ASKIT: AN EFFICIENT, PARALLEL LIBRARY FOR HIGH-DIMENSIONAL KERNEL SUMMATIONS

WILLIAM B. MARCH*, BO XIAO*, CHENHAN D. YU†, AND GEORGE BIROS*

Abstract. Kernel-based methods are a powerful tool in a variety of machine learning and computational statistics methods. A key bottleneck in these methods is computations involving the kernel matrix, which scales quadratically with the problem size. Previously, we introduced `ASKIT`, an efficient, scalable, kernel-independent method for approximately evaluating kernel matrix-vector products. `ASKIT` is based on a novel, randomized method for efficiently factoring off-diagonal blocks of the kernel matrix using approximate nearest neighbor information. In this context, ASKIT can be viewed as an algebraic fast multipole method for arbitrary dimensions.

In this paper, we introduce our open-source implementation of `ASKIT`. Features of our `ASKIT` library include: linear dependence on the input dimension of the data, the ability to approximate kernel functions with no prior information on the kernel, and scalability to tens of thousands of compute cores and data with billions of points or hundreds of dimensions. We also introduce some new extensions and improvements of `ASKIT`, included in our library. We introduce a new method for adaptively selecting approximation ranks and correctly partition the nearest neighbor information, both of which improve the performance of ASKIT over our previous implementation.

We describe the `ASKIT` algorithm in detail, and collect and summarize our previous theoretical complexity and error bounds in one place. We present a brief selection of experimental results illustrating the accuracy and scalability of `ASKIT`. We then provide some details and guidance for users of `ASKIT`.

1. Introduction.

In this paper, we introduce a new, open-source library implementing the `ASKIT` algorithm for fast kernel summations. `ASKIT` (Approximate Skeletonization Kernel Independent Treecode) is a novel N-body code for arbitrary dimension, general kernel summations, previously described in [25–28].

A kernel summation is the following problem: given a set of N data points $x_i \in \mathbb{R}^d$, a real-valued function $\mathcal{K}(\cdot)$ of pairs of data points, and a charge vector $w \in \mathbb{R}^N$, we compute the N potentials:

$$u(i) = \sum \mathcal{K}(x_i, x_j) w_j. \tag{1.1}$$

Equivalently, one can view this as a matrix-vector product $u = Kw$, where K is an $N \times N$ matrix of kernel interactions with entries $K_{ij} = \mathcal{K}(x_i, x_j)$. Clearly, evaluating this product directly will require $\mathcal{O}(N^2)$ kernel evaluations. Our task is to compute approximate sums in $\mathcal{O}(N \log N)$ (or $\mathcal{O}(N)$) work.

Motivation. Kernel summations are a fundamental calculation in the solution of partial differential equations [13]. Furthermore, they frequently appear in kernel-based machine learning methods for classification, regression, and density estimation [10,36]; spatial statistics [4]; and Gaussian process modeling [31], among other applications. In particular, kernels are a popular way to transform data non-linearly before applying a linear learning method [17].

Data analysis problems typically involve both large number of points N and dimension d (the number of features). Furthermore, they require a variety of kernel functions $\mathcal{K}$, including kernels that are learned from the data [19] or kernels using variable bandwidths [34]. Therefore, we require a method that scales with N and d and that is kernel independent.

*Institute for Computational Engineering and Sciences, The University of Texas, Austin, TX
†School of Computer Science, The University of Texas, Austin, TX

Treecodes (and Fast Multipole Methods) are a class of efficient methods for kernel summation problems. However, these were generally developed for problems in computational physics, where d is typically small (≤ 3). These methods tend to scale exponentially with d [12, 33, 40], making them ineffective for high-dimensional problems. Furthermore, many of these methods use series expansions which require derivatives or other information on the kernel function. Both of these issues limit the applicability of existing treecodes to high-dimensional problems.

`ASKIT` is a treecode which overcomes these issues with a novel approximation method based on randomized linear algebra. Our previous work on `ASKIT` demonstrated that the algorithm is efficient and accurate as long as the kernel matrix admits a hierarchical low-rank decomposition. We have also shown scalable parallel versions of `ASKIT`. For example, in [28], we show that `ASKIT` can scale to two billion points in 64 dimensions, on 32K cores. This is well beyond the capabilities of previously existing methods. We also show that the kernel matrix-vector product used in `ASKIT` can be used in iterative methods for solving linear systems [26, 27].

Contributions. Here, in a companion paper to our software release, we give an overview of `ASKIT`, summarizing the different versions presented in previous work in a coherent whole. We also present some new improvements to the algorithm, and include a discussion for users of `ASKIT`. In particular, we provide the following:

- We release an open-source, C++ implementation of `ASKIT`*.
- We provide guidance for practical parameter selection for users of `ASKIT` (§5).
- We correct an inaccuracy in our previous use of nearest neighbor information. By partitioning the neighbor information into points used for sampling and point used for pruning, we improve the quality of our approximation over our previous implementation of `ASKIT` §2.3.
- We introduce a new adaptive rank selection criterion and level restriction algorithm to improve users' control over the approximation error §2.3.

Overall, these modifications significantly improve over the original algorithm [25] and make the parameter selection much easier.

Limitations. `ASKIT` is still under development, both in terms of software and the algorithm. Here, we summarize a few of the outstanding questions and issues with our implementation. We discuss some of these in more detail in §5. Our adaptive level restriction algorithm (§2.3) is currently only implemented for a single MPI process. The multiple process version is currently under development. Optimal sampling in terms of both complexity and accuracy is an open problem. Our heuristic of using nearest neighbors for smooth monotonically-decreasing-with-distance kernels to sample works very well but we don't have yet a proof on rigorous error bounds using this sapling. This makes error control much harder than traditional low-dimensional N-body methods, if one wants to control the work complexity.

ASKIT approximates off-diagonal blocks. If the whole matrix admits a low-rank approximation, Nystrom methods are an excellent alternative [27]. As we discuss in §5, `ASKIT` can be modified to behave like a Nystrom meth'od. Currently, this process requires the user to choose between these methods; in the future, the selection between these methods will be automated in our library.

Most of the components of ASKIT have been optimized for performance on x86

*Available at `http://padas.ices.utexas.edu/libaskit/`.

architectures. However, there are still three main opportunities for speed-ups: (1) the algebraic skeletonization (far field approximation) is done using a standard pivoted QR BLAS, which obtains very low performance; (2) we explore no GPU acceleration; and (3) all calculations are done in double precision.

Another limitation is that for problems in lower dimensions, classical geometric range-based pruning will be much more memory-scalable than precomputing nearest neighbor information.

Related work. Previously, we introduced `ASKIT` in [25] and in parallel in [28]. We also performed a comparison against Nystrom methods in [27] and introduced a more rigorous theoretical error analysis of the method. In [26], we extended `ASKIT` to a fast multipole method and introduced several improvements to our parallel implementation. We show experimental results on our nearest neighbor-based sampling heuristic in [24]. The current paper summarizes and builds on all of these.

`ASKIT` is an example of a *treecode*, a family of methods which approximates the sum in (1.1) by partitioning it into exact and approximate evaluations. Examples include the Barnes-Hut algorithm [3] and FMM [11] for the Laplace kernel, the kernel-independent FMM [40], and Fast Gauss Transforms [12, 14, 20, 21, 39]. Another line of work is *Nystrom methods*. These methods efficiently compute a global, low-rank factorization of the matrix K. For more discussion on these methods see [25, 28] and for a comparison with the Nystrom methods see [27]. The main limitation of Nystrom methods is that they assume that K has (from a practical point of view) a global low-rank structure. This is often not the case, especially for large data sets.

Outline. In §2, we review `ASKIT` in some detail. We describe the key features of the method, its implementation in parallel, and some performance improvements. We then discuss our new modifications. In §3, we summarize our theoretical error and complexity bounds for `ASKIT`. In §4, we give some experimental results on the latest version of `ASKIT`. Finally, we provide some guidance for users of `ASKIT` in §5 and conclude in §6.

Notation. We briefly summarize the notation used in the remainder of the paper in Table 1.1. Throughout, we refer to a point for which we compute a potential (the first argument of the kernel function in (1.1)) as a *target*. The point contributing to the potential (second argument in (1.1)) is a *source*.

2. Methods.

We now turn to a description of the `ASKIT` algorithm. We begin with an outline of treecodes in general, then describe the basic parts of `ASKIT`. We then move to the parallel implementation of `ASKIT` in §2.2 and describe some key optimizations of the method. We introduce our new modifications of `ASKIT` in §2.3.

2.1. Treecodes and `ASKIT`.

`ASKIT` is an example of a *treecode* – a method which efficiently approximates kernel summations with $\mathcal{O}(N \log N)$ work. Having introduced the two key features of a treecode—the *outgoing representation* and the *pruning rule*—we describe the novel versions of these in `ASKIT`.

Near-far decomposition. Treecodes decompose the kernel summation (1.1) into the *near field*—interactions which are computed directly—and the *far field*—interactions which can be approximated for each target:

$$u_i = \sum_{j \in \mathrm{Near}(i)} K_{ij} w_j + \sum_{j \in \mathrm{Far}(i)} K_{ij} w_j. \tag{2.1}$$

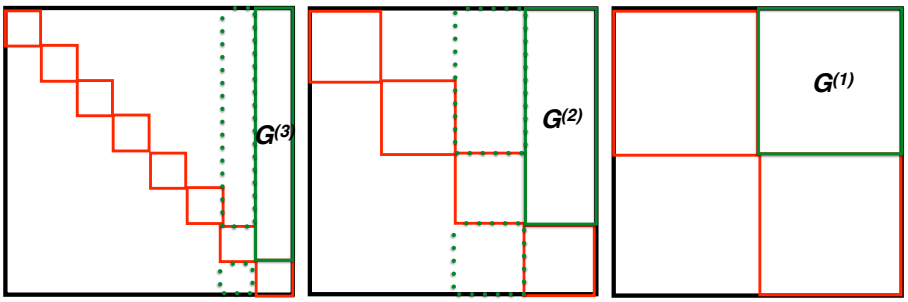
Data and Kernel Functions	
N	number of points
d	ambient dimension of the data
h	bandwidth of Gaussian kernel
x_i, x_j	data points in $\mathbb{R}^d$
Matrices and Vectors	
w	charges or weights on source points
$\tilde{w}$	skeleton weights computed by `ASKIT`
u	potentials of target points
$\tilde{u}$	approximate potentials computed by `ASKIT`
$\bar{u}$	skeleton potentials computed in FMM version of `ASKIT`
K	full $N \times N$ kernel matrix with entries $\mathcal{K}(x_i, x_j)$
G	an off-diagonal block of K
G'	a subsampled off-diagonal block
$\sigma_i(G)$	i^{th} singular value of matrix G
$\tilde{\sigma}_i(G)$	an estimated singular value (2.25)
`ASKIT` Parameters	
κ	number of nearest neighbors per point
m	number of points per leaf
s	number of skeleton points
τ	tolerance used in adaptive rank selection
ℓ	the number of rows sampled in skeletonization
p	number of distributed processes
L	minimum depth at which to skeletonize a node (level restriction)
Tree Notation	
α	a node in the tree
$\mathcal{X}_\alpha$	points in node α
$\mathcal{N}_i$	κ nearest neighbor list for point i
$\mathtt{l}(\alpha)$, $\mathtt{r}(\alpha)$	left or right child of node α
$\mathcal{D}$	the depth of the tree ($\log(N/m)$)
SIBLINGS(α)	nodes with the same parent as α
$\mathcal{A}(\alpha)$	ancestors of α

Table 1.1: *A summary of the notation used in the remainder of the text. We also use $w(\mathcal{I})$ to indicate the components of vector w determined by an index set $\mathcal{I}$ and we use similar notation for matrices.*

Entries in Far(i) are approximated by an *outgoing representation*—a low-rank factorization of the contribution of the node. Clearly, both the speed and the accuracy of a treecode depend on the outgoing representation and the method used to decompose into near and far fields.

Space-partitioning tree. Treecodes derive their name from a *space-partitioning tree*, such as a quadtree, *kd*-tree, or ball tree. The tree partitions the points hierarchically. These tree nodes are then used to partition the points in (2.1). For each target i, we traverse the tree from the root to the leaves. At each node, we apply a *pruning rule* to determine if the node can be placed in the set Far(i). If a leaf node cannot be placed in Far, then it goes in Near.

In our implementation of `ASKIT`, we use a balanced, binary ball tree. We partition



(a) **Level 3.** (b) **Level 2.** (c) **Level 1.**

Fig. 2.1: *We show the off-diagonal matrix blocks approximated by the skeletonization step in* `ASKIT`. *We show on-diagonal blocks in red and off-diagonal in green. We highlight an off-diagonal block $G^{(i)}$ at each level, along with the block corresponding to its sibling (dashed line), which extends above and below the diagonal. At the leaf level (Fig. 2.1(a)), we compute an approximate ID by sampling rows of $G^{(3)}$ to obtain skeleton points. Moving up the tree, (Fig. 2.1(b)), we skeletonize $G^{(2)}$ by merging the skeleton points of block $G^{(3)}$ and its sibling, then computing another approximate ID. We continue (Fig. 2.1(c)) this process all the way up the tree (See Alg. 2.1).*

each node by estimating the farthest pair of points belonging to it. We compute its centroid (x_c), then the farthest point from x_c (x_l), then the farthest point from x_l (x_r). We project all the points onto the line (x_l, x_r), compute the median of the projected points, and split them into two equal-sized groups. We recursively split each node until every leaf contains no more than m points.

Matrix block decomposition. It is instructive to take an alternative view of the partitioning in (2.1). The partitioning of the data into tree nodes corresponds to a partitioning of the kernel matrix K into blocks. More concretely, consider an even split of the points into two groups—*i.e.* the first split in our tree. This corresponds to a matrix partitioning

$$K = \begin{bmatrix} K_1 & G \\ G & K_2 \end{bmatrix}. \tag{2.2}$$

We refer to the blocks K_1 and K_2 as *on-diagonal* blocks. K_1 corresponds to the interactions between the points in the left child node and themselves (and likewise for K_2). The *off-diagonal* blocks correspond to the interactions between the nodes. Further splits of each group of points correspond to further partitionings of the on-diagonal blocks K_1 and K_2 (see Figure 2.1). We can view the outgoing representation of the source points in some tree node as a low-rank factorization of an off-diagonal matrix block G. We employ this view in the construction of our outgoing representation.

We now give a concrete description of the two key features of a treecode mentioned above. First, we describe the novel outgoing representation used in `ASKIT`. We then discuss the combinatorial pruning rule used to create the partitioning in (2.1).

2.1.1. Outgoing Representation. The outgoing representation in `ASKIT` is based on a linear-algebraic decomposition of the off-diagonal blocks G. We define the decomposition we use, then discuss our randomized algorithm to compute it efficiently.

Interpolative Decomposition. Consider a node α with q points. This node corresponds to an off-diagonal block $G = \mathcal{K}(\mathcal{X} \setminus \mathcal{X}_\alpha, \mathcal{X}_\alpha)$—*i.e.* G is the $(N-q) \times q$ block of K with columns corresponding to source points in α and rows corresponding

to all points not in α. Our task is to form a low-rank decomposition of G so that we may apply it to vectors of charges efficiently.

The singular value decomposition provides the optimal reconstruction error for a given rank. However, the SVD constructs a new basis for the column space of G, which requires $\mathcal{O}(N-q)$ storage per vector. Storing this basis for every tree node would require $\mathcal{O}(N^2)$ space, so we require a different factorization.

`ASKIT` uses the *interpolative decomposition* (ID) [29] as a factorization of G. The ID is a rank s decomposition

$$G \approx G_{\text{col}} P \tag{2.3}$$

where G_{col} consists of s columns of G and $P \in \mathbb{R}^{s \times q}$. Since the ID represents G in terms of a subset of its columns, we can store it by simply storing the indices of the source points corresponding to these columns in $\mathcal{O}(s)$ extra storage.

We refer to the columns selected in G_{col} (or, equivalently, the source points corresponding to these columns) as the *skeleton* of α, which we denote $\mathcal{S}_\alpha$. We compute *s skeleton weights*

$$\tilde{w}(\alpha) = P w(\alpha) \tag{2.4}$$

obtained from the original weights of points in α. We can then approximate the contribution of node α in $\text{Far}(i)$ for some target i as

$$\tilde{u}(i) = \mathcal{K}(x_i, \mathcal{S}_\alpha)\tilde{w}(\alpha) \tag{2.5}$$

in $O(s)$ work and with $\mathcal{O}(s)$ storage per node. We refer to the construction of the skeleton and skeleton weights of a node as *skeletonization*.

Combining outgoing representations. The method above can be used directly for a leaf node α. In a treecode, we construct outgoing representations for internal nodes from the representations of the node's children. Let $\mathcal{S}_{\mathtt{l}(\alpha)}$ and $\mathcal{S}_{\mathtt{r}(\alpha)}$ be the skeletons of the children of node α, each of size s. We form the matrix $G = \mathcal{K}(\mathcal{X} - \mathcal{X}_\alpha, \mathcal{S}_{\mathtt{l}(\alpha)} \cup \mathcal{S}_{\mathtt{r}(\alpha)})$ – *i.e.* the matrix with columns corresponding to these skeleton points. We form an ID of this matrix to obtain a new skeleton consisting of a subset of the children's skeleton points. We obtain the skeleton weights from the children using

$$\tilde{w}(\alpha) = P \begin{bmatrix} \tilde{w}(\mathtt{l}(\alpha)) \\ \tilde{w}(\mathtt{r}(\alpha)) \end{bmatrix}. \tag{2.6}$$

Computing the ID. In order to compute an ID such that $\|G - G_{\text{col}}P\|$ is small, we employ a rank-revealing QR factorization to obtain $G\Pi = QR$ for a permutation Π, an orthonormal matrix Q, and upper triangular matrix R. The skeleton $\mathcal{S}_\alpha$ corresponds to the first s columns of $G\Pi$, and the matrix P can be computed from R by solving the system

$$R_{11} P = R_{12} \tag{2.7}$$

where R_{11} is the $s \times s$ upper-left block of R and R_{12} is the $s \times (q-s)$ upper-right block. It can be shown [5, 15] that

$$\|G - G_{\text{col}}P\| \leq \sqrt{1 + qs(q-s)}\,\sigma_{s+1}(G) \tag{2.8}$$

using this method, where σ_{s+1} is the $(s+1)^{\text{st}}$ singular value of G.

Random sampling. While the ID can be used as a compact and accurate outgoing representation, its construction is much too expensive. The QR factorization of G requires $O(Nm^2)$ work for each leaf node. Since there are N/m leaf nodes, this task is more expensive than simply evaluating the kernel summation directly. Thus, we require a more efficient approximation to compute the ID.

We draw on the rich literature on randomized matrix approximation [16, 23]. We subsample rows of G to form an $\ell \times q$ matrix G'. This is equivalent to sampling a subset of the target points. If our sample accurately captures the row space of G, then we can use the ID of G' in place of the ID of the full matrix. In addition to accurately capturing the row space of G, we must be able to obtain our samples efficiently enough to form a part of a fast matrix-vector multiplication algorithm.

Importance sampling distributions based on column norms and statistical leverage scores can be shown to accurately capture the row space [8, 9, 23]. On the other hand, these distributions require at least one access of every matrix entry, which necessarily scales as $\mathcal{O}(N^2)$. Sampling uniformly at random is cheap, but not effective for all matrices.

As a heuristic approximation, we employ nearest-neighbor information to form an importance sampling distribution. When the kernel is a decreasing function of distance, we expect the nearest neighbors to provide a good approximation of the rows with largest norm. Additionally, the nearest neighbors can be approximated without touching each entry of the matrix.

We store the κ nearest neighbors of each point. For a node, we collect the ℓ closest points among the nearest neighbors of points in the node. We use these points as the sampled rows to construct G'. If additional samples are required, we choose them uniformly at random. See [24] for a more thorough exploration of this sampling scheme and empirical evidence that it is effective.

Using this subsampled approximation, we can efficiently compute an interpolative decomposition for each tree node. We summarize this in Algorithm 2.1.

Algorithm 2.1 SKELETONIZE(α)

1: `if` (α) is not a leaf
2: SKELETONIZE($\mathtt{l}(\alpha)$); SKELETONIZE($\mathtt{r}(\alpha)$)
3: $\mathcal{X}_\alpha = \mathcal{S}_{\mathtt{l}(\alpha)} \cup \mathcal{S}_{\mathtt{r}(\alpha)}$
4: $\mathcal{N}_\alpha = \left(\mathcal{N}_{\mathtt{l}(\alpha)} \cup \mathcal{N}_{\mathtt{r}(\alpha)}\right) / \mathcal{X}_\alpha$
5: Collect the first ℓ points in $\mathcal{N}_\alpha \cup \mathcal{U}$ into $\mathcal{T}_\alpha$
6: Form $K(\mathcal{T}_\alpha, \mathcal{X}_\alpha)$ and compute ID $K_{\text{col}}P$
7: Store skeleton $\mathcal{S}_\alpha$ and skeleton weights $\tilde{w}_\alpha$

Nearest neighbors. The nearest neighbors of all points can be computed exactly or approximately (in high d) using randomized projection methods [1, 10, 18, 37]. We employ a greedy search using random projection trees [6]. We build a tree and, for each point x_i, we collect κ-nearest neighbors found by exhaustive search among the other points in the leaf node that contains x_i. Then we discard the tree (we do not perform top-down searches) and iterate, keeping the best candidate neighbors found at each step. For simplicity, in the remainder of our discussion, we consider the nearest neighbors of each point to be an input to `ASKIT`. They can be pre-computed by any

method.

2.1.2. Pruning rule. Having described the method used to approximate off-diagonal blocks, we now turn to the decomposition used in `ASKIT` to split nodes into Near and Far in (2.1). In existing treecodes, nodes are separated into Near(i) and Far(i) according to distances from the target point x_i. For each target, the treecode traverses the tree, starting from the root. If a node is "far enough" from the target (according to some algorithm-specific criterion), it is *pruned—i.e.* placed in Far(i). If a node cannot be pruned, then the algorithm traverses to its children. Leaves which cannot be pruned are placed in Near(i).

Pairwise distance bounds tend to become less effective at separating groups of points as d grows. Due to the concentration of measure effect, the pairwise distances between points will tend to cluster around a single value. In this case, bounding regions around points will overlap more frequently, making pruning increasingly difficult and resulting in too many contributions to Near.

Neighbor-based pruning. In [25], we introduced a novel combinatorial pruning condition based on the nearest-neighbor graph of the input points. Recall the list $\mathcal{N}_i$ of nearest neighbors of point i we collected in the skeletonization step. If a node owns a point $x \in \mathcal{N}_i$, then this node *cannot be pruned* for i. This pruning condition does not depend on any distances between the target and a bounding box or ball, so it avoids the issues with high dimensions mentioned above.

Constructing interaction lists and evaluation. The explicit, recursive traversal of the tree is not necessary with our combinatorial pruning rule. Instead, using *Morton IDs*, we can construct the lists Near and Far for each target. The Morton ID of a node is a bit string that encodes the path from the root to the node. The i^{th} bit of the Morton ID of a node is 1 if the node's ancestor at level i is the right child of its parent and 0 otherwise. We assign a point the same Morton ID as the leaf which owns it.

A node's Morton ID and level are sufficient to uniquely identify it in the tree. We collect the Morton IDs of the points in $\mathcal{N}_i$ for each i. Then, we can construct the near and far field interaction lists for a target in $\mathcal{O}(\kappa)$ time without explicitly traversing the tree using:

$$\begin{aligned} \text{Near}(i) &= \text{MORTONIDS}(\mathcal{N}_i) \\ \text{Far}(i) &= \text{SIBLINGS}(\mathcal{A}(\mathcal{N}_i)) \setminus \mathcal{A}(\mathcal{N}_i) \end{aligned} \tag{2.9}$$

where $\mathcal{A}(\mathcal{X})$ is the set of all ancestors of nodes in the set $\mathcal{X}$ and SIBLINGS($\mathcal{X}$) is the set of siblings of nodes in $\mathcal{X}$. Given these interaction lists, we simply compute the approximate potential for each target:

$$\tilde{u}(i) = \sum_{\alpha \in \text{Near}(i)} \mathcal{K}(x_i, \mathcal{X}_\alpha) w(\alpha) + \sum_{\alpha \in \text{Far}(i)} \mathcal{K}(x_i, \mathcal{S}_\alpha) \tilde{w}(\alpha). \tag{2.10}$$

Neighbors for sampling and pruning. We use nearest neighbor information for both sampling to construct the approximate ID and for partitioning interactions into Near and Far. We split the neighbor list for each point into two components. We use the first κ_{pr} neighbors of each point for pruning, and the remaining κ_{sa} neighbors for sampling. Throughout, we use $\kappa = \kappa_{\text{pr}} + \kappa_{\text{sa}}$ and $\kappa_{\text{pr}} = \kappa/2$.

More concretely, we split the list $\mathcal{N}_i$ of nearest neighbors of each point i. into two lists, ${\mathcal{N}_i}^P$ and ${\mathcal{N}_i}^S$, each with $\kappa/2$ points[†]. The list ${\mathcal{N}_i}^P$ is used for pruning, and has the $\kappa/2$ closer neighbors. The list ${\mathcal{N}_i}^S$ is used for sampling, and has the farther $\kappa/2$ neighbors. Note that each point has itself in its list ${\mathcal{N}_i}^P$.

For skeletonization of a leaf node α, we form the list of targets $\mathcal{T}_\alpha$ as

$$\mathcal{T}_\alpha = \left(\cup_i {\mathcal{N}_i}^S\right) \setminus \left(\cup_i {\mathcal{N}_i}^P\right) \tag{2.11}$$

We then keep the ℓ closest points in $\mathcal{T}_\alpha$ (as measured by the distance from the point to its neighbor in α). These are the points used to form the skeleton.

For internal nodes, we collect the children's lists $\mathcal{T}_{\mathtt{l}(\alpha)}$ and $\mathcal{T}_{\mathtt{r}(\alpha)}$. We also need to remove points in the pruning list of this node. However, storing the lists ${\mathcal{N}_i}^P$ as we go up the tree will be prohibitively expensive. Instead, we store these lists only for the skeleton points. For any node α, we define

$${\mathcal{N}_\alpha}^P = \cup_{i \in \mathcal{S}_\alpha} {\mathcal{N}_i}^P. \tag{2.12}$$

Then, for an internal node, we merge the sample neighbor lists of the children, then remove any points in $\mathcal{N}^P_{\mathtt{l}(\alpha)}$ or $\mathcal{N}^P_{\mathtt{r}(\alpha)}$:

$$\mathcal{T}_\alpha = \left(\mathcal{T}_{\mathtt{l}(\alpha)} \cup \mathcal{T}_{\mathtt{r}(\alpha)}\right) \setminus \left(\mathcal{N}^P_{\mathtt{l}(\alpha)} \cup \mathcal{N}^P_{\mathtt{r}(\alpha)}\right). \tag{2.13}$$

As before, we can add additional uniform samples to the target list if needed.

2.2. Efficiently implementing `ASKIT`. We see that `ASKIT` consists of two main steps: the *skeletonization* step (Algorithm 2.1), in which we construct an approximate ID for each tree node, and the *evaluation* step (2.10), in which we compute the approximate potential. We now discuss some important extensions and improvements to this basic algorithm which we use in our software.

2.2.1. ASKIT in parallel. We begin by discussing our parallel implementation of `ASKIT` [27]. The algorithm consists of four major steps which need to be parallelized: construction of the space-partitioning tree, skeletonization of each node (Alg. 2.1), construction of the interaction lists (2.9), and evaluation of the approximate potentials (2.10). Our parallel implementation of `ASKIT` uses a hybrid MPI / OpenMP scheme with p distributed processes and multiple threads per process. We also introduced several improvements to our original parallel implementation in [26]. These improvements are a part of our software release and the results in this paper.

For the construction of the space-partitioning tree, we refer the reader to [28,38]. For the remainder of our discussion, we point out that our tree construction assigns a subtree containing N/p points to each process. We refer to this subtree as the *local tree* and the portion of the tree above level $\log p$ as the *distributed tree*. As a part of the tree construction, we ensure that each processor has the Morton IDs and coordinates of the nearest neighbors of all of its points.

Skeletonization in parallel. In the local tree, the skeletonization of each node on a level can be done in parallel. We assign the skeletonizations of different nodes to different OpenMP threads. No communication between processes is required. When we reach the distributed tree, the skeletonization proceeds as a reduction. Every node

[†] In the case that $\kappa = 1$, we put the only neighbor in the list ${\mathcal{N}_i}^P$ and sample uniformly.

collects skeleton points and skeleton weights from its children by communicating with its sibling process, constructs its skeleton, then passes this information up the tree. Once we collect the points, the subsampled QR factorization is performed locally.

Interaction lists with a Local Essential Tree. Given the set of target points assigned to a single MPI process, the local essential tree (LET) is the portion of the global tree (leaves, internal nodes and skeleton data) that is required to evaluate the potential at the targets [35].

The construction of our interaction lists using Morton IDs allows us to efficiently collect the information needed for the LET. Let $\mathcal{X}_p$ be the set of all points owned by process p and $\mathcal{N}_p$ be the set of all nearest neighbors of points in $\mathcal{X}_p$. Then, let $\mathcal{L}$ be the set of all leaves containing points in $\mathcal{N}_p \setminus \mathcal{X}_p$. The LET is the set of nodes

$$\text{LET} = \mathcal{L} \cup \left[\cup_{\alpha \in \mathcal{L}} \left[\text{SIBLINGS}(\mathcal{A}(\alpha)) \setminus \mathcal{A}(\alpha)\right]\right]. \tag{2.14}$$

Since the Morton ID and level of a node uniquely identify it, and since each process has the Morton IDs of all of the nearest neighbors of its target points, the process can identify locally the nodes it needs for its LET.

Each node can then obtain the skeletons and coordinates of the nodes in its LET in two communication phases. First, an all-to-all primitive allows each process to send its node requests to all other processes. Then, in a second all-to-all, each process answers these requests with either the point coordinates and charges (for a leaf node), or skeleton points and skeleton weights (for an approximated node).

Evaluation. Once each process has the contents of its LET, no further communication is required to compute the approximate potentials. We can compute (2.10) in an embarrassingly parallel fashion on each process.

2.2.2. `ASKIT` efficiency improvements. We now discuss some further improvements to `ASKIT` used in our implementation.

Blocking interaction lists. For more efficient evaluation, we can block the interaction lists over source nodes. Concretely, for each node α, we form the lists

$$\text{Near}(\alpha) = \{i : \alpha \in \text{Near}(i)\}. \tag{2.15}$$

We form similar lists for far-field interactions. Then, in order to evaluate the potentials, we simply compute the matrix vector products

$$\begin{aligned} \tilde{u}(\text{Near}(\alpha)) &\mathrel{+}= \mathcal{K}(\text{Near}(\alpha), \mathcal{X}_\alpha)\, w(\alpha) \\ \tilde{u}(\text{Far}(\alpha)) &\mathrel{+}= \mathcal{K}(\text{Far}(\alpha), \mathcal{S}_\alpha)\, \tilde{w}(\alpha). \end{aligned} \tag{2.16}$$

By blocking the interaction lists in this way, the evaluation phase can be carried out more efficiently. An interaction consists of indices for the source and target points, weights (or skeleton weights), and indices in the final potential (or skeleton potential). On an MPI process, the individual interactions can be computed in parallel in any order. We employ a new kernel summation library in our code to compute these interactions efficiently [41].

Extending ASKIT to a Fast Multipole Method. Fast Multipole Methods are similar to treecodes—kernel interactions are partitioned into near and far sets, which are evaluated directly or approximately with an outgoing representation. Additionally, they employ an *incoming representation*. The outgoing representation compactly summarizes the contribution of a group of sources to the potential at some

distant point. The incoming representation efficiently approximates the contribution at a group of targets from distant source points. The combination of these two representations allows an FMM to approximate the contribution between a large number of targets and a large number of sources efficiently. This allows FMM's to have an evaluation phase which scales as $\mathcal{O}(N)$.

We present the extension of `ASKIT` to an FMM in detail elsewhere [26]. Here, we only mention that an interpolative decomposition can be used as an incoming representation as well[‡]. Let α be a node of target points and β a node of source points, each with an interpolative decomposition. Then, if all points in β belong to $\text{Far}(i)$ for all points in α, we can compute the approximate interaction as

$$\tilde{u}(\alpha) = P_\alpha^T \mathcal{K}(\mathcal{S}_\alpha, \mathcal{S}_\beta) P_\beta w(\beta) \tag{2.17}$$

We make use of this observation by updating the interaction lists. We begin by forming the interaction lists $\text{Far}(i)$ for each target i in (2.9). We then merge the lists from leaves up the tree. Concretely, for a leaf node α, we compute a list

$$\text{FMMFar}(\alpha) = \cap_{i \in \alpha} \text{Far}(i). \tag{2.18}$$

We remove the merged nodes from each target list:

$$\text{Far}(i) = \text{Far}(i) \setminus \text{FMMFar}(\alpha). \tag{2.19}$$

Similarly, we merge common entries for the children of each node and put them in the list of the parent. This requires a second upward pass of the tree after skeletonization and the construction of the interaction lists.

In addition to the computations in (2.16), for the evaluation step we now loop over all nodes $\beta \in \text{FMMFar}(\alpha)$ and compute

$$\bar{u}(\mathcal{S}_\alpha) \mathrel{+}= \mathcal{K}(\mathcal{S}_\alpha, \mathcal{S}_\beta) \tilde{w}(\beta) \tag{2.20}$$

where $\bar{u}$ are the *skeleton potentials* for node α. As a final post-processing step, we then pass down the tree and apply the matrix P_α^T to obtain skeleton potentials for the children of node α:

$$[\bar{u}(\mathcal{S}_{\mathtt{l}(\alpha)}), \bar{u}(\mathcal{S}_{\mathtt{r}(\alpha)})] \mathrel{+}= P_\alpha^T \bar{u}(\mathcal{S}_\alpha). \tag{2.21}$$

Applying P^T at the leaf level results in the final potentials:

$$u(\alpha) \mathrel{+}= P_\alpha^T \bar{u}(\mathcal{S}_\alpha). \tag{2.22}$$

The complete algorithm is given in Algorithm 2.2.

ASKIT in iterative methods. We mention one more important feature of `ASKIT`. Frequently, kernel sums are used inside a loop. For instance, kernel regression requires solving the system $Kx = y$. Since we cannot explicitly construct K, let alone invert it, we use an iterative solver [26, 27]. In this case, we need to compute a kernel sum for a fixed K for many different charge vectors w.

[‡] If the kernel function is symmetric, then we can re-use the same ID for both incoming and outgoing representations. If it is not, then we compute one ID for the matrix G' and one for G'^T. For simplicity, we restrict our attention to the symmetric case here.

Algorithm 2.2 u=ASKIT$(\mathcal{X}, w, s, \ell, m)$

1:	Read or compute the κ nearest neighbors of all points	
2:	Construct tree and compute Morton IDs	
3:	SKELETONIZE(α)	Alg. 2.1
4:	Construct interaction lists Near(i) and Far(i) for all i	(2.9)
5:	Merge interaction lists to obtain FMMFar(α)	(2.18)
6:	Invert and block lists	(2.15)
7:	Compute interactions in lists	(2.16)
8:	Apply P^T to skeleton potentials	(2.21), (2.22)

In ASKIT, the skeletonization, LET construction and exchange, and blocking of interaction lists do not need to be repeated for a new charge vector. For the skeletonization step, we simply store the matrix P in each node. Then, for a new charge vector, we simply need to compute $\tilde{w}(\alpha) = Pw(\alpha)$. This step is much cheaper than the QR factorization required for the initial skeletonization. In the LET, we only need to exchange the updated charges. We record which exchanges were necessary to obtain the LET. We use this information to perform a single all-to-all communication to obtain the updated skeleton weights.

Bichromatic kernel summations. So far, we have described ASKIT in the case that the source and target sets are identical. However, the method can be extended to the case where they are distinct (the *bichromatic* case) [26]. This case typically occurs in a learning setting where we first *train* a model using a monochromatic evaluation, then *test* the model on some new set of targets.

We perform the training exactly as described above. Then, we read in the new test targets. We compute the nearest neighbors of each testing point in the training set. We assign each testing point the Morton ID of its nearest neighbor. We can then compute interaction lists for the test points using (2.9) and update the LET with any additional nodes needed.

2.3. New features of ASKIT. While the primary purpose of this paper is to introduce our open-source ASKIT library and provide some guidance for future users, we also introduce a few improvements to the existing ASKIT algorithm.

Adaptive rank skeletonization. In the original implementation of ASKIT [25], we chose the skeleton size s as an input parameter. This approach makes controlling the final error in the approximation extremely difficult. Also, for some data sets and kernels, some nodes may be easier to compress than others—*i.e.* they will require fewer skeleton points to achieve a given accuracy.

In [27], we introduced an adaptive rank selection method. The user specifies a tolerance τ. Then, when skeletonizing an off-diagonal block G, we estimate the singular values of G from the QR factorization. We use a *relative adaptive rank criterion* to select s by:

$$s = \min \{ i : \sigma_{i+1}(G)/\sigma_1(G) < \tau \} . \tag{2.23}$$

In our previous work, we estimated the singular values of G using the diagonal elements of the factor R in the QR factorization [15].

While this method can be effective, consider a case where the off-diagonal block G makes very little contribution in the far field. In other words, the first singular

value $\sigma_1(G)$ is small. This can occur, for instance, if the kernel is extremely thin or if G corresponds to an isolated cluster of source points. In this case, we will have to choose a large approximation rank to satisfy the condition in (2.23). However, because G makes a small contribution at target points, a smaller rank would result in little additional error in the final potentials.

We now describe a new alternative method to estimate the singular values of the off-diagonal block G and use them to choose an approximation rank. We again specify a rank tolerance parameter τ. When skeletonizing, we construct the subsampled off-diagonal block G' as before. If we knew the singular values of the true off-diagonal block G, then we could control the error by choosing s such that

$$\sigma_{s+1}(G) < \tau \tag{2.24}$$

for every off-diagonal block G.

In practice, we estimate the singular values of G using our subsampled approximation and the diagonal elements in the upper triangular factor in the QR factorization computed for skeletonization [15]. In the relative error implementation, we did not need to scale our estimated singular values to account for missing rows and columns because the scaling factors in the numerator and denominator in (2.23) will cancel. In the absolute condition, we must account for these missing rows and columns.

Let q be the total number of source points owned by the node being skeletonized, and let q' be the number of columns of G – *i.e.* $q' = q$ for a leaf node and q' is the sum of the children's skeleton sizes for an internal node. We then form an approximate estimate of the singular values of G as

$$\tilde{\sigma}_i(G) = R_{ii} \left(\frac{q}{q'}\right)^{\frac{1}{2}} \left(\frac{N-q}{\ell}\right)^{\frac{1}{2}} \tag{2.25}$$

The scaling factor is derived from the uniform sampling case: if we sample ℓ rows and q columns uniformly at random and scale the singular values by the factors in (2.25), then in expectation we will obtain the singular values of G. We discuss the difficulty in reconciling this approach with our neighbor-based sampling in §4 and §5. We then choose s by

$$s = \arg\min\{i : \tilde{\sigma}_i(G) < \tau\}. \tag{2.26}$$

In this way, we can account for nodes which make a small contribution to the final potential while still retaining the flexibility of the adaptive rank algorithm.

Adaptive level restriction. The cost of skeletonizing a single node is $\mathcal{O}(\ell s^2)$ if both children have skeletons of size s. In the formulation above, s may be as large as the entire node, leading to extremely high skeletonization costs. In order to govern this, we introduced a parameter $s_{\max}$, the largest possible skeleton size, in [27]. In that work, if the adaptively selected rank exceeded $s_{\max}$, then we switched over to the fixed rank algorithm with rank $s_{\max}$. We coupled this approach with a *level restriction* parameter L; we do not skeletonize any node that is less than L levels from the root. This reduces the error by reducing the number of nodes using the fixed rank skeletonization.

However, the level restriction method is difficult to choose *a priori*, since a user will not know how well nodes will skeletonize in advance. Here, we introduce an

adaptive level restriction method. If the condition in (2.26) chooses a rank larger than $s_{\max}$ for some node, then we mark the node as "unprunable". Any target which would put the node in its list Far instead interacts with the node's children.

This introduces a trade-off in terms of the complexity of `ASKIT`. We bound the cost of any individual skeletonization by $\mathcal{O}(\ell {s_{\max}}^2)$. However, the evaluation cost may grow, since we may not be able to prune some nodes. This approach also has the advantage of requiring less input from the user without sacrificing accuracy.

3. Theory. Our previous work proved theoretical guarantees on both the approximation error and complexity of `ASKIT`. We gather and summarize those results here.

3.1. Error Bounds. `ASKIT` incurs two main sources of error: the approximation of the off-diagonal block G in a basis of s of its columns using the ID and the approximate construction of the ID by subsampling G to form G'. In [27], we provide a more thorough accounting for these sources of error. We summarize these results, in particular a key theorem bounding the error of `ASKIT` in terms of the singular values of the off-diagonal blocks. We introduce this bound as a foundation for our discussion of the control of error in `ASKIT` in practice.

Notation and preliminaries. For simplicity, we consider a fixed approximation rank s and we construct interaction lists for $\kappa = 1$. We will discuss how our bounds generalize to arbitrary κ and adaptively chosen s below. Since we always consider a point to be its own nearest neighbor, in the $\kappa = 1$ case, a target interacts directly with the points in its leaf and interacts with the skeletons of all of the siblings of its ancestors.

The first split in the tree creates the submatrices given in (2.2). The tree recursively splits collections of points, which corresponds to splitting the on-diagonal blocks K_1 and K_2 in the same fashion. We also partition the weight vector w in the same way. We refer to an on-diagonal block at level i as $K^{(i)}$—*i.e.* a block corresponding to the interactions of points in a node with themselves. For the same node, we use $G^{(i)}$ to refer to the interactions between points in the node and all other points. These are the blocks approximated in `ASKIT` (in the $\kappa = 1$ case). We illustrate these blocks in Fig. 2.1. In our discussion, it is useful to identify a node with its off-diagonal block.

For our discussion, it is useful to consider the structure of the approximate matrix $\tilde{K}$ computed by `ASKIT`. In the $\kappa = 1$ case, the approximation computed by ASKIT is

$$\tilde{K}w = \sum_{i=0}^{\mathcal{D}} \tilde{G}^{(i)} w^{(i)} + K^{(\mathcal{D})} w^{(\mathcal{D})} \tag{3.1}$$

where the summation runs over all levels of the tree and the matrices $G^{(i)}$ and $K^{(i)}$ are interpreted as direct sums over the subblocks corresponding to individual matrix nodes. Since the term $K^{(\mathcal{D})} w^{(\mathcal{D})}$ is exact (corresponding to direct evaluations at the leaves), we only need to bound the error of each term $\tilde{G}^{(i)}$.

Error from low-rank approximations. In the best case (SVD), we have that the error due to a rank s approximation of G is the largest omitted singular value of G: $\sigma_{s+1}(G)$. Our error bounds depend on this quantity throughout. The error for an ID is bounded by (2.8). In [27], we proved a bound on the error from combining ID's in the skeletonization phase, which we incorporate in our bound below.

Sampling matrix rows. We now turn to the error due to the random sampling used to approximate the ID. The number of samples needed to achieve high accuracy with high probability depends on the structure of the matrix. Consider, as an example, a matrix with r rows, where the first $r-1$ rows are parallel and the last row v_r is orthogonal to them. In this case, unless our sample includes v_r, we will not accurately capture the row space of the matrix. In order to be likely to capture v_r with a uniform sampling scheme, we will require $\mathcal{O}(r)$ samples.

Leverage scores. The *leverage scores* of a matrix measure the "importance" of a row to the overall structure of the row space. Let G be any matrix, and let U be a matrix of left singular vectors of G. Then, the statistical leverage scores of rows of G with respect to rank s are defined as:

$$\ell_j = \|U(j, 1{:}s)\|. \tag{3.2}$$

In our example above, the score ℓ_r corresponding to row v_r will be large, capturing the importance of this row to the overall space.

As we mentioned in §2, an effective method is to sample from an importance distribution based on leverage scores. However, in the context of `ASKIT`, constructing or approximating these scores is too expensive. Therefore, we consider two simpler sampling methods: uniform sampling and an importance distribution based on nearest neighbor information.

Error due to uniform sampling. We begin by considering a uniform sampling distribution, which is what we use in `ASKIT` in the $\kappa = 1$ case. The accuracy of uniform matrix sampling can be bounded in terms of the leverage scores using a theorem from [24].

THEOREM 3.1. *Let G be $n \times q$. Sample ℓ rows of G to form G'. Let ζ be the coherence of G with respect to a given rank s, defined as*

$$\zeta = \max_j \ell_j^2, \tag{3.3}$$

where the ℓ_j are the leverage scores of rows of G.

Let the number of samples satisfy

$$\ell \geq 10q\zeta \log(2s/\delta) \tag{3.4}$$

and let Π be a projection onto the sampled rows. Then, with probability at least $(1-\delta)$

$$\|G(I - \Pi)\| \leq \left(1 + 6\frac{n}{\ell}\right)^{\frac{1}{2}} \sigma_{s+1}(G). \tag{3.5}$$

Intuitively, a small value of ζ means that the action of the matrix is more evenly distributed among the rows. In this case, a small uniform sample can be successful. On the other hand, a large value of ζ means that there is some part of the row space of G that is spanned by very few of the rows of G. Therefore, we must sample some or all of these rows in order to accurately capture the row space.

Note that the quantity ζ is bounded between s/q and 1. In the case that ζ is small, the number of samples needed is proportional to $s \log s$.

Error from neighbor-based sampling. Proving a bound on the sampling based on nearest neighbor information is more difficult. Sampling from an importance

distribution based on leverage scores can lead to $(1+\epsilon)$ error [22, 23]. Distributions based on Euclidean distances [7] also have theoretical bounds. In [24], we explore the performance of neighbor-based sampling schemes empirically. Further theoretical analysis depends on the ability to understand the relationship between the neighbor distribution and the distribution of leverage scores.

ASKIT error bound. We prove a bound on the overall error of `ASKIT` in [27].

THEOREM 3.2. *[27]. Compute an approximation with* `ASKIT` *using* $\kappa = 1$ *and assume that the number of uniform samples satisfies* (3.4)*. Then:*

$$\epsilon_A \leq \{c_{samp} + c_{id}\} \log(N/m) \max \sigma_{s+1}(G) \|w\| \tag{3.6}$$

where the maximum is taken over all off-diagonal blocks G *and with the sampling error and ID error terms defined as:*

$$c_{samp} = 2 + \left(1 + 6\frac{N-m}{\ell}\right)^{\frac{1}{2}} \tag{3.7}$$

$$c_{id} = (1 + ms(m-s))^{\frac{1}{2}} + \log(N/m)\left(1 + 2s^3\right)^{\frac{1}{2}}. \tag{3.8}$$

In this result, the term c_{samp} is from sampling targets uniformly at random, c_{id} is the error of an interpolative decomposition, and the factor $\log(N/m)$ is from the accumulation of this error up the levels of the tree.

The term $\log(N/m) \max \sigma_{s+1}(G)$—the decay of the spectrum of the of the off-diagonal blocks—is key to the approximation error of `ASKIT`. The term c_{id} is a worst-case bound; while matrices exist for which (2.8) is not loose, empirical work on kernel matrices shows that the ID is nearly as accurate as the SVD in practice [24]. The term c_{samp} is more difficult to understand and is discussed further below.

Effect of increasing κ. The number of neighbors κ plays two roles with respect to the error of `ASKIT`. First, it reduces the term c_{samp} in the case that the neighbor distribution captures the leverage score distribution. Second, increasing κ means that we prune fewer nodes and perform more direct evaluations. In terms of (3.6), we expect the singular values of G to decay more quickly, since G only corresponds to interactions between points that are more distant.

3.2. Complexity Bounds. We have previously discussed the computational and storage complexity of `ASKIT` in [25] and in parallel in [28]. We also discuss some of the improved versions of the algorithm in [27]. Here, for completeness, we summarize these results.

Notation. We use t_s for the communication latency and t_w for the reciprocal of the bandwidth. We will assume a hypercube topology for our complexity results, although nothing in the implementation assumes this. In these results, we assume that a single kernel evaluation requires $\mathcal{O}(d)$ work, which is the only explicit dependence on the ambient dimension.

We let $M = N/m$ represent the number of leaves. Note that the total number of nodes in the tree is $2M - 1 = \mathcal{O}(M)$. The tree has depth $\mathcal{D} = \log(N/m)$. We let $n = N/p$ be the number of points owned by each distributed process.

Nearest-neighbor search. We assume that the nearest-neighbor list $\mathcal{N}_i$ for each point is given. We either precompute these, or read them from storage along

with the point coordinates. Note that exact nearest-neighbors can be computed in $\mathcal{O}(N)$ time for low-intrinsic dimensional sets [30] and approximation schemes, such as the one we use, are even faster [1, 18, 37]. Since ASKIT can use exact or approximate neighbor information obtained from any source, we do not consider this cost further.

After reading the data and neighbor information, we first construct the tree in parallel, then we create the skeletonization (outgoing and incoming representations), then construct and exchange the local essential tree, and finally we evaluate the approximate potentials.

Tree construction. The parallel tree construction method used in ASKIT is described in detail in [37, 38] and requires

$$(t_s + t_w) \log^2 p \log N + (t_w \log p)\,(d + \kappa)\, n$$

time. The storage is $\mathcal{O}(d\kappa n)$.

Skeletonization. The skeletonization of a leaf node requires $\mathcal{O}(d\ell m + \ell m^2)$ work, where the first term is the cost of forming the $\ell \times m$ matrix G' and the second is for the QR factorization. Similarly, the cost for an internal node is $\mathcal{O}(d\ell s + \ell s^2)$. In keeping with our use of ASKIT in practice, we assume that $\ell = \mathcal{O}(s)$ for the remainder of our discussion.

Nodes up to level $\log p$ (from the root) are skeletonized locally with no communication required. At and above this level, we skeletonize as a reduction. We combine the skeletons of two siblings with a message of size $\mathcal{O}(ds)$, so the total communication is bounded by $\mathcal{O}((t_s + t_w) sd \log p)$. Since we have a total of $2(n/m + \log p)$ nodes per MPI process, and assuming that $m \leq s$ (to simplify the expressions), the total skeletonization time is

$$T_S = \left(\frac{n}{m} + \log p\right)(ds^2 + s^3) \tag{3.9}$$

and the storage cost is $s^2 n + (ds + s^2) \log p$.

In the adaptive rank algorithm, we can replace s with $s_{\max}$ to obtain a worst-case bound over all computations. However, in the case that many nodes have skeletons that are much smaller than $s_{\max}$, this bound will be pessimistic.

LET construction. In [25], we showed that the number of nodes in the near and far interaction lists for any target i are bounded by

$$\begin{aligned} |\text{Near}(i)| &= \mathcal{O}(\kappa) \\ |\text{Far}(i)| &= \mathcal{O}(\kappa \mathcal{D}). \end{aligned} \tag{3.10}$$

Note that these bounds are also generally pessimistic, in that they assume that all of the neighbors of the target belong to different nodes. When the data have low intrinsic dimension, this will generally not be the case.

We can obtain the entries in the interaction lists for a point in $\mathcal{O}(\kappa\mathcal{D})$ time using the Morton IDs of the nearest neighbors. Therefore, the total work for the LET construction step is

$$\mathcal{O}\left(\kappa \frac{N}{p} \log\left(\frac{N}{m}\right)\right). \tag{3.11}$$

Once we have formed these lists (which require no communication), each process must obtain the skeleton coordinates and charges for every node in Far(i) and the

coordinates and input charges for every node in Near(i) for every target i. In the worst case, all of these nodes must be received from another MPI process. Thus, the worst case of the communication during the LET exchange is

$$t_s p + t_w d\kappa(m + s\mathcal{D})n. \tag{3.12}$$

The LET requires additional storage of size $d\kappa\,(m + s\mathcal{D})\,n$.

Once again, these bounds are pessimistic if there is significant overlap between the interaction lists on a given MPI process. This will occur more often in the case that the data have low intrinsic dimension. The list blocking step requires no additional communication and can be done in time proportional to the size of the lists.

Evaluation. Given the LET, we require no additional communication to evaluate the approximate potentials. The evaluation step itself requires time

$$T_E = \mathcal{O}(dn\kappa(m + s\mathcal{D})). \tag{3.13}$$

Complexity of `ASKIT-FMM`. The FMM version of `ASKIT` used in this paper requires an additional step to merge lists Far(α) over nodes. This requires $\mathcal{O}(\kappa\mathcal{D}N)$ work and is part of the setup phase—*i.e.* it does not need to be repeated when updating the charge vector. The total work in the evaluation phase depends on how well the lists can be merged. In the worst case, no merging occurs, and the complexity of the evaluation phase is the same as the treecode version. In the other extreme, when $\kappa = 1$, we obtain a bound of $\mathcal{O}(nd(\kappa + s^2/m))$, since each node only needs to interact with its sibling. See [26] for more details on this analysis.

Total storage. A key bottleneck in the use of `ASKIT` is the storage requirement, particularly as κ increases. Here, we state the total storage cost per distributed process of the algorithm for the user's reference. We assume that we store the matrix P in the ID step and that we have a maximum skeleton size s. In this case, the worst-case memory requirement is bounded by:

$$d(\kappa+1)\frac{N}{p} + \left(2\frac{N}{pm} + \log p\right)s^2 + 2d\kappa\frac{N}{p}\log\left(\frac{N}{m}\right)(s+m)\,. \tag{3.14}$$

The first term is the space required for the point coordinates in the case that all of the nearest neighbors belong to another process. The second term is the cost of storing the skeletons (including the matrix P), and the last term is for the coordinates and charges of each node in the LET. As before, this bound is pessimistic in the case that there is significant overlap between nearest neighbors.

4. Experiments. We now briefly report selected experimental results on `ASKIT`. Also we present convergence results with our new method for partitioning nearest neighbors between sampling and pruning sets and our new adaptive rank selection criterion. For a more thorough discussion of the empirical performance of`ASKIT`, see our previous papers [25–28].

Datasets. We use the following data sets in our experiments, chosen to reflect a range of values of d and varying structures and intrinsic dimensions. See Table 4.1 for details. We generate synthetic data from a uniform distribution in 3 dimensions, **(Uniform 3D)** and a uniform distribution in 6 dimensions, embedded in 64 ambient dimensions **(Uniform 64D)**[§]. We also use four datasets derived from learning tasks.

[§] We sample the six dimensional data, pad it with zeroes into 64 dimensions, then apply a random rotation.

Set	N	d
UNIFORM 3D	1,000,000	3
UNIFORM 64D	1,000,000	64
COVTYPE	500,000	54
SUSY	4,500,000	18
HIGGS	10,500,000	28
BRAIN	10,584,046	246

Table 4.1: *Details of the data sets used in our experiments. We give the number of points N and dimension d of each set. Note the points in the UNIFORM 64D set are drawn from a 6 dimensional distribution, then embedded in 64 dimensions. The COVTYPE, SUSY, and HIGGS sets are from [2].*

The **COVTYPE** set is derived from a forest cover classification task. The **SUSY** and **HIGGS** sets are from a classification task in particle physics. These three sets are from the UCI ML repository [2]. Each data point of the **BRAINS** set corresponds to a single pixel of an MRI image of one of 50 brains. The features are based on correlations with neighboring pixels. This set is from a segmentation task to identify white from gray matter in brain images. Detailed results on these data will be reported elsewhere.

Setup. Our experiments use the Maverick cluster at the Texas Advanced Computing Center. The nodes have two Intel Xeon E5-2680 v2 (2.8GHz) CPUs and 256GB RAM each. All tests were done in double-precision arithmetic. Our implementation is written in C++ using OpenMP, MPI, the Intel MKL library, and vectorization using x86 intrinsics.

Error measurements. In keeping with our previous work, we report the relative error in our matrix approximation:

$$\epsilon_2 = \frac{\|Kw - \tilde{K}w\|}{\|Kw\|} \tag{4.1}$$

where K is the exact kernel matrix and $\tilde{K}$ is the approximation computed by `ASKIT`. Since Kw is prohibitively expensive to compute, we sample $n_s = 1000$ entries of the vector Kw and compute vector norms on this subsample. All of our experiments are repeated over 10 independent charge vectors w with independent standard normal entries. We also use the same sample points in different experiments with the same data set.

Timing measurements. We report timings for our experiments as well. We break the timing of `ASKIT` into four parts. The skeletonization time T_S is the cost to sample targets and compute the interpolative decomposition of each tree node. The LET construction time T_{LET} is the time to construct the interaction lists for each target point (or node) and exchange the local essential tree (in parallel experiments). The list blocking time T_L is the time to block the interaction lists for efficient evaluation. Finally, the time T_E is the cost to perform the kernel evaluations and obtain the approximate potentials. Note that this last cost is the only one that needs to be repeated for new right hand sides (see §2.2).

Kernels. Here, we show results for two kernel functions. The Gaussian is characterized by a bandwidth h, and is given by

$$\mathcal{K}(x_i, x_j) = \exp\left(-\frac{\|x_i - x_j\|^2}{2h^2}\right). \tag{4.2}$$

#	τ	ϵ_2	T_S	T_{LET}	T_L	T_E	%K
1	1E-11	5E-10	439	53	7	4	2.1%
2	1E-07	5E-05	73	16	1	1	0.6%
3	1E-05	2E-04	29	15	1	1	0.4%
4	1E-03	1E-03	14	15	1	1	0.3%
5	1E-01	6E-03	10	15	1	1	0.2%

Table 4.2: ***Uniform 3D data, Gaussian kernel,*** $\mathbf{h = 0.1}$. *All experiments use absolute adaptive error criteria with adaptive level restriction. We set* $\mathbf{m = 512}$ *and* $\kappa = \mathbf{64}$*, evenly split between pruning and sampling. The reported columns are: "#" to index the experiments for reference in the text, "*$\boldsymbol{\tau}$*" for the adaptive error tolerance, "*$\boldsymbol{\epsilon_2}$*" for the estimated approximation error (4.1), "*$\boldsymbol{T_S}$*" for the skeletonization time, "*T_{LET}*" for the Local Essential Tree construction, "*$\boldsymbol{T_L}$*" for the list blocking, and "*$\boldsymbol{T_E}$*" for the evaluation of approximate potentials. "*%$\boldsymbol{K}$*" is the percentage of kernel evaluations performed in the evaluation step relative to the number required for a direct evaluation. All timings are in seconds. All experiments are performed on a single compute node.*

As h tends to zero, the kernel approaches a delta function. As h increases, then the kernel takes on a single constant value for all arguments.

We also show some results for the Laplace kernel in three dimensions, given by

$$\mathcal{K}(x_i, x_j) = \|x_i - x_j\|^{-1}. \tag{4.3}$$

This kernel has a singularity as the distance between the points goes to zero. This presents a challenge for our nearest-neighbor sampling and pruning schemes, discussed in §2.3 and below. See [26] for results on more kernel functions.

4.1. Discussion. We now discuss our experimental results. Our intent is simply to illustrate the effectiveness of our `ASKIT` library on a range of data sets and kernel parameters. We show both that `ASKIT` converges in terms of error and that it can scale to large data sets.

All of our experiments here use the newest version of `ASKIT` using the improvements in §2.3: neighbor lists are partitioned into pruning and sampling lists; approximation ranks are selected adaptively using the absolute cutoff criterion; we use the FMM version of `ASKIT`; we use the adaptive level restriction; and kernel interactions are blocked and optimized.

Convergence of error. We begin with some simple convergence and timing results, along the lines of the studies of `ASKIT` in our previous work. Our goal here is to simply illustrate that our new absolute error criterion allows more sensitive error control than our previous work without significant increase in cost. In Table 4.2, we show results on a uniform distribution in 3D, and in Table 4.3, for uniform data in 6 dimensions embedded in 64 dimensions. In both cases, we explore a Gaussian kernel with a single value of h.

In these experiments, we see that the error decreases smoothly with decreasing τ in Table 4.2, suggesting that our adaptive ID condition can be used to control the error. In Table 4.3, however, we see that in order to get more than 5 digits of accuracy for the 64D data, we must perform more than half of the total kernel interactions (#6). If this much accuracy is desired, it would be preferable to perform the summation directly.

In the adaptive rank selection algorithm (§2.3) with tolerance τ, and in the event that our samples satisfy the conditions of Theorem 3.2, the approximation error would

#	τ	ϵ_2	T_S	T_{LET}	T_L	T_E	%K
6	1E-05	9E-06	1068	395	149	260	56%
7	1E-03	4E-04	486	67	11	29	6.2%
8	1E-01	5E-03	57	30	1	9	1.6%

Table 4.3: ***Uniform 64D (6 intrinsic) data, Gaussian kernel,*** $\mathbf{h = 0.385}$. *All experiments use absolute adaptive error criteria with adaptive level restriction. We set* $\mathbf{m = 512}$ *and* $\kappa = \mathbf{64}$*, evenly split between pruning and sampling. The reported columns are: "#" to index the experiments for reference in the text, "*$\boldsymbol{\tau}$*" for the adaptive error tolerance, "*$\boldsymbol{\epsilon_2}$*" for the estimated approximation error (4.1), "*$\boldsymbol{T_S}$*" for the skeletonization time, "*$\boldsymbol{T_{LET}}$*" for the Local Essential Tree construction, "*$\boldsymbol{T_L}$*" for the list blocking, and "*$\boldsymbol{T_E}$*" for the evaluation of approximate potentials. "*%$\boldsymbol{K}$*" is the percentage of kernel evaluations performed in the evaluation step relative to the number required for a direct evaluation. All timings are in seconds. All experiments are performed on a single compute node.*

be bounded by

$$\|Kw - \tilde{K}w\| \leq (c_{\text{samp}} + c_{\text{id}}) \log(N/m)\tau\|w\|. \tag{4.4}$$

However, our experimental results show that controlling the error in `ASKIT` is not always this simple. While we have shown that our nearest-neighbor-based sampling heuristic is effective empirically, we do not currently have a theoretical guarantee of the error due to this approach. Additionally, we scale our estimated singular values using (2.25). While this scaling is correct in expectation for uniformly sampled rows, it does not currently account for the influence of the neighbor information. Inaccuracy in the estimation of the singular values can translate into an incorrect selection of the approximation rank as well. Obtaining tighter control over the approximation error is a subject of ongoing work and is discussed further in §5.

Dependence on h. In Table 4.4 we show results for `ASKIT` on the COVTYPE data set for a range of values of h to briefly highlight the dependence of the method on the kernel function. These bandwidths are chosen based on our previous work on the COVTYPE set [27].

At one extreme, when the kernel is extremely thin, nearest neighbor information is sufficient (#9). As h gets larger, we enter a more difficult regime. Even including 45% of the kernel interactions (# 18) only gets two digits of accuracy for $h = 0.35$. As h increases further, we see the far-field interactions compress more easily (# 22).

Both the COVTYPE and UNIFORM results are in line with our previous experimental work on `ASKIT`. For example, in [27], we observed that some Gaussian kernel bandwidths can be difficult to compress. This is due to the lack of a rapidly decaying spectrum of the off-diagonal blocks. In this case, (3.6) predicts that a large approximation rank will be necessary to achieve small error. See [24] for a study of the spectra of these off-diagonal blocks.

Laplace kernel. We also look at the Laplace kernel in Table 4.5. These results particularly highlight the advantage of our new partitioning of nearest neighbors when compared with our work in [26]. In that work, using the same neighbors for pruning and sampling, we achieve $\epsilon_2 =$ 8E-3 using 5% of the total kernel evaluations (see [26], Table 5, #23). Here, using 4% of the total evaluations, we achieve more than three orders of magnitude improvement in the error (#27). Furthermore, the results in [26] required a very large value of L and small value of κ to overcome the difficulties associated with not using the partitioned neighbor list.

#	h	τ	ϵ_2	T_S	T_{LET}	T_L	T_E	%K
9	0.02	1E-01	3E-11	8	25	0	2	1.6%
10	0.05	1E-07	9E-04	207	41	1	5	5.1%
11	0.05	1E-05	1E-03	101	31	0	4	3.2%
12	0.05	1E-03	2E-03	24	25	0	2	2.1%
13	0.05	1E-01	3E-03	8	25	0	2	1.6%
14	0.16	1E-07	5E-02	532	70	4	19	18%
15	0.16	1E-05	5E-02	505	65	3	17	16%
16	0.16	1E-03	6E-02	275	43	1	7	7.0%
17	0.16	1E-01	8E-02	22	26	0	3	2.7%
18	0.35	1E-07	3E-02	625	108	8	47	45%
19	0.35	1E-05	9E-02	319	53	2	13	12%
20	0.35	1E-03	9E-02	74	26	0	4	3.7%
21	0.35	1E-01	1E-01	10	26	0	2	2.2%
22	10.0	1E-07	7E-03	11	25	0	3	2.3%
23	10.0	1E-05	2E-02	9	26	0	2	1.9%
24	10.0	1E-03	1E-02	9	25	0	2	1.7%
25	10.0	1E-01	3E-02	9	26	0	2	1.6%

Table 4.4: ***COVTYPE data, Gaussian kernel.*** *All experiments use absolute adaptive error criteria with adaptive level restriction. We set* $\mathbf{m = 512}$ *and* $\kappa = \mathbf{1024}$*, evenly split between pruning and sampling. The reported columns are: "#" to index the experiments for reference in the text, "*$\boldsymbol{\tau}$*" for the adaptive error tolerance, "*$\boldsymbol{\epsilon_2}$*" for the estimated approximation error (4.1), "*$\boldsymbol{T_S}$*" for the skeletonization time, "*T_{LET}*" for the Local Essential Tree construction, "*$\boldsymbol{T_L}$*" for the list blocking, and "*$\boldsymbol{T_E}$*" for the evaluation of approximate potentials. "*%$\boldsymbol{K}$*" is the percentage of kernel evaluations performed in the evaluation step relative to the number required for a direct evaluation. All timings are in seconds. All experiments are performed on a single compute node.*

In these results, we also look at two different values of κ. We see that for the same value of τ, $\kappa = 512$ tends to result in smaller error than $\kappa = 2048$ (*e.g.*.#26 vs. #30). This is counter to our intuition that larger κ results in smaller error. However, we see from the number of kernel evaluations that the $\kappa = 512$ results in larger approximation rank sizes, which in turn results in the smaller approximation error.

We also point out that `ASKIT` is both slower and less accurate than specialized codes for the Laplace kernel. For low-dimensional problems, standard FMM's are a better tool. We include these results to demonstrate the generality of `ASKIT` and as a point of comparison with existing methods.

Single node timings. In Tables 4.2–4.5, we also see that the skeletonization and evaluation times increase with decreasing values of τ. This is because smaller values of τ result in larger approximation ranks throughout the tree. We see that the skeletonization time increases more drastically, since it is proportional to s^2 (3.9), while the evaluation step is only proportional to s (3.13). For example, if we compare #17 and #14, we see a factor of 24 increase in the skeletonization time but only a 6x increase in evaluation due to the larger skeletons. Overall, we see that skeletonization accounts for most of the evaluation time. However, this cost can be amortized over many different right hand sides in an iterative method, since the evaluation step is the only work that needs to be repeated for new charges. Furthermore, as we discuss in §5, our skeletonization code is currently less optimized than the evaluation code.

ASKIT on large data sets. Finally, we show selected results for larger data

#	τ	κ	ϵ_2	T_S	T_{LET}	T_L	T_E	%K
26	1E-05	512	3E-08	833	207	51	73	11%
27	1E-03	512	5E-06	402	115	12	27	4.1%
28	1E-01	512	1E-03	140	47	2	9	1.3%
29	1E-00	512	8E-03	92	30	1	6	0.9%
30	1E-05	2048	5E-07	403	114	10	25	3.6%
31	1E-03	2048	2E-05	187	90	3	12	1.9%
32	1E-01	2048	1E-03	62	82	1	8	1.2%
33	1E-00	2048	1E-02	31	83	1	7	1.0%

Table 4.5: ***Uniform 3D data, Laplace kernel.*** *All experiments use absolute adaptive error criteria with adaptive level restriction. We set* $\mathbf{m = 512}$*. The reported columns are: "#" to index the experiments for reference in the text, "*τ*" for the adaptive error tolerance, "*κ*" for the total number of nearest neighbors split evenly between pruning and sampling, "*ϵ_2*" for the estimated approximation error (4.1), "*T_S*" for the skeletonization time, "*T_{LET}*" for the Local Essential Tree construction, "*T_L*" for the list blocking, and "*T_E*" for the evaluation of approximate potentials. "*%K*" is the percentage of kernel evaluations performed in the evaluation step relative to the number required for a direct evaluation. All timings are in seconds. All experiments are performed on a single compute node.*

#	h	τ	L	ϵ_2	T_S	T_{LET}	T_L	T_E	%K
34	0.05	1E-05	7	1E-03	981	43	16	34	5.1%
35	0.05	1E-00	2	5E-03	17	31	2	4	0.4%
36	0.15	1E-03	10	2E-02	959	94	132	184	30%
37	0.15	1E-00	7	8E-02	514	46	17	31	4.7%

Table 4.6: ***SUSY data, Gaussian kernel.*** *All experiments use absolute adaptive error criteria with fixed level restriction. These experiments use* $\mathbf{p = 8}$ *nodes of Maverick. We set* $\mathbf{m = 1024}$ *and* $\kappa = \mathbf{1024}$*, evenly split between pruning and sampling. The reported columns are: "#" to index the experiments for reference in the text, "*h*" for the kernel bandwidth, "*τ*" for the adaptive error tolerance, "*L*" for the level restriction, "*ϵ_2*" for the estimated approximation error (4.1), "*T_S*" for the skeletonization time, "*T_{LET}*" for the Local Essential Tree construction, "*T_L*" for the list blocking, and "*T_E*" for the evaluation of approximate potentials. "*%K*" is the percentage of kernel evaluations performed in the evaluation step relative to the number required for a direct evaluation. All timings are in seconds.*

sets using our parallel implementation of `ASKIT`. In Table 4.6, we show results for SUSY data. We also show results for the HIGGS and BRAIN sets in Table 4.7 and Table 4.8 using bandwidths from our previous work [26]. These experiments use a fixed level restriction L. We choose L to ensure that no nodes have a skeleton larger than $s_{\max} = 2048$ for the given value of τ.

We see in these experiments that the HIGGS set is extremely difficult for the bandwidth we have chosen; doing 25% of the kernel interactions still results in nearly 10% error (#38). On the other hand, the BRAIN set compresses more easily (#41). Note also that for the BRAIN set, the evaluation step takes longer relative to the skeletonization step (*e.g.* #40) than for our other data sets. Since this set has much larger d than the others, the evaluation of the kernel functions is more expensive. The evaluation step scales linearly with the number of evaluations, while the skeletonization is dominated by the cost of the QR factorization.

Scaling of ASKIT. We also explore the scalability of ASKIT with the size of the input data. More detailed scaling results are in [26, 28]. We perform a weak scaling experiment using the UNIFORM 64D distribution and Gaussian kernel. We choose the bandwidth h according to a theoretically optimal value for kernel density

#	τ	L	ϵ_2	T_S	T_{LET}	T_L	T_E	$\%K$
38	1E-03	10	9E-02	110	236	37	259	24%
39	1E-01	8	1E-01	18	196	6	127	11%

Table 4.7: ***HIGGS data, Gaussian kernel,*** $\mathbf{h = 1.0}$. *All experiments use absolute adaptive error criteria with fixed level restriction. These experiments use* $\mathbf{p = 32}$ *nodes of Maverick. We set* $\mathbf{m = 512}$ *and* $\kappa = \mathbf{1024}$*, evenly split between pruning and sampling. The reported columns are: "#" to index the experiments for reference in the text, "*$\boldsymbol{\tau}$*" for the adaptive error tolerance, "*$\boldsymbol{L}$*" for the level restriction, "*$\boldsymbol{\epsilon_2}$*" for the estimated approximation error (4.1), "*$\boldsymbol{T_S}$*" for the skeletonization time, "*$\boldsymbol{T}_{LET}$*" for the Local Essential Tree construction, "*$\boldsymbol{T_L}$*" for the list blocking, and "*$\boldsymbol{T_E}$*" for the evaluation of approximate potentials. "*$\boldsymbol{\%K}$*" is the percentage of kernel evaluations performed in the evaluation step relative to the number required for a direct evaluation. All timings are in seconds.*

#	τ	L	ϵ_2	T_S	T_{LET}	T_L	T_E	$\%K$
40	1E-07	10	2E-03	596	58	29	320	5.2%
41	1E-03	8	5E-03	247	28	3	69	0.9%

Table 4.8: ***BRAIN data, Gaussian kernel,*** $\mathbf{h = 3.5}$. *All experiments use absolute adaptive error criteria with fixed level restriction. These experiments use* $\mathbf{p = 32}$ *nodes of Maverick. We set* $\mathbf{m = 512}$ *and* $\kappa = \mathbf{1024}$*, evenly split between pruning and sampling. The reported columns are: "#" to index the experiments for reference in the text, "*$\boldsymbol{\tau}$*" for the adaptive error tolerance, "*$\boldsymbol{L}$*" for the level restriction, "*$\boldsymbol{\epsilon_2}$*" for the estimated approximation error (4.1), "*$\boldsymbol{T_S}$*" for the skeletonization time, "*$\boldsymbol{T}_{LET}$*" for the Local Essential Tree construction, "*$\boldsymbol{T_L}$*" for the list blocking, and "*$\boldsymbol{T_E}$*" for the evaluation of approximate potentials. "*$\boldsymbol{\%K}$*" is the percentage of kernel evaluations performed in the evaluation step relative to the number required for a direct evaluation. All timings are in seconds.*

estimation for Gaussian distributions of points [32]. We fix the number of points per core at 50,000, and report timing results in Table 4.9. We show two sets of experiments. In the first, we fix the approximation rank $s = 512$. This has the effect of fixing the amount of work per node in both skeletonization and evaluation. In the second set, we use the adaptive rank selection method with $\tau = 0.1$ to fix the overall matrix approximation error.

The scalings in both sets of experiments are qualitatively similar. The skeletonization phase scales linearly up to 320 cores, then increases slightly for 640 cores. This is in line with the prediction of (3.9), which says this phase will scale linearly until the $\log p$ term becomes too large. The list construction and LET exchange steps scale worse than linearly; again, (3.11) predicts $N \log N$ scaling. Since $\kappa = 128$, we see slightly worse than linear scaling in the evaluation phase. This corresponds to a situation between the linear $\kappa = 1$ case and the same scaling as the treecode version of the algorithm.

The major difference we see between the two experiments is that the parallel efficiency seems to be better for the fixed-rank (fixed s) experiments. In the fixed rank-experiments, each node requires a large skeleton, while nodes low in the tree for the adaptive rank have small skeletons. The cost of skeletonization in the fixed s set is therefore dominated by the large number of expensive leaf node skeletonizations. This work is completely independent on each process, so we expect this step to scale well. Since the skeletonization is responsible for most of the total runtime in these experiments, this scaling accounts for the better parallel efficiency. However, these experiments are still informative, because as we see in experiments with lower error (*e.g.* #6 and #34), skeletonization is often the dominant cost.

5. User's Guide to ASKIT. We now turn to a practical discussion of the `ASKIT` open-source library. `ASKIT` is available at `http://padas.ices.utexas.edu/`

	Fixed s				Fixed ϵ_2			
N	**1M**	**4M**	**16M**	**32M**	**1M**	**4M**	**16M**	**32M**
#cores	**20**	**80**	**320**	**640**	**20**	**80**	**320**	**640**
h	0.31	0.24	0.19	0.17	0.31	0.24	0.19	0.17
ϵ_2	7E-02	1E-01	2E-01	3E-01	4E-01	4E-01	4E-01	4E-01
T_S	207	207	208	214	15	15	17	19
T_L	2	2	4	5	1	1	2	3
T_{LET}	13	17	31	40	12	15	21	27
T_E	14	17	21	23	5	6	7	9
Total	235	244	263	281	33	37	47	58
Efficiency	1.00	0.97	0.90	0.84	1.00	0.89	0.70	0.57

Table 4.9: ***Uniform, 64d (6 intrinsic) data, Gaussian kernel.*** *We report two sets of weak-scaling results on* `ASKIT-FMM`*. In the first set, we use a fixed approximation rank* $\mathbf{s} = \mathbf{512}$*. In the second, we use the adaptive rank algorithm with* $\boldsymbol{\tau} = \mathbf{0.1}$ *to achieve fixed error. We sample 50K points per compute core. The experiments were run on Maverick with one MPI process per node and 20 threads per process. We set* $\boldsymbol{\kappa} = \mathbf{128}$ *and* $\mathbf{m} = \mathbf{512}$*. The reported values are: "*T_S*" for the skeletonization time, "*T_{LET}*" for the Local Essential Tree construction, "*T_L*" for the list blocking, and "*T_E*" for the evaluation of approximate potentials. We also report the observed parallel efficiency relative to the 20 core (1 MPI rank) case. All timings are in seconds.*

`libaskit/`, along with its dependencies. The documentation in the code contains more details for users. Here we simply outline the basic concepts.

Estimating nearest neighbors. Our implementation of `ASKIT` reads neighbor information for each point from a file. These files can be generated from any source. In our experiments, we estimate neighbors using the RKDT library, available along with `ASKIT` at the link above. This library is a dependency for `ASKIT` since it contains the tree construction code, so this is an easy choice for users as well. The memory for the nearest neighbors can quickly increase, necessitating distributed memory parallelism.

Using `ASKIT` for a single kernel summation. The `ASKIT` library comes with a command line function to compute kernel matrix vector products. The user specifies the data, neighbor file, and kernel function and parameters. This functionality can be used to estimate the error and runtime of the method, as in our experiments. In particular, users may want to do a preliminary run with this approach to verify their parameter selections before using `ASKIT` in their applications.

Using `ASKIT` inside another method. Typically, `ASKIT` will be used as an inner loop in another computation. For example, we demonstrate embedding `ASKIT` in a kernel regression application in [26, 27]. In this case, for a given kernel function, the skeletonization and list construction steps only need to be performed once. `ASKIT` can be provided with a new vector of charges, which it uses to update the skeleton weights efficiently. Then, the only additional computation required is the evaluation step. Our implementation of `ASKIT` generates a static library which can be linked against other code as desired.

Adding kernel functions. `ASKIT` currently supports several kernel functions. However, adding additional functions is straightforward. The `ASKIT` code is encapsulated in a C++ class which takes a template argument. This argument is a class which computes kernel functions. In order to extend `ASKIT` with a new kernel function, a user only needs to implement this kernel class.

Choice of `ASKIT` variants. In our previous work (and this paper), we have

introduced several variants of the ASKIT method. For this release, we recommend that users use the FMM version of ASKIT §2.2 with the absolute, adaptive skeletonization rank selection criterion and adaptive level restriction (§2.3). These approaches will all be enabled as the defaults, with other options available for advanced users.

ASKIT as a Nystrom-like method. ASKIT can also be used in a way that is similar to a Nystrom method. Our implementation has an option to compress the on-diagonal blocks as well. With this option, no direct evaluations are performed; instead, all interactions are approximated with the node's skeleton. In this case, we sample points from within the node as well as outside it to construct the skeleton.

By using a small number of splits in the tree (*e.g.* one or two), ASKIT only partitions the matrix into a few large blocks. Furthermore, we can set $\kappa = 1$, so that all of the samples are chosen uniformly. Using these options, together with the skeletonization of on-diagonal blocks, we obtain factorizations of the (2 or 4) matrix blocks from uniform samples. We then evaluate using only these factorizations. This is similar to a Nystrom method, in that it consists of a (nearly) global low-rank factorization, chosen by sampling a small matrix uniformly and factoring it.

5.1. Parameter Selection. We now turn to some guidance for users for parameter selection in ASKIT. The main parameters of interest to a user are:

- κ – the number of nearest neighbors per point,
- τ – the absolute adaptive rank selection tolerance,
- m – the number of points per tree leaf,
- $s_{\max}$ – the maximum possible skeleton size,
- ℓ – the number of sampled targets for approximate ID construction.

We address each of these parameters briefly, drawing on our theoretical and empirical results. Overall, the selection of parameters will depend on both the data set and the difficulty of the kernel function. As our previous work [27] and §4.1 have shown, some kernel functions are simply difficult to compress. If the number of kernel evaluations required for a desired accuracy is a large fraction of the total (30% or more), then there is not a more efficient approach than to evaluate the sum directly. As noted previously, we recommend that users do a "dry run" with their parameter selection before using ASKIT in their application.

Neighbors – κ. In general, more nearest neighbors result in more accuracy, since there are more direct evaluations and the sampling for ID construction is typically more accurate. We recommend that one uses as many neighbors as possible. However, it should be noted that depending on the kernel and data, it may be possible to obtain good error with no neighbor information.

The storage requirements increase drastically with the number of neighbors, particularly for high-dimensional data. As κ increases, more neighbors of each point will belong to another distributed process, as shown in (3.14). This cost drastically limits the number of neighbors a user will be able to employ. In our experiments, we generally limit κ to at most 2048, even with our large compute nodes.

ID tolerance – τ. The choice of the adaptive skeletonization tolerance is up to the user—it determines the accuracy that ASKIT will achieve, but comes at a price of both increased skeletonization and evaluation time. Our experiments give some representative values of τ with the resulting approximation accuracy. This parameter may require some direct experimentation on the part of the user. As we discuss below, it is a topic of ongoing work to provide a more rigorous connection between the input

tolerance and the actual error achieved by the method.

In general, reducing τ will decrease the error, but at an increased cost. As the size of a node's skeleton increases, the cost of evaluating an approximate interaction with the node increases linearly. This cost will be reflected in an increase in the cost of the evaluation phase (T_E in our experiments). The cost of skeletonization will increase more drastically. Since QR factorization for an internal node requires $\mathcal{O}(\ell s^2)$ time, an increase in the skeleton size of a node causes a quadratic increase in the cost of skeletonizing its parent.

As the total number of kernel evaluations required for a desired accuracy increases, the additional costs of `ASKIT` will outweigh the cost of performing a direct evaluation. If very high accuracy (for a difficult kernel function and data set) is truly needed, direct evaluation may be preferable.

Leaf size – m. The parameter m appears in both the error and complexity bounds. In the error, however, it appears as part of the tree depth in the worst case that errors accumulate up the tree in the combination of IDs. In practice, this is generally pessimistic. Therefore, we recommend choosing m based on complexity considerations. It is important for m to be large enough for direct evaluations to be dense enough to be efficient, but no so large that the number of direct evaluations is too large. It is also useful to keep m from being too large because we have to skeletonize each leaf node at a cost of $\mathcal{O}(\ell m^2)$. We find $m = 512$ to represent a good compromise value.

Maximum skeleton size – $s_{\max}$. The maximum skeleton size is also determined by complexity considerations. In principle, the error does not depend on $s_{\max}$ because the adaptive rank selection criterion (2.26) holds. In terms of complexity, we have a tradeoff between the cost of skeletonizing nodes that have rank close to $s_{\max}$ and the increased cost of evaluation from marking many nodes as "unprunable". However, the skeletonization cost is only incurred once in an iterative computation, while the evaluation step occurs in each iteration. In our experiments, we use $s_{\max} = 2048$.

Number of samples – ℓ. The sampling step is the least well understood in terms of its effect on the error. If the number of samples is too small, then we do not have a bound on the term c_{samp} in the error bound (3.7). Additionally, our adaptive rank selection criterion depends on our estimate of the singular values of G obtained from G'. If we sample too few rows, then this estimate may be very inaccurate.

In terms of complexity, the cost is straightforward. The skeletonization cost scales linearly with ℓ, and this term does not appear in the evaluation phase. We use $\ell = 2q$ for a matrix of q columns in our experiments. It is also possible to force `ASKIT` to take some additional uniform samples along with those from the nearest neighbor lists. While this may improve the quality of the samples (at increased cost), we defer a more detailed analysis of the sampling to future work.

5.2. Outstanding Issues. `ASKIT` is still under development, both in terms of software and the algorithm. Here, we summarize a few of the outstanding questions and issues with our implementation.

Timing of skeletonization phase. In typical use cases of `ASKIT`, such as in our experiments, the skeletonization phase is extremely expensive. For each node, we must perform a pivoted QR factorization of a potentially large matrix. Currently, our implementation is relatively unoptimized for this part of the algorithm, as opposed to the evaluation phase. This is an area of ongoing work.

Sampling. The sampling step is currently the least well understood part of ASKIT. If the samples accurately capture the row space of the off-diagonal block, then we have rigorous theoretical guarantees on the error of the method. While we have guarantees in the case of uniform sampling for bounded coherence (Thm. 3.1), this does not address the biased, neighbor-based sampling we use in practice or the case when the coherence is large. A better understanding of this sampling distribution will lead to a more reliable method for choosing the number of samples ℓ, rather than the heuristic method used currently.

Scaling singular values. A related issue is the scaling of singular values used in our new adaptive rank selection criterion (2.25). For uniformly chosen samples, this scaling will provide the correct value of σ in expectation. However, when we use neighbor information, we sample from a distribution that is quite different from uniform. Understanding how to correctly scale the estimated singular values is crucial to more rigorous error control in ASKIT.

6. Conclusion. In this paper, we presented our open-source implementation of ASKIT, the Approximate Skeletonization Kernel Independent Treecode. In our previous work, we have show that ASKIT is scalable, efficient, and accurate. Additionally, there are important kernel summation problems, such as arise in supervised learning applications, where existing methods fail to provide good accuracy [27], but ASKIT is more successful. We have also demonstrated that ASKIT can be successfully embedded in kernel-based supervised learning methods [26, 27].

Here, we summarized the ASKIT method and its complexity and error bounds. We also illustrated the performance of the method on some representative kernel summation applications and provided guidance to users of our software.

Our ongoing development of ASKIT will focus on a better understanding of the quality of sampling used to construct the approximate interpolative decomposition in the skeletonization step. In particular, we are exploring a better theoretical understanding of the neighbor-based sampling method used in ASKIT. This will in turn lead to more reliable error control from the user's standpoint, and a more robust way to choose the number of samples required in skeletonization.

REFERENCES

[1] A. Andoni and P. Indyk, *Near-Optimal Hashing Algorithms for Approximate Nearest Neighbor in High Dimensions*, Communications of the ACM, 51 (2008), p. 117.

[2] K. Bache and M. Lichman, *UCI machine learning repository*, 2013.

[3] Josh Barnes and Piet Hut, *A hierarchical o (n log n) force-calculation algorithm*, Nature, 324 (1986), pp. 446–449.

[4] Jie Chen, Lei Wang, and Mihai Anitescu, *A fast summation tree code for Matérn kernel*, SIAM Journal on Scientific Computing, 36 (2014), pp. A289–A309.

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

[6] S. Dasgupta and Y. Freund, *Random projection trees and low dimensional manifolds*, in Proceedings of the 40th annual ACM symposium on Theory of computing, ACM, 2008, pp. 537–546.

[7] Amit Deshpande and Santosh Vempala, *Adaptive sampling and fast low-rank matrix approximation*, in Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques, Springer, 2006, pp. 292–303.

[8] Petros Drineas, Ravi Kannan, and Michael W Mahoney, *Fast monte carlo algorithms for matrices i: Approximating matrix multiplication*, SIAM Journal on Computing, 36 (2006), pp. 132–157.

[9] Alan Frieze, Ravi Kannan, and Santosh Vempala, *Fast monte-carlo algorithms for finding low-rank approximations*, Journal of the ACM (JACM), 51 (2004), pp. 1025–1041.

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

[11] L. Greengard and V. Rokhlin, *A Fast Algorithm for Particle Simulations*, Journal of Computational Physics, 73 (1987).

[12] Leslie Greengard and John Strain, *The fast gauss transform*, SIAM Journal on Scientific and Statistical Computing, 12 (1991), pp. 79–94.

[13] Greengard, L., *Fast Algorithms For Classical Physics*, Science, 265 (1994), pp. 909–914.

[14] Michael Griebel and Daniel Wissel, *Fast approximation of the discrete gauss transform in higher dimensions*, Journal of Scientific Computing, 55 (2013), pp. 149–172.

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

[16] N. Halko, P. Martinsson, and J. Tropp, *Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions*, SIAM Review, 53 (2011), p. 217.

[17] Thomas Hofmann, Bernhard Schölkopf, and Alexander J Smola, *Kernel methods in machine learning*, The annals of statistics, (2008), pp. 1171–1220.

[18] Peter W Jones, Andrei Osipov, and Vladimir Rokhlin, *A randomized approximate nearest neighbors algorithm*, Applied and Computational Harmonic Analysis, 34 (2013), pp. 415–444.

[19] Gert RG Lanckriet, Nello Cristianini, Peter Bartlett, Laurent El Ghaoui, and Michael I Jordan, *Learning the kernel matrix with semidefinite programming*, The Journal of Machine Learning Research, 5 (2004), pp. 27–72.

[20] Dongryeol Lee, Alexander Gray, and Andrew Moore, *Dual-tree fast gauss transforms*, Advances in Neural Information Processing Systems, 18 (2006), p. 747.

[21] Dongryeol Lee, Piyush Sao, Richard Vuduc, and Alexander G Gray, *A distributed kernel summation framework for general-dimension machine learning*, Statistical Analysis and Data Mining, (2013).

[22] M.W. Mahoney and P. Drineas, *Cur matrix decompositions for improved data analysis*, Proceedings of the National Academy of Sciences, 106 (2009), p. 697.

[23] Michael W Mahoney, *Randomized algorithms for matrices and data*, Foundations and Trends® in Machine Learning, 3 (2011), pp. 123–224.

[24] William B. March and George Biros, *Far-field compression for fast kernel summation methods in high dimensions*, Submitted for publication, (2014), pp. 1–43. arxiv.org/abs/1409.2802v1.

[25] William B. March, Bo Xiao, and George Biros, *ASKIT: Approximate skeletonization kernel-independent treecode in high dimensions*, SIAM Journal on Scientific Computing, 37 (2015), pp. A1089–A1110.

[26] William B. March, Bo Xiao, Sameer Tharakan, Chenhan Yu, and George Biros, *A kernel-independent FMM in general dimensions*, in Proceedings of Supercomputing (to appear), 2015.

[27] ———, *Robust treecode approximation for kernel machines*, in Proceedings of KDD 2015, 2015. http://padas.ices.utexas.edu/static/papers/kdd15askit.pdf.

[28] William B. March, Bo Xiao, Chenhan Yu, and George Biros, *An algebraic parallel treecode in arbitrary dimensions*, in Proceedings of IPDPS 2015, 29th IEEE International Parallel and Distributed Processing Symposium, Hyderabad, India, May 2015. http://padas.ices.utexas.edu/static/papers/ipdps15askit.pdf.

[29] Per-Gunnar Martinsson, Vladimir Rokhlin, and Mark Tygert, *A randomized algorithm for the decomposition of matrices*, Applied and Computational Harmonic Analysis, 30 (2011), pp. 47–68.

[30] Parikshit Ram, Dongryeol Lee, William B. March, and Alexander G Gray, *Linear-time algorithms for pairwise statistical problems*, in Advances in Neural Information Processing Systems, 2009, pp. 1527–1535.

[31] Carl Edward Rasmussen and Christopher Williams, *Gaussian Processes for Machine Learning*, MIT Press, 2006.

[32] Bernard W. Silverman, *Density Estimation for Statistics and Data Analysis*, Chapman and Hall, 1986.

[33] Marina Spivak, Shravan K Veerapaneni, and Leslie Greengard, *The fast generalized Gauss transform*, SIAM Journal on Scientific Computing, 32 (2010), pp. 3092–3107.

[34] George R Terrell and David W Scott, *Variable kernel density estimation*, The Annals of Statistics, (1992), pp. 1236–1265.

[35] Michael S. Warren and John K. Salmon, *A parallel hashed octree N-body algorithm*, in Proceedings of Supercomputing, The SCxy Conference series, Portland, Oregon, November 1993, ACM/IEEE.

[36] Larry Wasserman, *All of Statistics: A Concise Course in Statistical Inference*, Springer Science & Business Media, 2004.

[37] Bo Xiao, *Parallel algorithms for the generalized n-body problem in high dimensions and their applications for bayesian inference and image analysis*, PhD thesis, Georgia Institute of Technology, Atlanta, GA, USA, 8 2014.

[38] Bo Xiao, Logan Moon, and George Biros, *Parallel algorithms for nearest neighbor searches*, 2014. Submitted for publication,padas.ices.utexas.edu/static/papers/knn.pdf.

[39] Changjiang Yang, Ramani Duraiswami, Nail A Gumerov, and Larry Davis, *Improved fast gauss transform and efficient kernel density estimation*, in Computer Vision, 2003. Proceedings. Ninth IEEE International Conference on, IEEE, 2003, pp. 664–671.

[40] Lexing Ying, George Biros, and Denis Zorin, *A kernel-independent adaptive fast multipole method in two and three dimensions*, Journal of Computational Physics, 196 (2004), pp. 591–626.

[41] Chenhan D. Yu and George Biros, *Optimized direct kernel summation on x86 architectures*, 2015. in preparation, code available at https://github.com/ChenhanYu/ks.