end{bmatrix} + \begin{bmatrix} U_j & & \\ & \ddots & \\ & & U_{j'} \end{bmatrix} \underline{\Lambda} \begin{bmatrix} V_j^* & & \\ & \ddots & \\ & & V_{j'}^* \end{bmatrix},$$

where $\underline{\Lambda}$ is a block matrix with the (j,j') block being $\Sigma_{jj'}$. Then, clearly,

$$B_{ii} = \begin{bmatrix} B_{jj} & & \\ & \ddots & \\ & & B_{j'j'} \end{bmatrix} + \begin{bmatrix} U_j & & \\ & \ddots & \\ & & U_{j'} \end{bmatrix} \Lambda \begin{bmatrix} V_j^* & & \\ & \ddots & \\ & & V_{j'}^* \end{bmatrix}, \tag{4.5}$$

where the block matrix Λ is defined in the proposition. Thus, we conclude the proposition by applying the Sherman-Morrison-Woodbury formula on B_{ii}. □

The use of Proposition 4.2 is that we can apply almost the same rationale as in the preceding subsection to derive the recursive formulas for constructing B_{ii}^{-1} level by level, such that eventually when i is the root, $A^{-1} = A_{ii}^{-1}$ is easily obtained from B_{ii}^{-1}. We say "almost the same rationale" but not "exactly the same" because we need one more tool to handle the concluding case. See the following proposition, whose verification is straightforward. The use of the proposition is to apply to the root node.

PROPOSITION 4.3. *For any node i,*

$$A_{ii}^{-1} = B_{ii}^{-1} - B_{ii}^{-1}U_i[(I + \Sigma_{ii}V_i^*B_{ii}^{-1}U_i)^{-1}\Sigma_{ii}]V_i^*B_{ii}^{-1},$$

if all the involved inverses are well defined.

We are now ready to sketch the recursive computational process. Borrowing the derivations in the preceding subsection, we define for all nodes k

$$\tilde{U}_k = B_{kk}^{-1}U_k, \quad \tilde{V}_k = (B_{kk}^{-1})^* V_k, \quad \tilde{\Theta}_k = V_k^* B_{kk}^{-1} U_k. \tag{4.6}$$

Then, when j is the parent of k, we have

$$\tilde{U}_j(I_k,:) = \tilde{U}_k\tilde{W}_{kj}, \quad \text{where} \quad \tilde{W}_{kj} = W_{kj} + \sum_{l\in Ch(j)} \tilde{\Sigma}_{kl}\tilde{\Theta}_l W_{lj}, \tag{4.7}$$

$$\tilde{V}_j(I_k,:) = \tilde{V}_k\tilde{Z}_{kj}, \quad \text{where} \quad \tilde{Z}_{kj} = Z_{kj} + \sum_{l\in Ch(j)} \tilde{\Sigma}_{lk}^*\tilde{\Theta}_l^* Z_{lj}, \tag{4.8}$$

$$\tilde{\Sigma}_{jj'} = -D_{jj'}, \quad \text{where block matrix } D \text{ is defined in Proposition 4.2.} \tag{4.9}$$

Furthermore, $\tilde{\Theta}_j$ can be computed from the $\tilde{\Theta}_k$'s as

$$\tilde{\Theta}_j = \sum_{k \in Ch(j)} Z_{kj}^* \tilde{\Theta}_k \tilde{W}_{kj}. \tag{4.10}$$

In the base case, for all leaf nodes k we intialize $\tilde{A}_{kk} \leftarrow B_{kk}^{-1}$ and compute $\tilde{U}_k$, $\tilde{V}_k$, and $\tilde{\Theta}_k$ according to (4.6). We then recursively compute $\tilde{W}_{**}$, $\tilde{Z}_{**}$, and $\tilde{\Sigma}_{**}$ based on (4.7)–(4.9), with the help of (4.10). We move up the tree level by level until reaching the root node i. At the root, we must in addition compute

$$\tilde{\Sigma}_{ii} = -(I + \Sigma_{ii}\tilde{\Theta}_i)^{-1}\Sigma_{ii},$$

according to Proposition 4.3. At this point, the computation of A^{-1} is almost complete except for the corrections of the $\tilde{\Sigma}_{**}$'s.

4.3. Improved computation: part 2. Recall (4.2) in Section 4.1. When i is the current node of interest and j is a child of i, one performs a correction $\tilde{\Sigma}_{kl} \leftarrow \tilde{\Sigma}_{kl} + \tilde{E}_{kl}$ for all descendant pairs k, l of j, where the correction term is

$$\tilde{E}_{kl} = \tilde{W}_{kj_s} \cdots \tilde{W}_{j_1 j} \tilde{\Sigma}_{jj} \tilde{Z}_{j_1 j}^* \cdots \tilde{Z}_{lj_s}^*.$$

For a fixed pair k, l, this correction must be applied every time we construct B_{ii}^{-1}, for all nodes i that are at least two levels above k, l. The repeating corrections for different i's moving toward the root is time consuming; hence, we consider consolidating the corrections. To this end, we let initially

$$\tilde{E}_{kl} \leftarrow \tilde{W}_{kj} \tilde{\Sigma}_{jj} \tilde{Z}_{lj}^*$$

where k, l refer to a pair of children of j, when i is the current node of interest. After the upward calculation reaches the tree root, we perform a downward cascade correction. Suppose $\tilde{E}_{jj}$ has accumulated all the corrections to $\tilde{\Sigma}_{jj}$. Then clearly with the update

$$\tilde{E}_{kl} \leftarrow \tilde{E}_{kl} + \tilde{W}_{kj} \tilde{E}_{jj} \tilde{Z}_{lj}^*,$$

$\tilde{E}_{kl}$ will accumulate all the corrections to $\tilde{\Sigma}_{kl}$. Hence, we move down the tree and update $\tilde{E}_{**}$ in this recursive manner. Each $\tilde{E}_{**}$ is updated only once, and hence the corresponding $\tilde{\Sigma}_{**}$ suffices to be corrected only once. When we finish visiting all the tree nodes, all the corrections have been completed. At a leaf node k, we in addition update $\tilde{A}_{kk} \leftarrow \tilde{A}_{kk} + \tilde{U}_k \tilde{\Sigma}_{kk} \tilde{V}_k^*$ and conclude the overall calculation.

4.4. Formal algorithm. We summarize the discussions in the preceding subsections and present the formal algorithm here.

We first augment the tree data structure. When we use $\tilde{A}$ to represent the inverted matrix, the augmented storage is posted on $\tilde{A}$. These additional contents are

1. $\tilde{\Theta}_i \in \mathbb{C}^{r \times r}$, for all nodes i, and
2. $\tilde{E}_{jj'} \in \mathbb{C}^{r \times r}$, for all sibling pairs j, j', including $j = j'$.

Furthermore, we require the additional storage of Σ_{ii} in A and $\tilde{\Sigma}_{ii}$ in $\tilde{A}$ for all nodes i. Different from Σ_{ij} (resp. $\tilde{\Sigma}_{ij}$), which is stored in the parent node, Σ_{ii} (resp. $\tilde{\Sigma}_{ii}$) is stored in node i itself. Then, for consistency, $\tilde{E}_{jj'}$ is stored in the parent node of j and j' if $j \neq j'$, but it is stored in the node j itself if $j = j'$. Naturally, $\tilde{\Theta}_i$ is stored

with node i. Clearly, the extra storage maintains the linear complexity of the data structure.

Algorithm 2 presents the detailed steps. Similar to Algorithm 1, Algorithm 2 also comprises an upward and a downward pass, which are equivalent to a postorder and a preorder tree traversal, respectively. Hence, one is free to choose a recursive or an iterative implementation style.

Although the algorithm is derived based on Section 4.2, not surprisingly one sees that the discussions in Section 4.1 correspond to the special case where all Σ_{ii}'s are zero (that is, $A_{ii} = B_{ii}$). The Σ_{ii}'s are nonessential in representing the original matrix A; hence in theory they can be arbitrary. The role of Σ_{ii}'s is to encourage numerical stability by transforming the inversions on the A_{ii}'s to those on the better-conditioned B_{ii}'s. The B_{ii}'s do not explicitly appear in the algorithm.

Algorithm 2 Computing $\tilde{A} = A^{-1}$

UPWARD(`root`)
DOWNWARD(`root`)

subroutine UPWARD(i)
  if i is leaf **then**
    $\tilde{A}_{ii} \leftarrow (A_{ii} - U_i \Sigma_{ii} V_i^*)^{-1}$; $\tilde{U}_i \leftarrow \tilde{A}_{ii} U_i$; $\tilde{V}_i \leftarrow \tilde{A}_{ii}^* V_i$; $\tilde{\Theta}_i \leftarrow V_i^* \tilde{U}_i$
    return
  end if
  for all children j of i **do**
    UPWARD(j)
    $\tilde{W}_{kj} \leftarrow W_{kj} + \sum_{l \in Ch(j)} \tilde{\Sigma}_{kl} \tilde{\Theta}_l W_{lj}$ for all children k of j
    $\tilde{Z}_{kj} \leftarrow Z_{kj} + \sum_{l \in Ch(j)} \tilde{\Sigma}_{lk}^* \tilde{\Theta}_l^* Z_{lj}$ for all children k of j
    $\tilde{\Theta}_j \leftarrow \sum_{k \in Ch(j)} Z_{kj}^* \tilde{\Theta}_k \tilde{W}_{kj}$ **if** j is not leaf
  end for
  Compute D that is defined in Proposition 4.2
  for all children j, j' of i (including $j = j'$) **do** $\tilde{\Sigma}_{jj'} \leftarrow -D_{jj'}$ **end for**
  for all children j of i **do**
    $\tilde{E}_{kl} \leftarrow \tilde{W}_{kj} \tilde{\Sigma}_{jj} \tilde{Z}_{lj}^*$ for all children k, l of j (including $k = l$)
  end for
  if i is root **then**
    $\tilde{W}_{ji} \leftarrow W_{ji} + \sum_{j' \in Ch(i)} \tilde{\Sigma}_{jj'} \tilde{\Theta}_{j'} W_{j'i}$ for all children j of i
    $\tilde{Z}_{ji} \leftarrow Z_{ji} + \sum_{j' \in Ch(i)} \tilde{\Sigma}_{j'j}^* \tilde{\Theta}_{j'}^* Z_{j'i}$ for all children j of i
    $\tilde{\Theta}_i \leftarrow \sum_{j \in Ch(i)} Z_{ji}^* \tilde{\Theta}_j \tilde{W}_{ji}$
    $\tilde{\Sigma}_{ii} \leftarrow -(I + \Sigma_{ii} \tilde{\Theta}_i)^{-1} \Sigma_{ii}$
    $\tilde{E}_{jj'} \leftarrow \tilde{W}_{ji} \tilde{\Sigma}_{ii} \tilde{Z}_{j'i}^*$ for all children j, j' of i (including $j = j'$)
    $\tilde{E}_{ii} \leftarrow 0$
  end if
end subroutine

Continued in Algorithm 3...

Algorithm 3 Computing $\tilde{A} = A^{-1}$, continued from Algorithm 2

subroutine DOWNWARD(i)
  if i is leaf **then**
    $\tilde{A}_{ii} \leftarrow \tilde{A}_{ii} + \tilde{U}_i \tilde{\Sigma}_{ii} \tilde{V}_i^*$
  else
    for all children j, j' of i (including $j = j'$) **do**
      $\tilde{E}_{jj'} \leftarrow \tilde{E}_{jj'} + \tilde{W}_{ji} \tilde{E}_{ii} \tilde{Z}_{j'i}^*$
      $\tilde{\Sigma}_{jj'} \leftarrow \tilde{\Sigma}_{jj'} + \tilde{E}_{jj'}$
    end for
    for all children j of i **do** DOWNWARD(j) **end for**
  end if
end subroutine

4.5. Time cost. The time cost of Algorithm 2 is linear in n. To see this, note that the number of elements for each category: $\tilde{A}_{**}$, $\tilde{U}_*$, $\tilde{V}_*$, $\tilde{W}_*$, $\tilde{Z}_*$, $\tilde{\Theta}_*$, $\tilde{\Sigma}_{**}$, and $\tilde{E}_{**}$, is linear in the tree size. Furthermore, computing each element requires a constant time. Therefore, we conclude that the overall time is linear.

5. Determinant calculation. The computation of $\det(A)$ is a simple application of the Sylvester's determinant theorem.

PROPOSITION 5.1. *For all nodes i,*

$$\det(A_{ii}) = \det(I + \Sigma_{ii} \tilde{\Theta}_i) \det(B_{ii}).$$

In addition, if i is not a leaf,

$$\det(B_{ii}) = \det(H) \prod_{j \in Ch(i)} \det(B_{jj}),$$

where the matrix H is defined in Proposition 4.2.

Proof. The first equation is a direct consequence of the Sylvester's determinant theorem $\det(C+DE) = \det(C)\det(I+EC^{-1}D)$ on (4.3), whereas the second equation is a result of the same theorem on (4.5). □

Hence, $\det(A)$ comes almost for free with the calculation of A^{-1}. We attach to each tree node i a scalar value δ_i, defined as follows:

$$\delta_i = \begin{cases} \det(B_{ii}) & \text{if } i \text{ is a leaf node,} \\ \det(H) & \text{if } i \text{ is neither a leaf nor the root,} \\ \det(H)\det(I + \Sigma_{ii}\tilde{\Theta}_i) & \text{if } i \text{ is the root.} \end{cases}$$

Then, $\det(A)$ is simply the product of the δ_i's in all tree nodes i.

5.1. Formal algorithm. It is well known that the determinant of a matrix easily overflows or underflows. Hence, a better approach representing the determinant is to take logarithm. Let $\delta = \det(A)$. We use two values to represent a complex δ:

$$\log|\delta| \quad \text{and} \quad \arg(\delta),$$

where $|\cdot|$ and $\arg(\cdot)$ denote the magnitude and the argument of a complex number, respectively. The argument allows 2π ambiguity. Then, we augment the data structure

by adding the storage of $\log|\delta_i|$ and $\arg(\delta_i)$ to each node i and compute that

$$\log|\delta| = \sum_i \log|\delta_i| \quad \text{and} \quad \arg(\delta) = \sum_i \arg(\delta_i).$$

In Algorithm 4, we present the calculation of $\log|\delta|$ and $\arg(\delta)$ by using a post-order tree traversal, which can be rewritten in any form of tree traversal, as along as the traversal visits all the tree nodes.

Algorithm 4 Computing $\delta = \det(A)$

```
Patch Algorithm 2:
    Line 6: Store log|δ_i| and arg(δ_i) before return, where δ_i = det(A_ii − U_i Σ_ii V_i^*)
    Line 14: Store log|δ_i| and arg(δ_i), where δ_i = det(H)
    Line 23: Update log|δ_i| and arg(δ_i), where δ_i ← δ_i · det(I + Σ_ii Θ̃_i)
Initialize log|δ| ← 0, arg(δ) ← 0
UPWARD(root)

subroutine UPWARD(i)
    if i is leaf then
        log|δ| ← log|δ| + log|δ_i|;  arg(δ) ← arg(δ) + arg(δ_i)
    else
        for all children j of i do
            UPWARD(j)
            log|δ| ← log|δ| + log|δ_j|;  arg(δ) ← arg(δ) + arg(δ_j)
        end for
    end if
end subroutine
```

5.2. Time cost. Clearly, the time cost is linear in the tree size, that is, $O(n)$.

6. Generation of compressed matrix. We have completed presenting the proposed data structure and several operations it supports. In this section, we consider how a recursively low-rank compressed matrix A is instantiated. Here, we discuss two separate examples, each of which serves a purpose. To avoid excessive technicalities and repetitions with existing methods, we organize the technical details of the examples in the appendix.

The first example constructs a random, complex, and non-Hermitian matrix. The constructed matrix complies only with the recursively low-rank structure, but it does not necessarily represent a compression of a kernel matrix. Such a matrix can be used for empirically verifying the correctness of the developed algorithms and testing the impact of matrix conditioning on the numerical behavior. This matrix also demonstrates that the proposed data structure is tied to neither a real symmetric matrix nor a positive definite matrix. Simply speaking, we first generate a random tree and then the random elements in each tree node. For algorithmic details, see Appendix A.

The second example constructs a compressed form of a kernel matrix with a given kernel function. This example shows how the proposed data structure is used in practice. The two parts of the construction are tree generation and matrix approximation. Each part can be achieved with a considerable number of choices. We consider, for the former part, using a binary k-d tree [6] to partition the data points, whereas for the latter part, using Chebyshev approximation to derive the low-rank compression [11].

We note that this construction is only one among many other possibilities proposed in the literature.

Particularly useful to our case of the Chebyshev approximation is that the definition of the Σ_{ii} matrices used for matrix inversion (cf. Section 4.2) shares the same formula for that of the Σ_{ij}'s. Empirically, defining the Σ_{ii}'s in this way significantly improves the conditioning situation. See Appendix B on how the definitions of the matrix components are derived.

7. Related methods. Matrices with a compressed structure similar to that of this work have been studied under at least four different names: fast multipole methods (FMM) [13], tree code methods [3], hierarchical matrices (H and H^2 matrices) [14], and hierarchically semiseparable matrices (HSS matrices) [9]. FMM was originally developed for computing a matrix-vector product, and thus the product can be used in an iterative solver for solving linear systems. The compressed structure was later explored for developing a direct solver for linear systems [22]. A majority of the FMM development does not use a language that is algebraically oriented; however, equivalence is easily identified. The S2M, M2M, M2L, L2L, and L2T operators in FMM corresponds to, in our case, the matrix components V_*, Z_{**}, Σ_{**}, W_{**}, and U_*, respectively. A distinction is that our work does not have the notion of neighboring boxes and interaction lists.

Early FMM methods exploit known series expansions for compressing a kernel matrix; hence, only specialized kernel functions are applicable. Later developments lean toward a kernel-independent nature. Ying et al. [28] replaced the analytic expansions and translations with equivalent density representation, which were computed by using Tikhonov regularizations for stability. Fong and Darve [11] proposed using Chebyshev approximation to derive the compression. Other algebraically oriented methods for compression include the use of SVD [12] and interpolative decompositions [10, 24, 23, 17]. Some recent work also explores the use of a black-box matrix-vector multiplication to reversely construct the matrix; see [20, 23]. We note that the work of this paper is orthogonal to how the matrix is compressed. We assume the matrix preexists with a compressed form, and we are concerned mainly with numerically stable algorithms for the matrix.

The concept of hierarchical matrices [14, 15, 7] (and also the ancestor—panel clustering [16]) lays a comprehensive framework for matrix algebra, where more operations other than matrix-vector multiplications and linear system solutions are considered. The basic H matrix does not impose the same-subspace requirement on the low-rank approximations (that is, no W_{**} and Z_{**}); hence, it is not surprising that both the storage and the floating point operations for various matrix algorithms have an additional $\log^\tau n$ factor for some small integer τ. The tree structure of hierarchical matrices in principle can be more complicated than ours; for example, an off-diagonal block may be further subdivided because of admissibility requirements. It is unclear in such a case, however, how the various matrix operations are performed. On the other hand, the H^2 matrices (a restricted subset of hierarchical matrices) are equivalent to the matrix structure of this work. However, it has not been demonstrated how the various matrix operations supported by H matrices are extended to H^2 matrices, except for matrix-vector multiplications [7].

The tree code method [3] was developed mainly for a fast kernel summation (equivalently a matrix-vector product), at approximately the same time FMM became known. The method is based on analytic series expansions of specialized kernels and entails an $O(n \log n)$ complexity. When the analytic expansion is amended for cluster-

cluster interactions [18], in the matrix form, it is equivalent to an H matrix.

We consider our work closest related to HSS matrices [9]. The only difference from the data structure point of view is that we incorporate a more general tree than the binary HSS tree. From the algorithmic point of view, however, the development of HSS matrices leans toward using factorizations [8, 26] for solving linear systems or for composing preconditioners [27], whereas we directly invert the matrix. Moreover, we describe the algorithms by using a graph theoretic language, which simplifies the index notations used for HSS matrices in general.

8. Numerical experiments. In this section we demonstrate a comprehensive set of experiments, focusing on numerical stability and computational costs. The program is written in C++, where the basic matrix operations are called from BLAS and LAPACK. We still use Φ to denote a kernel matrix and A a recursively low-rank compressed matrix. We add a tilde to denote matrix inverse (e.g., $\tilde{\Phi}$ and $\tilde{A}$). We sometimes need a fine distinction between the matrix stored in the tree data structure and one converted to an explicit dense matrix form. Then, we add a subscript t for the former case and a subscript m for the latter case (e.g., A_t and A_m). We also distinguish the matrix inverses computed by different algorithms with subscripts: "Alg2" means an inverse computed by using Algorithm 2, "Kry" means an inverse computed by using a Krylov solver with the preconditioner $\tilde{A}$ on every column of the identity matrix, and "LU" means an inverse computed based on LU factorization with partial pivoting. We use x to denote the vector solution of a linear system, with the aforementioned subscripts to distinguish results from different methods. The machine precision `eps = 2.2e-16`.

8.1. Random complex matrix. Table 8.1 shows the results of a randomly generated compressed matrix (cf. Appendix A). The matrix has a size approximately equal to $2,000$ with $r = 10$. The matrix is complex and non-Hermitian. It is ill conditioned, with a 2-norm condition number `3.8e+08`. The relative error `1.9e-15` $\approx$ `eps` on the top left corner of the table indicates that the tree form and the matrix form of A can be used interchangeably (here, b is a random vector). Then, by performing an LU factorization of A, we see that the inverse $\tilde{A}_{\mathrm{LU}}$ reaches an accuracy `4.0e-08`. On the other hand, we perform the inversion by using the proposed algorithm to obtain $\tilde{A}_{\mathrm{Alg2}}$. This inverse attains an accuracy `2.7e-06`, which is moderately close to that of $\tilde{A}_{\mathrm{LU}}$. The preconditioned matrix $A\tilde{A}_{\mathrm{Alg2}}$, however, already has an almost perfect condition number. Applying GMRES with the preconditioner $\tilde{A}_{\mathrm{Alg2}}$, we see that it converges in two iterations for each column of the identity matrix, and the computed inverse $\tilde{A}_{\mathrm{Kry}}$ has an accuracy `5.2e-07` that is further close to that of $\tilde{A}_{\mathrm{LU}}$.

We compare the determinants computed according to the tree form A_t and that according to the matrix form A_m. The relative difference `2.5e-14` indicates that the log magnitudes of the two determinants are sufficiently close, and the differences `1.1e-10` and `1.0e+00` collectively show that the argument of the two determinants are close, too. If the two matrices are real, the cosine term is particularly useful for verifying whether the determinant has flipped sign.

We also compare the diagonals of the inverse of A, one computed from Algorithm 2 and the other from an LU factorization. We see that the 2-norm error and the trace error are both on the order of `1e-11`; thus they agree reasonably well.

8.2. Indefinite kernel. After the experiment with a random matrix, we start to work on kernel matrices. They are compressed by using the technique discussed in

TABLE 8.1
Computation results for a random matrix. $n = 2,076$. Parameters: budgeted number of leaves $= 50$, probability $p = 0.5$, range of number of children $[3, 5]$, range of leaf size $[30, 50]$, rank $r = 10$.

$\frac{\|A_t b - A_m b\|_2}{\|A_m b\|_2}$	1.9e-15	$\frac{\text{abs}[\log\|\det(A_t)\| - \log\|\det(A_m)\|]}{\text{abs}[\log\|\det(A_m)\|]}$	2.5e-14
$\text{cond}_2(A)$	3.8e+08	$\tan[\arg(\det(A_t)) - \arg(\det(A_m))]$	1.1e-10
$\text{cond}_2(A\tilde{A}_{\text{Alg2}})$	1.0e+00	$\cos[\arg(\det(A_t)) - \arg(\det(A_m))]$	1.0e+00
$\|A\tilde{A}_{\text{LU}} - I\|_2$	4.0e-08	$\frac{\|\text{diag}(\tilde{A}_{\text{Alg2}}) - \text{diag}(\tilde{A}_{\text{LU}})\|_2}{\|\text{diag}(\tilde{A}_{\text{LU}})\|_2}$	6.4e-11
$\|A\tilde{A}_{\text{Alg2}} - I\|_2$	2.7e-06		
$\|A\tilde{A}_{\text{Kry}} - I\|_2$	5.2e-07	$\frac{\|\text{tr}(\tilde{A}_{\text{Alg2}}) - \text{tr}(\tilde{A}_{\text{LU}})\|}{\|\text{tr}(\tilde{A}_{\text{LU}})\|}$	6.6e-11
Krylov solver	GMRES		
iter. per r.h.s.	2		

Appendix B. Table 8.2 shows the results for a one-dimensional multiquadric kernel

$$\phi(\boldsymbol{x}, \boldsymbol{y}) = (\|\boldsymbol{x} - \boldsymbol{y}\|_2^2 + c^2)^{1/2}. \tag{8.1}$$

A multiquadric kernel yields a symmetric matrix that is only conditionally positive definite. We use $n = 10^3$ points uniformly distributed on $[0, 1]$ to generate the matrix Φ. By using a Chebyshev order $k = 15$, the compressed matrix A yields a relative accuracy 4.9e-09. From the table we see that Φ and A have approximately the same condition number (which is not always the case, as we will see in a later example) and that again $A\tilde{A}_{\text{Alg2}}$ is almost perfectly conditioned. The accuracy of the inverse of A computed by using the proposed algorithm is close to that by using the LU factorization, and a Krylov method further improves the accuracy. Note that even though A is not positive definite, the preconditioned conjugate gradient method still converges (and in one iteration only). Moreover, note that even though A is close to Φ and the inverse $\tilde{A}_{\text{Kry}}$ of A is numerically accurate, $\tilde{A}_{\text{Kry}}$ is not necessarily an accurate inverse of Φ, as the 2-norm of $\Phi\tilde{A}_{\text{Kry}} - I$ indicates. The determinant and diagonal results in the table suggest that the accuracies are only moderate if one compares A with the original matrix Φ, in contrast to the good accuracy in the previous experiment, where the tree form of A is compared with the matrix form.

8.3. Positive definite kernel. Table 8.3 shows the results for a two-dimensional Matérn kernel (with nugget)

$$\phi(\boldsymbol{x}, \boldsymbol{y}) = M_\nu(\hat{\boldsymbol{x}} - \hat{\boldsymbol{y}}) + \delta(\hat{\boldsymbol{x}}, \hat{\boldsymbol{y}}). \tag{8.2}$$

Here, the pure Matérn kernel M_ν of order ν, the point $\hat{\boldsymbol{x}}$ under coordinate scaling, and the nugget δ are defined as, respectively,

$$M_\nu(\boldsymbol{r}) = \frac{\|\boldsymbol{r}\|_2^\nu K_\nu(\|\boldsymbol{r}\|_2)}{2^{\nu-1}\Gamma(\nu)}, \quad \hat{\boldsymbol{x}} = \left[\frac{x_1}{\ell_1}, \ldots, \frac{x_d}{\ell_d}\right], \quad \delta(\boldsymbol{x}, \boldsymbol{y}) = \begin{cases} \delta, & \boldsymbol{x} = \boldsymbol{y} \\ 0, & \boldsymbol{x} \neq \boldsymbol{y}, \end{cases}$$

where K_ν is the modified Bessel function of second kind of order ν. The pure Matérn kernel is positive definite; however, the matrix generated from the pure kernel often has tiny eigenvalues when n is large. Hence, a small nugget δ is often used to preserve the positive definiteness of the kernel matrix. The coordinate scaling makes it possible

TABLE 8.2
Computation results for a kernel matrix. 1D Multiquadric kernel (8.1). *Points uniform on* $[0,1]$. *Parameters:* $c = 10^{-5}$, $n = 1,000$, $n_0 = 60$, $k = 15$.

$\frac{\|A-\Phi\|_2}{\|\Phi\|_2}$	4.9e-09	$\frac{\mathrm{abs}[\log\lvert\det(A)\rvert - \log\lvert\det(\Phi)\rvert]}{\mathrm{abs}[\log\lvert\det(\Phi)\rvert]}$	3.6e-05
$\mathrm{cond}_2(\Phi)$	8.3e+08	$\tan[\arg(\det(A)) - \arg(\det(\Phi))]$	-7.3e-12
$\mathrm{cond}_2(A)$	8.3e+08	$\cos[\arg(\det(A)) - \arg(\det(\Phi))]$	1.0e+00
$\mathrm{cond}_2(A\tilde{A}_{\mathrm{Alg2}})$	1.0e+00	$\frac{\lVert\,\mathrm{diag}(\tilde{A}_{\mathrm{Alg2}}) - \mathrm{diag}(\tilde{\Phi}_{\mathrm{LU}})\rVert_2}{\lVert\,\mathrm{diag}(\tilde{\Phi}_{\mathrm{LU}})\rVert_2}$	2.6e-03
$\lVert A\tilde{A}_{\mathrm{LU}} - I\rVert_2$	5.7e-09		
$\lVert A\tilde{A}_{\mathrm{Alg2}} - I\rVert_2$	3.3e-08	$\frac{\lvert\,\mathrm{tr}(\tilde{A}_{\mathrm{Alg2}}) - \mathrm{tr}(\tilde{\Phi}_{\mathrm{LU}})\rvert}{\lvert\,\mathrm{tr}(\tilde{\Phi}_{\mathrm{LU}})\rvert}$	9.1e-04
$\lVert A\tilde{A}_{\mathrm{Kry}} - I\rVert_2$	1.5e-08		
Krylov solver	PCG		
iter. per r.h.s.	1		
$\lVert\Phi\tilde{A}_{\mathrm{Kry}} - I\rVert_2$	1.3e-01		

to yield an anisotropic kernel, as is the case for our experiment here. We generate a matrix of size $n = 4 \times 10^3$ and use Chebyshev order $k = 15$. Note that in two dimensions, this order means that the "rank" $r = (k+1)^2 = 256$, which is larger than the leaf size $n_0 = 200$.

TABLE 8.3
Computation results for a kernel matrix. 2D Matérn kernel (8.2). *Points uniform on* $[0,1]^2$. *Parameters:* $\nu = 1$, $\boldsymbol{\ell} = [1;2]$, $\delta = 10^{-4}$, $n = 4,000$, $n_0 = 200$, $k = 15$.

$\frac{\|A-\Phi\|_F}{\|\Phi\|_F}$	2.7e-05	$\frac{\mathrm{abs}[\log\lvert\det(A)\rvert - \log\lvert\det(\Phi)\rvert]}{\mathrm{abs}[\log\lvert\det(\Phi)\rvert]}$	6.8e-04
$\mathrm{cond}_2(\Phi)$	3.2e+07	$\tan[\arg(\det(A)) - \arg(\det(\Phi))]$	-4.5e-11
$\mathrm{cond}_2(A)$	2.8e+08	$\cos[\arg(\det(A)) - \arg(\det(\Phi))]$	1.0e+00
$\mathrm{cond}_2(A\tilde{A}_{\mathrm{Alg2}})$	1.0e+00	$\frac{\lVert\,\mathrm{diag}(\tilde{A}_{\mathrm{Alg2}}) - \mathrm{diag}(\tilde{\Phi}_{\mathrm{LU}})\rVert_2}{\lVert\,\mathrm{diag}(\tilde{\Phi}_{\mathrm{LU}})\rVert_2}$	1.4e-01
$\lVert A\tilde{A}_{\mathrm{LU}} - I\rVert_F/\sqrt{n}$	1.2e-10		
$\lVert A\tilde{A}_{\mathrm{Alg2}} - I\rVert_F/\sqrt{n}$	4.8e-04	$\frac{\lvert\,\mathrm{tr}(\tilde{A}_{\mathrm{Alg2}}) - \mathrm{tr}(\tilde{\Phi}_{\mathrm{LU}})\rvert}{\lvert\,\mathrm{tr}(\tilde{\Phi}_{\mathrm{LU}})\rvert}$	8.3e-03
$\lVert A\tilde{A}_{\mathrm{Kry}} - I\rVert_F/\sqrt{n}$	1.6e-10		
Krylov solver	PCG		
iter. per r.h.s.	2		
$\lVert\Phi\tilde{A}_{\mathrm{Kry}} - I\rVert_F/\sqrt{n}$	7.0e-01		

We use uniformly random points on $[0,1]^2$ to generate the kernel matrix Φ. The approximation A to Φ has a relative error 2.7e-05. We see that the condition number of A and Φ are no longer similar. However, $A\tilde{A}_{\mathrm{Alg2}}$ is still almost perfectly conditioned.

For measuring the accuracy of the inverses, we change the 2-norm to the Frobenius norm divided by $\sqrt{n}$. The latter measure not only is more economic to compute but also has a stochastic interpretation, because for any matrix M,

$$\mathop{\mathrm{E}}_{x\sim\mathcal{N}} \frac{\|Mx\|_2}{\|x\|_2} = \frac{\|M\|_F}{\sqrt{n}}, \tag{8.3}$$

where $\mathrm{E}_{x\sim\mathcal{N}}$ means taking expectation for all vector x drawn from the standard multivariate normal distribution. See Appendix C for a proof. Then,

$$\frac{\|A\tilde{A} - I\|_F}{\sqrt{n}} = \mathop{\mathrm{E}}_{b\sim\mathcal{N}} \frac{\|A(\tilde{A}b) - b\|_2}{\|b\|_2}, \tag{8.4}$$

for any numerical inverse $\tilde{A}$. The equality (8.4) means that if we solve a linear system with respect to A and a random vector b from standard normal, then the relative residual is the left-hand side of (8.4) on average. Hence, for large matrices we cannot afford computing $\tilde{A}$, but testing with a random vector b is justified.

Reading Table 8.3, we see that the inverse computed by using the proposed algorithm has an accuracy far inferior than that computed by using the LU factorization. However, two iterations of preconditioned conjugate gradient substantially boosts the accuracy, which matches that of the LU factorization.

8.4. Unsymmetric kernel. The fourth experiment is run with a nonstationary (that is, non-translational-invariant) kernel

$$\phi(\boldsymbol{x}, \boldsymbol{y}) = \exp(-\tau\|\hat{\boldsymbol{x}}\|_2) \cdot \exp(-\|\hat{\boldsymbol{y}}\|_2) \cdot M_\nu(\hat{\boldsymbol{x}} - \hat{\boldsymbol{y}}) + \delta(\hat{\boldsymbol{x}}, \hat{\boldsymbol{y}}), \tag{8.5}$$

where the pure Matérn term M_ν and the nugget term δ have been defined in the previous experiment. We set $\tau = 2$, such that the matrix is unsymmetric. The points for this experiment are uniformly random on the unit sphere. See Table 8.4 for results.

Since $n = 10^4$ is not small, we do not invert A, but we test the inversion accuracy by only solving linear systems with a random right-hand side b. One sees that the solutions x_{LU} and x_{Kry} attain almost the same accuracy, with x_{Kry} slightly better. Note that even though A is unsymmetric, preconditioned conjugate gradient still converges (and in two iterations only). Comparing the residuals in Table 8.4 with those in Table 8.3, we suggest that the modified kernel (8.5) yields a matrix much better conditioned than that resulting from the Matérn kernel (8.2), when the same nugget δ is applied.

TABLE 8.4
Computation results for a kernel matrix. 2D nonstationary kernel (8.5)*. Points uniform on the unit sphere. Parameters:* $\tau = 2$, $\nu = 1$, $\boldsymbol{\ell} = [1; 2]$, $\delta = 10^{-4}$, $n = 10,000$, $n_0 = 200$, $k = 15$.

$\dfrac{\|A - \Phi\|_F}{\|\Phi\|_F}$	`2.9e-04`	$\dfrac{\mathrm{abs}[\log\lvert\det(A)\rvert - \log\lvert\det(\Phi)\rvert]}{\mathrm{abs}[\log\lvert\det(\Phi)\rvert]}$	`5.0e-05`
$\|Ax_{\mathrm{LU}} - b\|_2/\|b\|_2$	`5.1e-15`	$\tan[\arg(\det(A)) - \arg(\det(\Phi))]$	`-5.8e-12`
$\|A(\tilde{A}_{\mathrm{Alg2}}b) - b\|_2/\|b\|_2$	`1.7e-12`	$\cos[\arg(\det(A)) - \arg(\det(\Phi))]$	`1.0e+00`
$\|Ax_{\mathrm{Kry}} - b\|_2/\|b\|_2$	`4.5e-15`		
Krylov solver	`PCG`		
iter. per r.h.s.	`2`		
$\|\Phi x_{\mathrm{Kry}} - b\|_2/\|b\|_2$	`6.8e-04`		

8.5. Scaling. We use the kernel (8.5) to perform a scaling test with varying n. The configuration of the kernel is similar to that in Table 8.4, but the leaf size is changed to $n_0 = 128$, the Chebyshev order is changed to $k = 7$, and the points are uniform on $[0, 1]^2$. The computations are serial and are performed on one compute

node (64 GB local memory) of a computing cluster. The processor is Intel Sandy Bridge with a clock rate of 2.6 GHz. The timings are plotted in Figure 8.1. With dashed lines indicating a linear increase, we see that the time costs of all the algorithms proposed in this paper closely follow the linear scaling. Some markers in the plot appear to be "missing" because the recorded times are zero. Furthermore, inverting a matrix is substantially more expensive than performing matrix-vector multiplications, whereas the determinant and trace calculations have a negligible cost once the matrix is inverted.

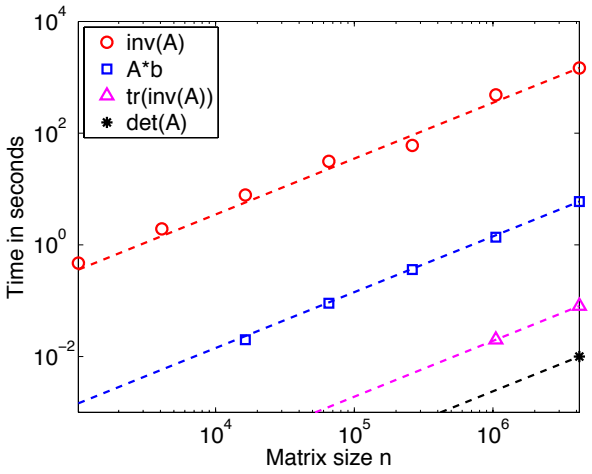


FIG. 8.1. *Running time versus matrix dimension n. 2D nonstationary kernel* (8.5). *Points uniform on* $[0,1]^2$. *Parameters:* $\tau = 2$, $\nu = 1$, $\boldsymbol{\ell} = [1;2]$, $\delta = 10^{-4}$, $n_0 = 128$, $k = 7$. *Dashed lines indicate linear scaling.*

9. Concluding remarks. We have defined a class of matrices that entail a multilevel structure, where the off-diagonal blocks on all levels are low rank and the corresponding blocks across levels share the same row and column subspace. Such a matrix requires only a linear storage, as opposed to the requirement for a general dense matrix, which is quadratic in n. We also show that the inverse of such a matrix belongs to the same class. Hence, matrix-vector multiplications and matrix inversions can both be carried out in strictly linear time. We design the underlying data structure that supports these two matrix operations and also the determinant calculation. The proposed algorithms execute a level-by-level working flow, which implies both intranode and internode parallelism that applies to massively parallel and heterogeneous computer architectures.

The matrix inversion algorithm is not based on factorizations, but it is demonstrated to have a good numerical stability in practice. The key of its favorable numerical property is that we decompose a diagonal block of the matrix as a new block plus a low-rank correction; see (4.3). When the low-rank correction is defined appropriately, the new block can be much better conditioned than the original block; hence the propagation of numerical errors caused by inversions is better controlled. The Chebyshev compression scheme proposed in [11] naturally defines the low-rank correction. Without the use of such a decomposition, the errors in the tables of Section 8 are unacceptably large, except for well-conditioned instances. From the tables, we see that the proposed algorithm yields reasonable accuracies for $\tilde{A}$, the numerical

inverse of A. One can directly multiply $\tilde{A}$ with the right-hand side as a solution of the linear system with respect to A. If higher accuracy is required, a Krylov iteration with $\tilde{A}$ being the preconditioner can converge in one or two iterations and can reach an accuracy matching that of a dense direct solver.

The matrix inversion algorithm consequently enables computing the determinant of A and the diagonal of the inverse of A with negligible cost. Hence, the contribution of this paper spans beyond matrix-vector multiplications and solving linear systems with kernel matrices. The determinant and diagonal frequently appear in statistical applications, such as data analysis and uncertainty quantification [1, 25, 2, 4].

We have demonstrated various computational scenarios, including uniform versus nonuniform point sets, kernel versus non-kernel matrices, isotropic versus anisotropic versus nonstationary kernels, real versus complex matrices, Hermitian versus non-Hermitian matrices, and positive-definite versus indefinite matrices. Most of the experimented matrices are reasonably ill conditioned (condition number $O(10^8)$). The comprehensive experiments show that the proposed data structure and algorithms serve in a general purpose.

The compression quality is a key for the successful calculation with kernel matrices. In this paper we adapt the compression scheme proposed in [11] combined with the use of a k-d tree. A benefit of such a compression is that the cost is strictly linear, whereas most of other methods cannot achieve this complexity in general. In this compression, however, the "rank" r grows exponentially with the Chebyshev order k, which makes the application in high dimensions still challenging. Nevertheless, compression is a subject orthogonal to the matrix operations, and it can be independently developed. An avenue of future work is to design better compression schemes. This topic has attracted considerable research, but methods and theory for dimensions higher than three are rare. We remark that in the context of machine learning, kernels are typically defined on a *very* high-dimensional ambient space but the data points are assumed to be embedded on a low-dimensional manifold. How to apply this work for kernel machine learning is an interesting subject.

The ultimate use of the proposed data structure and algorithms is in the large-scale setting, for which parallelization is an essential component. We will consider the parallel implementation in a separate paper. Because all the algorithms are designed with a working flow similar to that of FMM, which has been demonstrated to scale to $O(10^5)$ processor cores [19], we foresee a similar parallel scalability of our work here.

Acknowledgments. We gratefully acknowledge the use of the Blues cluster in the Laboratory Computing Resource Center at Argonne National Laboratory. This work was supported by the U.S. Department of Energy under Contract DE-AC02-06CH11357.

Appendix A. Generation of a random and recursively low-rank structured matrix. The generation of a recursively low-rank compressed matrix in a random nature follows these steps:

1. Specify parameters: a budgeted number m of tree leaves, a probability p for a node being a leaf, a range $[s_1, s_2]$ of the number of children a node has (with $s_1 > 1$), a range $[n_1, n_2]$ of leaf size, and the rank r.

2. Generate a random tree by creating nodes (and links) in a fashion similar to performing a breadth-first search of a graph. Specifically, maintain a queue of nodes and initialize it with the tree root. We iteratively pop a node out of the queue; with probability p ignore this node or with probability $1-p$ generate children for it, where the number of children is an integer uniformly random in $[s_1, s_2]$. The first node (the

root) must have children. The generated children are pushed into the queue. The current set of leaf nodes include the ones popped off the queue with no children and the ones staying in the queue. This procedure is continued as long as the current number of leaf nodes is less than m. If the queue is empty but the number has not reached m, the last node popped off the queue must have children so that more leaf nodes can be generated.

3. After the tree is generated, for all leaf nodes k, generate an integer n_k uniformly random in $[n_1, n_2]$. Generate random matrices $A_{kk} \in \mathbb{C}^{n_k \times n_k}$ and $U_k, V_k \in \mathbb{C}^{n_k \times r}$. For all nonleaf nodes i, generate random $\Sigma_{jj'} \in \mathbb{C}^{r \times r}$ for all pairs of children j, j' of i. For all non-root nodes k, generate random $W_{ki}, Z_{ki} \in \mathbb{C}^{r \times r}$ where i is the parent of k. Each random element (particularly the A_{kk}'s) can be constructed with a prescribed condition number.

Appendix B. Compressing a kernel matrix. Denoting by Φ a kernel matrix, the goal is to construct a recursively low-rank compressed A that approximates Φ. Let there be a set of d-dimensional points $\{\boldsymbol{x}_p\}_{l=1,\ldots,n}$ and a kernel function $\phi : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$ so that

$$(p, q) \text{ entry of } \Phi = \phi(\boldsymbol{x}_p, \boldsymbol{x}_q).$$

Hence, the partitioning tree in effect recursively partitions the set of points. We use the k-d tree [6] to perform the partitioning, in which case attached to every tree node is a d-dimensional bounding box that tightly bounds the points contained in this node. A k-d tree is a binary tree.

For all leaf nodes l of the partitioning tree, the matrix block $A(I_l, I_l)$ is equal to $\Phi(I_l, I_l)$. On the other hand, for every pair of sibling nodes l, m, we construct a rank-r matrix block $A(I_l, I_m)$ that approximates $\Phi(I_l, I_m)$. Such an approximation uses an extension of the Chebyshev interpolating polynomial that approximates the kernel $\phi(\boldsymbol{x}, \boldsymbol{y})$ for any $\boldsymbol{x} \in B_l$ and $\boldsymbol{y} \in B_m$, where B_l and B_m are the bounding boxes associated with nodes l and m, respectively.

To build the approximation, we first note that in the $\mathbb{R}^1$ case, the degree-k Chebyshev interpolating polynomial ϕ_k that interpolates a function ϕ on an interval $[a, b]$ is defined as

$$\phi_k(x) = \sum_{i=0}^{k} \phi(\xi(\bar{x}_i))\ R_k(\bar{x}_i, \xi^{-1}(x)), \quad x \in [a, b],$$

where $\bar{x}_i = \cos((2i + 1)\pi/(2k + 2))$, $i = 0, \ldots, k$, are the $k + 1$ Chebyshev points on the interval $[-1, 1]$, ξ is the affine mapping that maps $[-1, 1]$ to $[a, b]$, and R_k is the Lagrange polynomial

$$R_k(\bar{x}_i, x) = \frac{2}{k+1} \left[\frac{1}{2} + \sum_{j=1}^{k} T_j(\bar{x}_i) T_j(x) \right],$$

with T_j being the Chebyshev polynomial of the first kind of order j. The Chebyshev interpolating polynomial ϕ_k interpolates ϕ at the mapped Chebyshev points $\xi(\bar{x}_i)$ and it converges to ϕ as k increases.

We extend ϕ_k to accept two arguments as ϕ does:

$$\phi_k(x,y) = \sum_{i=0}^{k}\sum_{j=0}^{k} \phi(\xi_1(\bar{x}_i),\xi_2(\bar{x}_j))\ R_k(\bar{x}_i,\xi_1^{-1}(x))\ R_k(\bar{x}_j,\xi_2^{-1}(y)),$$
$$x\in[a,b],\ y\in[e,f],$$

in which case ξ_1 and ξ_2 are the affine mappings that map $[-1,1]$ to $[a,b]$ and $[e,f]$, respectively. We can further extend ϕ_k to accept two $\mathbb{R}^d$ arguments, if ϕ does so, too. In this case, we must use d-dimensional vectors, such as $\boldsymbol{i}=[i_1,\ldots,i_d]$, to denote indices; an interval such as $[\boldsymbol{a},\boldsymbol{b}]$ means $[a_1,b_1]\times\cdots\times[a_d,b_d]$. We write

$$\phi_{\boldsymbol{k}}(\boldsymbol{x},\boldsymbol{y}) = \sum_{\boldsymbol{i}=\boldsymbol{0}}^{\boldsymbol{k}}\sum_{\boldsymbol{j}=\boldsymbol{0}}^{\boldsymbol{k}} \phi(\xi_1(\bar{\boldsymbol{x}}_{\boldsymbol{i}}),\xi_2(\bar{\boldsymbol{x}}_{\boldsymbol{j}}))\ R_{\boldsymbol{k}}(\bar{\boldsymbol{x}}_{\boldsymbol{i}},\xi_1^{-1}(\boldsymbol{x}))\ R_{\boldsymbol{k}}(\bar{\boldsymbol{x}}_{\boldsymbol{j}},\xi_2^{-1}(\boldsymbol{y})),$$
$$\boldsymbol{x}\in[\boldsymbol{a},\boldsymbol{b}],\ \boldsymbol{y}\in[\boldsymbol{e},\boldsymbol{f}], \quad \text{(B.1)}$$

where ξ_1 and ξ_2 denote the affine mappings that map $[-1,1]^d$ to $[\boldsymbol{a},\boldsymbol{b}]$ and $[\boldsymbol{e},\boldsymbol{f}]$, respectively, the d-dimensional Chebyshev points $\bar{\boldsymbol{x}}_{\boldsymbol{i}}$ are Cartesian products of the 1-dimensional Chebyshev points $\bar{x}_i$, and the Lagrange polynomial $R_{\boldsymbol{k}}$ for $\mathbb{R}^d$ is the product of the usual Lagrange polynomials for $\mathbb{R}^1$:

$$R_{\boldsymbol{k}}(\boldsymbol{z},\boldsymbol{w}) = R_{k_1}(z_1,w_1)\cdots R_{k_d}(z_d,w_d).$$

We have that $\phi_{\boldsymbol{k}}$ is a degree $2(k_1+\cdots+k_d)$-polynomial that approximates ϕ.

The equation (B.1) is used to build the rank-r approximation of a block of the kernel matrix, with $r=(k_1+1)(k_2+1)\cdots(k_d+1)$. Specifically, let two sibling leaf nodes l,m of the partitioning tree be associated with bounding boxes $[\boldsymbol{a},\boldsymbol{b}]$ and $[\boldsymbol{e},\boldsymbol{f}]$. Then, for any $\boldsymbol{x}_p\in[\boldsymbol{a},\boldsymbol{b}]$ and $\boldsymbol{x}_q\in[\boldsymbol{e},\boldsymbol{f}]$, $\phi(\boldsymbol{x}_p,\boldsymbol{x}_q)$ is approximated by $\phi_{\boldsymbol{k}}(\boldsymbol{x}_p,\boldsymbol{x}_q)$ as defined in (B.1). This approximation defines the elements of the compressed matrix:

1. the $(\boldsymbol{i},\boldsymbol{j})$ entry of Σ_{lm} is $\phi(\xi_1(\bar{\boldsymbol{x}}_{\boldsymbol{i}}),\xi_2(\bar{\boldsymbol{x}}_{\boldsymbol{j}}))$,
2. the $(p,\boldsymbol{i})$ entry of U_l is $R_{\boldsymbol{k}}(\bar{\boldsymbol{x}}_{\boldsymbol{i}},\xi_1^{-1}(\boldsymbol{x}_p))$,
3. the $(q,\boldsymbol{j})$ entry of V_m is $R_{\boldsymbol{k}}(\bar{\boldsymbol{x}}_{\boldsymbol{j}},\xi_2^{-1}(\boldsymbol{x}_q))$.

Furthermore, we can approximate $\phi(\xi_1(\bar{\boldsymbol{x}}_{\boldsymbol{i}}),\xi_2(\bar{\boldsymbol{x}}_{\boldsymbol{j}}))$ in (B.1) by $\phi_{\boldsymbol{k}}$ by using (B.1) recursively, which leads to the definitions of other Σ_{**}'s and also of the change-of-basis matrices. Specifically, item 1 of the above list (definition of Σ_{lm}) still holds for any pair of sibling nodes l,m. For any pair of child node l and parent node u, if ξ_1 and ξ_3 are the mappings that maps $[-1,1]^d$ to the bounding boxes associated with l and u, respectively, then

4. the $(\boldsymbol{i},\boldsymbol{m})$ entry of W_{lu} is $R_{\boldsymbol{k}}(\bar{\boldsymbol{x}}_{\boldsymbol{m}},\xi_3^{-1}(\xi_1(\bar{\boldsymbol{x}}_{\boldsymbol{i}})))$,
5. Z_{lu} is the same as W_{lu}.

We thus have defined the recursively low-rank compressed matrix A. In addition, we note that the definition of Σ_{lm} in item 1 extends to Σ_{ll} for all nodes l.

Appendix C. Proof of (8.3). If x follows the standard multivariate normal, then $x/\|x\|_2$ is uniform on the unit sphere. Denote by $\mathcal{S}$ the latter distribution. Then, we have

$$\mathop{\mathrm{E}}_{x\sim\mathcal{N}}\frac{\|Mx\|_2}{\|x\|_2} = \mathop{\mathrm{E}}_{x\sim\mathcal{S}}\|Mx\|_2 = \mathop{\mathrm{E}}_{x\sim\mathcal{S}}\mathrm{tr}(Mxx^*M^*)^{1/2} = \mathrm{tr}\left\{M\left(\mathop{\mathrm{E}}_{x\sim\mathcal{S}}xx^*\right)M^*\right\}^{1/2}.$$

Because $\mathrm{E}_{x\sim\mathcal{S}}\, xx^* = I/n$, we immediately have that

$$\mathrm{tr}\left\{M\left(\mathop{\mathrm{E}}_{x\sim\mathcal{S}} xx^*\right)M^*\right\}^{1/2} = \frac{\|M\|_F}{\sqrt{n}}.$$

Thus, the proof is completed.

REFERENCES

[1] M. Anitescu, J. Chen, and L. Wang, *A matrix-free approach for solving the parametric Gaussian process maximum likelihood problem*, SIAM J. Sci. Comput., 34 (2012), pp. A240–A262.

[2] E. Aune, D. P. Simpson, and J. Eidsvik, *Parameter estimation in high dimensional Gaussian distributions*, Statistics and Computing, 24 (2014), pp. 247–263.

[3] J. E. Barnes and P. Hut, *A hierarchical $O(N \log N)$ force-calculation algorithm*, Nature, 324 (1986), pp. 446–449.

[4] C. Bekas, A. Curioni, and I. Fedulova, *Low-cost data uncertainty quantification*, Concurrency and Computation: Practice and Experience, 24 (2011), pp. 908–920.

[5] C. Bekas, E. Kokiopoulou, and Y. Saad, *An estimator for the diagonal of a matrix*, Appl. Numer. Math., 57 (2007), pp. 1214–1229.

[6] J. L. Bentley, *Multidimensional binary search trees used for associative searching*, Communications of the ACM, 18 (1975), pp. 509–517.

[7] S. Börm, L. Grasedyck, and W. Hackbusch, *Introduction to hierarchical matrices with applications*, Engineering Analysis with Boundary Elements, 27 (2003), pp. 405–422.

[8] S. Chandrasekaran, P. Dewilde, M. Gu, W. Lyons, and T. Pals, *A fast solver for HSS representations via sparse matrices*, SIAM J. Matrix Anal. Appl., 29 (2006), pp. 67–81.

[9] S. Chandrasekaran, M. Gu, and T. Pals, *A fast ULV decomposition solver for hierarchically semiseparable representations*, SIAM J. Matrix Anal. Appl., 28 (2006), pp. 603–622.

[10] H. Cheng, Z. Gimbutas, P. G. Martinsson, and V. Rokhlin, *On the compression of low rank matrices*, SIAM J. Sci. Comput., 26 (2005), pp. 1389–1404.

[11] W. Fong and E. Darve, *The black-box fast multipole method*, J. Comput. Phys., 228 (2009), pp. 8712–8725.

[12] Z. Gimbutas and V. Rokhlin, *A generalized fast multipole method for nonoscillatory kernels*, SIAM J. Sci. Comput., 24 (2002), pp. 796–817.

[13] L. Greengard and V. Rokhlin, *A fast algorithm for particle simulations*, J. Comput. Phys., 73 (1987), pp. 325–348.

[14] W. Hackbusch, *A sparse matrix arithmetic based on $\mathcal{H}$-matrices, part I: Introduction to $\mathcal{H}$-matrices*, Computing, 62 (1999), pp. 89–108.

[15] W. Hackbusch and S. Börm, *Data-sparse approximation by adaptive $\mathcal{H}^2$-matrices*, Computing, 69 (2002), pp. 1–35.

[16] W. Hackbusch and Z. P. Nowak, *On the fast matrix multiplication in the boundary element method by panel clustering*, Numerische Mathematik, 54 (1989), pp. 463–491.

[17] L. Y. Kenneth L Ho, *Hierarchical interpolative factorization for elliptic operators: integral equations*. arXiv preprint arXiv:1307.2666, 2013.

[18] R. Krasny and L. Wang, *Fast evaluation of multiquatric RBF sums by a Cartesian treecode*, SIAM J. Sci. Comput., 33 (2011), pp. 2341–2355.

[19] I. Lashuk, A. Chandramowlishwaran, H. Langston, T.-A. Nguyen, R. Sampath, A. Shringarpure, R. Vuduc, L. Ying, D. Zorin, and G. Biros, *A massively parallel adaptive fast multipole method on heterogeneous architectures*, Communications of the ACM, 55 (2012), pp. 101–109.

[20] L. Lin, J. Lu, and L. Ying, *Fast construction of hierarchical matrix representation from matrix-vector multiplication*, J. Comput. Phys., 230 (2011), pp. 4071–4087.

[21] L. Lin, C. Yang, J. C. Meza, J. Lu, L. Ying, and W. E, *Selinv—an algorithm for selected inversion of a sparse symmetric matrix*, ACM Trans. Math. Software, 37 (2011), pp. 40:1–40:19.

[22] P. Martinsson and V. Rokhlin, *A fast direct solver for boundary integral equations in two dimensions*, J. Comput. Phys., 205 (2005), pp. 1–23.

[23] P. G. Martinsson, *A fast randomized algorithm for computing a hierarchically semiseparable representation of a matrix*, SIAM J. Matrix Anal. Appl., 32 (2011), pp. 1251–1274.

[24] P. G. Martinsson and V. Rokhlin, *An accelerated kernel-independent fast multipole method in one dimension*, SIAM J. Sci. Comput., 29 (2007), pp. 1160–1178.

[25] M. L. Stein, J. Chen, and M. Anitescu, *Stochastic approximation of score functions for Gaussian processes*, Annals of Applied Statistics, 7 (2013), pp. 1162–1191.
[26] J. Xia, S. Chandrasekaran, M. Gu, and X. S. Li, *Fast algorithms for hierarchically semiseparable matrices*, Numer. Lin. Alg. Appl., 17 (2010), pp. 953–976.
[27] J. Xia and M. Gu, *Robust approximate Cholesky factorization of rank-structured symmetric positive definite matrices*, SIAM J. Matrix Anal. Appl., 31 (2010), pp. 2899–2920.
[28] L. Ying, G. Biros, and D. Zorin, *A kernel-independent adaptive fast multipole algorithm in two and three dimensions*, J. Comput. Phys., 196 (2004), pp. 591–626.

The submitted manuscript has been created by UChicago Argonne, LLC, Operator of Argonne National Laboratory ("Argonne"). Argonne, a U.S. Department of Energy Office of Science laboratory, is operated under Contract No. DE-AC02-06CH11357. The U.S. Government retains for itself, and others acting on its behalf, a paid-up nonexclusive, irrevocable worldwide license in said article to reproduce, prepare derivative works, distribute copies to the public, and perform publicly and display publicly, by or on behalf of the Government.