# H2Pack: High-Performance $\mathcal{H}^2$ Matrix Package for Kernel Matrices Using the Proxy Point Method

HUA HUANG, XIN XING, and EDMOND CHOW, School of Computational Science and Engineering, Georgia Institute of Technology

Dense kernel matrices represented in $\mathcal{H}^2$ matrix format typically require less storage and have faster matrix-vector multiplications than when these matrices are represented in the standard dense format. In this paper, we present H2Pack, a high-performance, shared-memory library for constructing and operating with $\mathcal{H}^2$ matrix representations for kernel matrices defined by non-oscillatory, translationally-invariant kernel functions. Using a hybrid analytic-algebraic compression method called the proxy point method, H2Pack can efficiently construct an $\mathcal{H}^2$ matrix representation with linear computational complexity. Storage and matrix-vector multiplication also have linear complexity. H2Pack also introduces the concept of "partially admissible blocks" for $\mathcal{H}^2$ matrices to make $\mathcal{H}^2$ matrix-vector multiplication mathematically identical to the fast multipole method (FMM) if analytic expansions are used. We optimize H2Pack from both the algorithm and software perspectives. Compared to existing FMM libraries, H2Pack generally has much faster $\mathcal{H}^2$ matrix-vector multiplications since the proxy point method is more effective at producing block low-rank approximations than the analytic methods used in FMM. As a trade-off, $\mathcal{H}^2$ matrix construction in H2Pack is typically more expensive than the setup cost in FMM libraries. Thus, H2Pack is ideal for applications that need a large number of matrix-vector multiplications for a given configuration of data points.

CCS Concepts: • **Mathematics of computing → Mathematical software**;

Additional Key Words and Phrases: rank-structured matrix, $\mathcal{H}^2$ matrix, proxy point method, N-body problem, fast multipole method, high-performance computing

**ACM Reference Format:**
Hua Huang, Xin Xing, and Edmond Chow. 2020. H2Pack: High-Performance $\mathcal{H}^2$ Matrix Package for Kernel Matrices Using the Proxy Point Method. *ACM Trans. Math. Softw.* 0, 0, Article 0 ( 2020), 30 pages. https://doi.org/10.xxxx/xxxxxxx.xxxxxxx

## 1 INTRODUCTION

Many problems in scientific computing and data analytics, such as particle simulations with long-range interactions, the numerical solution of integral equations, and Gaussian process modeling lead to dense *kernel matrices*. Given two sets of points, $X$ and $Y$, and a non-compact kernel function $K(x, y)$, the kernel matrix $K(X, Y)$ has entries defined as $K(x_i, y_j)$ with all $(x_i, y_j) \in X \times Y$. Usually, kernel matrices have block low-rank structure, i.e., certain blocks of the matrices are numerically low-rank. For such a kernel matrix, representing these blocks in low-rank form gives a *rank-structured matrix representation* that asymptotically reduces the quadratic cost of matrix storage and matrix-vector multiplication. Different kernel matrices can be effectively stored in different rank-structured matrix representations such as $\mathcal{H}$ [18, 20], $\mathcal{H}^2$ [19, 21], and HSS [8]. In this

Authors' address: Hua Huang, huangh223@gatech.edu; Xin Xing, xxing33@gatech.edu; Edmond Chow, echow@cc.gatech.edu, School of Computational Science and Engineering, Georgia Institute of Technology, Atlanta, Georgia, 30332.


0098-3500/2020/0-ART0 $15.00
https://doi.org/10.xxxx/xxxxxxx.xxxxxxx

paper, we focus on the development of a library for efficiently constructing and using $\mathcal{H}^2$ matrix representations defined by non-oscillatory, translationally-invariant kernel functions with points in low-dimensional space (e.g., 2D or 3D).

$\mathcal{H}^2$ matrix representations are constructed by compressing specific matrix blocks into low-rank form via a nested approach. The compression of these blocks can be computed either *analytically* based on degenerate approximations of the kernel function such as multipole expansions and polynomial expansions, or *algebraically* based on matrix decomposition methods such as such as SVD, QR, and ACA [2]. It is worth noting that, when analytic compression methods are used, the fast matrix-vector multiplication of the constructed $\mathcal{H}^2$ matrix can be viewed as an algebraic variant of the fast multipole method (FMM) [11, 15, 16, 32]. Compared to algebraic compression, analytic compression usually requires less intermediate storage and computation, but is limited to specific kernel functions and can give approximation rank much larger than the numerical rank of the matrix block to be compressed. Algebraic compression, instead, is usually more effective in terms of range of applicability and optimality of the approximation rank. Due to these differences, the matrix-vector multiplication with an $\mathcal{H}^2$ matrix constructed by algebraic methods is usually faster than FMM, since a lower-rank approximation leads to more cost reduction in the multiplication. As a sacrifice, algebraic methods usually lead to much more expensive $\mathcal{H}^2$ matrix construction than analytic methods. For example, simply evaluating all matrix entries has quadratic cost, making many algebraic methods such as SVD unfavorable.

To balance between analytic methods and algebraic methods, we use a hybrid analytic-algebraic compression method called the *proxy point method* [31] to construct $\mathcal{H}^2$ matrix representations. For kernel functions from potential theory, such as the Laplace and Stokes kernels, Martinsson and Rokhlin [24] introduced the *proxy surface method* to efficiently compress specific kernel blocks into a low-rank form called interpolative decomposition (ID) [17]. Two variants of the proxy surface method were proposed later by Corona [10] and Minden [26]. All three methods belong to the class of the proxy point methods that has been formalized and studied in recent work [31]. Compared to algebraic methods, the proxy point method avoids forming a kernel block explicitly before compressing it and also requires far less data communication in parallel $\mathcal{H}^2$ matrix construction. Compared to analytic methods, the proxy point method can obtain better approximation ranks and is kernel-independent. As a result, it can efficiently construct an $\mathcal{H}^2$ matrix representation with linear complexity, while the constructed $\mathcal{H}^2$ matrix can have faster matrix-vector multiplications than FMM. Another common hybrid analytic-algebraic approach is to combine an analytic method with algebraic recompression [1, 4, 6]. Such an approach gives better approximation rank but is also restricted to certain kernels, like analytic methods. In comparison, the proxy point method incorporates the kernel function numerically when constructing the $\mathcal{H}^2$ matrix representation. This allows the construction to be kernel-independent.

H2Pack is a shared-memory parallel library for kernel matrices based on constructing $\mathcal{H}^2$ matrix representations using the proxy point method. The kernel functions must be non-oscillatory and *translationally-invariant* (i.e., $K(x, y) = k(x - y)$ with a univariate function $k(\cdot)$) with points in low-dimensional space. H2Pack library works for both scalar and tensor kernel functions. Presently, the library further requires the input kernel function to be symmetric, i.e, $K(x, y) = K(y, x)$, and the kernel matrix to be defined by one set of points $X$, i.e., $K(X, X)$. These two requirements can be easily lifted via a simple extension of H2Pack which will be addressed in the next version.

More precisely, H2Pack implements the following two components:

- $\mathcal{H}^2$ matrix construction based on the proxy point method with inputs being a kernel function $K(x, y)$, a set of points $X$, and an error threshold for the low-rank approximations;
- $\mathcal{H}^2$ matrix-vector multiplication.

We optimize H2Pack from both the algorithm and software perspectives. Different parallelization and load-balancing strategies are applied for different computation phases in H2Pack. Moreover, two running modes for H2Pack are available to adapt to different computing platforms and different problem settings for better performance. The ahead-of-time mode precomputes and stores all the components of an $\mathcal{H}^2$ matrix. The just-in-time mode calculates a large portion of the components dynamically when needed in $\mathcal{H}^2$ matrix-vector multiplication. These two modes provide a trade-off between storage and computation. The performance difference between the two modes depends on the memory bandwidth, CPU speed, and the complexity of the kernel function evaluations. It is worth noting that this two-mode approach has been proposed before in Refs [6, 25]. Lastly, we also exploit intrinsic functions for better vectorization to further improve the performance of H2Pack on multi-core and many-core processors.

Our numerical tests with H2Pack show that its $\mathcal{H}^2$ matrix construction cost is only around 5 to 15 times the corresponding $\mathcal{H}^2$ matrix-vector multiplication cost. Comparisons of H2Pack with two state-of-the-art FMM libraries, PVFMM [23] and FMM3D [13], show that H2Pack has asymptotically more expensive $\mathcal{H}^2$ matrix construction but faster $\mathcal{H}^2$ matrix-vector multiplications. More precisely, the $\mathcal{H}^2$ matrix construction cost in H2Pack is similar to the FMM setup costs in FMM3D and PVFMM for a low or moderate relative multiplication accuracy, e.g., $10^{-5}$ and $10^{-8}$, and is just 2 to 5 times more expensive for a high relative multiplication accuracy, e.g., $10^{-11}$. Meanwhile, the approximation ranks of blocks in H2Pack are 5 to 10 times smaller than those in PVFMM and FMM3D. As a result, the $\mathcal{H}^2$ matrix-vector multiplication in H2Pack is 1.5 to 5 times faster than in PVFMM and 5 to 25 times faster than in FMM3D in various tests. In practice, H2Pack is ideal for problems where many matrix-vector multiplications are required per configuration of data points, e.g., numerical solution of integral equations and Gaussian process modeling, so that the relatively expensive $\mathcal{H}^2$ matrix construction cost can be amortized.

*Related work.* There are several libraries for FMM and its variants. FMM3D [13] implements the classical FMM [14, 16] for three key kernel functions in 3D from potential theory, the Laplace, Helmholtz, and Stokes kernels. PVFMM [23] implements the kernel-independent FMM [32] and works for kernel functions from potential theory. BBFMM3D [29] implements the black-box FMM [11] and works for smooth, translationally-invariant kernel functions. All these FMM libraries support OpenMP shared memory parallelization. PVFMM further supports MPI distributed memory parallelization and GPU acceleration of major FMM subroutines.

There are also several libraries for working with rank-structured matrices. H2Lib [3] constructs $\mathcal{H}^2$ matrix representations algebraically but only works for matrices from the boundary element method whose entries are kernel-defined interactions between compact basis functions in integral form. H2Lib supports OpenMP shared memory parallelization. SMASH [6] uses a heuristic hybrid compression method to construct both $\mathcal{H}^2$ and HSS matrix representations for kernel matrices. SMASH is written in MATLAB and its C language implementation is still under development. STRUMPACK [12, 28] uses a randomized algebraic compression method to efficiently construct HSS matrix representations for a general class of dense matrices. STRUMPACK supports MPI distributed memory parallelization for fast matrix-vector multiplications and fast matrix solve. Recently, an $\mathcal{H}^2$ matrix library for GPUs has also been developed [5].

## 2 $\mathcal{H}^2$ MATRIX REPRESENTATION AND $\mathcal{H}^2$ MATRIX-VECTOR MULTIPLICATION

Consider a kernel matrix $K(X, X)$ defined by a non-oscillatory kernel function $K(x, y)$ that is *translationally-invariant and symmetric*, and a set of points $X$ in a low-dimensional space. This section describes an $\mathcal{H}^2$ matrix representation of $K(X, X)$, $\mathcal{H}^2$ matrix construction based on the

proxy point method, and $\mathcal{H}^2$ matrix-vector multiplication. The following discussion applies to both scalar and tensor kernel functions $K(x, y)$, e.g., both the Laplace and the Stokes kernels.

### 2.1 $\mathcal{H}^2$ matrix representation

*Interpolative decomposition.* An interpolative decomposition (ID) [9, 17] represents or approximates a matrix $A \in \mathbb{R}^{n\times m}$ in the low-rank form $UA_J$, where $U \in \mathbb{R}^{n\times k}$ has bounded entries, $A_J \in \mathbb{R}^{k\times m}$ contains $k$ rows of $A$, and $k$ is the rank. An ID approximation defined this way is said to have error below the *error threshold* $\varepsilon_0$ if the 2-norm of each row of $A - UA_J$ is bounded by $\varepsilon_0$. Using an algebraic approach, an ID approximation with a given rank or a given error threshold can be calculated using the strong rank-revealing QR (SRRQR) decomposition [17] or using the pivoted QR decomposition. Specifically, an ID approximation of a kernel matrix block $K(X_0, Y_0)$ can be written as $K(X_0, Y_0) \approx UK(X_{\text{id}}, Y_0)$ where $K(X_{\text{id}}, Y_0)$ contains a subset of the rows in $K(X_0, Y_0)$ and $X_{\text{id}}$ is a subset of $X_0$.

*Hierarchical partitioning of $X$ and $K(X, X)$.* To construct an $\mathcal{H}^2$ matrix representation, the first step is to hierarchically partition the points in $X$. Assume $X$ is in a $d$-dimensional space and let $\mathcal{B}$ be a box with equal-length edges that encloses $X$. The box $\mathcal{B}$ is partitioned into $2^d$ smaller same-sized boxes by bisecting all its edges. Each smaller box is further partitioned recursively in the same way until the number of points in a box is less than a prescribed constant. This hierarchical partitioning of $\mathcal{B}$ can be represented by a $2^d$-ary *partition tree* $\mathcal{T}$ whose nodes correspond to the boxes. We define the root node of $\mathcal{T}$ to be at level 0, its children nodes to be at level 1, etc. We also define the leaf level to be level $L$.

Each level of the partition tree defines a non-overlapping partitioning of the set of points $X$. This partitioning is defined using the set of nodes at a given level of the partition tree. To generalize the concept of the set of nodes at a given level to the case of possibly non-perfect partition trees, let $\text{level}^+(l)$ denote the union of all the nodes in level $l$ and all the leaf nodes above level $l$ (toward the root). The caption of Figure 1 gives examples of $\text{level}^+(l)$ for an example partition tree.

Now, let $X_i$ denote the set of points lying in box $i$ and corresponding to node $i$ in the tree. At any level $l$, $\{X_i\}_{i\in\text{level}^+(l)}$ defines a non-overlapping partitioning of the set of points $X$, i.e.,

$$X_i \cap X_j = \emptyset \;\text{ for distinct } i, j \in \text{level}^+(l) \quad \text{and} \quad \cup_{i\in\text{level}^+(l)} X_i = X.$$

For the kernel matrix itself, $\{K(X_i, X_j)\}_{i,j\in\text{level}^+(l)}$ defines a non-overlapping partitioning of $K(X, X)$. See Figure 1 for an example of a partition tree and the associated matrix partitioning at each level.

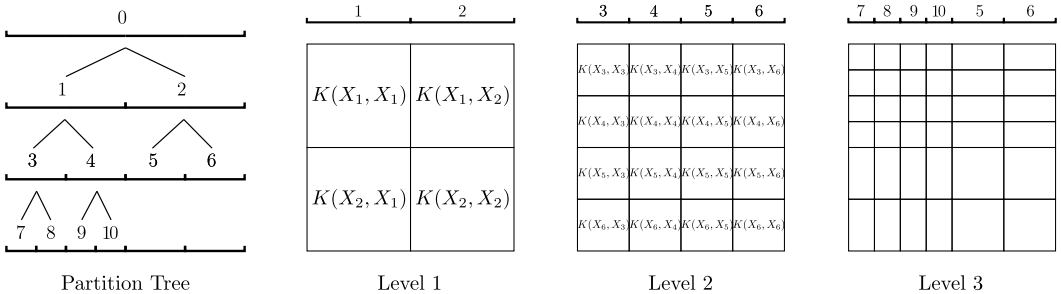



Fig. 1. Illustration of a 3-level hierarchical partitioning of a set of points $X$ in 1-dimensional space and the associated partitioning of a kernel matrix $K(X, X)$. In this partition tree, $\text{level}^+(1) = \{1, 2\}$, $\text{level}^+(2) = \{3, 4, 5, 6\}$, and $\text{level}^+(3) = \{7, 8, 9, 10, 5, 6\}$. In each level $l$, $K(X, X)$ is partitioned into non-overlapping blocks $K(X_i, X_j)$ with $i, j \in \text{level}^+(l)$.

*Inadmissible, admissible, and partially admissible blocks.* In an $\mathcal{H}^2$ matrix representation, a kernel block $K(X_*, Y_*)$ is considered numerically low-rank if $X_*$ is in a box and $Y_*$ is in the *far field* of the box. The far field of a box is defined as the area of all the boxes that are least one box width away from the box. For any box $i$ in some level $l$, we split boxes in $\text{level}^+(l)$ into two subsets $\mathcal{F}_i$ and $\mathcal{N}_i$ as

$$\mathcal{F}_i = \{k \in \text{level}^+(l) \mid \text{box } k \text{ is in the far field of box } i\} \quad \text{and} \quad \mathcal{N}_i = \text{level}^+(l) \setminus \mathcal{F}_i.$$

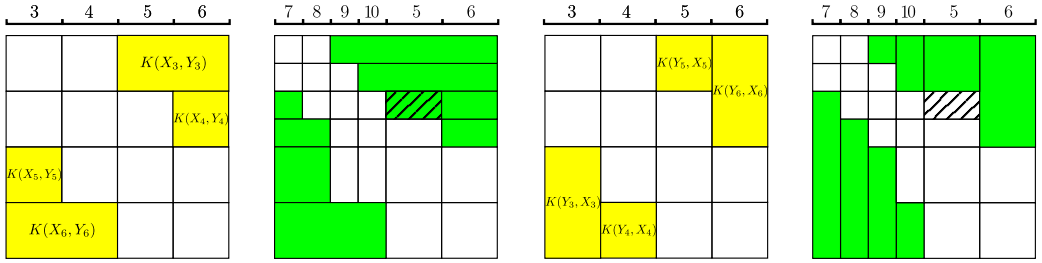
Let $Y_i = \cup_{k \in \mathcal{F}_i} X_k$ be the set of all points in the far field of box $i$. Then, $K(X_i, Y_i)$ for each box $i$ with nonempty $\mathcal{F}_i$ is considered to be numerically low-rank. Thus, the numerically low-rank blocks at each level can be denoted as $K(X_i, Y_i)$ or $K(Y_i, X_i)$ for all nodes $i \in \text{level}^+(l)$. Note that if $K(X, X)$ is symmetric, then $K(Y_i, X_i) = K(X_i, Y_i)^T$. See Figure 2 for an illustration of these low-rank blocks.

A block $K(X_i, X_j)$ that is contained in the low-rank blocks $K(X_i, Y_i)$ or $K(Y_j, X_j)$ is thus also low-rank. Based on this observation, the blocks in $\{K(X_i, X_j)\}_{i,j \in \text{level}^+(l)}$ can be categorized into three classes:

- *inadmissible blocks*, if $K(X_i, X_j)$ is not within $K(X_i, Y_i)$ and not within $K(Y_j, X_j)$ (equivalent to $X_j \cap Y_i = \emptyset$ and $X_i \cap Y_j = \emptyset$);
- *admissible blocks*, if $K(X_i, X_j)$ is within both $K(X_i, Y_i)$ and $K(Y_j, X_j)$ (equivalent to $X_j \subseteq Y_i$ and $X_i \subseteq Y_j$);
- *partially admissible blocks*, if $K(X_i, X_j)$ is within $K(X_i, Y_i)$ but not within $K(Y_j, X_j)$ (equivalent to $X_j \subseteq Y_i$ and $X_i \cap Y_j = \emptyset$), or if $K(X_i, X_j)$ is not within $K(X_i, Y_i)$ but within $K(Y_j, X_j)$ (equivalent to $X_j \cap Y_i = \emptyset$ and $X_i \subseteq Y_j$).

See the hatched block in Figure 2 for an example of a partially admissible block.

The concept of "partially admissible blocks" is new to the standard $\mathcal{H}^2$ matrix representation. More details follow later in this section.




Level 2 Level 3 Level 2 Level 3

(a) blocks $K(X_i, Y_i)$ with $i \in \text{level}^+(l)$, $l = 2$ or 3 (b) blocks $K(Y_i, X_i)$ with $i \in \text{level}^+(l)$, $l = 2$ or 3

Fig. 2. Illustrations of the low-rank blocks $K(X_i, Y_i)$ and $K(Y_i, X_i)$ for the partition tree in Figure 1. The low-rank blocks are colored yellow for level 2 and green for level 3. For level 2, these blocks are labeled explicitly. In level 3, note that some of these blocks are not contiguous. The hatched block $K(X_9, X_5)$ is a partially admissible block since it is within $K(X_9, Y_9)$ in (a) but not within $K(Y_5, X_5)$ in (b).

*Compression of low-rank blocks.* We express a low-rank approximation of each $K(X_i, Y_i)$ in an ID form,

$$K(X_i, Y_i) \approx U_i K(X_i^{\text{id}}, Y_i), \tag{1}$$

where $X_i^{\text{id}}$ is a subset of $X_i$ and $K(X_i^{\text{id}}, Y_i)$ contains a subset of the rows in $K(X_i, Y_i)$. For a non-leaf box $i$ with children $\{i_1, i_2, \ldots, i_s\}$, the above ID approximation is formed and computed by a nested

approach (to be described in Section 2.2) in an $\mathcal{H}^2$ matrix representation. Precisely, the two ID components $U_i$ and $X_i^{\text{id}}$ are recursively defined in the nested form,

$$U_i = \begin{bmatrix} U_{i_1} & & \\ & \ddots & \\ & & U_{i_s} \end{bmatrix} R_i \quad \text{and} \quad X_i^{\text{id}} \subseteq X_{i_1}^{\text{id}} \cup X_{i_2}^{\text{id}} \ldots \cup X_{i_s}^{\text{id}} \subseteq X_i, \tag{2}$$

with some matrix $R_i$ to be computed. Based on eq. (2), $U_i$ for each non-leaf node is not explicitly formed but can be recovered recursively from quantities at all the descendants of node $i$.

Each inadmissible block $K(X_i, X_j)$ is considered to be full-rank. Each admissible block $K(X_i, X_j)$ is numerically low-rank and can be compressed as

$$K(X_i, X_j) \approx U_i K(X_i^{\text{id}}, X_j^{\text{id}}) U_j^T, \tag{3}$$

based on the compression of $K(X_i, Y_i)$ and $K(Y_j, X_j)$ in eq. (1). Each partially admissible block $K(X_i, X_j)$ can be compressed as

$$K(X_i, X_j) \approx \begin{cases} U_i K(X_i^{\text{id}}, X_j) & \text{if } K(X_i, X_j) \text{ is within } K(X_i, Y_i) \\ K(X_i, X_j^{\text{id}}) U_j^T & \text{if } K(X_i, X_j) \text{ is within } K(Y_j, X_j) \end{cases}, \tag{4}$$

based on the compression of $K(X_i, Y_i)$ or $K(Y_j, X_j)$ in eq. (1).

*$\mathcal{H}^2$ matrix representation.* The $\mathcal{H}^2$ matrix representation of $K(X, X)$ consists of three parts:

- dense inadmissible blocks $K(X_i, X_j)$ with both $i$ and $j$ being leaf nodes.
- low-rank approximations eq. (3) of all the admissible blocks $K(X_i, X_j)$ that are not contained in larger admissible or partially admissible blocks.
- low-rank approximations eq. (4) of all the partially admissible blocks $K(X_i, X_j)$ that are not contained in larger admissible or partially admissible blocks.

Denote the three sets of the node pairs $(i, j)$ associated with the above three sets of kernel blocks as $\mathcal{D}$, $\mathcal{A}$, and $\mathcal{A}_p$, respectively. See Figure 3 for an example of these three sets of blocks making up an $\mathcal{H}^2$ matrix representation. As can be easily verified, these three sets of kernel blocks exactly form a non-overlapping partitioning of $K(X, X)$. The components stored by an $\mathcal{H}^2$ matrix include:

- $U_i$ and $X_i^{\text{id}}$ for each leaf node $i$ with nonempty $\mathcal{F}_i$;
- $R_i$ and $X_i^{\text{id}}$ for each non-leaf node $i$ with nonempty $\mathcal{F}_i$;
- *intermediate blocks* denoted by $B_{i,j}$ for each $(i, j) \in \mathcal{A} \cup \mathcal{A}_p$. Block $B_{ij}$ is one of blocks $K(X_i^{\text{id}}, X_j^{\text{id}})$, $K(X_i^{\text{id}}, X_j)$, or $K(X_i, X_j^{\text{id}})$ in the low-rank approximation eq. (3) or eq. (4) of $K(X_i, X_j)$;
- *inadmissible blocks* $K(X_i, X_j)$ denoted by $D_{i,j}$ for each $(i, j) \in \mathcal{D}$.

All the intermediate and inadmissible blocks can be computed using only the sets $\{X_i\}$ and $\{X_i^{\text{id}}\}$ for all $i$, which can be stored economically. Instead of precomputing and storing these intermediate and inadmissible blocks, they can be dynamically computed when needed, using only $\{X_i\}$ and $\{X_i^{\text{id}}\}$. This provides a trade-off between storage and computation.

*More details on partially admissible blocks.* In the standard $\mathcal{H}^2$ matrix representation, all the partially admissible blocks characterized above are treated as admissible blocks and are compressed into the form eq. (3) (instead of eq. (4)) where the corresponding $U_i$ and $X_i^{\text{id}}$ for each node $i$ are computed by the ID approximation of $K(X_i, \tilde{Y}_i)$ with $\tilde{Y}_i$ defined as some superset of $Y_i$.

Taking the partially admissible block $K(X_5, X_9)$ in Figure 3 as an example, we have $Y_5 = X_3$, $\tilde{Y}_5 = X_3 \cup X_9$, and $\tilde{Y}_9 = Y_9 = X_7 \cup X_5 \cup X_6$. Note that $K(X_5, X_9)$ is within $K(X_5, \tilde{Y}_5)$ but not within $K(X_5, Y_5)$. Thus, by the ID approximation of $K(X_5, Y_5)$ and $K(X_9, Y_9)$, the block $K(X_5, X_9)$ can only

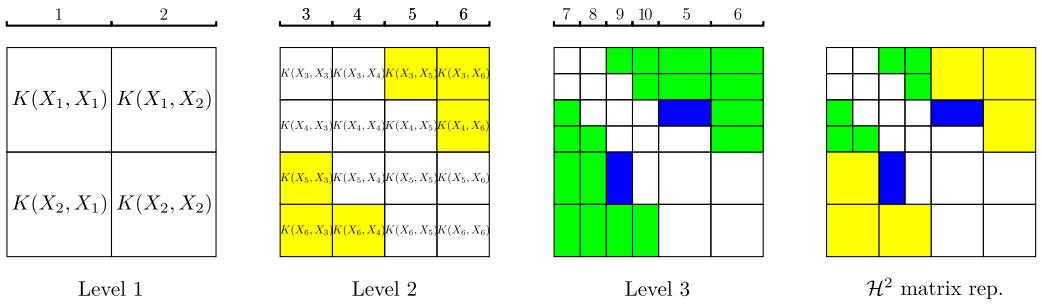



Fig. 3. Illustration of a 3-level $\mathcal{H}^2$ matrix representation associated with the partition tree in Figure 1. Inadmissible blocks are white in all levels, level 2 has admissible blocks (yellow), and level 3 has admissible blocks (green) and partially admissible blocks (blue). The $\mathcal{H}^2$ matrix representation is made up of specific inadmissible and admissible blocks in levels 2 and 3 and the partially admissible blocks in level 3.

be compressed into the form eq. (4). Meanwhile, by the ID approximation of $K(X_5, \tilde{Y}_5)$ and $K(X_9, \tilde{Y}_9)$ in the standard $\mathcal{H}^2$ matrix representation, the block $K(X_5, X_9)$ can be compressed into the form eq. (3).

Since $\tilde{Y}_i$ in the standard $\mathcal{H}^2$ matrix representation is defined as a superset of $Y_i$ for each node $i$, $K(X_i, \tilde{Y}_i)$ has larger numerical rank than $K(X_i, Y_i)$ (can be much larger in rare cases), leading to a larger rank for the approximation of each admissible or partially admissible block $K(X_i, X_j)$. Thus, the $\mathcal{H}^2$ matrix representation using partially admissible blocks introduced in this paper typically has smaller storage cost and faster matrix-vector multiplications than the standard $\mathcal{H}^2$ matrix representation. The concept of partially admissible blocks has a counterpart in FMM and is necessary for the exact equivalence between $\mathcal{H}^2$ matrix-vector multiplication and FMM [30].

### 2.2 $\mathcal{H}^2$ matrix construction

$\mathcal{H}^2$ matrix construction consists of two parts: (1) computing the ID approximation of $K(X_i, Y_i)$ for each node $i$ with non-empty $\mathcal{F}_i$ via a nested approach and (2) computing the intermediate blocks associated with $\mathcal{A} \cup \mathcal{A}_p$ and the inadmissible blocks associated with $\mathcal{D}$. As just mentioned in the previous paragraph, the second part is optional. The nested approach to computing these ID approximations is as follows.

For a leaf node $i$, the ID approximation of $K(X_i, Y_i)$ is directly computed using the proxy point method (to be described in Section 2.3). For a non-leaf node $i$ with children $\{i_1, i_2, \ldots, i_s\}$, the ID approximations associated with all these children nodes must be computed first. Then, since $X_i = X_{i_1} \cup \ldots \cup X_{i_s}$, $K(X_i, Y_i)$ can be split into blocks $K(X_{i_a}, Y_i)$ with $i_a \in \{i_1, i_2, \ldots, i_s\}$. By definition, the points in $Y_i$ are in the far field of box $i$ and thus are also in the far field of each child box $i_a$, i.e., $Y_i \subseteq Y_{i_a}$. As a result, the computed ID approximation $K(X_{i_a}, Y_{i_a}) \approx U_{i_a} K(X^{\mathrm{id}}_{i_a}, Y_{i_a})$ associated with $i_a$ gives the approximation $K(X_{i_a}, Y_i) \approx U_{i_a} K(X^{\mathrm{id}}_{i_a}, Y_i)$. Together, $K(X_i, Y_i)$ is split and approximated as,

$$K(X_i, Y_i) = \begin{bmatrix} K(X_{i_1}, Y_i) \\ K(X_{i_2}, Y_i) \\ \vdots \\ K(X_{i_s}, Y_i) \end{bmatrix} \approx \begin{bmatrix} U_{i_1} K(X^{\mathrm{id}}_{i_1}, Y_i) \\ U_{i_2} K(X^{\mathrm{id}}_{i_2}, Y_i) \\ \vdots \\ U_{i_s} K(X^{\mathrm{id}}_{i_s}, Y_i) \end{bmatrix} = \begin{bmatrix} U_{i_1} & & & \\ & U_{i_2} & & \\ & & \ddots & \\ & & & U_{i_s} \end{bmatrix} \begin{bmatrix} K(X^{\mathrm{id}}_{i_1}, Y_i) \\ K(X^{\mathrm{id}}_{i_2}, Y_i) \\ \vdots \\ K(X^{\mathrm{id}}_{i_s}, Y_i) \end{bmatrix}. \tag{5}$$

Denoting $\hat{X}_i = X_{i_1}^{\mathrm{id}} \cup X_{i_2}^{\mathrm{id}} \cup \ldots \cup X_{i_s}^{\mathrm{id}}$, an ID approximation of the last block above $K(\hat{X}_i, Y_i)$ is computed using the proxy point method as,

$$K(\hat{X}_i, Y_i) \approx R_i K(X_i^{\mathrm{id}}, Y_i), \quad X_i^{\mathrm{id}} \subseteq \hat{X}_i \subseteq X_i.$$

Plugging this approximation into eq. (5), we get the ID approximation $K(X_i, Y_i) \approx U_i K(X_i^{\mathrm{id}}, Y_i)$ with $U_i$ defined in the nested form eq. (2) using the computed $R_i$.

### 2.3 The proxy point method

The $\mathcal{H}^2$ matrix construction above is dominated by the ID approximation of $K(X_i, Y_i)$ for leaf nodes $i$ and $K(\hat{X}_i, Y_i)$ for non-leaf nodes $i$. All these approximated kernel blocks share the same form $K(X_*, Y_*)$ where $X_*$ is a set of points in a box $\mathcal{X}$ and $Y_*$ is a set of points in a compact subdomain $\mathcal{Y}$ of the far field of $\mathcal{X}$, as illustrated in Figure 4. In general, $Y_*$ has far more points than $X_*$. The proxy point method [31] can efficiently construct an ID approximation of $K(X_*, Y_*)$ with $X_* \times Y_*$ lying in a pair of compact domains $\mathcal{X} \times \mathcal{Y}$ as follows.

First select a set of so-called *proxy points* $Y_p$ in $\mathcal{Y}$ following the selection scheme Algorithm 1 (to be described later). Then compute an ID approximation of $K(X_*, Y_p)$ algebraically using the pivoted QR decomposition as $K(X_*, Y_p) \approx U_* K(X_*^{\mathrm{id}}, Y_p)$ with $X_*^{id} \subseteq X_*$. Using the computed $U_*$ and $X_*^{\mathrm{id}}$, the ID approximation of $K(X_*, Y_*)$ is then directly defined as $K(X_*, Y_*) \approx U_* K(X_*^{\mathrm{id}}, Y_*)$. In most cases, $Y_p$ has far fewer points that $Y_*$ and thus the above proxy point method is far more efficient than the direct ID approximation of $K(X_*, Y_*)$. Numerically, when a relative error threshold $\varepsilon_{\mathrm{id}}$ is used for the algebraic ID approximation of $K(X_*, Y_p)$, the defined ID approximation of $K(X_*, Y_*)$ usually has relative error approximately $\varepsilon_{\mathrm{id}}$.

*Selection of the proxy points.* The selection scheme given in Algorithm 1 was proposed in Ref. [31]. The basic idea is to first discretize $K(x, y)$ in $\mathcal{X} \times \mathcal{Y}$ into matrix $K(X_1, Y_1)$. Steps 2 and 3 in this algorithm compresses this matrix as $K(X_1, Y_1) \approx K(X_1, Y_p) K(X_p, Y_p)^{-1} K(X_p, Y_1)$ with $O(\varepsilon_p)$ error. Due to the low-rank property of $K(x, y)$, it can be proved that, if $|X_1|$ and $|Y_1|$ are sufficiently large,

$$K(x, y) \approx K(x, Y_p) K(X_p, Y_p)^{-1} K(X_p, y) + O(\varepsilon_p), \quad (x, y) \in \mathcal{X} \times \mathcal{Y},$$

$$\xrightarrow{\text{plug in } X_*, Y_*} K(X_*, Y_*) \approx K(X_*, Y_p) K(X_p, Y_p)^{-1} K(X_p, Y_*) + O(\varepsilon_p).$$

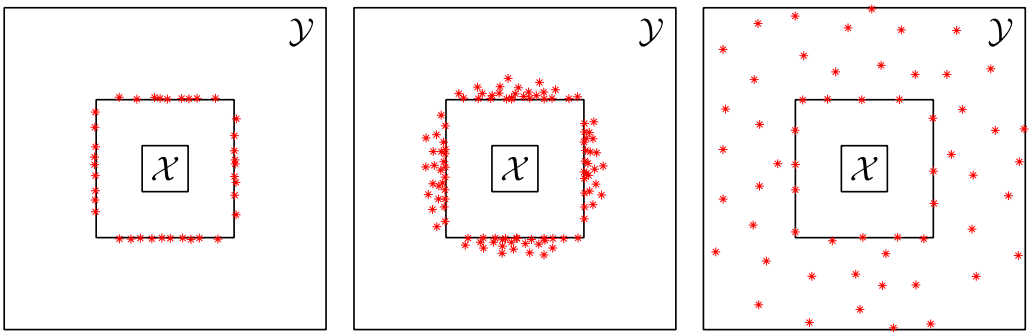
The proxy point method exactly computes an ID approximation of $K(X_*, Y_p)$ and thus can also be viewed as a recompression of the above $O(\varepsilon_p)$-accuracy approximation of $K(X_*, Y_*)$. Usually, the parameter $\varepsilon_p$ can be set to one or two orders of magnitudes smaller than the error threshold specified for the proxy point method. The sizes of $X_1$ and $Y_1$ should be large enough to guarantee the accuracy $O(\varepsilon_p)$ of the above function approximation to $K(x, y)$, and also to guarantee well-boundedness of this specific vector function $K(X_p, Y_p)^{-1} K(X_p, y)$ in $\mathcal{Y}$ which is critical for the accuracy of the proxy point method. More explanations can be found in [31].

This selection scheme is computationally expensive and only depends on $K(x, y)$ and $\mathcal{X} \times \mathcal{Y}$. With more sample points $X_1$ and $Y_1$, the set of proxy points $Y_p$ selected by Algorithm 1 is more effective in terms of controlling the accuracy of the proxy point method based on $Y_p$, but Algorithm 1 becomes more expensive. In H2Pack, the numbers of sample points in $X_1$ and $Y_1$ in Algorithm 1 are heuristically chosen. We used $|X_1| = 1000$ and $|Y_1| = 15000$ for the various kernel functions and pairs of domains that were tested numerically (see Section 4). An adaptive choice of the number of sample points can be developed and applied if necessary. The ID approximation of $K(X_1, Y_1)$ at Step 2 of Algorithm 1 is computed using a randomized method [22] instead of the pivoted QR decomposition, for better efficiency. Figure 4 illustrates several examples of the selected proxy points for different kernel functions.

**Algorithm 1** Proxy point selection scheme

**Input:** $K(x, y)$, $\mathcal{X}$, $\mathcal{Y}$, $\varepsilon_p$.
**Output:** proxy points $Y_p$.

1: Sample domains $\mathcal{X}$ and $\mathcal{Y}$ to obtain two sets of uniformly distributed points $X_1$ and $Y_1$ with high point density, respectively.
2: Compute an ID approximation $K(X_1, Y_1) \approx U_1 K(X_p, Y_1)$ with error threshold $\varepsilon_p \sqrt{|Y_1|}$.
3: Compute a pivoted QR decomposition $K(X_p, Y_1)P = Q(R_1, R_2)$ where $P$ is a permutation matrix, $Q$ is an orthogonal matrix, and $R_1$ is an $|X_p| \times |X_p|$ upper-triangular matrix.
4: Let $Y_p$ be the subset of points in $Y_1$ that corresponds to the $|X_p|$ columns of $R_1$ after permutation.




(a) $K(x, y) = \log(|x - y|)$ (b) $K(x, y) = \exp(-|x - y|^2)$ (c) $K(x, y) = \exp(-0.1|x - y|^2)$

Fig. 4. Examples of the proxy points selected by Algorithm 1 for various kernel functions with $\mathcal{X} = [-1, 1]^2$, $\mathcal{Y} = [-7, 7]^2 \setminus [-3, 3]^2$, and $\varepsilon_p = 10^{-10}$. The three sets have 37, 103, and 58 proxy points, respectively.

Applying Algorithm 1 to select proxy points for each ID approximation in $\mathcal{H}^2$ matrix construction is expensive and impractical. Instead, we can reuse a set of selected proxy points $Y_p$ for all the ID approximations associated with nodes in one level of the construction. Specifically, note that all the boxes in the same level are of the same size and $K(x, y)$ is assumed to be translationally-invariant. Thus, at each level $l$, we select $\mathcal{X}$ as a box in level $l$ and $\mathcal{Y}$ as a large compact subdomain of the far field of $\mathcal{X}$, and apply Algorithm 1 with $\mathcal{X} \times \mathcal{Y}$ to select a set of proxy points $Y_p^l$. For each node $i$ in level $l$, let $z_i$ be a translation vector such that $X_i + z_i$ lies in $\mathcal{X}$ and $Y_i + z_i$ lies in $\mathcal{Y}$ ($\mathcal{Y}$ should be selected large enough to contain $Y_i + z_i$ for each node $i$). Since $K(X_i, Y_i) = K(X_i + z_i, Y_i + z_i)$, we can apply the proxy point method with the shifted proxy points $Y_p^l - z_i$ to compute the ID approximation of $K(X_i, Y_i)$ (or $K(\hat{X}_i, Y_i)$).

As a result, at each level, we only need to construct a set of proxy points $Y_p^l$ for just one pair of domains $\mathcal{X} \times \mathcal{Y}$. The corresponding proxy points for all the nodes in one level can be obtained by proper translation of $Y_p^l$. Also, another option is to precompute and store multiple sets of proxy points for box domains $\mathcal{X}$ of different sizes (with sufficiently large domains $\mathcal{Y}$) given a kernel function. In $\mathcal{H}^2$ matrix construction, we simply need to load the corresponding proxy point set based on the box domain size in each level. Combining the proxy point method with the $\mathcal{H}^2$ matrix construction described in the last subsection, the overall $\mathcal{H}^2$ matrix construction for a kernel matrix $K(X, X)$ is summarized in Algorithm 2.

**Algorithm 2** $\mathcal{H}^2$ matrix construction for $K(X, X)$

• construct a hierarchical partitioning of $X$ which gives a $L$-level partition tree $\mathcal{T}$.
**for** $l = L, L-1, \ldots, 1$ **do**
  • construct a set of proxy points $Y_p^l$ for just one box in level $l$.
  **parfor all** nodes $i$ in level $l$ **do** (dynamic scheduling)
    **if** $i$ is a leaf node **then**
      • compute $U_i$ and $X_i^{\text{id}}$ from an ID approximation of $K(X_i, Y_i)$ using the proxy point method with a proper translation of $Y_p^l$.
    **else if** $i$ is a non-leaf node with children $\{i_1, i_2, \ldots, i_s\}$ **then**
      • construct $\hat{X}_i^{\text{id}} = X_{i_1}^{\text{id}} \cup \ldots \cup X_{i_s}^{\text{id}}$.
      • compute $R_i$ and $X_i^{\text{id}}$ from an ID approximation of $K(\hat{X}_i, Y_i)$ using the proxy point method with a proper translation of $Y_p^l$.
    **end if**
  **end parfor**
**end for**
• (optional, can be dynamically computed) compute the inadmissible blocks $D_{i,j}$ for all $(i, j) \in \mathcal{D}$ and compute the intermediate blocks $B_{i,j}$ for all $(i, j) \in \mathcal{A} \cup \mathcal{A}_p$.

### 2.4 $\mathcal{H}^2$ matrix-vector multiplication

Consider computing $b = K(X, X)q$. For each node $i \in \mathcal{T}$, let $q_i$ and $b_i$ denote the subvectors of $q$ and $b$, respectively, corresponding to the point subset $X_i$ in $X$. The $\mathcal{H}^2$ matrix-vector multiplication algorithm [21], summarized in Algorithm 3, traverses all three sets of kernel blocks $K(X_i, X_j)$ corresponding to $\mathcal{D}$, $\mathcal{A}$, and $\mathcal{A}_p$ in the $\mathcal{H}^2$ matrix representation and accumulates the products $K(X_i, X_j)q_j$.

First, initialize the result vector $b$ to zero. For each inadmissible block $K(X_i, X_j)$ with $(i, j) \in \mathcal{D}$, the dense matrix computation is straightforward: $b_i = b_i + K(X_i, X_j)q_j$. For each admissible block $K(X_i, X_j)$ with $(i, j) \in \mathcal{A}$, the computation

$$b_i = b_i + K(X_i, X_j)q_j \approx b_i + U_i B_{i,j} U_j^T q_j,$$

can be computed in three steps $U_j^T q_j$, $B_{i,j}(U_j^T q_j)$, and $b_i = b_i + U_i \left( B_{i,j} \left( U_j^T q_j \right) \right)$ giving three phases in $\mathcal{H}^2$ matrix-vector multiplication: forward transformation, intermediate multiplication, and backward transformation.

*Forward transformation.* This phase computes $y_j = U_j^T q_j$ for all the nodes $j \in \mathcal{T}$. Note that $y_j$ can be used for all the admissible blocks with columns defined by $X_j$. For each leaf node $j$, $y_j$ is directly computed. For each non-leaf node $j$ with children $\{j_1, j_2, \ldots, j_s\}$, $y_j$ is recursively computed using $y_{j_1}, y_{j_2}, \ldots, y_{j_s}$ associated with the children as

$$y_j = U_j^T q_j = R_j^T \begin{bmatrix} U_{j_1}^T & & \\ & \ddots & \\ & & U_{j_s}^T \end{bmatrix} \begin{bmatrix} q_{j_1} \\ \vdots \\ q_{j_s} \end{bmatrix} = R_j^T \begin{bmatrix} U_{j_1}^T q_{j_1} \\ \vdots \\ U_{j_s}^T q_{j_s} \end{bmatrix} = R_j^T \begin{bmatrix} y_{j_1} \\ \vdots \\ y_{j_s} \end{bmatrix}.$$

*Intermediate multiplication.* This phase computes $z_{i,j} = B_{i,j} y_j$ for each admissible block $K(X_i, X_j)$ with $(i, j) \in \mathcal{A}$. Note that all the $z_{i,j}$ sharing the node $i$ are to be multiplied by $U_i$ and added to $b_i$ as

$$b_i = b_i + \textstyle\sum_{(i,j)\in\mathcal{A}} U_i z_{i,j} = b_i + U_i \left( \sum_{(i,j)\in\mathcal{A}} z_{i,j} \right), \quad \text{for each node } i \in \mathcal{T}.$$

Only multiplying $U_i$ once, it is more efficient to first sum over all these $z_{i,j}$, then apply $U_i$, and lastly add to $b_i$. Thus, for each node $i \in \mathcal{T}$, this phase further computes $z_i = \sum_{(i,j)\in\mathcal{A}} z_{i,j}$.

*Backward transformation.* This phase computes $b_i = b_i + U_i z_i$ for each node $i \in \mathcal{T}$. For a non-leaf node $i$ with children $\{i_1, i_2, \ldots, i_s\}$, $b_i$ is recursively accumulated as

$$b_i = b_i + U_i z_i = b_i + \begin{bmatrix} U_{i_1} & & \\ & \ddots & \\ & & U_{i_s} \end{bmatrix} R_i z_i = b_i + \begin{bmatrix} U_{i_1}[R_i z_i]_{i_1} \\ \vdots \\ U_{i_s}[R_i z_i]_{i_s} \end{bmatrix}$$

where $[R_i z_i]_{i_a}$ denotes the subvector of $R_i z_i$ associated with $U_{i_a}$. Thus, $b_i = b_i + U_i z_i$ is reduced to $b_{i_a} = b_{i_a} + U_{i_a}[R_i z_i]_{i_a}$ with all the children $i_a$. Meanwhile, $b_{i_a} = b_{i_a} + U_{i_a} z_{i_a}$ needs to be computed as well. Only multiplying $U_{i_a}$ once, it is more efficient to first overwrite $z_{i_a}$ by $z_{i_a} = z_{i_a} + [R_i z_i]_{i_a}$ and then multiply $U_{i_a}$ by $z_{i_a}$. Recursively, this phase traverses the tree from the root to the leaves to overwrite each $z_i$ by $z_i = z_i + [R_p z_p]_i$ with $p$ the parent of $i$. As a result, for each leaf node $i$, $z_i$ accumulates the intermediate multiplication results from all its ancestors. Adding $U_i z_i$ to $b_i$ for all the leaf nodes in $\mathcal{T}$ finishes this phase. See the lines 21-28 in Algorithm 3 for the exact calculation.

For each partially admissible block $K(X_i, X_j)$ with $(i, j) \in \mathcal{A}_p$, its multiplication by $p_j$,

$$b_i = b_i + U_i B_{i,j} q_j \quad \text{or} \quad b_i = b_i + B_{i,j} U_j^T q_j$$

can be similarly computed following the above process for the admissible blocks. In fact, these multiplications can be merged into the above three phases for admissible blocks.

*$\mathcal{H}^2$ matrix-matrix multiplication.* Consider computing $C = K(X, X)Q$. It is straightforward to extend the above $\mathcal{H}^2$ matrix-vector multiplication to the multiplication by multiple vectors simultaneously. We only need to replace vectors $q_i$, $b_i$, $y_i$, and $z_i$ in Algorithm 3 by matrices $Q_i$, $C_i$, $Y_i$, and $Z_i$, respectively, where $Q_i$ and $C_i$ are the row subsets of $Q$ and $C$ associated with $X_i$.

## 3 PARALLEL IMPLEMENTATION

### 3.1 Parallelization and load-balancing

In Section 2, we presented $\mathcal{H}^2$ matrix construction ($\mathcal{H}^2$-construction) and $\mathcal{H}^2$ matrix-vector multiplication ($\mathcal{H}^2$-matvec). For the parallel implementation of these two operations, we consider calculation dependencies associated with each node in the partition tree. In $\mathcal{H}^2$-construction, the first step is to compute specific ID approximations associated with each node $i$ with nonempty $\mathcal{F}_i$. In this step, the ID approximation at a non-leaf node cannot be computed until the ID approximations at all its children nodes are computed, corresponding to a post-order traversal of the partition tree. The optional step of computing inadmissible and intermediate blocks has no restriction on calculation orders for each block. In $\mathcal{H}^2$-matvec, the forward transformation phase has the same calculation order as the ID approximations in $\mathcal{H}^2$-construction, i.e., the calculation of $y_i$ for a non-leaf node $i$ requires the calculation of $\{y_{i_k}\}$ with the children $\{i_k\}$ of $i$. The backward transformation phase has calculation order reverse to that of the forward transformation phase, corresponding to a pre-order traversal of the partition tree. Meanwhile, there is no restriction on the calculation order in the intermediate and dense multiplication phases, since the matrix-vector multiplications by different $B_{i,j}$ and $D_{i,j}$ are completely independent.

Based on the above observations, the calculations in $\mathcal{H}^2$-construction and $\mathcal{H}^2$-matvec can be categorized into two types. The first type is *level-by-level calculation*, where the calculation at node $i$ rely on the calculations at nodes on the level above or below. The second type is *independent calculation*, where the calculations associated with different $B_{i,j}$ or $D_{i,j}$ are independent. We apply different strategies to parallelize these two types of calculations within the OpenMP framework.

**Algorithm 3** $\mathcal{H}^2$ matrix-vector multiplication

1: • Initialize result vector $b$ to zero.
2: • Initialize temporary vectors $z_i, \forall i \in \mathcal{T}$ to zero.

▷ **Step 1:** Forward transformation

3: **for** $l = L, L-1, \ldots, 1$ **do**
4: **parfor all** nodes $i$ in level $l$ **do** (greedy static partitioning)
5: **if** $i$ is a leaf node **then**
6: $y_i = U_i^T q_i$.
7: **else**
8: $y_i = R_i^T (y_{i_1}^T, y_{i_2}^T, \ldots, y_{i_s}^T)^T$ with children $\{i_1, i_2, \ldots, i_s\}$ of node $i$.
9: **end if**
10: **end parfor**
11: **end for**

▷ **Step 2:** Intermediate multiplication

12: **parfor all** $(i, j) \in \mathcal{A}$ **do** (hybrid load balancing)
13: $z_i = z_i + B_{i,j} y_j$.
14: **end parfor**
15: **parfor all** $(i, j) \in \mathcal{A}_p$ **do** (hybrid load balancing)
16: **if** $K(X_i, X_j) \approx U_i B_{i,j}$ **then**
17: $z_i = z_i + B_{i,j} q_j$.
18: **else** *(note:* $K(X_i, X_j) \approx B_{i,j} U_j^T$*)*
19: $b_i = b_i + B_{i,j} y_j$.
20: **end if**
21: **end parfor**

▷ **Step 3:** Backward transformation

22: **for** $l = 1, 2, \ldots, L$ **do**
23: **parfor all** non-leaf node $i$ in level $l$ **do** (greedy static partitioning)
24: $z_{i_a} = z_{i_a} + [R_i z_i]_{i_a}$ with all children $i_a \in \{i_1, i_2, \ldots, i_s\}$ of node $i$.
25: **end parfor**
26: **end for**
27: **parfor all** leaf node $i$ in $\mathcal{T}$ **do** (hybrid load balancing)
28: $b_i = b_i + U_i z_i$.
29: **end parfor**

▷ **Step 4:** Dense multiplication

30: **parfor all** $(i, j) \in \mathcal{D}$ **do** (hybrid load balancing)
31: $b_i = b_i + D_{i,j} q_j$.
32: **end parfor**

*Level-by-level calculations.* Let the calculation at node $i$ in a level-by-level computation phase be referred to as *task i*. In the following phases, task $i$ needs the results of multiple tasks in a lower level or the result of a task in an upper level:

- the ID approximations in $\mathcal{H}^2$-construction (lines 2-12 in Algorithm 2),
- the forward transformation in $\mathcal{H}^2$-matvec (lines 3-11 in Algorithm 3),
- the backward transformation in $\mathcal{H}^2$-matvec (lines 22-29 in Algorithm 3).

In these computational tasks, the accessed matrices $K(X_i, Y_p)$, $K(\hat{X}_i, Y_p)$, $U_i$, and $R_i$ usually have size smaller than $1000 \times 1000$. For such small matrices, the column-pivoted QR and matrix-vector multiplications usually have poor parallel performance when using a large number of processors. Instead of parallelizing these elementary computational kernels, we choose to parallelize across the tasks in each level of $\mathcal{T}$. Specifically, we parallelize the for-loops in line 4 of Algorithm 2 and in lines 4, 23, and 27 of Algorithm 3 with OpenMP.

We use different load-balancing strategies for the three computation phases listed above. In $\mathcal{H}^2$-construction, since the size of $K(\hat{X}_i, Y_p)$ at each non-leaf node is not known in advance, we use OpenMP dynamic scheduling to balance the workload in the parallel loop of ID approximations in each level. In $\mathcal{H}^2$-matvec, the performance bottleneck of the forward and backward transformations is the transfer of $U_i$ and $R_i$ from memory to processors. Since the sizes of $U_i$ and $R_i$ are known at this stage, we use a greedy static partitioning scheme to approximately balance the total sizes of matrices that each processor needs to load from memory.

*Independent calculations.* Let calculations associated with a block $B_{i,j}$ or $D_{i,j}$ in an independent computation phase be referred to as *task* $(i, j)$ with $(i, j)$ in the node pair sets $\mathcal{A} \cup \mathcal{A}_p$ or $\mathcal{D}$. In the following phases, all tasks are independent and can be performed in parallel without restriction:

- the optional construction of $B_{i,j}$ and $D_{i,j}$ in $\mathcal{H}^2$-construction (lines 13 in Algorithm 2),
- the intermediate multiplication phase in $\mathcal{H}^2$-matvec (lines 12-21 in Algorithm 3),
- the dense multiplication phase in $\mathcal{H}^2$-matvec (lines 30-32 in Algorithm 3).

Note that, for each $B_{i,j}$ or $D_{i,j}$, both the computation cost of forming it and the communication cost of transferring it from memory to a processor are proportional to its block size which is known after the ID approximations in $\mathcal{H}^2$-construction.

We first consider exploiting the symmetry property of these blocks $B_{i,j}$ and $D_{i,j}$. Since $K(X, X)$ is symmetric, $B_{i,j} = B_{j,i}^T$ if $(i, j)$ is in $\mathcal{A} \cup \mathcal{A}_p$ (corresponding to an admissible or partially admissible block) and $D_{i,j} = D_{j,i}^T$ if $(i, j)$ is in $\mathcal{D}$ (corresponding to an inadmissible block). Thus, for each pair of $(i, j)$ and $(j, i)$ in $\mathcal{A} \cup \mathcal{A}_p$, only $B_{i,j}$ is computed and the following two matrix-vector multiplications in the intermediate multiplication phase will be performed on one processor simultaneously:

$$z_i = z_i + B_{i,j} y_j, \quad z_j = z_j + B_{i,j}^T y_i.$$

The same approach applies to each pair of $(i, j)$ and $(j, i)$ in $\mathcal{D}$ with blocks $D_{i,j}$.

We use a hybrid approach for parallelizing and load-balancing the independent calculations. In this hybrid approach, a static partitioning is used for approximately balancing the workload on each processor and a dynamic task scheduler is used for polishing the load balance. We use the construction of blocks $B_{i,j}$ to illustrate this approach. For $P$ processors, we partition all tasks into $kP$ disjoint task units ($1 \leq k \leq 20$ is a prescribed constant) with a greedy algorithm such that the total size of matrix blocks in each task unit is approximately the same. Each processor has $k - 1$ initial task units, which leads to approximately the same computation time for initial task units on each processor. The last $P$ task units form a task pool for dynamic task scheduling. After finishing all its $k - 1$ initial task units, a processor starts to steal task units one by one from the task pool until all task units have been consumed. If $k = 1$, the hybrid approach is equivalent to a static task partitioning scheme. The construction of blocks $D_{i,j}$, the intermediate multiplication phase, and the dense multiplication phase are all parallelized in the same way.

Combining the utilization of the symmetry property and the hybrid parallelization approach causes a new problem. In the intermediate and dense multiplication phases, two or more processors may update the same $z_i$ or $b_i$ simultaneously, leading to incorrect results. Three solutions to this problem are available. The first is to discard utilizing the symmetry property and then to partition the corresponding tasks in a way that each $z_i$ and $b_i$ can be updated by only one processor. This

approach is unfavorable since it leads to more data transfer between memory and processors and higher computation cost. The second solution is to use atomic operations for updating $z_i$ and $b_i$. However, atomic operations are much slower than their non-atomic counterparts. In H2Pack, we use the third solution that each processor uses local copies of $z_i$ and $b_i$ to accumulate local matrix-vector multiplication results. All local copies of $z_i/b_i$ are summed after the intermediate/dense multiplication phase to form the actual $z_i/b_i$. The additional cost for summing local copies of $z_i$ and $b_i$ is negligible compared to the main phases in $\mathcal{H}^2$-matvec.

For multiplying multiple vectors simultaneously, H2Pack provides a separate $\mathcal{H}^2$ matrix-matrix multiplication ($\mathcal{H}^2$-matmul) function. In $\mathcal{H}^2$-matmul, vectors $q_i$, $b_i$, $y_i$, and $z_i$ in $\mathcal{H}^2$-matvec are replaced by blocks $Q_i$, $C_i$, $Y_i$, and $Z_i$, and the matrix-vector multiplications in $\mathcal{H}^2$-matvec are replaced by matrix-matrix multiplications. The multiplicand matrix $Q$ could be stored in either row-major or column-major format, with the output matrix $C$ stored in the same format. $\mathcal{H}^2$-matmul adopts almost the same parallelization and load-balancing scheme as $\mathcal{H}^2$-matvec. One exception is that $\mathcal{H}^2$-matmul does not utilize the symmetry property of $B_{i,j}$ and $D_{i,j}$ blocks. Instead, independent calculation tasks with $B_{i,j}$ and $D_{i,j}$ are partitioned in a way that each $Z_i$ and $C_i$ is only updated by one processor. Processor-local $Z_i$ and $C_i$ copies are not used since they could require a large amount of memory.

### 3.2 Performance optimizations

We optimize H2Pack for state-of-the-art multi-core and many-core architectures. We first introduce the H2Pack kernel function interface in Section 3.2.1. Next, we discuss two running modes of H2Pack in Section 3.2.2. Then, we illustrate the use of intrinsic functions for better vectorization in Section 3.2.3.

*3.2.1 Kernel function interface.* The performance of H2Pack relies on the performance of evaluating the kernel function. $\mathcal{H}^2$-construction and $\mathcal{H}^2$-matvec using just-in-time mode (to be discussed in Section 3.2.2) both need to evaluate a large number of kernel matrix blocks. H2Pack provides an optimized implementation of the 3D Laplace kernel $K(x, y) = 1/|x - y|$ which can be modified easily for other kernel functions. In the following, we use the 3D Laplace kernel as an example to show the H2Pack kernel function interface.

H2Pack provides a C language interface. A driver program must provide a pointer to a *kernel matrix evaluation* (KME) function to use H2Pack. Listing 1 shows a KME function for the 3D Laplace kernel. Lines 2-4 in Listing 1 are parameters of a KME function: two sets of point coordinates `coord0` and `coord1` and the kernel matrix `kmat` to be returned. Input `coord0` is a `3×n0` row-major matrix with leading dimension `ld0` and contains the coordinates of `n0` points. Each column of `coord0` stores a point coordinate. The same storage scheme applies to `coord1`. The function returns an `n0×n1` kernel matrix stored in a row-major matrix `kmat` with leading dimension `ldm`. Note that a KME function should be single-threaded and should only use variables or memory that can be updated by the current thread.

The above design of a KME function is in order to facilitate the vectorization of multiple kernel function evaluations. It would be easier for users to program a function evaluating the kernel function for just a single pair of points. However, such a single-value function must be compiled together with H2Pack so that the compiler can auto-vectorize multiple kernel function evaluations. Using KME functions is more flexible: H2Pack only needs to be compiled once for different KME functions, and a KME function can be auto-vectorized by the compiler (line 14 in Listing 1) or manually vectorized (to be discussed in Section 3.2.3).

*3.2.2 Ahead-of-time and just-in-time running modes.* H2Pack provides two running modes of $\mathcal{H}^2$-construction and $\mathcal{H}^2$-matvec: (1) ahead-of-time (AOT) mode computes and stores all $B_{i,j}$

Listing 1. Sample KME function for the 3D Laplace kernel

```
void Laplace_3D_krnl_eval(
    const double *coord0, const int ld0, const int n0,
    const double *coord1, const int ld1, const int n1,
    double * restrict kmat, const int ldm
)
{
    const double *x0 = coord0 + ld0 * 0, *x1 = coord1 + ld1 * 0;
    const double *y0 = coord0 + ld0 * 1, *y1 = coord1 + ld1 * 1;
    const double *z0 = coord0 + ld0 * 2, *z1 = coord1 + ld1 * 2;
    for (int i = 0; i < n0; i++)
    {
        double x0i = x0[i], y0i = y0[i], z0i = z0[i];
        double *kmat_i = kmat + i * ldm;
        #pragma omp simd   // Requires the compiler to vectorize this loop
        for (int j = 0; j < n1; j++)
        {
            double dx = x1[j] - x0i;
            double dy = y1[j] - y0i;
            double dz = z1[j] - z0i;
            double r2 = dx * dx + dy * dy + dz * dz;
            double rinv = (r2 == 0.0) ? 0.0 : 1.0 / sqrt(r2);
            kmat_i[j] = rinv;
        }
    }
}
```

and $D_{i,j}$ in $\mathcal{H}^2$-construction, and (2) just-in-time (JIT) mode computes each $B_{i,j}$ and $D_{i,j}$ when needed in $\mathcal{H}^2$-matvec without storing them. These two modes give flexibility in how to obtain performance on different computing platforms for different kernel functions. We note that any implementations of $\mathcal{H}^2$ matrix representations based on ID approximations, e.g., the SMASH library and the STRUMPACK library, can also use both AOT and JIT modes.

The AOT mode is designed to avoid redundant calculation when the cost of kernel function evaluation is high. To form $B_{i,j}$ and $D_{i,j}$, a large number of kernel function evaluations are needed. Kernel functions with transcendental arithmetic (e.g., the Gaussian kernel $K(x, y) = \exp(|x - y|^2)$ and the logarithm kernel $K(x, y) = \log(|x - y|)$) have much higher evaluation cost than those without transcendental arithmetic. In such cases, using AOT mode could be more efficient than using JIT mode for $\mathcal{H}^2$-matvec. As a trade-off, AOT mode has much larger storage cost than JIT mode due to the storage of $B_{i,j}$ and $D_{i,j}$.

The performance bottleneck of $\mathcal{H}^2$-matvec in AOT mode is the transfer of $B_{i,j}$ and $D_{i,j}$ from memory to processors. Two optimizations are proposed for $\mathcal{H}^2$-matvec in AOT mode, targeting the intermediate and dense multiplication phases. First, we optimize for non-uniform memory access (NUMA) architectures. The memory for $B_{i,j}$ and $D_{i,j}$ blocks used by a processor is bound to its NUMA node to reduce memory access latency and to fully utilize memory bandwidth of all NUMA nodes in a computer. Second, we implement a bi-matrix-vector multiplication (BMV) function that computes $Ax_0$ and $A^T x_1$ with a matrix $A$ and two input vectors $x_0$, $x_1$ simultaneously to avoid loading the same $B_{i,j}$ or $D_{i,j}$ block twice from memory or processor cache. This function is not available in any existing optimized linear algebra library.

The JIT mode is designed to reduce the storage cost of an $\mathcal{H}^2$ matrix representation. The total size of all $B_{i,j}$ and $D_{i,j}$ blocks is usually 10 to 100 times larger than that of other $\mathcal{H}^2$ matrix

Listing 2. Sample BKM function for the 3D Laplace kernel

```
void Laplace_3D_bi_krnl_matvec(
    const double *coord0, const int ld0, const int n0,
    const double *coord1, const int ld1, const int n1,
    const double *xin0, const double *xin1,
    double * restrict xout0, double * restrict xout1
)
{
    const double *x0 = coord0 + ld0 * 0, *x1 = coord1 + ld1 * 0;
    const double *y0 = coord0 + ld0 * 1, *y1 = coord1 + ld1 * 1;
    const double *z0 = coord0 + ld0 * 2, *z1 = coord1 + ld1 * 2;
    for (int i = 0; i < n0; i++)
    {
        double x0i = x0[i], y0i = y0[i], z0i = z0[i], xin1_i = xin1[i];
        double sum_i = 0.0;
        #pragma omp simd   // Requires the compiler to vectorize this loop
        for (int j = 0; j < n1; j++)
        {
            double dx = x1[j] - x0i;
            double dy = y1[j] - y0i;
            double dz = z1[j] - z0i;
            double r2 = dx * dx + dy * dy + dz * dz;
            double rinv = (r2 == 0.0) ? 0.0 : 1.0 / sqrt(r2);
            sum_i    += rinv * xin0[j];
            xout1[j] += rinv * xin1;
        }
        xout0[i] += sum_i;
    }
}
```

components. For a given memory size, by not storing $B_{i,j}$ and $D_{i,j}$ blocks, H2Pack in JIT mode can handle problems with far more points. We use the cache-blocking technique to optimize the intermediate and dense multiplication phases in JIT mode. Specifically, we partition $B_{i,j}$ or $D_{i,j}$ into multiple subblocks such that each subblock and the coordinates associated with this subblock can fit in processor L2 data cache. A small processor-private buffer is used for each processor to temporarily store a dynamically generated subblock. Once a subblock is generated, we use this subblock and the BMV function to compute two matrix-vector multiplications immediately. Since only the point coordinates need to be transferred from memory to processors, the intermediate and dense multiplication phases also have much smaller memory bandwidth pressure in JIT mode than in AOT mode.

We further design a *matrix-free* approach for $\mathcal{H}^2$-matvec in JIT mode using a *bi-kernel matvec* (BKM) function (note that $\mathcal{H}^2$-matmul does not use the BKM function). For two point sets $X_0$ and $X_1$, a BKM function calculates two matrix-vector multiplications $K(X_0, X_1)x_0$ and $K(X_1, X_0)x_1$ at the same time without explicitly storing any subblock of $K(X_0, X_1)$ or $K(X_1, X_0)$. Compared to using a KME function, using a BKM function eliminates the transferring of the dynamically generated subblocks of $B_{i,j}$ and $D_{i,j}$ between a processor and its L2 data cache. Listing 2 shows a sample BKM function for the 3D Laplace kernel. Lines 2-4 in Listing 2 are parameters of a BKM function: two sets of point coordinates `coord0` and `coord1` stored in the same way as in the KME function, two input vectors `xin0`, `xin1`, and two output vectors `xout0`, `xout1`. Input `xin0` stores $x_0$ and `xout0` stores the result of $K(X_0, X_1)x_0$. Input `xin1` stores $x_1$ and `xout0` stores the result of $K(X_1, X_0)x_1$. The only difference between a KME function and a BKM function is that once a kernel function

Listing 3. Sample KME function for the 3D Laplace kernel using vector wrapper functions

```
1  void Laplace_3D_krnl_eval_vec(
2      const double *coord0, const int ld0, const int n0,
3      const double *coord1, const int ld1, const int n1,
4      double * restrict kmat, const int ldm
5  )
6  {
7      const double *x0 = coord0 + ld0 * 0, *x1 = coord1 + ld1 * 0;
8      const double *y0 = coord0 + ld0 * 1, *y1 = coord1 + ld1 * 1;
9      const double *z0 = coord0 + ld0 * 2, *z1 = coord1 + ld1 * 2;
10     int n1_vec = (n1_vec / SIMD_LEN_D) * SIMD_LEN_D;
11     for (int i = 0; i < n0; i++)
12     {
13         double *kmat_i = kmat + i * ldm;
14         // Vectorized loop
15         vec_d x0i_v = vec_set1_d(x0[i]);
16         vec_d y0i_v = vec_set1_d(y0[i]);
17         vec_d z0i_v = vec_set1_d(z0[i]);
18         for (int j = 0; j < n1_vec; j += SIMD_LEN_D)
19         {
20             vec_d dx_v = vec_sub_d(vec_loadu_d(x1 + j), x0i_v);
21             vec_d dy_v = vec_sub_d(vec_loadu_d(y1 + j), y0i_v);
22             vec_d dz_v = vec_sub_d(vec_loadu_d(z1 + j), z0i_v);
23             vec_d r2_v = vec_mul_d(dx_v, dx_v);
24             r2_v = vec_fmadd_d(dy_v, dy_v, r2_v);
25             r2_v = vec_fmadd_d(dz_v, dz_v, r2_v);
26             vec_d rinv_v = vec_frsqrt_d(r2_v);
27             vec_storeu_d(kmat_i + j, rinv_v);
28         }
29         // Remainder loop
30         double x0i = x0[i], y0i = y0[i], z0i = z0[i];
31         for (int j = n1_vec; j < n1; j++)
32         {
33             double dx = x1[j] - x0i;
34             double dy = y1[j] - y0i;
35             double dz = z1[j] - z0i;
36             double r2 = dx * dx + dy * dy + dz * dz;
37             double rinv = (r2 == 0.0) ? 0.0 : 1.0 / sqrt(r2);
38             kmat_i[j] = rinv;
39         }
40     }
41 }
```

value is calculated (line 21 in both Listing 1 and 2), a KME function stores this value to a matrix but a BKM function consumes this value and discards it immediately. If the kernel function evaluation is cheap (for example, for the 3D Laplace kernel) and we have fast processors but moderate memory bandwidth, $\mathcal{H}^2$-matvec in JIT mode using a BKM function could be faster than $\mathcal{H}^2$-matvec in AOT mode.

*3.2.3 Vector intrinsics.* Effectively vectorizing the KME and BKM functions is critical to high performance of H2Pack. In general, KME and BKM functions for scalar kernels (kernels that return a single value for a pair of points) using the same framework as Listing 1 and 2 can be auto-vectorized by compilers. As an alternative, H2Pack provides a set of *vector wrapper functions* independent of

the processor instruction set for manually vectorizing calculations with intrinsic functions. The optimized KME and BKM functions provided in H2Pack for the 3D Laplace kernel use these vector wrapper functions. Currently the vector wrapper functions supports AVX, AVX2, and AVX-512 instruction sets on x86 architecture. (Other architectures can also be supported in the future.)

Listing 3 shows a sample KME function for the 3D Laplace kernel using vector wrapper functions. This function has two major parts in its inner loop: a manually vectorized loop using vector wrapper functions (lines 15-28) and a remainder loop (lines 30-39) using scalar operations. All vector wrapper functions are named as `vec_{opname}_{d/s}`, where `opname` is the operation name and the suffix indicates the floating point data type (*double (d)* or *float (s)*). Constant value `SIMD_LEN_D` indicates the number of double words in each `vec_d` vector data type. This constant is determined according to the processor instruction set and H2Pack compilation options. Vector wrapper functions used in Listing 3 are the most useful vector wrapper functions for programming KME and BKM functions. A detailed list of all vector wrapper functions and their usage can be found in the H2Pack user manual. For BKM functions, H2Pack automatically pads artificial points in `coord0`, `coord1` and pads extra zeros in `xin0`, `xin1` to guarantee that `n0` and `n1` are multiples of *SIMD_LEN_D*. The padding aims to simplify the programming of BKM functions since the remainder loop can be eliminated.

Calculating the reciprocal square root (RSQRT) is an expensive step in evaluating $1/|x-y|$, which appears in many kernel functions. We thus implement a fast RSQRT function with x86 intrinsic functions based on the approach proposed in Ref. [27]. In this fast RSQRT function, a dedicated intrinsic function is first used to calculate an approximate RSQRT value with relative error less than $4\times10^{-4}$. Then, two Newton-Raphson iterations are performed using the approximate RSQRT value as an initial guess to obtain a more accurate RSQRT result with $O(10^{-14})$ relative error. The same or similar approaches to computing RSQRT have also been used in some existing FMM implementations [7, 23].

## 4 NUMERICAL EXPERIMENTS

We consider two types of point distributions: random distributions on the unit sphere in 3D (*sphere* point sets) and random distributions in the unit ball in 3D (*ball* point sets). For experiments in Sections 4.1 and 4.5, we use an Intel Skylake node on the Stampede2 supercomputer at Texas Advanced Computing Center. This node has two sockets and 192 GB DDR4 memory. Each socket has an Intel Xeon Platinum 8160 processor with 24 cores and 2 hyperthreads per core. For experiments in Section 4.2, we use an Intel Skylake node described above and an Intel Knights Landing node. The latter has an Intel Xeon Phi 7210 many-core processor with 64 cores and 4 hyperthreads per core, 16 GB MCDRAM high-bandwidth memory, and 96 GB DDR4 memory. H2Pack is compiled using Intel C compiler 2018.0.2 with optimization flags "-xHost -O3" on both nodes. Intel MKL 2018.0.2 is used in H2Pack to perform general matrix-vector multiplications (xGEMV) and general matrix-matrix multiplications (xGEMM). Double precision floating point is used for storage and calculations in H2Pack.

### 4.1 Accuracy tests

We first measure the accuracy of $\mathcal{H}^2$ matrix representations constructed by H2Pack under different settings. We consider three kernel functions:

- 3D Laplace kernel: $K(x,y) = \frac{1}{|x-y|}$,
- 3D Gaussian kernel: $K(x,y) = \exp(-|x-y|^2)$,
- 3D Stokes kernel: $K(x,y) = \frac{1}{|x-y|}I + \frac{(x-y)(x-y)^T}{|x-y|^3}$.

Table 1 shows the relative error of $\mathcal{H}^2$-matvec for the two types of point sets with different prescribed relative error thresholds for the ID approximation in $\mathcal{H}^2$-construction. Both the *sphere* and *ball* point sets contain $1 \times 10^6$ points. All the multiplicand vectors for $\mathcal{H}^2$-matvec have entries randomly and uniformly generated between $[-1, 1]$. Each reported relative error is the average result obtained by 10 independent $\mathcal{H}^2$-matvec tests. The prescribed relative error threshold varies from $1 \times 10^{-2}$ to $1 \times 10^{-12}$. As can be observed, for all tested kernel functions and types of point sets, the relative errors of $\mathcal{H}^2$-matvec are controlled by the prescribed threshold. Further, when the relative error threshold is above $1 \times 10^{-8}$, the actual relative error is usually an order of magnitude smaller than the threshold.

Table 1. Relative error of $\mathcal{H}^2$-matvec in H2Pack for several kernel functions with different prescribed relative error thresholds for the ID approximation in $\mathcal{H}^2$-construction ("ID approx. relerr"). Both sphere and ball points sets are tested. Each point set contains $1 \times 10^6$ points.

| ID approx. relerr | | 1.00E-2 | 1.00E-4 | 1.00E-6 | 1.00E-8 | 1.00E-10 | 1.00E-12 |
|---|---|---|---|---|---|---|---|
| 3D Laplace | sphere | 8.42E-4 | 3.68E-6 | 4.30E-8 | 6.35E-10 | 2.97E-11 | 9.20E-13 |
| | ball | 8.21E-4 | 4.13E-6 | 4.54E-8 | 8.05E-10 | 4.27E-11 | 5.30E-13 |
| 3D Gaussian | sphere | 3.38E-3 | 1.89E-5 | 2.35E-7 | 4.25E-9 | 1.73E-11 | 2.38E-13 |
| | ball | 3.57E-3 | 1.53E-5 | 1.46E-7 | 1.13E-9 | 1.09E-11 | 3.85E-12 |
| 3D Stokes | sphere | 1.26E-3 | 7.06E-6 | 6.02E-8 | 3.73E-10 | 2.46E-12 | 2.71E-12 |
| | ball | 1.42E-3 | 7.72E-6 | 3.20E-7 | 2.61E-9 | 2.69E-11 | 2.76E-12 |

### 4.2 Scalability tests

We now demonstrate the strong scalability (fixed problem size) of H2Pack. We test the 3D Laplace kernel with a ball point set of size $2 \times 10^5$ points and with $1 \times 10^{-6}$ prescribed matvec relative error. Under this setting, the constructed $\mathcal{H}^2$ matrix representation in AOT mode can be completely stored in the 16 GB MCDRAM high-bandwidth memory of the Knights Landing node. On the Skylake node, we run H2Pack using one thread per core on all 48 cores. On the Knights Landing node, we run H2Pack using one thread per core on all 64 cores. Figure 5 shows the timings of $\mathcal{H}^2$-construction ("build") and $\mathcal{H}^2$-matvec ("matvec") of H2Pack in AOT and JIT modes on the two different nodes.

For $\mathcal{H}^2$-construction, JIT mode is faster than AOT mode on both types of compute nodes since AOT mode additionally calculates and stores $B_{i,j}$ and $D_{i,j}$ blocks. For both modes, however, $\mathcal{H}^2$-construction does not fully scale to all the cores on both types of nodes. The reason is that the performance of $\mathcal{H}^2$-construction is limited by memory bandwidth. The major computational kernel in $\mathcal{H}^2$-construction in both modes is the column-pivoted QR factorization to compute ID approximations (lines 6 and 9 in Algorithm 2). On both nodes, this computational kernel takes more than 95% and 50% of $\mathcal{H}^2$-construction time in JIT mode and AOT mode, respectively. Meanwhile, this computational kernel has a low computation-to-memory-access ratio and thus its performance is determined by the memory access bandwidth. Intel VTune (Intel performance profiling software) reports that the achieved memory bandwidth of this computational kernel is more than 80% of the peak memory bandwidth when using all cores on both nodes.

For $\mathcal{H}^2$-matvec, JIT mode is faster than AOT mode on the Skylake node while AOT mode is faster than JIT mode on the Knights Landing node, which is due to hardware differences between the Skylake node and the Knights Landing node. The Knights Landing node has high memory

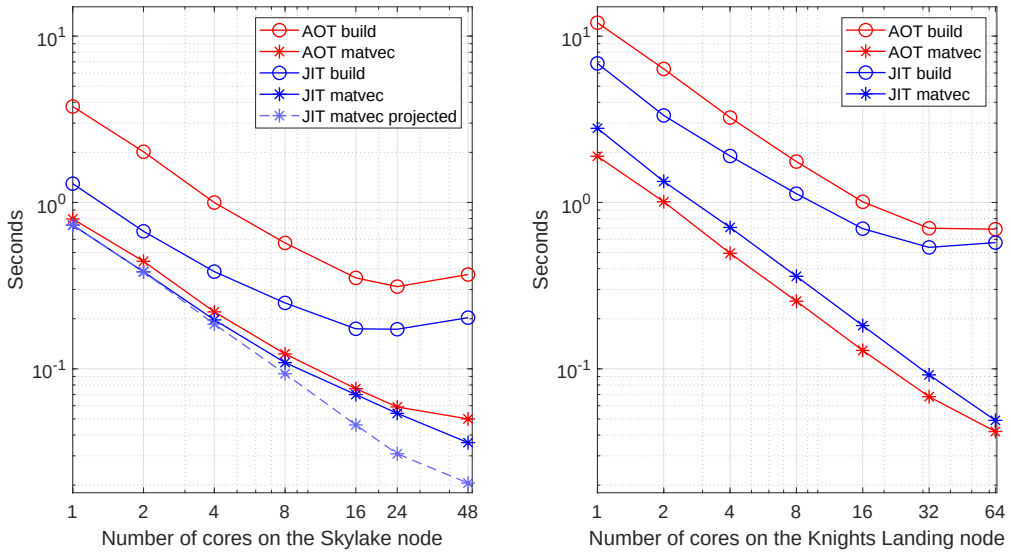



Fig. 5. H2Pack $\mathcal{H}^2$ construction ("build") and $\mathcal{H}^2$ matvec ("matvec") timings in AOT mode and JIT mode on an Intel Skylake node (left) and an Intel Knights Landing node (right) using different numbers of cores. Projected $\mathcal{H}^2$-matvec time in JIT mode assuming all processors always run at 3.5GHz ("JIT matvec projected") on the Skylake node is also plotted as a reference. A *ball* point set with $2 \times 10^5$ points and a $10^{-6}$ prescribed matvec relative error are used.

bandwidth but its single core performance is only moderate. On this node, the parallel efficiencies of $\mathcal{H}^2$-matvec in JIT and AOT modes are 89.0% and 70.5%, respectively, showing excellent scalability. Intel VTune reports that $\mathcal{H}^2$-matvec in AOT mode only utilizes about 65% of the peak MCDRAM memory bandwidth on the Knights Landing node when using all 64 cores. The Skylake node has powerful processor cores with moderate memory bandwidth. On this node, the parallel efficiencies of $\mathcal{H}^2$-matvec in JIT and AOT modes are only 33.1% and 42.3%, respectively. Intel VTune reports that $\mathcal{H}^2$-matvec in AOT mode achieved 79% and 90% of the peak memory bandwidth when using 24 and 48 cores on the Skylake node, suggesting that the lower parallel efficiency in AOT mode than in JIT mode is caused by the memory bandwidth limit. Furthermore, the lower parallel efficiency in JIT mode on the Skylake node than on the Knights Landing node (i.e., 33.1% v.s. 89.0%) is due to Intel Turbo Boost technology on Intel Xeon Platinum processors. If only one core is active on a Xeon Platinum 8160 processor (on the Skylake node), this core runs at 3.5 GHz. The more active cores, the lower the clock frequency of the cores. If all 24 cores are active, all the cores run at only 2.0 GHz. In comparison, all cores on the Knights Landing node always run at 1.4 GHz. This decrease of core frequency reduces the parallel efficiency of $\mathcal{H}^2$-matvec in JIT mode which requires a large number of kernel function evaluations. In Figure 5, we also plot the projected execution time for $\mathcal{H}^2$-matvec in JIT mode on the Skylake node assuming that all its cores always run at 3.5 GHz. The projected parallel efficiency of $\mathcal{H}^2$-matvec in JIT mode is 72.5% when using 48 cores.

Lastly, we also measure the performance of $\mathcal{H}^2$-matvec in JIT mode in terms of giga floating-point operations per second (GFLOPS). To measure this performance, we note that evaluating one value of the 3D Laplace kernel requires 19 floating-point operations (8 for the squared distance, 1 for the approximate RSQRT and 10 for two Newton iterations). On the Skylake node, $\mathcal{H}^2$-matvec in JIT mode achieved 1047 GFLOPS (34.9% of the peak performance). On the Knights Landing node, $\mathcal{H}^2$-matvec in JIT mode achieved 651 GFLOPS (24.5% of the peak performance).

Table 2. Timing results (in seconds) of $\mathcal{H}^2$-construction and $\mathcal{H}^2$-matvec using different implementations of KME and BKM functions for 3D Gaussian kernel: no vectorization ("no-vec"), automatic vectorization by the Intel C compiler ("auto-vec"), and manual vectorization by vector wrapper functions ("wrap-vec"). The JIT mode and relative error threshold $10^{-6}$ are used in all the tests.

| #pts $\times 10^5$ | ball | | | sphere | | |
|---|---|---|---|---|---|---|
| | 1 | 4 | 16 | 1 | 4 | 16 |
| *$\mathcal{H}^2$-construction* | | | | | | |
| KME no-vec | 0.046 | 0.142 | 0.583 | 0.051 | 0.127 | 0.400 |
| KME auto-vec | 0.045 | 0.131 | 0.574 | 0.049 | 0.123 | 0.396 |
| KME wrap-vec | 0.043 | 0.128 | 0.566 | 0.050 | 0.123 | 0.394 |
| *$\mathcal{H}^2$-matvec* | | | | | | |
| KME no-vec | 0.086 | 0.211 | 0.504 | 0.035 | 0.117 | 0.499 |
| KME auto-vec | 0.028 | 0.075 | 0.188 | 0.012 | 0.041 | 0.172 |
| KME wrap-vec | 0.018 | 0.057 | 0.146 | 0.008 | 0.035 | 0.119 |
| BKM no-vec | 0.092 | 0.264 | 0.735 | 0.040 | 0.138 | 0.586 |
| BKM auto-vec | 0.028 | 0.076 | 0.209 | 0.012 | 0.041 | 0.178 |
| BKM wrap-vec | 0.013 | 0.037 | 0.118 | 0.006 | 0.029 | 0.089 |

### 4.3 Performance improvements by BKM and vectorization

The efficient evaluation of kernel functions is crucial to the overall performance of $\mathcal{H}^2$-construction and $\mathcal{H}^2$-matvec (in JIT mode). In this section, we study the performance improvements brought by the BKM interface and vector wrapper functions. Table 2 shows the timing results of $\mathcal{H}^2$-construction and $\mathcal{H}^2$-matvec using different implementations of KME and BKM functions for the 3D Gaussian kernel $K(x, y) = \exp(-|x - y|^2)$: no vectorization, automatic vectorization by the Intel C compiler, and our manual vectorization by vector wrapper functions.

As explained in Section 4.2, $\mathcal{H}^2$-construction in JIT mode is dominated by the column-pivoted QR factorization, and thus only gains minor performance improvements from vectorization. Meanwhile, both automatic and manual vectorization of KME and BKM functions can lead to 300%-400% speedup in $\mathcal{H}^2$-matvec, with the manual vectorization being 20%-50% faster than the automatic vectorization. Comparing KME and BKM functions for $\mathcal{H}^2$-matvec, using KME functions without vectorization or with automatic vectorization can be even faster than using BKM functions. This is because the dense matrix-vector multiplication after evaluating a kernel block by KME functions is vectorized in the BLAS library. On the other hand, based on the manual vectorization, using BKM functions is 20%-35% faster than using KME functions.

### 4.4 Comparison between $\mathcal{H}^2$-matvec and $\mathcal{H}^2$-matmul

In this section, we compare the performance of $\mathcal{H}^2$-matvec and $\mathcal{H}^2$-matmul in H2Pack for multiplying multiple vectors. In the latter, the vectors are assumed to be available at the same time and the multiplications are performed simultaneously. Figure 6 shows the timings of $\mathcal{H}^2$-matvec and $\mathcal{H}^2$-matmul to multiply different numbers of vectors in both AOT and JIT modes. The runtime of $\mathcal{H}^2$-matmul increases much more slowly with the number of vectors compared to $\mathcal{H}^2$-matvec. This indicates that calculating $B_{i,j}$ and $D_{i,j}$ blocks in JIT mode or transferring these blocks from main memory to processor cache in AOT mode are very expensive compared to the actual multiplication. For a single vector, $\mathcal{H}^2$-matmul is slower than $\mathcal{H}^2$-matvec because the symmetry property of $B_{i,j}$ and $D_{i,j}$ is not exploited (see Section 3.1). In AOT mode, the performance of $\mathcal{H}^2$-matmul is

further affected by NUMA. $\mathcal{H}^2$-matvec uses a fixed workload partitioning and $B_{i,j}$ and $D_{i,j}$ blocks are optimized for this fixed partitioning using the first-touch policy. $\mathcal{H}^2$-matmul uses a different workload partitioning, making it hard to optimize for NUMA without almost doubling the storage.

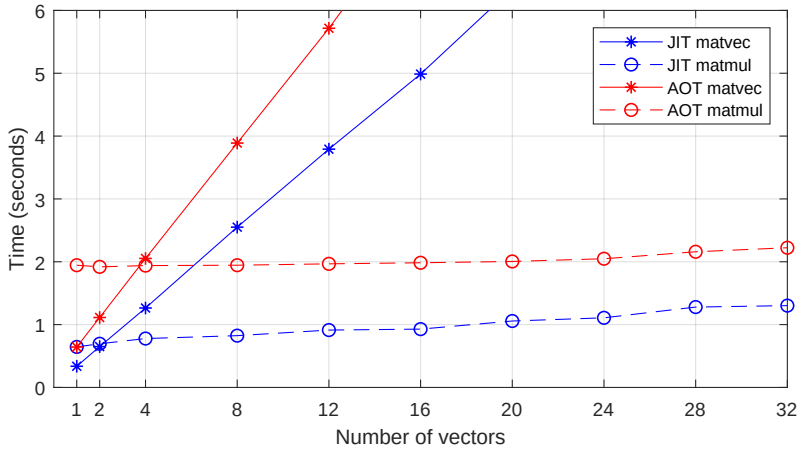



Fig. 6. Timings of $\mathcal{H}^2$-matvec and $\mathcal{H}^2$-matmul (in seconds) in AOT and JIT modes for multiplying different numbers of vectors. The test settings are: the 3D Laplace kernel, a *ball* point set with $1.6\times10^6$ points, prescribed relative error threshold $10^{-6}$, and the Skylake node with 48 cores. The results are qualitatively similar for other kernel functions and point sets. Column-major format for the matrix of vectors was used; the timings for row-major format are very similar. The corresponding $\mathcal{H}^2$-construction in AOT and JIT modes take 2.54 seconds and 1.11 seconds, respectively.

### 4.5 Comparison with fast multipole methods

In this section, we compare the performance of H2Pack with two fast multipole method (FMM) libraries: the FMM3D library implements the standard FMM and the PVFMM library implements the kernel independent FMM (KIFMM). We note that FMM3D works for the 3D Laplace, Stokes, and Helmholtz kernels, PVFMM works for kernel functions from potential theory, and H2Pack can work for non-oscillatory kernel functions in general. For all the libraries, we use the same sets of points and test using the 3D Laplace kernel. The number of points in $X$ ranges from $1 \times 10^5$ to $1.6 \times 10^6$. For all three libraries, we specify that a box is further partitioned into smaller boxes if it contains more than 400 points in the hierarchical partitioning of $X$. For H2Pack, JIT mode is used. All three libraries are compiled using Intel C/C++/Fortran compilers and Intel MPI 2018.0.2 with optimization flags "-xHost -O3". Intel MKL 2018.0.2 is used to perform optimized general matrix-vector multiplications (xGEMV), general matrix-matrix multiplications (xGEMM), and fast Fourier transformations that appear in these three libraries. Double precision floating point is used for storage and calculations in all three libraries.

We run all three libraries using one thread per core on all 48 cores on a Skylake node. Tables 3 to 5 show the test results corresponding to relative multiplication accuracy of approximately $10^{-5}$, $10^{-8}$, and $10^{-11}$, respectively. The tables show results for the following quantities:

- **Precomputation cost**. The runtime of specific precomputations in H2Pack and PVFMM that can be reused for different sets of points but not for different accuracy requirements and for different kernel functions. (FMM3D does not have precomputations.) In H2Pack,

the precomputation involves computing the proxy points. In PVFMM, the precomputation involves computing fixed translation operators in KIFMM and storing them into a file.
- **Setup cost**. The runtime of all the computations other than the precomputations above before matrix-vector multiplications, i.e., hierarchical partitioning of $X$ in all the libraries and $\mathcal{H}^2$-construction in H2Pack.
- **Peak memory**. The peak memory usage recorded by the operating system during the entire program execution (precomputation, setup, and matrix-vector multiplication).
- **Storage cost**. The storage cost of the translation operators in PVFMM and of the $\mathcal{H}^2$ matrix components in H2Pack. (FMM3D does not report its storage cost.)
- **Runtime and relative error of the multiplication**. These results are averaged over 5 multiplications by random vectors for each point set $X$. For each multiplication, denoted as an approximation $A_{\mathcal{H}^2}v \approx b = K(X,X)v$, its relative error is measured as

$$\text{relerr} = \frac{\sqrt{\sum_{i\in S}(b_i - (A_{\mathcal{H}^2}v)_i)^2}}{\sqrt{\sum_{i\in S} b_i^2}},$$

where $S$ is a set of 10000 indices randomly chosen from 1 to the length of $b$ and the entries $\{b_i\}_{i\in S}$ are computed via direct matrix-vector multiplication.
- **Degree and rank**. The "degree" in PVFMM and FMM3D is an input parameter characterizing the number of expansion terms used for analytic compression of kernel matrix blocks. In PVFMM, a degree of $k$ corresponds to a rank-$6k^2$ analytic approximation of each block to be compressed in the equivalent $\mathcal{H}^2$ matrix representation. In FMM3D, a degree of $k$ corresponds to the approximation rank being $(k+1)^2$. For H2Pack, the resulting maximum and average ranks of all the low-rank approximations in each constructed $\mathcal{H}^2$ matrix are listed.

From the results, the cost for $\mathcal{H}^2$-construction ("setup") in H2Pack scales linearly in the number of points and increases with higher relative multiplication accuracy. For points in a unit ball, the H2Pack setup cost can be much more expensive than the setup costs in PVFMM and FMM3D. However, the setup cost of H2Pack is much cheaper for points on the unit sphere than in the unit ball. This is due to the smaller approximation ranks for all the blocks compressed in $\mathcal{H}^2$-construction.

The maximum and average approximation ranks in H2Pack are all much smaller than those in PVFMM and FMM3D. The approximation ranks in H2Pack are different with different point distributions, while PVFMM and FMM3D have fixed approximation ranks for both types of point distributions. As a result, H2Pack is the fastest library for matrix-vector multiplications among the three and this efficiency advantage becomes even greater when dealing with points on the unit sphere, i.e., around 5 times faster than PVFMM and 25 times faster than FMM3D.

The storage cost of H2Pack is proportional to the number of points and the approximation ranks in the constructed $\mathcal{H}^2$ matrices. In comparison, the storage cost of PVFMM changes very mildly under different problem settings. H2Pack has much smaller storage cost for small problems compared with PVFMM but ultimately can have larger storage cost when the number of points or the relative accuracy increases. For example, H2Pack begins to have more storage cost for $8\times10^5$ points in the unit ball with relative accuracy $10^{-11}$. FMM3D does not report its storage cost but theoretically only has very small storage cost for temporary components.

It is worth noting that the peak memory recorded by the operating system depends on the actual implementations of these libraries and can only be used as a rough reference for comparing the three different methods. As can be noted, H2Pack has its peak memory increasing much faster than PVFMM and eventually has larger peak memory than PVFMM when dealing with large numbers of points and high relative accuracy, e.g., $8\times10^5$ points in the unit ball with relative accuracy $10^{-11}$.

Meanwhile, FMM3D also has increasing peak memory with more points but has the smallest peak memory among the three libraries when dealing with a large number of points.

Compared to FMM3D, both H2Pack and PVFMM have relatively expensive precomputations. For H2Pack, the precomputation involves the kernel-related proxy point selection. However, for a given kernel function, the selected proxy points in H2Pack can be saved to a file and reused in future computations. For certain kernel functions such as the Laplace and Stokes kernels, H2Pack can also apply the proxy surface method [24] to generate the proxy points with negligible computation cost. For PVFMM, the precomputation involves computing fixed translation operators and its complexity depends on the kernel function and the degree parameter (which controls the relative accuracy). These precomputed results in PVFMM are stored in files for reuse. Since the precomputations in H2Pack and PVFMM can be reused when the kernel function and the relative accuracy are fixed, the precomputation costs typically make no impact in practice.

Figure 7 shows the timings and parallel efficiencies of the "setup" and "matvec" procedures of the three tested libraries. For H2Pack, the results in Figure 7 are similar to the results in Figure 5, but the parallel efficiency of $\mathcal{H}^2$-matvec when using 48 cores is higher in Figure 7 (49.9% v.s. 33.1%) due to more points and a larger parallelism. The setup procedure is not parallelized in FMM3D and not fully parallelized in PVFMM, leading to poor parallel efficiencies in FMM3D and PVFMM for the setup. Although FMM3D has slightly better parallel efficiency in matvec compared to PVFMM and H2Pack, its absolute matvec time is much larger than the matvec time of PVFMM and H2Pack.

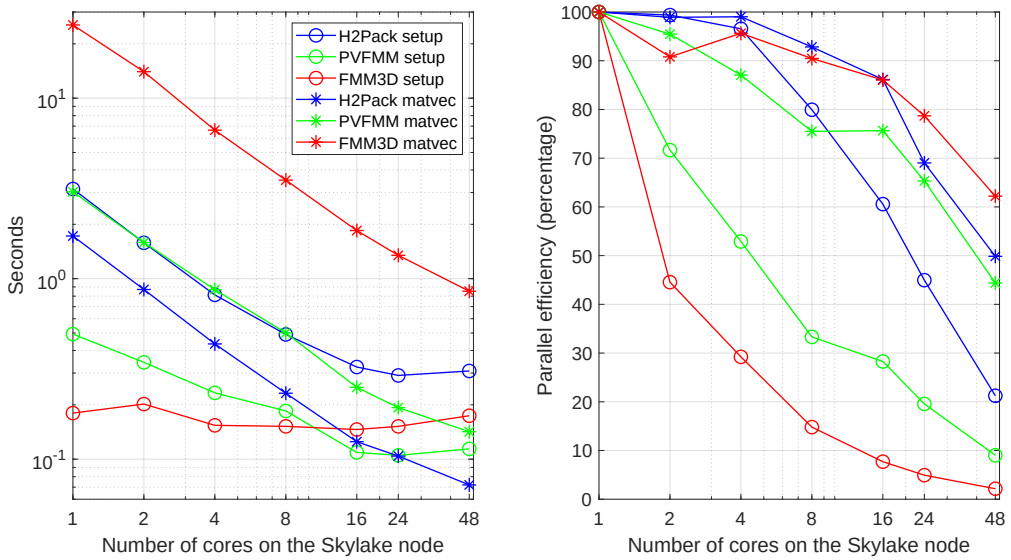



Fig. 7. Setup and matvec timings (in seconds) and parallel efficiency (in percentage) using different numbers of cores on a Skylake node for FMM3D, PVFMM, and H2Pack. A *ball* point set with $4 \times 10^5$ points and a $10^{-8}$ prescribed matvec relative error threshold are used.

Table 3. Numerical results of the three libraries with relative accuracy around $10^{-5}$. "Precomp" refers to the precomputations in H2Pack and PVFMM. "Mem" refers to the peak memory usage recorded by the operating system. "Storage" refers to the storage cost of translation operators in PVFMM and that of $\mathcal{H}^2$ matrix components in H2Pack.

| H2Pack | | | | | | | |
|---|---|---|---|---|---|---|---|
| #pts $\times 10^5$ | precomp(s) | setup(s) | matvec(s) | mem(MB) | storage(MB) | relerr | max/avg rank |
| sphere 1 | 0.185 | 0.062 | 0.005 | 519 | 15 | 1.63E-05 | 29/15 |
| sphere 2 | 0.194 | 0.082 | 0.010 | 436 | 30 | 1.91E-05 | 28/15 |
| sphere 4 | 0.187 | 0.125 | 0.018 | 569 | 59 | 2.12E-05 | 28/15 |
| sphere 8 | 0.239 | 0.223 | 0.037 | 839 | 119 | 2.34E-05 | 28/15 |
| sphere 16 | 0.278 | 0.444 | 0.075 | 1345 | 234 | 2.64E-05 | 28/15 |
| ball 1 | 0.159 | 0.073 | 0.010 | 669 | 43 | 1.68E-05 | 69/40 |
| ball 2 | 0.162 | 0.160 | 0.019 | 823 | 89 | 1.86E-05 | 69/35 |
| ball 4 | 0.157 | 0.173 | 0.035 | 836 | 163 | 2.14E-05 | 71/39 |
| ball 8 | 0.194 | 0.249 | 0.090 | 1181 | 308 | 2.56E-05 | 70/38 |
| ball 16 | 0.189 | 0.784 | 0.149 | 2418 | 723 | 2.84E-05 | 70/37 |

| PVFMM | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| #pts $\times 10^5$ | precomp(s) | setup(s) | matvec(s) | mem(MB) | storage(MB) | relerr | degree | rank |
| sphere 1 | 1.161 | 0.069 | 0.020 | 1164 | 1138 | 7.51E-06 | 5 | 150 |
| sphere 2 | 1.117 | 0.087 | 0.027 | 1388 | 1186 | 8.85E-06 | 5 | 150 |
| sphere 4 | 1.114 | 0.134 | 0.056 | 1799 | 1271 | 6.41E-06 | 5 | 150 |
| sphere 8 | 1.163 | 0.328 | 0.132 | 2753 | 1461 | 9.91E-06 | 5 | 150 |
| sphere 16 | 1.115 | 0.686 | 0.230 | 4528 | 1845 | 9.70E-06 | 5 | 150 |
| ball 1 | 1.113 | 0.042 | 0.035 | 1128 | 1134 | 1.99E-05 | 5 | 150 |
| ball 2 | 1.113 | 0.083 | 0.030 | 1355 | 1186 | 1.49E-05 | 5 | 150 |
| ball 4 | 1.114 | 0.113 | 0.075 | 1751 | 1252 | 1.85E-05 | 5 | 150 |
| ball 8 | 1.113 | 0.221 | 0.267 | 2547 | 1394 | 2.98E-05 | 5 | 150 |
| ball 16 | 1.115 | 0.691 | 0.219 | 4518 | 1832 | 1.54E-05 | 5 | 150 |

| FMM3D | | | | | | |
|---|---|---|---|---|---|---|
| #pts $\times 10^5$ | setup(s) | matvec(s) | mem(MB) | relerr | degree | rank |
| sphere 1 | 0.041 | 0.120 | 238 | 8.81E-06 | 15 | 256 |
| sphere 2 | 0.085 | 0.163 | 414 | 8.88E-06 | 15 | 256 |
| sphere 4 | 0.183 | 0.329 | 747 | 9.41E-06 | 15 | 256 |
| sphere 8 | 0.441 | 0.626 | 1397 | 8.61E-06 | 15 | 256 |
| sphere 16 | 1.025 | 1.259 | 2784 | 9.56E-06 | 15 | 256 |
| ball 1 | 0.042 | 0.167 | 302 | 6.55E-06 | 15 | 256 |
| ball 2 | 0.081 | 0.168 | 353 | 6.85E-06 | 15 | 256 |
| ball 4 | 0.170 | 0.192 | 554 | 6.77E-06 | 15 | 256 |
| ball 8 | 0.443 | 1.266 | 1830 | 6.84E-06 | 15 | 256 |
| ball 16 | 0.955 | 1.261 | 2025 | 6.75E-06 | 15 | 256 |

Table 4. Numerical results of the three libraries with relative accuracy around $10^{-8}$.

| H2Pack | | | | | | | |
|---|---|---|---|---|---|---|---|
| #pts $\times 10^5$ | precomp(s) | setup(s) | matvec(s) | mem(MB) | storage(MB) | relerr | max/avg rank |
| sphere 1 | 0.349 | 0.095 | 0.006 | 778 | 47 | 1.20E-08 | 77/39 |
| sphere 2 | 0.419 | 0.143 | 0.012 | 717 | 90 | 1.43E-08 | 78/36 |
| sphere 4 | 0.612 | 0.199 | 0.023 | 908 | 176 | 1.74E-08 | 78/36 |
| sphere 8 | 0.727 | 0.326 | 0.047 | 1320 | 352 | 1.83E-08 | 79/37 |
| sphere 16 | 0.948 | 0.589 | 0.097 | 2046 | 687 | 2.04E-08 | 77/36 |
| ball 1 | 0.417 | 0.135 | 0.021 | 857 | 137 | 1.71E-08 | 194/96 |
| ball 2 | 0.352 | 0.247 | 0.044 | 1294 | 331 | 1.68E-08 | 201/78 |
| ball 4 | 0.339 | 0.312 | 0.078 | 1652 | 561 | 2.19E-08 | 203/94 |
| ball 8 | 0.500 | 0.417 | 0.141 | 2246 | 984 | 2.58E-08 | 206/90 |
| ball 16 | 0.438 | 1.190 | 0.340 | 4642 | 2362 | 3.07E-08 | 205/78 |

| PVFMM | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| #pts $\times 10^5$ | precomp(s) | setup(s) | matvec(s) | mem(MB) | storage(MB) | relerr | degree | rank |
| sphere 1 | 2.981 | 0.048 | 0.030 | 1397 | 1211 | 2.35E-08 | 8 | 384 |
| sphere 2 | 2.967 | 0.089 | 0.052 | 1597 | 1266 | 2.46E-08 | 8 | 384 |
| sphere 4 | 2.966 | 0.138 | 0.122 | 2112 | 1358 | 1.75E-08 | 8 | 384 |
| sphere 8 | 2.966 | 0.471 | 0.201 | 3288 | 1574 | 2.57E-08 | 8 | 384 |
| sphere 16 | 2.967 | 0.658 | 0.420 | 4927 | 1994 | 2.70E-08 | 8 | 384 |
| ball 1 | 2.970 | 0.043 | 0.041 | 1233 | 1205 | 3.57E-08 | 8 | 384 |
| ball 2 | 2.957 | 0.087 | 0.058 | 1530 | 1264 | 2.48E-08 | 8 | 384 |
| ball 4 | 2.955 | 0.115 | 0.118 | 1948 | 1331 | 3.55E-08 | 8 | 384 |
| ball 8 | 2.959 | 0.336 | 0.330 | 2800 | 1477 | 4.07E-08 | 8 | 384 |
| ball 16 | 2.954 | 0.883 | 0.454 | 5052 | 1971 | 3.97E-08 | 8 | 384 |

| FMM3D | | | | | | |
|---|---|---|---|---|---|---|
| #pts $\times 10^5$ | setup(s) | matvec(s) | mem(MB) | relerr | degree | rank |
| sphere 1 | 0.041 | 0.156 | 298 | 1.10E-08 | 21 | 484 |
| sphere 2 | 0.084 | 0.295 | 454 | 1.21E-08 | 21 | 484 |
| sphere 4 | 0.184 | 0.535 | 860 | 1.14E-08 | 21 | 484 |
| sphere 8 | 0.418 | 1.099 | 1615 | 1.19E-08 | 21 | 484 |
| sphere 16 | 1.027 | 2.138 | 3235 | 1.28E-08 | 21 | 484 |
| ball 1 | 0.042 | 0.172 | 366 | 1.13E-08 | 21 | 484 |
| ball 2 | 0.081 | 0.210 | 359 | 1.10E-08 | 21 | 484 |
| ball 4 | 0.171 | 0.863 | 632 | 1.11E-08 | 21 | 484 |
| ball 8 | 0.452 | 1.037 | 2113 | 1.11E-08 | 21 | 484 |
| ball 16 | 0.926 | 1.322 | 2330 | 1.18E-08 | 21 | 484 |

Table 5. Numerical results of the three libraries with relative accuracy around $10^{-11}$.

| H2Pack | | | | | | | |
|---|---|---|---|---|---|---|---|
| #pts $\times 10^5$ | precomp(s) | setup(s) | matvec(s) | mem(MB) | storage(MB) | relerr | max/avg rank |
| sphere 1 | 0.962 | 0.159 | 0.009 | 1222 | 114 | 5.66E-12 | 165/73 |
| sphere 2 | 1.018 | 0.203 | 0.019 | 1160 | 236 | 6.06E-12 | 165/70 |
| sphere 4 | 1.151 | 0.292 | 0.035 | 1616 | 437 | 7.90E-12 | 165/69 |
| sphere 8 | 1.022 | 0.493 | 0.072 | 2398 | 864 | 8.43E-12 | 165/69 |
| sphere 16 | 1.728 | 0.903 | 0.144 | 4035 | 1707 | 9.13E-12 | 166/68 |
| ball 1 | 0.906 | 0.590 | 0.035 | 1676 | 391 | 2.35E-12 | 444/184 |
| ball 2 | 0.873 | 0.847 | 0.082 | 2270 | 796 | 6.65E-12 | 450/109 |
| ball 4 | 0.792 | 1.502 | 0.177 | 3590 | 1604 | 9.93E-12 | 444/168 |
| ball 8 | 1.011 | 2.438 | 0.305 | 5426 | 2709 | 1.86E-11 | 450/167 |
| ball 16 | 0.941 | 4.057 | 0.633 | 9442 | 5539 | 2.50E-11 | 449/109 |

| PVFMM | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| #pts $\times 10^5$ | precomp(s) | setup(s) | matvec(s) | mem(MB) | storage(MB) | relerr | degree | rank |
| sphere 1 | 9.700 | 0.055 | 0.054 | 1953 | 1445 | 1.29E-11 | 12 | 864 |
| sphere 2 | 9.559 | 0.104 | 0.126 | 2196 | 1517 | 1.47E-11 | 12 | 864 |
| sphere 4 | 9.558 | 0.159 | 0.234 | 2555 | 1624 | 9.75E-12 | 12 | 864 |
| sphere 8 | 9.562 | 0.600 | 0.491 | 3535 | 1893 | 1.44E-11 | 12 | 864 |
| sphere 16 | 9.575 | 1.014 | 0.890 | 5496 | 2392 | 1.69E-11 | 12 | 864 |
| ball 1 | 9.547 | 0.056 | 0.060 | 1527 | 1434 | 2.76E-11 | 12 | 864 |
| ball 2 | 9.578 | 0.158 | 0.151 | 2086 | 1510 | 2.22E-11 | 12 | 864 |
| ball 4 | 9.652 | 0.180 | 0.181 | 2514 | 1578 | 2.73E-11 | 12 | 864 |
| ball 8 | 9.595 | 0.410 | 0.430 | 3552 | 1880 | 4.31E-11 | 12 | 864 |
| ball 16 | 9.607 | 0.784 | 1.123 | 5544 | 2351 | 2.17E-11 | 12 | 864 |

| FMM3D | | | | | | |
|---|---|---|---|---|---|---|
| #pts $\times 10^5$ | setup(s) | matvec(s) | mem(MB) | relerr | degree | rank |
| sphere 1 | 0.034 | 0.272 | 278 | 9.90E-12 | 29 | 900 |
| sphere 2 | 0.078 | 0.472 | 553 | 1.08E-11 | 29 | 900 |
| sphere 4 | 0.167 | 0.899 | 907 | 1.09E-11 | 29 | 900 |
| sphere 8 | 0.375 | 1.698 | 1780 | 1.12E-11 | 29 | 900 |
| sphere 16 | 0.917 | 3.541 | 3366 | 1.11E-11 | 29 | 900 |
| ball 1 | 0.037 | 0.238 | 208 | 9.55E-12 | 29 | 900 |
| ball 2 | 0.098 | 0.522 | 678 | 1.07E-11 | 29 | 900 |
| ball 4 | 0.163 | 0.654 | 728 | 1.10E-11 | 29 | 900 |
| ball 8 | 0.346 | 2.117 | 994 | 1.08E-11 | 29 | 900 |
| ball 16 | 0.947 | 3.106 | 4502 | 1.14E-11 | 29 | 900 |

To summarize, the numerical comparisons above show that FMM libraries typically have less cost for setup and storage but also typically have slower matrix-vector multiplications than H2Pack. Thus, FMM libraries are more suitable for problems where only a few matrix-vector multiplications are required per set of points, e.g., particle simulations. Meanwhile, H2Pack is more suitable for problems where many matrix-vector multiplications are required per set of points, e.g., numerical solution of integral equations and Gaussian processes, so that the relatively expensive $\mathcal{H}^2$ construction cost can be amortized by many multiplications.

## 5 CONCLUSION

H2Pack provides linear-scaling matrix-vector multiplication for kernel matrices defined by non-oscillatory kernel functions. Such multiplications are needed on their own in many applications, but can also be used in iterative solvers for kernel matrix systems. The critical step for linear-scaling matrix-vector multiplication is constructing the $\mathcal{H}^2$ matrix representation of the kernel matrix. In H2Pack, this is done by using the recently-developed proxy point method. The advantages of using the proxy point method are (1) greater generality compared to other methods (e.g., it works for Gaussian kernels), and (2) more effective block low-rank compression compared to analytic methods such as those used in FMM. The latter is what makes H2Pack matrix-vector multiplication faster than kernel summation in FMM libraries. On the other hand, constructing the $\mathcal{H}^2$ matrix representation in H2Pack is often more expensive than the setup phase in FMM libraries.

We have focused on translationally-invariant kernels. This allows the proxy points for each box (in a given level of the partition tree) to be translates of each other, thus reducing the overall cost of proxy point selection. We have also focused on 2D and 3D problems, as is common for FMM libraries. H2Pack can be extended to higher dimensions if a cheap method of selecting proxy points in higher dimensions is available.

In standard $\mathcal{H}^2$ matrix representations, blocks of the matrix are either admissible (represented as a low-rank block) or inadmissible (represented as a dense block). In H2Pack, we introduce the concept of partially admissible blocks. Such blocks arise with non-uniform distributions of points, leading to non-perfect partition trees. By treating partially admissible blocks in the appropriate way (rather than as either admissible or inadmissible), the representation of these blocks is more efficient. The same technique exists in FMM libraries but not in existing $\mathcal{H}^2$ matrix libraries.

H2Pack has been optimized for high-performance on shared-memory parallel computers. Important considerations are vectorization of kernel function evaluations, reducing memory traffic, and load balancing. Just-in-time and ahead-of-time modes are provided to trade computation with storage and memory traffic. Vectorization of kernel function evaluations is particularly important in just-in-time mode, and a kernel function interface is described. Numerical tests show good scaling of H2Pack matrix-vector multiplication with the number of cores. For constructing the $\mathcal{H}^2$ matrix representations, the performance with large numbers of cores is limited by the high memory bandwidth requirement of the column-pivoted QR factorization used in the code.

## ACKNOWLEDGMENTS

Funding from the National Science Foundation grant ACI-1609842 is gratefully acknowledged.

## REFERENCES

[1] Mario Bebendorf and Stefan Kunis. 2009. Recompression techniques for adaptive cross approximation. *The Journal of Integral Equations and Applications* 21, 3 (2009), 331–357.

[2] Mario Bebendorf and Sergej Rjasanow. 2003. Adaptive low-rank approximation of collocation matrices. *Computing* 70, 1 (2003), 1–24.

[3] Steffen Börm. 2017, accessed: 2019-12-05. H2Lib. (2017, accessed: 2019-12-05). https://github.com/H2Lib/H2Lib/tree/community

[4] Steffen Börm and Lars Grasedyck. 2005. Hybrid cross approximation of integral operators. *Numer. Math.* 101, 2 (2005), 221–249.

[5] Wajih Boukaram, George Turkiyyah, and David Keyes. 2019. Hierarchical matrix operations on GPUs: Matrix-vector multiplication and compression. *ACM Trans. Math. Softw.* 45, 1, Article 3 (2019), 28 pages.

[6] Difeng Cai, Edmond Chow, Lucas Erlandson, Yousef Saad, and Yuanzhe Xi. 2018. SMASH: Structured matrix approximation by separation and hierarchy. *Numerical Linear Algebra with Applications* 25, 6 (2018), e2204.

[7] Aparna Chandramowlishwaran, Samuel Williams, Leonid Oliker, Ilya Lashuk, George Biros, and Richard Vuduc. 2010. Optimizing and tuning the fast multipole method for state-of-the-art multicore architectures. In *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*. 1–12.

[8] Shiv Chandrasekaran, Ming Gu, and Timothy P. Pals. 2006. A fast ULV decomposition solver for hierarchically semiseparable representations. *SIAM J. Matrix Anal. Appl.* 28, 3 (2006), 603–622.

[9] Hongwei Cheng, Zydrunas Gimbutas, Per-Gunnar Martinsson, and Vladimir Rokhlin. 2005. On the compression of low rank matrices. *SIAM Journal on Scientific Computing* 26, 4 (2005), 1389–1404.

[10] Eduardo Corona, Per-Gunnar Martinsson, and Denis Zorin. 2015. An $O(N)$ direct solver for integral equations on the plane. *Applied and Computational Harmonic Analysis* 38, 2 (2015), 284–317.

[11] William Fong and Eric Darve. 2009. The black-box fast multipole method. *J. Comput. Phys.* 228, 23 (2009), 8712–8725.

[12] Pieter Ghysels, Xiaoye S. Li, Francois-Henry Rouet, Samuel Williams, and Artem Napov. 2016. An efficient multicore implementation of a novel HSS-structured multifrontal solver using randomized sampling. *SIAM Journal on Scientific Computing* 38, 5 (2016), S358–S384.

[13] Zydrunas Gimbutas, Leslie Greengard, Jeremy Magland, Manas Rachh, and Vladimir Rokhlin. 2019, accessed: 2019-12-05. FMM3D. (2019, accessed: 2019-12-05). https://fmm3d.readthedocs.io

[14] Leslie F. Greengard and Jingfang Huang. 2002. A new version of the fast multipole method for screened Coulomb interactions in three dimensions. *J. Comput. Phys.* 180, 2 (2002), 642–658.

[15] Leslie F. Greengard and Vladimir Rokhlin. 1987. A fast algorithm for particle simulations. *J. Comput. Phys.* 73, 2 (1987), 325–348.

[16] Leslie F. Greengard and Vladimir Rokhlin. 1997. A new version of the fast multipole method for the Laplace equation in three dimensions. *Acta Numerica* 6 (1997), 229–269.

[17] Ming Gu and Stanley C. Eisenstat. 1996. Efficient algorithms for computing a strong rank-revealing QR factorization. *SIAM Journal on Scientific Computing* 17, 4 (1996), 848–869.

[18] Wolfgang Hackbusch. 1999. A sparse matrix arithmetic based on $\mathcal{H}$-matrices. Part I: Introduction to $\mathcal{H}$-matrices. *Computing* 62, 2 (1999), 89–108.

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

[20] Wolfgang Hackbusch and Boris N. Khoromskij. 2000. A sparse $\mathcal{H}$-matrix arithmetic. Part II: Application to multi-dimensional problems. *Computing* 64, 1 (2000), 21–47.

[21] Wolfgang Hackbusch, Boris N. Khoromskij, and Stefan A. Sauter. 2000. On $\mathcal{H}^2$-matrices. In *Lectures on Applied Mathematics: Proceedings of the Symposium Organized by the Sonderforschungsbereich 438 on the occasion of Karl-Heinz Hoffmann's 60th birthday, Munich, June 30 - July 1, 1999*, Hans-Joachim Bungartz, Ronald H. W. Hoppe, and Christoph Zenger (Eds.). Springer, Berlin, 9–29.

[22] N. Halko, P. Martinsson, and J. Tropp. 2011. Finding structure with randomness: probabilistic algorithms for constructing approximate matrix decompositions. *SIAM Rev.* 53, 2 (2011), 217–288.

[23] Dhairya Malhotra and George Biros. 2015. PVFMM: A parallel kernel independent FMM for particle and volume potentials. *Communications in Computational Physics* 18, 3 (2015), 808–830.

[24] Per-Gunnar Martinsson and Vladimir Rokhlin. 2005. A fast direct solver for boundary integral equations in two dimensions. *J. Comput. Phys.* 205, 1 (2005), 1–23.

[25] Per-Gunnar Martinsson and Vladimir Rokhlin. 2007. An accelerated kernel-independent fast multipole method in one dimension. *SIAM Journal on Scientific Computing* 29, 3 (2007), 1160–1178.

[26] Victor Minden, Anil Damle, Kenneth L. Ho, and Lexing Ying. 2017. Fast spatial Gaussian process maximum likelihood estimation via skeletonization factorizations. *Multiscale Modeling & Simulation* 15, 4 (2017), 1584–1611.

[27] Keigo Nitadori, Junichiro Makino, and Piet Hut. 2006. Performance tuning of N-body codes on modern microprocessors: I. Direct integration with a hermite scheme on x86_64 architecture. *New Astronomy* 12, 3 (2006), 169 – 181.

[28] François-Henry Rouet, Xiaoye S. Li, Pieter Ghysels, and Artem Napov. 2016. A distributed-memory package for dense hierarchically semi-separable matrix computations using randomization. *ACM Trans. Math. Softw.* 42, 4 (2016), 27:1–27:35.

[29] Ruoxi Wang. 2018, accessed: 2019-12-05. BBFMM3D. (2018, accessed: 2019-12-05). https://github.com/ruoxi-wang/BBFMM3D

[30] Xin Xing. 2019. *The proxy point method for rank-structured matrices.* Ph.D. Dissertation. Georgia Institute of Technology.

[31] Xin Xing and Edmond Chow. 2020. Interpolative decomposition via proxy points for kernel matrices. *SIAM J. Matrix Anal. Appl.* 41 (2020), 221–243.
[32] Lexing Ying, George Biros, and Denis Zorin. 2004. A kernel-independent adaptive fast multipole algorithm in two and three dimensions. *J. Comput. Phys.* 196, 2 (2004), 591–626.