An algebraic parallel treecode in arbitrary dimensions

William B. March, Bo Xiao, Chenhan D. Yu, and George Biros

Institute for Computational Engineering and Sciences
The University of Texas at Austin, Austin, TX 78712
march@ices.utexas.edu bo@ices.utexas.edu chenhan@cs.utexas.edu gbiros@acm.org

***Abstract*—We present a parallel treecode for fast kernel summation in high dimensions—a common problem in data analysis and computational statistics. Fast kernel summations can be viewed as approximation schemes for dense kernel matrices. Treecode algorithms (or simply treecodes) construct low-rank approximations of certain off-diagonal blocks of the kernel matrix. These blocks are identified with the help of spatial data structures, typically trees. There is extensive work on treecodes and their parallelization for kernel summations in three dimensions, but there is little work on high-dimensional problems. Recently, we introduced a novel treecode, ASKIT, which resolves most of the shortcomings of existing methods.**

We introduce novel parallel algorithms for ASKIT, derive complexity estimates, and demonstrate scalability on synthetic, scientific, and image datasets. In particular, we introduce a local essential tree construction that extends to arbitrary dimensions in a scalable manner. We introduce data transformations for memory locality and use GPU acceleration. We report results on the "Maverick" and "Stampede" systems at the Texas Advanced Computing Center. Our largest computations involve two billion points in 64 dimensions on 32,768 x86 cores and 8 million points in 784 dimensions on 16,384 x86 cores.

I. Introduction

Given a set of N points $\{x_j\}_{j=1}^N \in \mathbb{R}^d$, a *kernel* function $\mathcal{K}(x_i, x_j)$ that defines pairwise *interactions*, and weights $w_j \in \mathbb{R}$, we wish to compute a *kernel sum*:

$$u_i = u(x_i) = \sum_{j=1}^{N} \mathcal{K}(x_i, x_j) w_j, \quad \forall i = 1 \ldots N. \qquad (1)$$

From a linear algebraic viewpoint, kernel summation is equivalent to approximating $u = Kw$ where u an w are N-dimensional vectors and K is a $N \times N$ matrix with $K_{ij} = \mathcal{K}(x_i, x_j)$. The challenge is that for most practical applications, K is a dense matrix and thus computing (1) requires $\Omega(N^2)$ work. Fast kernel summation methods approximate Kw in $\mathcal{O}(N \log N)$ or even $\mathcal{O}(N)$ time.

The need for fast kernel summation methods first appeared in computational physics. There is extensive literature on scalable algorithms and high-performance implementations of fast kernel summation schemes (*e.g.*, treecodes, fast multipole methods, particle-mesh methods, Ewald sums, and others) which can scale to billions or trillions of points for problems in two or three dimensions.

Kernel sums are also fundamental in data analysis, non-parametric statistics and machine learning, for example in kernel density estimation, kernel regression, and kernel classification [11], [15], [23], [25]. However, a key difference between these and typical problems in computational physics is that data analysis problems can be very high dimensional. Algorithms which are efficient for $d = 3$ often do not scale to higher values of d because they depend superlinearly in d. For example, octree-like data structures scale as 2^d, tensor-product analytic series expansions scale as 2^d, and Taylor series expansions scale as d^c where c is the expansion order.

Furthermore, data analysis tasks can have billions of points in hundreds or thousands of dimensions, so parallel scalability of kernel summations is critical in terms of N, cores, and very large values of d.

Contributions: In [22], we introduced ASKIT ("Approximate Skeletonization Kernel-Independent Treecode"), a fast scheme for kernel summations in high dimensions that circumvents the explicit superlinear scaling with dimensionality. Here, we introduce algorithms for its parallelization, a high-performance implementation, and an empirical evaluation. In particular:

- we present an algorithm for the *efficient construction of local essential trees (LETs)* in high dimensions (§II-C);
- we present *distributed memory parallel algorithms* for ASKIT and implement them using MPI (§II-C);
- we present a *theoretical analysis of the complexity* of the algorithm (in terms of time, communication, and storage) (§II-D);
- we discuss *single node performance optimizations as well as a GPU implementation*; (§II-E);
- we present an *empirical evaluation of our scheme* on synthetic and real datasets, including a run with *two billion points in 64 dimensions on 32,768 cores* (§IV).

Our key innovations are a novel nearest-neighbor pruning that enables the use of indexing of the spatial tree (similar to Morton IDs for octrees) and the efficient calculation of the nodes that need to be part of the LET, which in turn dramatically improves scalability. Furthermore, the construction of the off-diagonal low-rank approximations is done using algebraic techniques that are both more accurate and more efficient than existing methods since their construction and evaluation uses BLAS and LAPACK routines. In all, our implementation achieves high arithmetic intensity, algorithmic optimality, and unprecedented scalability.

Outline of methodology: There are three main approaches to approximating (1) for general d: (a) creating sparse approximations using nearest-neighbor approximations [8], (b) using global low-rank approximations [10], [17], [26]; and (c) using hierarchical low-rank approximations of the off-diagonal blocks of K [11]. To make the ideas precise, throughout the paper we will consider a popular kernel in data analysis, the Gaussian kernel

$$K(x_i, x_j) = \exp(-\|x_i - x_j\|_2^2/(2h^2)), \tag{2}$$

where h is the *width* of the kernel. When h is small, K admits a good sparse approximation since for the ith row of $K(x_i, :)$, only the points that are nearest neighbors of x_i have significant contributions. On the other hand when h is large no matrix entries are small but K admits a global low-rank approximation. However, there is a range of values of h for which neither sparse nor global low-rank approximations work well. In cross-validation studies in data analysis, we search for the optimal width h, and we must test a range of values for h that are in this intermediate regime. For such cases, the method of choice is to use a *treecode* (or N-body solvers and related methods like dual-tree and fast multipole methods). They are robust for all values of h as long as K has low-rank off-diagonal blocks.

Treecodes exploit low-rank blocks of the matrix K by constructing compact, approximate representations of them. These blocks are related to the smoothness of the underlying kernel function $\mathcal{K}()$, which in turn is directly related to pairwise similarities between elements of a set, such as distances between points. Hierarchical data structures reveal these low rank blocks by rewriting (1) as

$$u_i = \sum_{j \in \text{Near}(i)} K_{ij} w_j + \sum_{j \in \text{Far}(i)} K_{ij} w_j, \tag{3}$$

where $\text{Near}(i)$ is the set of points x_j whose contributions cannot be approximated well by a low-rank scheme and $\text{Far}(i)$ indicates the set of points whose contributions can.

Roughly speaking, treecode algorithms for (1) can be categorized based on 1) how the $\text{Near}(i)$ and $\text{Far}(i)$ splittings are defined (typically with the help of a tree data structure), and 2) the low-rank representation used to approximate K_{ij} (for $j \in \text{Far}(i)$). Then, to evaluate the sum at a target point, the treecode splits the nodes of the tree into Near and Far sets.[1] The contributions from Near nodes are evaluated directly, and those from Far nodes are approximated from their low-rank representations. We give details in §II-A.

Related work: Classical works on sequential fast kernel summations in low dimensions include [6] and [9], [12]. In higher dimensions, related work includes [11], [13], [18], [19], [24], [30]. For further discussion on the merits of the different algorithms, see [21], [22]. For HPC algorithms in low dimensions, please see [16] and references therein.

Although there is work on tree-construction and nearest neighbors [1], [29] in high dimensions, little has been done for treecode-based kernel summations. To our knowledge, the only work is [20]. There, the authors describe general algorithms for schemes on binary trees and provide impressive scalability results, with the largest run being a synthetic dataset (uniformly distributed points) with one billion points in ten dimensions. They present a complexity analysis for the tree construction phase. They report results on up to 6,144 cores. The tree construction scales well, but the evaluation phase of their algorithm is complex and difficult to load balance. The particular treecode algorithm used does not scale well with the number of dimensions (yet, it does much better than a naive extension of 3D algorithms) and the global tree is replicated on each processor, which limits the algorithm to shallow trees.

Here, we present results for problems with nearly two-orders-of-magnitude higher dimensionality than [20] ($d = 784$ vs $d = 10$) on up to 32,768 cores on the Stampede system. Our algorithm does not replicate the tree, so it scales to arbitrary N and number of cores. It can and has been applied to kernels with variable bandwidth [22].

Limitations: Certain variants of our scheme (for narrow bandwidths or kernels with singularities) require that we provide nearest neighbor information (§II). Constructing these neighbor lists can be quite expensive. However, this is a one-time setup cost and does not depend on the kernel width. It can be amortized across multiple evaluations, for example classification or regression tasks and cross-validation studies. Parts of our algorithm are limited to metric spaces but not in a fundamental way. We are currently working on removing such restrictions. In practice a significant amount (if not the majority) of kernel functions are defined on metric spaces. Scaling for this class of problems is an important step toward overall scalability for kernel methods. In this implementation, we have used GPUs only for the direct interactions, and we have not overlapped communication with computation for the MPI part of the code. Finally, treecodes can be further improved by using dual-tree approaches [18], which resemble fast multipole methods. Such methods have complexity $\mathcal{O}(N)$ as opposed to $\mathcal{O}(N \log N)$ for treecodes.

II. METHODS

We begin with a summary of treecodes in §II-A and introduce the sequential ASKIT in §II-B. The parallel algorithms are introduced in §II-C and the complexity in is discussed in §II-D. The per-node performance optimizations are described in §II-E.

Notation: In the following, α will denote a tree node, $\text{LEVEL}(\alpha)$ its level (with the root at 0), m will denote the points per leaf (used to construct the tree and to determine the size of direct interactions), $\mathcal{A}_\alpha$ will denote the ancestors of α in the tree, $\mathcal{C}_\alpha$ will denote the children of α, $\mathcal{A}_{x_i}$ will be

[1] Fast Multipole Methods involve more complex logic that has complexity $\mathcal{O}(N)$. Our method can be extended to behave like the FMM.

the ancestors of point x_i defined as the ancestors of the leaf that contains x_i, $\mathcal{X}_\alpha$ will denote the set of points assigned to α during the tree construction (Algorithm II.2), w_α their corresponding weights, $\mathcal{S}_\alpha$ will denote the skeleton points of α, (these are points in $\mathcal{X}_\alpha$ that will be used to represent the far field), $\mathcal{N}_i$ will denote the nearest neighbors of points x_i, $\mathcal{N}_\alpha = \cup_{x_i \in \mathcal{X}_\alpha} \mathcal{N}_i$, and s will be the number of skeleton points (5) for each node α. All logarithms are base two.

A. *Treecodes and* ASKIT

In a treecode, we first construct a tree to hierarchically partition the input points. In high dimensions, typically we use a balanced binary tree in which internal nodes have two children. Once the tree is constructed, we perform a bottom-up tree traversal. In this traversal, we create a low-rank approximation $\tilde{K}_\alpha$ of the far field generated by all the source points in each node. In the evaluation phase (Algorithm II.1),

Algorithm II.1 EVAL(Target point x, Tree node α)

```
if PRUNE(x, X_α), return u(x) += K̃_α(x)
elseif (α is a leaf), u(x) += Σ_{x_j∈α} K(x, x_j) w_j
else, u(x) += Σ_{α'} EVAL(x, α'), ∀ α' ∈ C_α
```

for a target point x, we traverse the tree from the top down. At a node α, we apply a *pruning criterion* that determines if the node can be added to the set $\text{Far}(i)$ (3). If it can, then the node is *pruned*. Otherwise, the traversal continues to the children. Leaves which cannot be pruned are added to $\text{Near}(i)$. Together, the pruning criterion and low-rank factorization characterize the treecode.

State-of-the-art treecodes for high dimensional problems [13] use analytic series expansions to construct $\tilde{K}$. At best, these expansions scale as d^c where c indicates the order of accuracy of the approximation [13], and therefore either d or c have to be small. For large d, only $c = 1$ or $c = 2$ are feasible. Such low-order accuracy approximations have adverse implications for the pruning criterion and error control, forcing an excessive number of direct evaluations.

Bounding region-based pruning: Pruning is typically related to the convergence radius of series expansions. The error in the expansions depends on the distance between the target point and a bounding region (box or sphere) enclosing the sources in a node. A node can be pruned (i.e., its far field can be approximated) if the minimum distance between the target and the bounding region is sufficiently large. Bounding region-based pruning presents several difficulties for efficient treecode implementations. First, the traversal pattern is generally not known in advance. This makes it difficult to determine the time required for a tree traversal. In parallel, this can also lead to unbalanced and unpredictable communication requirements. Second, in high dimensions, bounding boxes and balls tend to become less effective at separating groups of points. Due to the concentration of measure effect, the pairwise distances between points will tend to concentrate 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.

B. *ASKIT*

We resolve these problems with a novel two-pronged approach. First, we use a nearest-neighbor based pruning that does not involve node-based bounding regions. Second, we approximate the far-field using a *kernel-independent algebraic* approximate low-rank factorization. Both of these features of ASKIT scale well with the dimensionality of the dataset and the kernel.[2]

Neighbor-based pruning: In [22], we introduced a novel combinatorial pruning condition based on the nearest-neighbor graph of the input points. For each target point x, we compute its κ nearest neighbors $\mathcal{N}_i$. If a node is an ancestor of a point $x_i \in \mathcal{N}_i$, then this node *cannot be pruned* for x, or more formally

$$\text{PRUNE}(x_i, \alpha) = \texttt{true} \text{ iff } \nexists j \in \{i \cup \mathcal{N}_i\} : x_j \in \alpha. \quad (4)$$

This check can be performed in $\mathcal{O}(\kappa)$ time as described in §II-C. The current implementation of ASKIT uses an approximate nearest neighbor search based on random projection trees [7], [29].

In fact, for many kernels that do not have singularities (e.g., the Gaussian), there is *no need* to use neighbor information. Then, the pruning criterion remains the same by setting x_i to be the only entry in $\mathcal{N}_i$, which means that all nodes will be pruned for a point x_i except those in $\mathcal{A}_{x_i}$. This is only possible due to the accurate algebraic low-rank factorization of the far field. If it were to be applied in standard treecodes, it would result in excessive errors because standard high-dimensional treecodes use low-order series expansions. We give such an example in §IV.

Low rank factorization: The low rank factorization used in ASKIT is based on a linear algebraic decomposition of the kernel submatrix corresponding to the interactions due to the sources. First, we describe the process for a leaf node α with m points. Consider the $N \times m$ matrix K of kernel interactions between all points and points in α.

We use the *interpolative decomposition* (ID) [14] of K. Using pivoted QR factorization, we compute a rank-s matrix K_{col} such that $K \approx K_{\text{col}} P$ where $P \in \mathbb{R}^{s \times m}$ contains the $s \times s$ identity matrix as a submatrix and K_{col} consists of s columns of K. That is, the ID represents K as a linear combination of a subset of its columns. The error is related to the size of the $(s+1)^{\text{st}}$ singular value of K [5].

Given an ID of K, we can represent this low rank factorization using $2sd$ storage per node. We refer to the

[2] In fact, the effectiveness of ASKIT depends only on the *intrinsic* dimensionality of the dataset as opposed to standard treecodes which depend on the *ambient* dimensionality [22].

columns selected in K_{col} as the *skeleton*, which we denote $\mathcal{S}_\alpha$. Then we can represent the far-field of node α using s *skeleton weights* $\tilde{w}_\alpha$ by

$$\tilde{u}(x) = K(x, \mathcal{S}_\alpha)\tilde{w}_\alpha, \quad \tilde{w}_\alpha = P w_\alpha. \tag{5}$$

For an internal node α, we form the ID of the merged skeletons of $\mathcal{C}_\alpha$ and form $2s$ columns. We then compute an ID of the matrix with these columns and choose s of them to form the skeleton of α.

The ID can be computed from the pivoted QR factorization of K. However, this requires $O(Nm^2)$ work, which is more expensive than computing all of the interactions directly. Therefore, we must approximate the ID. We do this by subsampling ℓ rows of K to form an $\ell \times m$ matrix K' and approximating the ID of K with the ID of K'. Sampling

Algorithm II.2 SKELETONIZE(α)

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

plays an important role as it needs to be accurate and fast. We drew from the rich literature in randomized matrix approximations [14] and designed a new sampling scheme for ASKIT [21]. In a nutshell, we sample points in α, the nearest neighbor list $\mathcal{N}_\alpha$, and points sampled uniformly at random from $\mathcal{X}$. Intuitively, the self-interactions and the nearest neighbors will make the largest contributions to K. Thus, we expect them to accurately capture the row space. We use a parameter ℓ to determine the number of samples needed. If $|\alpha| > \ell$, then we randomly choose ℓ points from α. Otherwise, we take the $\ell - |\alpha|$ closest points in $\mathcal{N}_\alpha$. We collect a set of uniform samples $\mathcal{U}$ if $|\alpha \cup \mathcal{N}_\alpha| < \ell$. Fast convergence of the scheme can be shown for $\ell = s + 20$ [21]. The overall algorithm for constructing the far-field low-rank decomposition is summarized in Algorithm II.2, it uses pivoted QR factorization, and its output is the skeleton points and skeleton weights at every node.

Algorithm II.3 FORMLISTS(Point x, Node α)

1: `if` PRUNE(x, α): $\quad \mathrm{Far}(x) \mathrel{+}= \alpha$
2: `else if` ISLEAF(α): $\quad \mathrm{Near}(x) \mathrel{+}= \alpha$
3: `else`: $\quad$ FORMLISTS($x, \mathtt{l}(\alpha)$); FORMLISTS($x, \mathtt{r}(\alpha)$)

Interaction lists: Let us now introduce the first improvement to ASKIT. In a standard treecode implementation, when a node is assigned to a set Near or Far in (3), the exact or approximate interaction of that node is immediately

Algorithm II.4 u=ASKIT($\mathcal{X}, w, s, \ell, \alpha_{\mathrm{root}}, \mathcal{N}(\mathcal{X})$)

1: SKELETONIZE(α_{root})
2: `for all` $i \in \mathcal{X}$
3: $\quad \{\mathrm{Near}(x_i), \mathrm{Far}(x_i)\}$ = FORMLISTS($x_i, \alpha_{\mathrm{root}}$)
4: Invert target lists to obtain lists $\{\mathrm{Near}(\alpha), \mathrm{Far}(\alpha)\}$
5: `for all` leaf nodes α
6: $\quad u(\mathrm{Near}(\alpha)) \mathrel{+}= K(\mathrm{Near}(\alpha), \mathcal{X}_\alpha) w_\alpha$
7: `for all` nodes α
8: $\quad u(\mathrm{Far}(\alpha)) \mathrel{+}= K(\mathrm{Far}(\alpha), \mathcal{S}_\alpha)\tilde{w}_\alpha$

calculated at the target. This is what we used in [22]. In this paper, we use a different approach. We first collect the near and far-field computations for each node, then perform these calculations in a batched fashion. This allows more efficient use of kernel calculations and linear algebra routines and improved overall arithmetic intensity.

First, we create target interaction lists for each target point x. We store lists $\mathrm{Near}(x)$ (the leaf nodes for which x could not prune) and $\mathrm{Far}(x)$ (the nodes for which x could prune). After we have constructed these lists for each target, we invert them. We compute lists $\mathrm{Near}(\alpha)$ for each leaf node and $\mathrm{Far}(\alpha)$ for each node. These lists contain all targets x which have α in $\mathrm{Near}(x)$ or $\mathrm{Far}(x)$. We then iterate over each of these lists and use a `DGEMM` operation instead of a `DGEMV` operation since the far or near field of α is applied to multiple targets simultaneously. The overall evaluation algorithm for ASKIT is summarized in Algorithm II.4. Its structure is similar to the standard treecode of Algorithm II.1, but by working with the lists we achieve better performance as we show in §II-E.

Accuracy and efficiency: The error will not be discussed here due to space limitations—see [22] for details. In a nutshell, the relative error up to small (but not constant) prefactors is proportional to $\max_\alpha(\sigma_{s+1}(\alpha)/\sigma_1(\alpha))$ where σ_i is the ith largest singular value of the far-field block of a leaf node α. This error is related to the properties of the kernel and the distance to the $(\kappa+1)^{\mathrm{st}}$ nearest neighbor of a target point. These quantities are computable and can be used (locally) to give a-priori error estimates and determine the appropriate values of s and κ.

Overall, the efficiency of ASKIT depends on the percentage of interactions that can be pruned in a given dataset. As discussed above, the pruning depends on κ, with smaller values of κ leading to more aggressive pruning. However, this will require s to be large enough for $\sigma_{s+1}(\alpha)$ to be small. If the dataset is truly high-dimensional and the bandwidth is within a certain rage, the off-diagonal blocks of K will not be low rank [21]. In this case, an accurate approximation of the kernel sum is inherently difficult. We assume that this is not the case and the matrix can be compressed. In practice, this can be checked numerically during the skeletonization phase. The overall complexity

of the algorithm will be presented after we discuss the parallelization of ASKIT.

C. *Parallel* ASKIT

ASKIT consists of four major steps: first we construct the tree, then we skeletonize, then we construct the local essential tree (LET), and finally we evaluate the potential at the targets. We now describe parallel algorithms for each of these steps. Without loss of generality, we assume that m, N, p are powers of two. Given an MPI process and target points assigned to it, the LET is the portion of the global tree (leaves, internal nodes and skeleton data) that is required to evaluate the potential at the targets. Once the process has its LET, the evaluation involves no additional communication. The LET can be constructed before evaluation using octrees in 3D treecodes [16], [28]. In existing high-dimensional treecodes, the LET cannot be computed without doing the actual evaluation.

Tree construction: For the tree construction, we use a binary tree scheme that we presented in [29]. We summarize it here for completeness. The overall tree structure among all processors is illustrated in Figure 1. Each MPI process stores a portion of the overall tree (the union of the ancestors its leaves). Each tree node α with level $\log p$ or smaller has an MPI communicator $\texttt{Comm}_\alpha$ and which is shared across multiple processes. Whenever we split a node, we split the communicator and we assign the new values to each child. We use a distributed top-down algorithm to construct the distributed tree. If α has more than m points, we split it (by projecting points on a line and then using their median) and we redistribute the points to $\mathcal{C}_\alpha$. Each MPI process maintains a local data structure for the tree node containing its MPI communicator and hyperplane splitting information including the projection direction and median. In each process, these data structures are stored in a doubly-linked list representing a path from the root to a node. Splitting and redistributing points between communicators can introduce load imbalance at an individual process. We repartition points to guarantee load balancing with respect the number of points per leaf node. For nodes at a level deeper than $\log p$, the construction is local and we use shared memory parallelism. For details, see [29].

Morton IDs: Our parallel implementation of ASKIT requires a global way to index the nodes. Each node α is identified by a *Morton ID* that is determined by the path from the root: the ith digit of the Morton ID is set to be 1 if the ith node on this path is on the right subtree or 0 if on the left subtree. The highlighted path in Figure 1 shows an example of the Morton ID of a leaf node. We define the Morton ID of a point to be the ID of the leaf node that owns the point. Upon the tree construction, every point is uniquely assigned to an MPI process and every point and every node have been assigned a Morton ID. The number of points per process and leaf is perfectly balanced across processes.

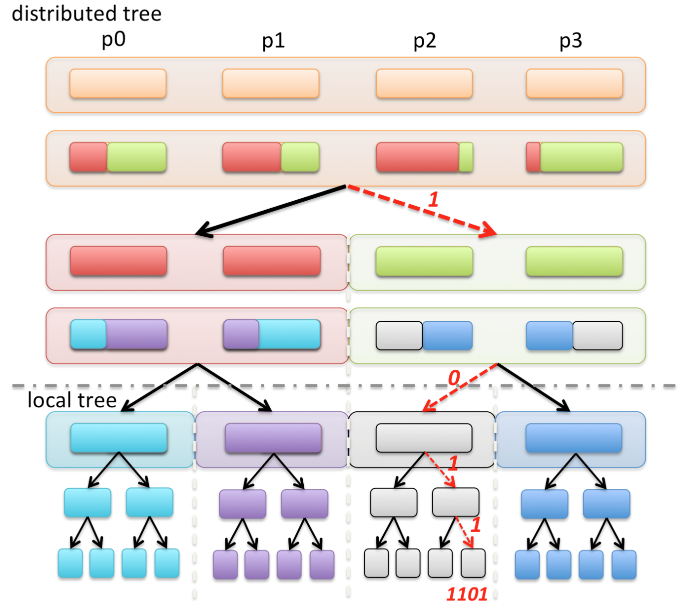

Figure 1: Recursive tree construction and the corresponding communicators. *An illustration of the splitting of communicators used for point redistribution among children. The first* $\log p$ *levels form the distributed tree. When splitting nodes, we first partition the points, then redistribute them for load balancing. All nodes at a given level share an MPI communicator, marked with a shaded rectangle. The highlighted path illustrates how the Morton ID of a leaf node is defined.*

Skeletonization: Up to $\log p$ levels from the root, the skeletonization is done in a level-by-level manner using OpenMP on a single MPI process. When we reach the $\log(p)$-th level in the upward pass, the distributed tree starts and the skeletonization proceeds as a reduction. Every node collects skeleton points and skeleton weights from its children. The only non-standard component here is the uniform sampling (Algorithm II.2). If needed, we uniformly sample a few points outside the parent node to add them into the neighbors. Once we collect points all skeletonization is performed locally.

Local essential tree: An example of a LET is given in Figure 2. Let x be a target point and assume its nearest neighbors are located on the red leaves. Using (4) to evaluate the kernel summation, only those three red nodes (direct evaluations), their siblings (skeletons) and five black internal nodes (skeletons) are required to evaluate $u(x)$. ASKIT does not need to visit any other nodes. This process is repeated (logically) for each target point and the union of all these nodes defines the LET.

Given the Morton IDs of a node α and a point x we can check whether $\alpha \in \mathcal{A}_x$ by comparing the first $\text{LEVEL}(\alpha)$ bits of their Morton IDs. If they are the same, α is an ancestor of x and the siblings of α can be pruned. Using these arguments, the LET construction can be formalized as follows. Let α be the root node of an MPI process so that $\text{LEVEL}(\alpha) = \log p$. Let $\mathcal{L}$ be the set of all the leaves that contain points in $\mathcal{N}_\alpha \backslash \mathcal{X}_\alpha$. No nodes in $\mathcal{A}_\mathcal{L}$ can be pruned,

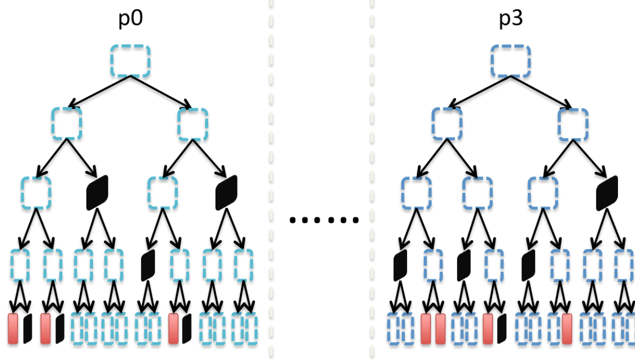

Figure 2: Illustration of local essential tree (LET). *Both left and right images are examples of LETs on different processes. Each process has its own local tree structure and only the necessary tree nodes are used. The LET on rank 0 gives an example of the pruning. Given a target point suppose all its nearest neighbors are located at the three red leaf nodes. Using* (6)*, the process collects the nodes in* $\text{SIBLINGS}(\mathcal{A}_\mathcal{L})\backslash\mathcal{A}_\mathcal{L}$ *(marked in black). On the right other processes (*e.g. *p3) also build their LETs.*

however any siblings of points in $\mathcal{A}_\mathcal{L}$ that are not themselves in $\mathcal{A}_\mathcal{L}$ can be pruned. More formally, we have:

$$\text{LET} = \mathcal{L} \cup (\text{SIBLINGS}(\mathcal{A}_\mathcal{L})\backslash\mathcal{A}_\mathcal{L}) . \qquad (6)$$

We must still be able to compute the Morton IDs of points in $\mathcal{N}_\alpha\backslash\mathcal{X}_\alpha$. Upon initialization (before the tree construction), every point is assigned to a process. We keep this information as we construct the tree. Upon tree construction, we send the Morton IDs of the $\mathcal{X}_\alpha$ to their original owner processes. Then we collect the Morton IDs of the nearest neighbors from the original processes.

Once we have the Morton IDs, (6) takes $\mathcal{O}(\kappa \log(N/p)N/p)$ time. For every node, we know its MPI rank due to our construction. This allows us to organize an all-to-all task to collect the LET skeleton and leaf information and insert it to the local tree. Then, the evaluation can proceed without any further communication using Algorithm II.4.

D. Complexity analysis

In [22], we derived complexity estimates and error estimates for sequential ASKIT. We summarize the leading terms for practical values of κ (the number of nearest neighbors), m (the points per leaf), and s (the skeleton size), which are the main parameters in ASKIT. The time to skeletonize is $\mathcal{O}\left(dsN(m + s^2/m)\right)$. The time to evaluate at N targets is bounded by $\mathcal{O}\left(dNm\,\kappa + dNs\,\kappa \log\left(\frac{N}{m}\right)\right)$, where the first term indicates direct evaluations and the second the far-field evaluations.

For the parallel complexity, we define t_s to be the latency and t_w to be the reciprocal of the bandwidth. The complexities below assume a hypercube topology for the interconnect.

The parallel complexity of the **tree construction** is $\mathcal{O}((t_s+t_w)\log^2 p \log N + (1+t_w \log p)N(d+\kappa)/p)$ for time and $\mathcal{O}((d+\kappa)N/p)$ for storage since we have N/p points per process and $\mathcal{O}(d+\kappa)$ data per point. The construction uses parallel sort, selection, median, and load balancing. This result has been derived in [29] for the nearest neighbor search. We do not discuss nearest neighbor search cost further since it an input to ASKIT (any method can be used), and can be amortized across multiple kernels and multiple evaluations per kernel. Upon completion of the construction, the worst-case storage per MPI process is $\mathcal{O}(d\kappa N/p)$.

The communication during the **skeletonization** resembles a reduction primitive, where the dominant term in message size in the reduction is equal to the skeleton information, which is $\mathcal{O}(ds)$. Therefore the worst case complexity of the communication is $\mathcal{O}\left((t_s + t_w)sd \log p\right)$ time. The worst-case additional storage due to skeletonization is $\mathcal{O}\left(ds(N/p + \log p)\right)$ since one node will store all its local skeletons plus all the skeletons from its local root to the global root.

The **LET construction** communication involves all-to-all exchanges to obtain Morton IDs of nearest neighbors, near interactions, and far interactions. The worst-case near interactions are κmd per target and thus $d\kappa mN/p$ per MPI rank. The far-field interactions are bounded by $\mathcal{O}(d\kappa s \log(N/m))$ per target. In the worst case, all interactions of all targets in the node require communication, requiring an all-to-all call with cost $\mathcal{O}\left(t_s p + t_w d\kappa(m + s\log(N/m))N/p\right)$. The total LET storage is $\mathcal{O}\left(d\kappa(m + \log(N/m)s)N/p\right)$.

The evaluation phase has no communication or additional storage and costs $\mathcal{O}\left(d\kappa(m + \log(N/m)s)N/p\right)$ time. Next, we discuss performance optimizations to get good FLOP efficiency for the Gaussian kernel.

E. Performance optimizations for the Gaussian kernel

Given the interaction lists Near and Far for a treenode α in Algorithm II.4, we can compute its near- and far-field contributions concurrently.

The GPU implementation takes advantage of the expansion $\|x_i - x_j\|_2^2 = \|x_i\|_2^2 + \|x_j\|_2^2 - 2x_i^T x_j$. Precomputing $\|x_i\|_2^2$ and reusing the result requires many fewer FMA operations than evaluating $\|x_i - x_j\|_2^2$ directly. Moreover, $-2x_i^T x_j$ and $u_i = \sum_{j\in\alpha} K_{ij} w_j$ can be computed by `DGEMM` and `DGEMV` which provides excellent performance on both the CPU and GPU architectures.

We first compute $-2x_i^T x_j$ for each pair of points using `DGEMM`. We store these quantities, and use them to evaluate the kernel matrix. Finally, we use `DGEMV` to apply the kernel matrix to the vector of weights. The only limitation of this BLAS approach is that every memory space has to be contiguous. Therefore, we require additional buffers in order to extract the point point coordinates and weights and to collect u. In addition to these four dense buffers, we require a $|\text{Near}(\alpha)|\max\{m, s\}$-sized buffer to store the intermediate result between `DGEMM` and `DGEMV` The extra storage requires extra memory accesses which may lead to significant device memory overhead.

The ideal way to implement this algorithm on the GPU is to embed `DGEMM` and `DGEMV` inside a GPU kernel by

$m = 256$				
	Embedded (ms)	Gflops	BLAS (ms)	Gflops
$d = 3$	1.0	329	1.5	218
$d = 64$	1.8	602	2.3	475
$d = 128$	2.9	667	3.1	605
$d = 1024$	19.2	686	14.7	880
$m = 1024$				
	Embedded (ms)	Gflops	BLAS (ms)	Gflops
$d = 3$	4.2	330	6.0	231
$d = 64$	7.1	628	8.5	524
$d = 128$	11.4	675	11.2	685
$d = 1024$	75.8	696	51.3	1029

Table I: *Runtime and Gflops comparison between the embedded and the BLAS kernel with different m, d and fixed $|\text{Near}(\alpha)| = 24,576$. All experiments are performed on a single compute node of the Maverick system at TACC in double precision.*

fetching x_i directly and storing $x_i^T x_j$ in the register. We implement a embedded kernel in CUDA which realizes this idea, and we show that even with a suboptimal `DGEMM` implementation, the embedded kernel is still able to reduce the runtime by reducing memory access and mixing different instructions. To illustrate the idea of implementing the kernel, we briefly summarize the parallel scheme used in CUDA. Our kernel computes $-2x_i^T x_j$ through a sequence of updates by iterating over d. A 16×16-thread block is mapped to a 64×64 submatrix of K, and in each iteration an 8×64 panel of targets and sources will be loaded to the shared memory buffers. Each thread computes a 4×4 update by reusing the panel in the shared memory, accumulating the value in a 4×4 register.

Table I compares the embedded kernel with the BLAS kernel for different dimensions and sizes. For values of d up to approximately 64, the embedded kernel results in improved GPU performance. This kernel has fewer device memory accesses and concurrently computes the update with the kernel evaluation. Kernels such as the Gaussian require SFUs (special function units) on the GPU which are separated from the double precision ALUs. Mixing updates with kernel evaluations activates both SFUs and ALUs at the same time, which provides a higher computing throughput. However, lacking optimization on the update pipeline, the embedded kernel loses its benefit once d is big enough for `DGEMM` to reach its steady state. To summarize, the near-field evaluation is accelerated using concurrent CPU-GPU execution. The embedded kernel helps boost the GPU performance in $d \leq 64$ by reducing memory access and mixing ALU and SFU operations. For larger d, the overhead is amortized, and the BLAS kernel performs better. While our implementation only accelerates the near-field the exact same algorithm can also be applied to the far-field.

III. Experimental setup

In this section, we give details on the experimental setup we used to test our methods.

Data Sets: We use both real-world and synthetic data sets to test our algorithm. The properties of the real-world sets are given in Table II. The `mnist8m` set consists of features from a handwritten digit recognition problem obtained from [4]. The `mnist2m` set consists of a subsample of the 8 million point set. The `susy` set is derived from Monte Carlo simulations in a particle physics context [3] and is available from the UCI Machine Learning Repository [2]. We also show results on synthetic data generated from a standard multivariate normal distribution for several values of d.

Set	N	d
`mnist2m`	2×10^6	784
`mnist8m`	8.1×10^6	784
`susy`	5×10^6	18

Table II: *The properties of the data sets used in our experiments.*

Implementation and Hardware: Our code is implemented in C++ and uses the Message Passing Interface (MPI) and OpenMP library for multithreading. The code is compiled with the Intel C++ compiler using the `-O3` flag. We carry out runtime experiments on the Stampede system at TACC, a cluster with 6,400 compute nodes, each with dual eight-core processors. The dual-CPUs in each host are Intel Xeon E5 (Sandy Bridge) processors running at 2.7GHz with 2GB/core of memory and a three-level cache. Stampede has a 56Gb/s FDR Mellanox InfiniBand network connected in a fat tree configuration which carries all high-speed traffic (including both MPI and parallel file-system data). We also show results on the Maverick system at TACC. Each compute node contains dual, ten-core processors. Each is an Intel Xeon E5 (Ivy Bridge) running at 2.8GHz with 12.8Gb/core of memory. Each node also has an NVIDIA Tesla K40. The extra memory allows us to explore the performance of more memory-intensive parameter settings for ASKIT, such as large values of κ.

Gaussian kernel parameters: The performance of kernel matrix-vector product approximation methods depends strongly on the choice of kernel bandwidth [21], [22]. For very large values of h, the kernel matrix approaches a scaled identity matrix. For small h, the matrix approaches a rank one matrix. However, as our complexity analysis in §II-D shows, the runtime and parallel efficiency of ASKIT does not depend on h. For our synthetic data sets, we use the asymptotically optimal bandwidth for kernel density estimation [27]. For the real data sets, we explore a range of h around values chosen for regression using cross-validation.

ASKIT *parameters:* We explore the performance over different values for the three main parameters in ASKIT: the number of points per leaf node m, the rank of the skeleton s, and the number of nearest neighbors per point κ, as discussed in §II-D. Due to space constraints, we present a small subset of results on a variety of parameters. In particular, we discuss the tradeoff between far-field and near-field computations as κ increases. For large values of κ, less pruning will occur (4). This will require more work for the near-field computation. On the other hand, for small

values of κ, more of the work will occur in the far-field computations.

IV. Results

We report three sets of results. In §IV-A, we discuss the impact of our optimizations on the single-node performance of ASKIT. We describe the performance gains due to our interaction list implementation and the use of the GPU for the evaluation of near-field interactions. In §IV-B, we report results that combine our single-node optimizations and GPU implementation in a distributed setting. Finally, in §IV-C, we show strong and weak scaling results for ASKIT on thousands of cores (without using GPUs).

We divide the wall clock timings for ASKIT into several parts, corresponding to parts of Algorithm II.4. We report the time to construct skeletons (line 1) as T_{skel}, the time to form and invert the interaction lists (lines 2 to 4) as T_{list}, the time to compute the near-field interactions (line 5) as T_{near}, and the time to compute the far-field (line 7) as T_{far}. We report the total time as T. In all our experiments, we do not report the time to compute the nearest neighbors. These are inputs to ASKIT and can be amortized across multiple kernel parameters and evaluations.

A. Node Optimizations

We present four implementations of our algorithm: 1) The original version of ASKIT [22] (with no interaction lists), 2) ASKIT with interaction lists so that near and far field interactions are batched (Alg. II.4), 3) ASKIT with interaction lists and near field computations performed on the GPU, and 4) ASKIT with interaction lists, near field on the GPU, and near and far field computations overlapped. We show two data sets with $N = 10^5$ Gaussian points and $d = 4$ and 256 in Table III. These experiments use a large value of κ (=512), so the near field interactions are dominant. However, we see that when we compute the near-field on the GPU, the runtimes are significantly less than the time for the far-field interactions. We see that we obtain the largest speedup from the interaction list implementations. As d increases, the speedups from both the interaction lists and the GPU increase. Additionally, when we overlap the GPU computation of the near-field and the CPU computation of the far-field, we see that the near-field time is almost completely hidden.

B. Effects of κ and h

We show results for ASKIT on two real-world data sets for a variety of parameters in Table IV. The runs use the optimized version of ASKIT with interaction lists (Algorithm II.4). These runs perform both the near and far field interactions on the CPU. We report the relative error of our approximate matrix-vector multiplication algorithms. Since the true error is prohibitively expensive to compute, we sample $n = 1,000$ random targets and report $\epsilon = 1/n \sum_{i=1}^{n} \left| \frac{u(i) - \tilde{u}(i)}{u(i)} \right|$, where u is computed exactly and $\tilde{u}$ is computed by ASKIT.

Alg	T_{skel}	T_{list}	T_{near}	T_{far}	T
4D Normal distribution, $N = 10^5$, $h = 0.21$					
Original	0.8	–	–	–	7.1
Lists	0.7	1.3	0.4	1.1	3.5
GPU	0.7	1.3	0.1	1.2	3.3
Overlap	0.7	1.3	0.1	1.2	3.2
256D Normal distribution, $N = 10^5$, $h = 0.93$					
Original	2.6	–	–	–	207.5
Lists	2.3	4.0	14.4	7.3	28.0
GPU	2.4	4.1	5.8	7.3	19.6
Overlap	2.4	4.0	5.8	7.3	13.7

Table III: ***Single node optimizations*** *for* ASKIT. *We show the timings for four implementations of* ASKIT *listed in §IV-A. For each experiment, we use $m = 256$, $s = 128$, $\ell = 148$, and $\kappa = 512$. All times are in seconds. All experiments were performed on a single node of Maverick with 20 OpenMP threads in double precision. For the 256D problem, the new version is over 15× faster than the original.*

In each of the `mnist2m` experiments, we achieve 10% relative error. In the first set, this is done by increasing κ, thus increasing the number of exact near-field calculations. In the second set, we choose $\kappa = 1$ and instead increase the approximation rank s. We note that either an increased number of exact evaluations or a higher rank approximation is necessary to achieve good error for these problems. For instance, we ran a computation on `mnist2m` with $s = 1$ (to imitate a low-order treecode) and $\kappa = 1$, which has a relative error $\epsilon = 91\%$. These examples highlight the advantage of the variable rank approximations in ASKIT. Without them, a treecode would have to incur a large number of direct interactions (and thus poor speedups), along with the resulting difficulties with load balancing and communication.

We also discuss results for varying bandwidths on the `susy` data set. For $h = 0.03$, the total interactions tend to be dominated by nearby points. On the other hand, for $h = 0.15$, the entire matrix K and its off-diagonal blocks tend to be difficult to compress. This is reflected in the relative error we are able to achieve for each set. With $s = 64$, we can approximate potentials to within 10^{-4} for $h = 0.03$. Despite increasing s by a factor of eight for $h = 0.15$, we are only able to achieve 10% error. This is an example of the inherent difficulty (for some kernels) of using low-rank approximations for off-diagonal blocks [21].

C. Scaling

We now turn to our strong and weak scaling results in Figure 3 and Figure 4. We break the execution time for ASKIT into three parts. We display the time to construct the tree as `Tree` and the time to compute a skeleton for each node and communicate the LET as `Skel + LET`. The time to evaluate the approximate potentials, both near and far field, as well as the time to construct interaction lists, is given as `List + Eval`. The total execution time is the sum of these three times. We also display the parallel efficiency of each computation, using the total time and the smallest run as the baseline. Although the tree construction is not the point of the paper, we included it to give the full picture of

cores	T_{skel}	T_{list}	T_{let}	T_{near}	T_{far}	T
mnist2m $m = 512, s = 128, \kappa = 64, h = 4, \epsilon =$1E-01						
20	18	11	0	109	592	750
320	5	<1	24	23	16	68
640	3	<1	22	11	8	44
mnist2m $m = 512, s = 256, \kappa = 1, h = 4, \epsilon =$1E-01						
20	16	2	0	6	389	413
320	7	<1	<1	3	10	20
640	4	<1	<1	1	4	10
susy $m = 256, s = 64, \kappa = 256, h = 0.03, \epsilon =$ 1E-04						
20	31	139	0	38	268	476
320	11	7	36	14	9	78
640	5	3	19	7	5	39
susy $m = 512, s = 512, \kappa = 64, h = 0.15, \epsilon =$ 1E-01						
20	84	35	0	24	910	1053
320	35	1	7	10	30	83
640	18	1	5	5	15	43

Table IV: ***Runtimes for ASKIT*** *on the* susy *and* mnist *data and varying numbers of cores. All experiments are performed on Maverick. The distributed (p = 320 and 640) runs use 2 MPI ranks per node and 10 OpenMP threads per rank. The 20-core experiments use a single MPI rank on one node. All times are in seconds. The superlinear speedup in the second set of* mnist2m *experiments is due to the improved performance from restricting OpenMP threads to a single socket.*

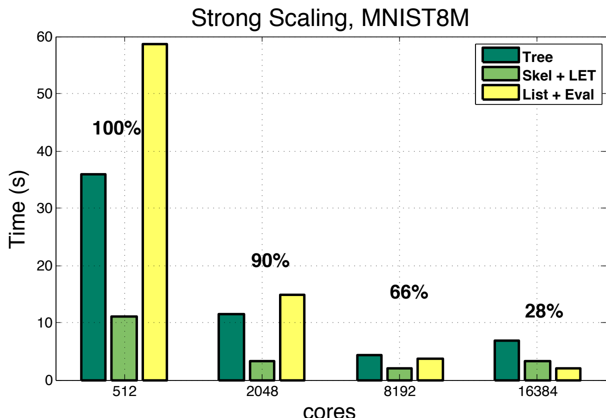

Figure 3: ***Strong scaling results*** *on the* mnist8m *data set with $d = 784$. The total time for each run is the sum of the three columns. We report the parallel efficiency (based on the total times) above each group of cores. The total runtimes are 106s for the 512-core run and 12s for the 16K-core run. All experiments were performed on Stampede.*

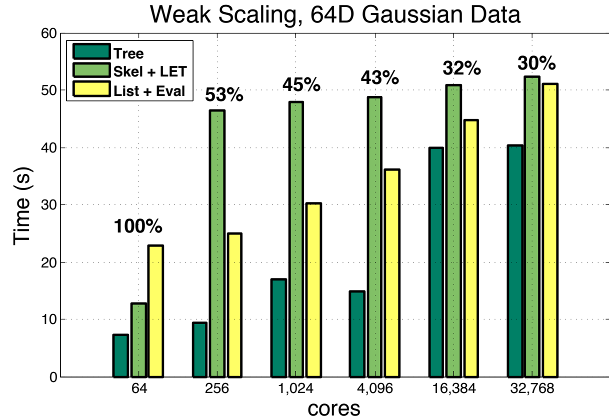

Figure 4: ***Weak scaling results*** *on Gaussian data with $d = 64$ and 31,250 points per core. The total time for each run is the sum of the three columns. We display the parallel efficiency (based on the total times) above each group of cores. The total runtimes are 43s for the 64-core run and 144s for the 32K-core run. All experiments were performed on Stampede. The run with 32K cores uses a data set of over one billion points.*

the performance of ASKIT. The errors for all these runs are below 5%.

For our strong scaling results, we use the mnist8m set, with 8.1 million points and $d = 784$. We use $s = 128$, $m = 1024$, and $\kappa = 1$. The kernel uses bandwidth $h = 4$. For up to $8,192$ cores, we see 66% parallel efficiency. After this point, we see a performance drop-off as the amount of work per core becomes too small.

In Figure 4, we show weak scaling results on randomly generated data from the standard multivariate normal distribution with $d = 64$. We set $s = 256$, $m = 256$, and $\kappa = 1$. We set $h = 1$ (which corresponds to a value for which kernel truncation is too inaccurate and global low-rank approximation too expensive). Observe that the runtime and communication of ASKIT do not depend on h.

From 64 to 256 cores, the time for skeleton and LET construction jumps because MPI kicks in. In our implementation, we do not construct skeletons for the top two levels of the tree. Since the 64 core case corresponds to a distributed tree with only two levels, there is no communication involved in the skeletonization step. Note that the skeletonization and LET time is relatively flat for 256 to 32K cores. We also run a 32K-core test two billion points, our largest run, and the communication scales similarly.

The evaluation cost increases since we use a $\mathcal{O}(N \log N)$ algorithm. Also, for our very largest runs, the parallel implementation suffers from our relatively unoptimized interaction list construction algorithm. For instance, in the weak scaling results, the far-field computation (which is much larger than near-field since κ is small) accounts for 21s and 39s in the 64 and 32K core runs, respectively. For the same runs, the interaction list construction accounts for 1s and 9s. Clearly, our scaling numbers can be improved substantially by optimizing this portion of the code. But fundamentally ASKIT scales well with d, N, and p.

V. Conclusions

We have described parallel algorithms for a new high-dimensional treecode and evaluated its empirical performance on synthetic and real-world data. These experiments include scaling to over 32K cores and calculations on very large data sets, results that are well beyond the capacity of existing kernel summation algorithms.

Our future work focuses on further optimizations of our implementation. In particular, there are opportunities to overlap the computation of some near and far field interactions with the communication of the LET. Also, we plan to explore the computation of both near and far field interactions on the GPU and implement a dual-tree/fast multipole variant.

ACKNOWLEDGMENTS

This material is based upon work supported by AFOSR grants FA9550-12-10484 and FA9550-11-10339; and NSF grants CCF-1337393, OCI-1029022; and by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Applied Mathematics program under Award Numbers DE-SC0010518, DE-SC0009286, and DE- FG02-08ER2585; and by the Technische Universität München - Institute for Advanced Study, funded by the German Excellence Initiative (and the European Union Seventh Framework Programme under grant agreement 291763). Any opinions, findings, and conclusions or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the AFOSR or the NSF. Computing time on the Texas Advanced Computing Centers Stampede system was provided by an allocation from TACC and the NSF.

REFERENCES

[1] I. AL-FURAJH, S. ALURU, S. GOIL, AND S. RANKA, *Parallel construction of multidimensional binary search trees*, Parallel and Distributed Systems, IEEE Transactions on, 11 (2000), pp. 136–148.

[2] K. BACHE AND M. LICHMAN, *UCI machine learning repository*, 2013.

[3] P. BALDI, P. SADOWSKI, AND D. WHITESON, *Searching for exotic particles in high-energy physics with deep learning*, Nature communications, 5 (2014).

[4] C.-C. CHANG AND C.-J. LIN, *LIBSVM: A library for support vector machines*, ACM Transactions on Intelligent Systems and Technology, 2 (2011), pp. 27:1–27:27. Software available at http://www.csie.ntu.edu.tw/~cjlin/libsvm.

[5] H. CHENG, Z. GIMBUTAS, P.-G. MARTINSSON, AND V. ROKHLIN, *On the compression of low rank matrices*, SIAM Journal on Scientific Computing, 26 (2005), pp. 1389–1404.

[6] H. CHENG, L. GREENGARD, AND V. ROKHLIN, *A fast adaptive multipole algorithm in three dimensions*, Journal of Computational Physics, 155 (1999), pp. 468–498.

[7] 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.

[8] J. FRIEDMAN, T. HASTIE, AND R. TIBSHIRANI, *The elements of statistical learning*, vol. 1, Springer Series in Statistics, 2001.

[9] Z. GIMBUTAS AND F. ROKHLIN, *A generalized fast mulipole method for nonoscillatory kernels*, SIAM Journal on Scientific Computing, 24 (2002), pp. 796–817.

[10] A. GITTENS AND M. MAHONEY, *Revisiting the Nystrom method for improved large-scale machine learning*, in Proceedings of the 30th International Conference on Machine Learning (ICML-13), 2013, pp. 567–575.

[11] A. GRAY AND A. MOORE, *N-body problems in statistical learning*, Advances in neural information processing systems, (2001), pp. 521–527.

[12] L. GREENGARD AND J. STRAIN, *The fast Gauss transform*, SIAM Journal on Scientific and Statistical Computing, 12 (1991), pp. 79–94.

[13] M. GRIEBEL AND D. WISSEL, *Fast approximation of the discrete Gauss transform in higher dimensions*, Journal of Scientific Computing, 55 (2013), pp. 149–172.

[14] N. HALKO, P. MARTINSSON, AND J. TROPP, *Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions*, SIAM Review, 53 (2011), p. 217.

[15] T. HOFMANN, B. SCHÖLKOPF, AND A. J. SMOLA, *Kernel methods in machine learning*, The annals of statistics, (2008), pp. 1171–1220.

[16] I. LASHUK, A. CHANDRAMOWLISHWARAN, H. LANGSTON, T.-A. NGUYEN, R. SAMPATH, A. SHRINGARPURE, R. VUDUC, L. YING, D. ZORIN, AND G. BIROS, *A massively parallel adaptive fast multipole method on heterogeneous architectures*, Communications of the ACM, 55 (2012), pp. 101–109.

[17] Q. LE, T. SARLÓS, AND A. SMOLA, *Fastfood: approximating kernel expansions in loglinear time*, in Proceedings of the international conference on machine learning, 2013.

[18] D. LEE, A. GRAY, AND A. MOORE, *Dual-tree fast gauss transforms*, Advances in Neural Information Processing Systems, 18 (2006), p. 747.

[19] D. LEE AND A. G. GRAY, *Fast high-dimensional kernel summations using the monte carlo multipole method.*, in NIPS, 2008, pp. 929–936.

[20] D. LEE, P. SAO, R. VUDUC, AND A. G. GRAY, *A distributed kernel summation framework for general-dimension machine learning*, Statistical Analysis and Data Mining, (2013).

[21] W. B. MARCH AND G. BIROS, *Far-field compression for fast kernel summation methods in high dimensions*, arXiv preprint, (2014), pp. 1–43. arxiv.org/abs/1409.2802v1.

[22] W. B. MARCH, B. XIAO, AND G. BIROS, *ASKIT: Approximate skeletonization kernel-independent treecode in high dimensions*, arXiv preprint, (2014), pp. 1–22. http://arxiv.org/abs/1410.0260.

[23] S. MIKA, B. SCHÖLKOPF, A. J. SMOLA, K.-R. MÜLLER, M. SCHOLZ, AND G. RÄTSCH, *Kernel pca and de-noising in feature spaces.*, in Neural Information Processing Systems, vol. 11, 1998, pp. 536–542.

[24] V. I. MORARIU, B. V. SRINIVASAN, V. C. RAYKAR, R. DURAISWAMI, AND L. S. DAVIS, *Automatic online tuning for fast Gaussian summation.*, in NIPS, 2008, pp. 1113–1120.

[25] B. SCHÖLKOPF AND A. J. SMOLA, *Learning with kernels: support vector machines, regularization, optimization, and beyond*, MIT press, 2002.

[26] S. SI, C.-J. HSIEH, AND I. DHILLON, *Memory efficient kernel approximation*, in Proceedings of The 31st International Conference on Machine Learning, 2014, pp. 701–709.

[27] B. W. SILVERMAN, *Density Estimation for Statistics and Data Analysis*, Chapman and Hall, 1986.

[28] M. S. WARREN AND J. K. SALMON, *A parallel hashed octree N-body algorithm*, in Proceedings of Supercomputing, The SCxy Conference series, Portland, Oregon, November 1993, ACM/IEEE.

[29] B. XIAO, L. MOON, AND G. BIROS, *Parallel algorithms for nearest neighbor searches*, 2014. Submitted for publication,padas.ices.utexas.edu/static/papers/knn.pdf.

[30] C. YANG, R. DURAISWAMI, N. A. GUMEROV, AND L. DAVIS, *Improved fast gauss transform and efficient kernel density estimation*, in Computer Vision, 2003. Proceed-

ings. Ninth IEEE International Conference on, IEEE, 2003, pp. 664–671.