PARALLEL ALGORITHMS FOR NEAREST NEIGHBOR SEARCH PROBLEMS IN HIGH DIMENSIONS.

BO XIAO* AND GEORGE BIROS†

Abstract. The nearest neighbor search problem in general dimensions finds application in computational geometry, computational statistics, pattern recognition, and machine learning. Although there is a significant body of work on theory and algorithms, surprisingly little work has been done on algorithms for high-end computing platforms and no open source library exists that can scale efficiently to thousands of cores. In this paper, we present algorithms and a library built on top of Message Passing Interface (MPI) and OpenMP that enable nearest neighbor searches to hundreds of thousands of cores for arbitrary dimensional datasets.

The library supports both exact and approximate nearest neighbor searches. The latter is based on iterative, randomized, and greedy KD-tree searches. We describe novel algorithms for the construction of the KD-tree, give complexity analysis, and provide experimental evidence for the scalability of the method. In our largest runs, we were able to perform an all-neighbors query search on a 13 TB synthetic dataset of 0.8 billion points in 2,048 dimensions on the 131K cores on Oak Ridge's XK6 "Jaguar" system. These results represent several orders of magnitude improvement over current state-of-the-art methods. Also, we apply our method to non-synthetic data from machine learning data repositories. For example, we perform an all-nearest-neighbor search on a variant of the "MNIST" handwritten digit dataset with 8 million points in 784 dimensions on 16,384 cores of the "Stampede" system at the Texas Advanced Computing Center, achieving less than one second per `PKDT` iteration.

Key words. Nearest Neighbor Algorithms, Computational Statistics, Tree Codes, Data Analysis, Parallel Algorithms, Machine Learning

AMS subject classifications. 65N30, 65N50, 65N55, 65Y05, 68W10, 68W15

1. Introduction .

Given a set $\mathcal{R}$ of n reference points $\{\mathbf{r}_i\}_{i=1}^n \in \mathbb{R}^d$, and a distance metric $d(\mathbf{r}_i, \mathbf{r}_j)$ (e.g., the Euclidean metric $\|\mathbf{r}_i - \mathbf{r}_j\|_2$), we seek to find the k-nearest neighbors (`KNN`) for points $\{\mathbf{q}_i\}_{i=1}^m \in \mathbb{R}^d$ from a query points set $\mathcal{Q}$. When the query points are the same as the reference points, `KNN` is commonly refer to as the *all nearest neighbors* problem. Solving the k-nearest neighbors problem is easy by direct search in $\mathcal{O}(mn)$ work. That is, for each query point all distances to the reference points are evaluated followed by a k-selection problem in a list of n numbers. The all-`KNN` problem requires $\mathcal{O}(n^2)$ work, which is prohibitively expensive when n is large. Spatial data structures can deliver $O(n \log n)$ or even $O(n)$ complexity, asymptotically, for fixed dimension d. But for *high dimensions* (say, $d \geq 10$), spatial data structures provably deteriorate; for large d all known exact schemes end up having the complexity of the direct algorithm [47].[1]

In many applications however, the distribution of points is mostly concentrated in a lower-dimensional subspace of $\mathbb{R}^d$. In such cases, find approximate nearest neighbors (`ANN`) using indexing techniques (tree-type data structures or hashing techniques) can be more realistic than direct searches. For each query point $\mathbf{q}_i$, `ANN` attempts to find some points $\mathbf{r}_j$ that have high probability to be the k closest points in the given metric. Those that are not among the "exact" nearest neighbors are close to being so. In the following of this paper, we refer to the *exact* k-nearest neighbors search

*The University of Texas at Austin, Institute for Computational Engineering and Sciences, Austin, TX 78712 (`bo@ices.utexas.edu`).

†The University of Texas at Austin, Institute for Computational Engineering and Sciences, Austin, TX 78712 (`gbiros@acm.org`).

[1] An easy way to see why is to consider a lattice of points. Every point has 2^d immediate neighbors.

problem as KNN, and the *approximate* nearest neighbors search problem as ANN. This paper mainly focus on a scheme that uses tree indexing to solve ANN.

Motivation and significance. The KNN problem is a fundamental problem that serves as a building block for higher-level algorithms in computational statistics (e.g., kernel density estimation), spatial statistics (e.g., n-point correlation functions), machine learning (e.g., classification, regression, manifold learning), high-dimensional and generalized N-body problems, dimension reduction for scientific datasets, and uncertainty estimation [13, 41, 38]. Examples of the applicability of these methods to science and engineering include image analysis and pattern recognition [45], materials science [15], cosmological applications [18], particulate flow simulations [36], and many others [1]. Despite nearest neighbors search being fundamental for many algorithms in computational data analysis, there is little work on scaling it to high-performance parallel platforms.

1.1. Our approach and contributions. We introduce algorithms, complexity analysis, and experimental validation for tree-construction and nearest neighbor searches in arbitrary dimensions.

- *Direct nearest neighbors.* We propose two parallel algorithms for the direct calculation of the KNN problem. The first one prioritizes computing time over memory by replicating the query and reference points while minimizing the synchronization costs. This method is useful when absolute wall-clock performance is important. It is work-, but not memory-, optimal. The second algorithm uses a cyclic iteration that is memory optimal but has more communication. Both algorithms can be used for modestly-sized datasets to compute exact distances and verify correctness of approximate algorithms. But since the perform direct evaluations they cannot scale with m and n.
- *Parallel tree construction.* We present PKDT, a set of parallel tree construction algorithms for indexing structures in arbitrary number of dimensions. The algorithm supports several types of trees (e.g., ball trees, metric trees, or KD-trees). It is a recursive, top-down algorithm in which every node corresponds to *a group* of processes and one group of reference points. The key feature of the algorithm is that the tree is *not* replicated across MPI ranks. The tree is used to partition and prune spatial searches across MPI processes.
- *Approximate randomized nearest-neighbor searches.* We present a randomized tree algorithm based on PKDT, randomization, and iterative greedy searches, for the solution of the ANN problem. Our implementation supports arbitrary query and reference points. We test the accuracy on both synthetic and machine-learning benchmark datasets and we report the results in §3.3. We are able to obtain good accuracies with less than 5% of the distance evaluations required for an exact search.
- *Scalability tests on up to 131K cores.* We conduct weak and strong scalability tests for the various components of the method. The largest run is up to nearly one billion 2048-dimensional reference points for the ANN problem for $k = 2048$. This result dwarfs the previously reported result with one million points in 10 dimensions for $k = 3$ [8] (see §3.4). Table 1.1 summarizes the largest data size we run on each method.
- *Software release.* We make this software available as part of a library for scalable data analysis tools. The library is under the GNU General Public License, it is open-source, available at RKDT. The library supports hierarchical kmeans trees, ball trees, KD trees, exact and approximate nearest neighbor

searches, and kmeans clustering with different seeding variants.

data size	KNN 2d partition	KNN cyclic partition	KNN PKDT	ANN randomized PKDT
number of points (million)	82	12	160	819
dimension	1,000	100	100	2,048
number of processes	16,384	12,288	12,288	16,384
million cycles/point/core	12	266	1	40

TABLE 1.1
Overall performance summarization of different nearest neighbors search. *We listed the latest data sizes in our experiments for each methods, for which the all nearest neighbors are selected.*

To our knowledge, our framework is the first scheme that enables such levels of performance and parallelism for arbitrary dimension computational geometry problems. Our tree construction and nearest-neighbors are already being used for kernel summation and treecodes in high dimensions [24, 25].

1.2. Limitations. PKDT has several limitations: **(1)** It has not be designed for frequent insertions or deletion of points. **(2)** PKDT does not resolve the curse of dimensionality. If the dataset has high intrinsic dimension, the approximation errors will be very large. **(3)** We have only examined ℓ_2 distance metrics and many applications require other distance metrics. However, this is a simple implementation issue, at least for standard distance metrics.

1.3. Related work. There is a very rich literature on nearest neighbor algorithms and theory. The most frequently used algorithm is the KD-tree [12], which at each level partitions the points into two groups according to one coordinate. Fukunaga et al [14] propose another tree structure that groups points by clustering points with kmeans into k disjoint groups. If the metric is non Euclidean, however, ball tree or metric tree might provide better performance [7, 46, 31]. As the dimensionality increases, each of these methods lose their effectiveness quickly, requiring visit almost all leaf nodes in a tree. In fact in high-dimensional spaces there are no known algorithms for exact nearest neighbor search that are more efficient than the direct exact search (or linear search per query point). Modern implementations of exact searches include [18, 37, 8, 27].

Avoiding this curse of dimensionality requires two ingredients. The first is to settle for approximate searches. The second is to introduce the assumption that the dataset has a lower *intrinsic dimensionality*, i.e., the dataset points lie on a lower dimensional subspace of $\mathbb{R}^d$ [20, 35, 6]. Given this, the search complexity *per query point* can be theoretically reduced from $\mathcal{O}(n)$ to $\mathcal{O}(\eta \log n)$, where η is determined only by the intrinsic dimensionality of data points [44, 39, 5, 9]. Several approximate algorithms have been proposed, with their main theoretical result being that, asymptotically, they can bound the relative error in the distance from the true nearest neighbor. No algorithm offers an actual practical accuracy guarantee, while simultaneously bounding the work complexity. There are two main classes of approximations, randomized tree based and hashing based algorithms.

Randomized tree algorithms were first proposed in [9]. Different variants of this approach have been used for the KNN problem: [30, 2, 19, 28]. In this paper we follow the work of [19]. Another tree-based algorithm, without randomization, that supports approximate KNN by pruning the tree search is presented in [27].

A different approach to approximate is KNN is to use hashing. Locality sensitive hashing (LSH) bins the reference points so that similar points are grouped into the same bucket with high probability. The details of the LSH algorithm can be found in [5]. The original implementation of LSH can be found in [4]. In [33], the authors present excellent analysis and implementation of the LSH algorithm on a GPU architecture. No open-source LSH based algorithms for HPC distributed memory architectures are available. We have implemented an MPI LSH algorithm, and we will discuss its comparison with randomized trees in a future article.

Scalability and available software. As we mentioned there is little on distributed memory scalable algorithms for nearest-neighbor searches. Also, while there is excellent theoretical work on parallel KD-tree construction [3], no implementations are available. Nearest neighbor algorithms using direct search or LSH on GPUs can be found in [16, 42, 34, 17]. The one exception is the FLANN package [30, 28], which supports multithreading and some MPI based parallelism in which the reference points are partitioned across processors and the query points are replicated. In [30] no all-KNN results are presented for the MPI version, and runs are done only up to eight nodes and for very small m (number of query points). FLANN's MPI parallel scheme resembles the scheme described in §2.2.1 but with approximate searches. This scheme does not scale well for the all-KNN problem as it requires replication of the query set on each MPI process. Other open-source libraries include MLPACK [8] and ANN [27]. These libraries do not support distributed memory parallelism. We discuss more the performance of these codes and FLANN in §3.4. In all, to the best of our knowledge, no scalable KNN libraries exist.

Outline of the paper. In §2, we introduce the basic algorithmic components of the method: the single node optimization of distance calculations and KNN search (shared memory parallelization) (§2.1); two schemes for direct exact parallel KNN search (§2.2.1 and §2.2.2); and our main contribution, the parallel tree construction and ANN search in §2.3. In §3, we apply our KNN algorithms on large UCI datasets [22] and discuss the relationship between the convergence and the intrinsic dimensions. Finally, we report more results on the performance and scalability of the method on very large synthetic datasets.

2. Methods.

We have implemented several scalable methods for both KNN and ANN. In the following sections, we discuss the different components and we provide weak and strong scaling results.

2.1. Single-node distance and k-nearest neighbor kernels.

We use single-node, multi-threaded kernels for both distance calculations and KNN searches on locally stored points. The distances between all $(\mathbf{q}_i, \mathbf{r}_j)$ pairs of points are computed as follows. The sets of reference points $\mathcal{R}$ and query points $\mathcal{Q}$ are stored as $n \times d$ and $m \times d$ matrices, respectively, with one point per row. Then, we compute $D_{ij} = |\mathbf{q}_i|^2 + |\mathbf{r}_j|^2 - 2\mathbf{r}_i \cdot \mathbf{q}_j$ for all $(i, j), i \in 1 \ldots m, j \in 1 \ldots n$, where D_{ij} is the *square* of the distance between query point i and reference point j. By expressing the $-2\mathbf{r}_i \cdot \mathbf{q}_j$ term as the matrix-matrix product $-2\mathcal{R}\mathcal{Q}^T$, we can use a BLAS DGEMM call, which delivers high single-node performance and portability to various homogeneous and heterogeneous platforms.

With the above squared-distance kernel, a single-node KNN calculation is quite simple. It can be implemented by calling the above distance routine and, for each query point, sorting the squared distances in ascending order while keeping track of the index of the reference point corresponding to each distance. Clearly, this approach exposes a great deal of parallelism. However, for sufficiently small k ($k << n$), we

would waste a significant amount of time by sorting each row of D in its entirety. Since we only care about the first k nearest points, we address this problem by maintaining a minimum heap of size k for each query point. Scanning the n reference points requires inserting every reference point into the minimum heap at a cost of $\mathcal{O}(\log k)$. Thus the total complexity is $\mathcal{O}(n \log k)$, which is less expensive than a sort of $\mathcal{O}(n \log n)$.

One implementation issue encountered in the development of the direct `KNN` kernel involves the fact that vendor tuned DGEMM routines provide very poor multi-threaded performance when multiplying a tall, skinny matrix by a relatively small matrix. In our code, this happens when computing distances between a small number of query points and a large number of reference points with low dimensionality. We have observed this problematic behavior in both Intel's MKL and GOTO BLAS. We are currently addressing this problem by developing a customized high performance kernel for nearest neighbor searches, which achieves more uniform performance. Those results will be reported elsewhere.

2.2. Brute-force Direct `KNN`. We have implemented two distributed direct evaluation (brute-force) nearest neighbor algorithms according to different data partition schemes: two-dimensional partitioning and cyclic partitioning.

2.2.1. Two dimensional partitioning with query point replication. We first consider a partitioning scheme in which the reference and query points are divided into r_{parts} and q_{parts} pieces, respectively, and distributed across $r_{\text{parts}} \cdot q_{\text{parts}}$ nodes (see Figure 2.1(a)). This scheme is more memory-intensive than the cyclic partitioning scheme we describe later since the reference points and the query points are replicated q_{parts} and r_{parts} times, respectively. However, all calculations performed are local until a reduction at the very end.

Algorithm 1 RECTDIRECTK$(n_{\text{global}}, m_{\text{global}}, k, d, r_{\text{parts}}, q_{\text{parts}})$

1: Choose n_{local} and m_{local}.
2: Read n_{local} reference points into $\mathbf{r}_{\text{local}}$ starting with $\lceil n_{\text{global}}/r_{\text{parts}} \rceil \cdot \text{id}$.
3: Read m_{local} query points into $\mathbf{q}_{\text{local}}$ starting with $\lceil m_{\text{global}}/q_{\text{parts}} \rceil \cdot \text{id}$.
4: D = computeDistances$(\mathbf{r}_{\text{local}}, \mathbf{q}_{\text{local}}, n_{\text{local}}, m_{\text{local}}, k, dim)$
5: **for** $i = 0 \ldots q_{\text{parts}} - 1$ **do** Sort ith row of D
6: Perform k-reduction among processes with ranks $\text{id}\%q_{\text{parts}}, \text{id}\%q_{\text{parts}} + q_{\text{parts}}, \ldots, \text{id}\%q_{\text{parts}} + (r_{\text{parts}} - 1) \cdot q_{\text{parts}}$.
7: Process has k-nearest neighbors for q_{local} if $\text{id} < q_{\text{parts}}$.

Algorithm 1 shows how this partitioning is used to compute the `KNN`. Each process computes a matrix containing the distance between each $(\mathbf{r}_i, \mathbf{q}_j)$ pair, and sorts each row of the matrix (the distances for each query point) to select the k minimum distances. Finally, we perform a k-min reduction among all processes which have the same query points.

Assuming the query and reference points are partitioned into an equal number of pieces, each process's memory consumption grows as $\mathcal{O}(\frac{n}{\sqrt{p}} + \frac{m}{\sqrt{p}} + \frac{nm}{p})$, since each set of points is replicated $\sqrt{p}$ times. Since our algorithm requires an all-pairs distance calculation, a sort of the computed distances (selection sort for small k and merge sort for large k), and a reduction on the k minimum of those distances, its time complexity is $\mathcal{O}\left(\frac{mnd}{p} + \frac{1}{\sqrt{p}}(m+n) + \frac{mn}{p}(\log \frac{n}{\sqrt{p}}) + k \log \sqrt{p}\right)$ for large k, and

$\mathcal{O}\left(\frac{mnd}{p}+\frac{1}{\sqrt{p}}(m+n)+\frac{mn}{p}+k\log\sqrt{p}\right)$ for small k.

	q1	q2	q3	q4
r1	r1q1	r1q2	r1q3	r1q4
r2	r2q1	r2q2	r2q3	r2q4
r3	r3q1	r3q2	r3q3	r3q4

(a) Rectangular partitioning

iter\node	1	2	3	4
1	r1q1	r2q2	r3q3	r4q4
2	r4q1	r1q2	r2q3	r3q4
3	r3q1	r4q2	r1q3	r2q4
4	r2q1	r3q2	r4q3	r1q4

(b) Cyclic partitioning

FIG. 2.1. Data partition for direct search. *(a) Diagram of rectangular partitioning. Here the query points are partitioned into four parts and the reference points into three. The processes in each column are part of the same group when the k-reduction is performed at the end of the algorithm. Depending on the mapping to MPI processes, the query points, the reference points, or both are replicated (b) Diagram of cyclic partitioning. Notice that now there is no replication but the method requires four steps and cyclic-shift of the query or reference points, depending on which one is of smaller cardinality.*

2.2.2. Cyclic partitioning for large problem sizes. By using a partitioning scheme that does not replicate any data across processes, we can solve the exact KNN for significantly larger problem sizes than would fit into a single nodes's memory when using the replicated partitioning scheme. However, the reduced memory footprint comes at the cost of additional computation time. As illustrated in Figure 2.1(b), in this method, both $\mathcal{R}$ and $\mathcal{Q}$ are partitioned into p nearly-equally sized partitions distributed among the processes.

Algorithm 2 shows how to use such a partitioning to compute the k-nearest neighbors. The algorithm works as follows. A given partition of $\mathcal{Q}$ remains pinned to its "home" process, while in each of p communication steps, each partition of $\mathcal{R}$ shifts one process to the left in a "ring" of processes. At each communication step, a process runs the local direct KNN kernel and merges the results with the current k minimum distances for each query point. Depending on which set is larger, we can cycle either the reference points or the query points.

The communication cost of this algorithm is $\mathcal{O}(pt_S+ndt_T)$, where t_S is the setup time (latency) required for each message, and t_T is the time required to transmit each double-precision value. Because communication and computation can be overlapped completely for sufficiently large m and n, the time complexity is determined by the time spent in the local direct KNN calculation, which is $\mathcal{O}(\frac{mnd}{p}+m+n+\frac{mn}{p}\log\frac{n}{p})$ for large k and $\mathcal{O}(\frac{mnd}{p}+m+n+\frac{mnk}{p})$ for small k. The memory cost for this approach is $\mathcal{O}(\frac{mn+m+n}{p})$.

2.3. Randomized KD tree (PKDT). Parallelizing tree operations with performance guarantees depends on the application and is hard even in low dimensions, especially in distributed memory architectures. Most of the tree algorithms for high dimensions follow a top-down approach in which the points are grouped recursively into tree nodes and then, during search, different pruning strategies are used [40]. Other techniques such as space-filling curves [21], graph partitioning [10] and geometric partitioning [26] have been used for parallelization, but for high-dimensional datasets these are not as successful and we are not aware of any parallel implementations. For this reason we have opted for the top-down basic approach.

Algorithm 2 CYCLICDIRECTK$(r, q, n_{\text{global}}, m_{\text{global}}, k, dim)$

1: Choose *send_partner* and *recv_partner*.
2: **for** $i = 0 \ldots p-1$ **do**
3:   **if** *kmin_set* **then**
4:     $kmin_new \leftarrow$ directKQuery$(\mathcal{R}_i, \mathcal{Q}_i, n_{\text{local}}, m_{\text{local}}, k, d)$
5:     $temp \leftarrow$ kmin_merge$(kmin, kmin_new, m_{\text{local}}, k)$
6:     $kmin \leftarrow temp$
7:   **else**
8:     $kmin \leftarrow$ directKQuery$(\mathcal{R}_i, \mathcal{Q}_i, n_{\text{local}}, m_{\text{local}}, k, d)$
9:     $kmin_set \leftarrow$ TRUE
10:   **end if**
11:   send$(r, send_partner)$
12:   $r \leftarrow$ recv$(recv_partner)$
13: **end for**
14: send$(r, send_partner)$
15: $r \leftarrow$ recv$(recv_partner)$

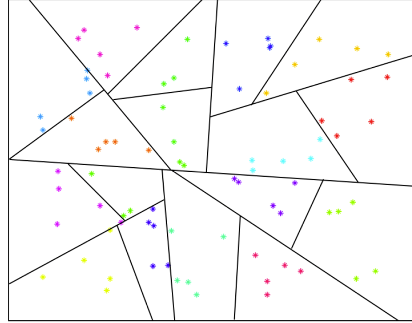


FIG. 2.2. Hyperplane splitting. *An illustration of the hyperplane splitting.*

Forgetting parallelism for a moment, the basic algorithm is a classical top-down recursive construction of a binary tree. Starting at the root. We split the points into two groups, and create two new nodes (the left and right child of the root). Then we assign one group of points to one child and the other group to the other child. We compared two point splitting schemes: clusters based (refered as $\texttt{PKDT}_C$) and hyperplane based (refered as $\texttt{PKDT}_H$).

In $\texttt{PKDT}_C$, we use kmeans with a large number of clusters (greater than two, so that we can control the number of points per leaf node) and then we assign clusters to different nodes. Hyperplane splitting is quite common and includes standard KD-trees. Clustering is used in `FLANN` [30, 29] and we want to test its performance. The clustering algorithm is given in the appendix. The hyperplane splitting is illustrated in Algorithm 3 and Figure 2.2.

In a hyperplane based scheme, we calculate a projection direction and project all points onto this direction, then split the points according to the median of the projected values. We tested three ways to choose the projection direction. The first one is selecting a coordinate axis at random, i.e., uniformly sampling a number from d integers. The second is to choose the coordinate with the largest variance considering all points within a node. The third alternative is to choose the direction

where the points are furthest apart. To do this, we first compute the centroid $\mathbf{c}$ of all points within a node, then find the furthest point $\mathbf{p}_1$ from $\mathbf{c}$, and a second point $\mathbf{p}_2$ which is furthest from $\mathbf{p}_1$. Finally we project points along the direction $\mathbf{p} = \mathbf{p}_1 - \mathbf{p}_2$. After projecting points onto this direction and calculate the median $\mathbf{m}$ of all projected values, we simply separate points into two groups by assigning points whose projected values is smaller than m to the left child, and the other to the right child. All of these three resulted in similar accuracies when used for the `ANN` problem. For the rest of this paper, the complexity estimates are computed using the random selection criterion.

In the end we found the hyperplane partition superior. First, partitioning points evenly using a hyperplane results in roughly the same number of points per group, which has implications to the parallel implementation and the load balancing of the algorithm. In other words, each child node has the same number of points as the others. Second, clustering based partition is more expensive as it requires solving kmeans problems multiple times. Third, we observe that the hyperplane splitting results in better pruning during neighbor searches. In the following parts, we only focus on $\texttt{PKDT}_H$, and the details of $\texttt{PKDT}_C$ can be found in Appendix B.

2.3.1. Tree construction. A standard tree construction uses the NodeSplit function summarized in Algorithm 3. Let $\mathcal{T}$ be a tree node; $\mathbf{X}_\mathcal{T}$ be all the points assigned to $\mathcal{T}$, $l_\mathcal{T}$ be the level of the node $\mathcal{T}$ in the tree. Hyperplane splitting requires a direction $\mathbf{p}_\mathcal{T}$ along which $\mathbf{X}_\mathcal{T}$ are projected. $m_\mathcal{T}$ is the median of all projected values $\mathbf{x}_{\mathbf{p}_\mathcal{T}}$. Denote $\mathbf{X}_l$ and $\mathbf{X}_r$ as the subset of $\mathbf{X}_\mathcal{T}$, which contains the points assigned to the left child node and right child node of $\mathcal{T}$ respectively. For bookeeping we use a `maxLevel` setting to indicate the maximum allowed level of a node and we use `minNumofPoints` (minimum number of points a node have to hold) to decide whether to split a node further or not.

Algorithm 3 NodeSplit($\mathcal{T}, X_\mathcal{T}$)

```
1: if l_T == maxLevel || |X_T| < minNumofPoints then
2:    store X_T in T
3:    return
4: end if
5: p_T = SplitDirection(T, X_T)
6: x_{p_T} = Project(X_T, p_T)
7: m_T = Median(x_{p_T})
8: for x_i ∈ X_T do
9:    if x_{p_T,i} < m_T then
10:      assign x_i to X_l
11:   else
12:      assign x_i to X_r
13:   end if
14: end for
15: NodeSplit(LeftChild(T), X_l)
16: NodeSplit(RightChild(T), X_r)
```

Shared memory $\texttt{PKDT}_H$ construction: Building a shared memory tree is straightforward. As mentioned before, we use the random selection to choose the projection direction, thus the cost of line 5 is $\mathcal{O}(1)$. The implementations of line 6 and 8-14 in Algorithm 3 are simply a OpenMP parallel-for, which have a complexity of $\mathcal{O}(n)$. To find the median, we use a select algorithm to find the $n/2$-th smallest element in the

input array (Appendix C). The only difference between the shared memory selection and Appendix C is there is no *Allreduce()* operation. The average complexity of select is $\mathcal{O}(n)$. The total work of Algorithm 3 is then $W(n) = \mathcal{O}(n) + 2W(n/2) = \mathcal{O}(n \log n)$. As a result, using the PRAM model, the time complexity of shared memory tree construction with p threads is $\mathcal{O}((n \log n)/p + \log p)$.

Distributed memory $\texttt{PKDT}_H$ construction: We parallelize Algorithm 3 using a distributed top-down algorithm. At each level, a node is split to two children, each of which contain one groups of points (in this implementation points locate on the lefthand side or the righthand side of the hyperplane). A tree node is shared among multiple processes. We implement this by assigning an MPI communicator to each node $\mathcal{T}$. This communicator is recursively split as illustrated in Figure 2.3. Each of those processes within one node's communicator maintains a local data structure for the tree node containing its MPI communicator and pruning information to traverse incoming queries. In each process, these data structures are stored in a doubly-linked list representing a path from root to leaf for that process's portion of the tree. The minimum granularity of $\texttt{PKDT}_H$ node is one MPI process. At the leaf, we switch to the local shared memory tree.

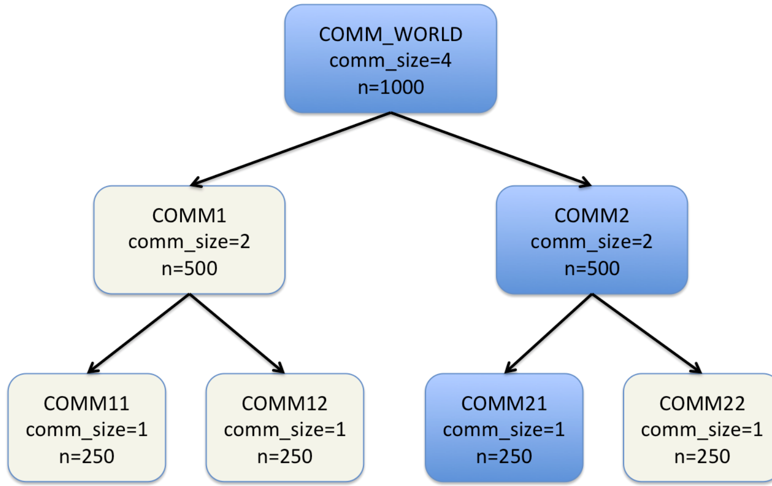


FIG. 2.3. Recursive tree construction and the corresponding communicators. *An illustration of the splitting of communicators used for reference point redistribution among children. The highlighted nodes represent a path from root to leaf as stored by a single process.*

The scheme is summarized in Algorithm 4, where $\mathcal{C}_\mathcal{T}$ is the communicators of node $\mathcal{T}$. Choosing the split direction requires a broadcast (process 0 choose a random coordinate axis and then broadcast the number to other processes). The projection is local and the parallel median (using randomized quick select in Appendix C) has an expected complexity of $\mathcal{O}(t_c \log p \log n + n/p)$, where $t_c = t_s + t_w$ [43]. Partitioning the points to the two children nodes is the most communication-intensive part. In POINTREPARTITION() (line 15 in Algorithm 4), the points are shuffled among processes. Each process would have have two subsets $\mathbf{X}_l$ and $\mathbf{X}_r$ after assigning membership by the median of projected values (line 8-14 in Algorithm 4). In order to split the current node $\mathcal{T}$ to two children nodes, the communicator $\mathcal{C}_\mathcal{T}$ will be divided to two sub communicators. As shown in Figure 2.4, the COMM_WORLD is split to

two communicators COMM1 and COMM2, each has two out of four processes. On process 0, 14 points (in red color) belong to the right node $\mathbf{X}_l$ and 6 points (in green color) go to $\mathbf{X}_r$. In the next level, p0 is assigned to COMM1, which is the communicator of left node of $\mathcal{T}$. Those 6 points in green must be shuffled to either p2 or p3, i.e, to any process assigned to be the right sub communicator (right child node).

Algorithm 4 DistributedNodeSplit($\mathcal{T}, \mathbf{X}_\mathcal{T}, \mathcal{C}_\mathcal{T}$)

1: **if** $size(\mathcal{C}_\mathcal{T}) == 1 \mid\mid l_\mathcal{T} ==$maxLevel $\mid\mid |\mathbf{X}_\mathcal{T}| <$ minNumofPoints **then**
2: SharedNodeSplit($\mathcal{T}, \mathbf{X}_\mathcal{T}$)
3: return
4: **end if**
5: $\mathbf{p}_\mathcal{T} =$ ParSplitDirection($\mathcal{T}, \mathbf{X}_\mathcal{T}$)
6: $\mathbf{x}_{\mathbf{p}_\mathcal{T}} =$ Project($\mathbf{X}_\mathcal{T}, \mathbf{p}_\mathcal{T}$)
7: $m_\mathcal{T} =$ ParMedian($\mathbf{x}_{\mathbf{p}_\mathcal{T}}$)
8: **for** $\mathbf{x}_i \in \mathbf{X}_\mathcal{T}$ **do**
9: **if** $\mathbf{x}_{\mathbf{p}_\mathcal{T},i} < m_\mathcal{T}$ **then**
10: assign $\mathbf{x}_i$ to $\mathbf{X}_l$
11: **else**
12: assign $\mathbf{x}_i$ to $\mathbf{X}_r$
13: **end if**
14: **end for**
15: $\mathbf{X}^{new} = PointRepartition(\mathbf{X}_l, \mathbf{X}_r, \mathcal{C}_\mathcal{T})$
16: $\mathcal{C}^{new} =$ CommSplit($\mathcal{C}_\mathcal{T}$)
17: DistributedNodeSplit(Kid($\mathcal{T}$), $\mathbf{X}^{new}, \mathcal{C}^{new}$)

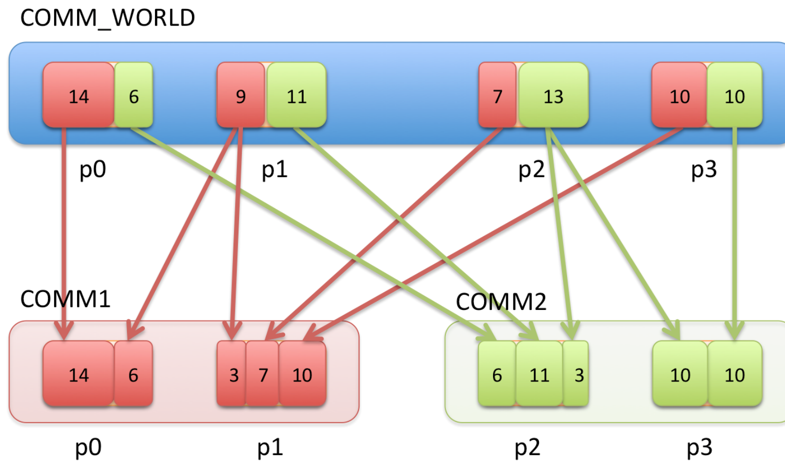


Fig. 2.4. *Here we illustrate the problem of load balancing when we split a node and its communicator. The blue region (with communicator* `COMM_WORLD`*) denotes the parent. Notice that every process in* `COMM_WORLD` *has an equal number of points. After the hyperplane split, some points go to the left child (red) and some to the left child (green). Also, we need to split* `COMM_WORLD` *to* `COMM1` *and* `COMM2`*, assign processes to each communicator, and redistribute the points from the parent to the children. There is no unique way to do this, but the goal is that upon completion each process in* `COMM1` *and* `COMM2` *has again the same number of points with as little communication as possible. Using an* `MPI_Alltoallv()` *is one possible way to do it.*

This will create severe load-balancing issues without carefully designed shuffling strategy. We discuss three ways to solve the data points shuffling problem (PointRepartition()), and compare different methods in details.

2.3.2. Load balanced data exchange. It is necessary to ensure the workload is evenly distributed in each process at every level of the tree. There are different ways to obtain the load balance, we present three variants for the PointRepartition() function. The problem is illustrated in Figure 2.4.

All-to-all data exchange: We repartition the points by using `MPI_Alltoallv()` directly at each level. Before calling this function, it is necessary to determine the MPI process that each point belongs to and ensure that after repartition, each process has an equal number of points. The algorithm that supports arbitrary spatial trees is given in Algorithm 5, where r_i^{target} is the target rank that point i have to be redistributed to.

Algorithm 5 Alltoall_exchange($\mathbf{X}_l, \mathbf{X}_r, \mathcal{C}_\mathcal{T}$)

1: $n_{loc} = |\mathbf{X}_l| + |\mathbf{X}_r|$
2: $n_{glb} = \text{MPI_Allreduce}(n_{loc}, \mathcal{C}_\mathcal{T})$
3: $n_{avg} = n_{glb}/size(\mathcal{C}_\mathcal{T})$
4: **for** each $\mathbf{X} \in \{\mathbf{X}_l, \mathbf{X}_r\}$ **do**
5: $\quad nc = |\mathbf{X}|$
6: $\quad \text{MPI_Scan}(nc, n_{scan}, \mathcal{C}_\mathcal{T})$
7: $\quad$ **for** i = 0 : nc-1 **do**
8: $\quad\quad r_i^{target} = \lfloor (i + n_{scan})/n_{avg} \rfloor$
9: $\quad$ **end for**
10: $\quad \text{MPI_Alltoallv}(\mathbf{X}_\mathcal{T}, \mathbf{r}^{target}, \mathcal{C}_\mathcal{T})$
11: **end for**

Assuming average message size of $\mathcal{O}(nd/p^2)$, the complexity of the exchange is $\mathcal{O}(t_w nd/p + t_s p)$. Therefore the expected complexity (omitting $\mathcal{O}()$) of one split at level ℓ with $p_\ell = p/2^\ell$ MPI tasks and n_ℓ points can be estimated as following: the projection costs $n_\ell d/p_\ell$ work, the median calculation costs $t_c \log^2 p_\ell \log n_\ell + n_\ell/p_\ell$ and the exchange cost is $t_w n_\ell d/p_\ell + t_s p_\ell$. Median-based splits result in perfect load balancing so that $n_\ell/p_\ell = n/p$. Summing over the levels of the tree $(1, \ldots, \log p)$, we obtain the expected complexity of the construction to be

$$\mathcal{O}\left((t_s + t_w) \log^2 p \log n + (1 + t_w \log p) \frac{nd}{p} + t_s p \right). \tag{2.1}$$

Figure 2.4 illustrates an example of the all-to-all data exchange strategy at one level. One drawback is that all processes have a chance to exchange data with each others. For large process counts and large dimensionality d, the communication cost will be excessive due to the $t_w nd/p \log p$ term. More importantly, MPI resources get taxed by managing such a massive exchange (essentially we're shuffling the whole dataset), which leads to having to use very small grain size to avoid running out of memory.

Pointwise data exchange: One way to resolve the massive exchange from `MPI_Alltoallv()` is a different point wise exchange approach in Algorithm 6: Suppose there are p processes. As shown in Figure 2.5 (a), process j with $j < p/2$ is assigned to $\mathcal{C}_l$ otherwise it is assigned to $\mathcal{C}_r$. To redistribute the points, a process

with $j < p/2$ sends its local $\mathbf{X}_r$ to $p-j$ and receives $\mathbf{X}'_l$ from $p-j$. This exchange introduces memory/work imbalance (worst case is dn/p), which as we traverse the tree may grow to $\mathcal{O}(nd/p \log p/2)$. In terms of memory, such complexity severely limits scalability. Hence, after exchanging points at each level, we need to rebalance the points among all processes within the same child communicator ($\mathcal{C}_l$ or $\mathcal{C}_r$). REBALANCE() in Algorithm 7 uses an iterative point-to-point communication to balance the load.

In order to find the parter rank in the point-to-point communication (Line 2 in Algorithm 7), we first sort the number of points on each process, then we can make a pair of every two processes by matching the process with the largest number of points to the process with the smallest number of points, then the process with the second largest number of points to the process with the second smallest number of points, etc.. Without loss of generality, we assume each process has n_i points and $n_0 < n_1 < n_2 < \cdots < n_{p-1}$, then the communication pairs are $(rank_0, rank_{p-1}), (rank_1, rank_{p-2}), \cdots, (rank_{p/2-1}, rank_{p/2})$. We can show that the maximum number of exchanges required to obtain perfect load balance is $\log p$ and the overall memory requirement for the construction is $\mathcal{O}(2nd/p)$.

Proof.

1) The first exchange in Algorithm 7: the number of points on each process will be $n_i^{(1)} = n_{p-1-i}^{(1)} = \frac{n_i^{(0)} + n_{p-1-i}^{(0)}}{2}$, each process has the number of points as

$$\frac{n_0^{(0)} + n_{p-1}^{(0)}}{2}, \cdots, \frac{n_{p/2-1}^{(0)} + n_{p/2}^{(0)}}{2}, |, \frac{n_{p/2-1}^{(0)} + n_{p/2}^{(0)}}{2}, \cdots, \frac{n_0^{(0)} + n_{p-1}^{(0)}}{2}$$

2) The second exchange: note the numbers of points on the first half of processes and the second half of processes are symmetric. A second exchange on all processes is equivalent to exchange points within each half of processes. After sorting the number of points on each half and reorder, the number of points on each process is an average of 4 $n_i^{(0)}$.
3) Similarly, after the third exchange, each process would have an average of 8 $n_i^{(0)}$. Finally at the t-th exchange ($t = \lceil \log p \rceil$), the number of points on each process would be $\sum_{i=0}^{p-1} n_i^{(0)}/p$. That is after $\log p$ point-to-point communications, perfect load balance is achieved. Figure 2.5 (b) gives an example of this rebalance procedure.

□

The sort can be done using a distributed bitonic sort with time complexity $\log^2 p$. This scheme removes $\mathcal{O}(t_s p)$ from equation (2.1):

$$\mathcal{O}\left((t_s + t_w)\log^2 p \log n + (1 + t_w \log p)\frac{nd}{p}\right). \tag{2.2}$$

Its main value however, is that it avoids collectives and has a lower memory footprint than `MPI_Alltoallv()`.

Replication of the complete tree at each process: Note that neither of the above schemes stores all nodes of the tree (the complete tree) at every single process, but only $\log p$ nodes (the path from the root to the leaf). By storing the whole tree, we can remove the $\log p$ factor from the $\mathcal{O}(t_w nd/p \log p)$ point-exchange cost. This algorithm is described in [3]. The basic structure of the algorithm is exactly the same as Algorithm 3 but now all the steps are parallelized using all of the processors. Each

Algorithm 6 POINTWISE_EXCHANGE($\mathbf{X}_l, \mathbf{X}_r, \mathcal{C}_\mathcal{T}$)

1: $\{\mathcal{C}_l, \mathcal{C}_r\}$ = COMMSPLIT($\mathcal{C}_\mathcal{T}$)
2: r =rank($\mathcal{C}_\mathcal{T}$)
3: **if** r $< p/2$ **then**
4: Send $\mathbf{X}_l$ to rank $(p-r)$
5: Receive $\mathbf{X}'_r$ from rank $(p-r)$
6: REBALANCE($\mathbf{X}', \mathcal{C}_l$)
7: **else**
8: Send $\mathbf{X}_r$ to rank $(p-r)$
9: Receive $\mathbf{X}'_l$ from rank $(p-r)$
10: REBALANCE($\mathbf{X}', \mathcal{C}_r$)
11: **end if**

Algorithm 7 REBALANCE($X_\mathcal{T}, C_\mathcal{T}$)

1: **for** $i = 1 : \log p$ **do**
2: find my_partner_rank
3: **if** $n_{my_rank} < n_{my_partner_rank}$ **then**
4: Receive $(n_{my_partner_rank} - n_{my_rank})/2$ points from my_partner_rank
5: **else**
6: Send $(n_{my_rank} - n_{my_partner_rank})/2$ points to my_partner_rank
7: **end if**
8: **end for**

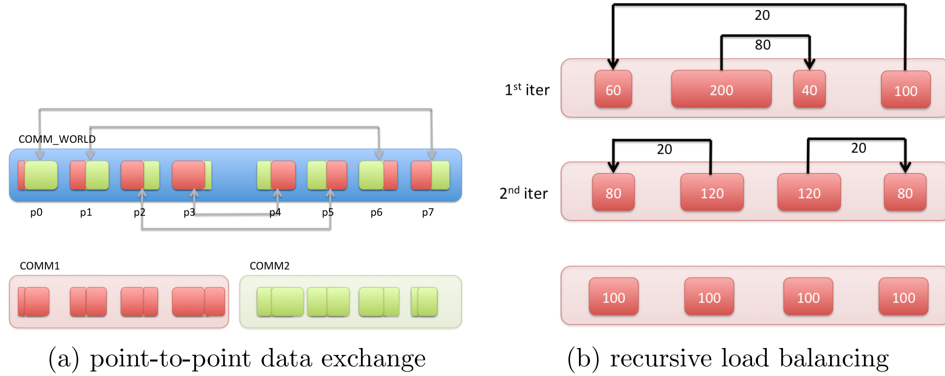
processor shuffles its points to the appropriate nodes using a top-down approach, with synchronizations at each node. At the leaf level, an `MPI_Alltoallv()` redistributes the points to their correct locations. All stages but the final all-to-all are perfectly balanced, and the complexity can be simplified to

$$\mathcal{O}\left(\log p \log n\,(t_s \log p + t_w p) + (1+t_w)\frac{dn}{p} + t_s p\right). \tag{2.3}$$

The last two terms come from the actual point redistribution at the leaves. The main drawback is now all the communications happened within the the global communicator MPI_COMM_WOLRD. During computing the median, this would significantly increase the communication costs if the number of processes is large. Experimental details to compare these three data exchanging method is given in §3.

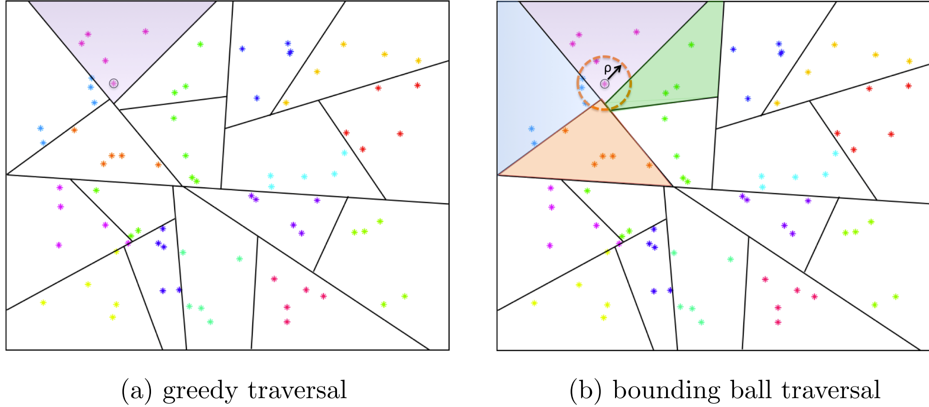
2.3.3. Nearest neighbors search algorithms. In this section, we introduce different tree traversal algorithms to find either exact (`KNN`) or approximate nearest neighbors (`ANN`). Two typical tree traversal strategies are the greedy traversal and the bounding ball traversal. In a greedy traversal (Algorithm 8), on the current node $\mathcal{T}$ a query point $\mathbf{q}$ only visits one of $\mathcal{T}$'s children nodes, i.e., $\mathbf{q}$ would only search nearest neighbors in one leaf node on the tree. In the bounding ball traversal (Algorithm 9), a bounding ball $\mathcal{B}_{\mathbf{q},\rho}$ is defined as an d dimensional solid sphere centered at $\mathbf{q}$ with a radius ρ. At node $\mathcal{T}$, $\mathbf{q}$ would visit all children nodes which have overlapped with $\mathcal{B}_{\mathbf{q},\rho}$. At a result, $\mathbf{q}$ would visit all leaf nodes that intersect with $\mathcal{B}_{\mathbf{q},\rho}$. These two traversal approaches are illustrated in Figure 2.6.

Based on these two tree traversal algorithms, we can solve both `KNN` and `ANN` problems.



(a) point-to-point data exchange (b) recursive load balancing

FIG. 2.5. Point-wise load balancing. *An illustration of point-wise load balancing scheme. First each pair of processes from each child node exchange corresponding data points with each other. Then a second pointwise data exchange occurs to reload balancing points within the child node. After* $\log p$ *iterations, the point are well balanced.*



(a) greedy traversal (b) bounding ball traversal

FIG. 2.6. *Illustration of the greedy and the bounding ball traversal strategies. In a greedy traversal, a query point (marked) as circle only visit the leaf node it belongs to, as in figure (a) the shaded region. In a bounding ball traversal, a query point would visit all the leaf nodes which overlap the query's bounding ball* $\mathcal{B}_{\mathbf{q},\rho}$.

1) KNN (exact nearest neighbors) by PKDT$_H$. In order to find the true nearest neighbors, we would visit all necessary leaf nodes which contain the true neighbors. Ideally if we know the distance between a query $\mathbf{q}$ and its k-th nearest neighbors $\mathbf{r}_k$, we can use a bounding ball traversal with the radius $\rho = \|\mathbf{q} - \mathbf{r}_k\|$. Of course $\mathbf{r}_k$ is unknown, but a good heuristic about $\mathbf{r}_k$ can be obtained by a greedy traversal at first. In other words we solve KNN problem by a two stage tree traversals. In the first pass, each query point would use a greedy traversal to go to one leaf node, and then search the exact k-nearest neighbors inside that leaf. Let $\mathbf{r}_k^{greedy}$ be the k-th nearest neighbor of $\mathbf{q}$ founded by greedy traversal, then a second bounding ball search with radius $\rho = \|\mathbf{q} - \mathbf{r}_k^{greedy}\|$ is performed to select all nearest neighbors inside $\mathcal{B}_{\mathbf{q},\|\mathbf{q}-\mathbf{r}_k^{greedy}\|}$. Finally the closest k out of all found neighbors are returned.

The effectiveness of the KNN using PKDT$_H$ depends on the number of nodes we visit, which in turn depends on the radius of the bounding balls. In the case that a significant overlap among bounding balls and leaf nodes occurs, which is typical

Algorithm 8 GreedyTraversal($\mathbf{q}$, $\mathcal{T}$)

```
1: if isLeafNode(T) then
2:    DirectSearch(q, X_T)
3: else
4:    for each query point q do
5:       if p_T · q − m_T < 0 then
6:          Assign q to T_l
7:       else
8:          Assign q to T_r
9:       end if
10:   end for
11:   Distribute q to appropriate process
12:   GreedyTraversal(q, T')
13: end if
```

Algorithm 9 BoundingBallTraversal($\mathbf{q}$, ρ, $\mathcal{T}$)

```
1: if isLeafNode(T) then
2:    DirectSearch(q, X_T)
3: else
4:    for each query point q do
5:       if p_T · q − m_T ≤ ρ then
6:          Assign q to T_l
7:       end if
8:       if m_T − p_T · q ≤ ρ then
9:          Assign q to T_r
10:      end if
11:   end for
12:   Distribute q to appropriate processes
13:   BoundingBallTraversal(q, ρ, T')
14: end if
```

in datasets with high intrinsic dimension, these exact searches using bounding balls will end up evaluating distances with all reference points. Then there is no difference between the tree based search and the direct search. To avoid memory exhaustion in these cases, we allow each process to store only a specified maximum number of query points during a tree traversal. If this limit is reached at an internal tree node, we stop the traversal early on that subtree and run an approximate search on that node.

2) `ANN` (approximate nearest neighbors) by randomized `PKDT`$_H$. The exact tree search algorithm might be in trouble if a query point have to visit many leaf nodes, which is likely in high dimensions. Instead of performing an exact search, we can visit only one leaf node, as what is described in Algorithm 8. The problem with this approach is it has very low accuracy. However, suppose there is a method which visit the leaf node randomly with an error ϵ, then after r independent runs, the accuracy can be improved to $1 - \epsilon^r$. This inspires the randomized `PKDT` algorithm.

Let's take the case in Figure 2.7 as an example. There are two points $\mathbf{r}_1$ and $\mathbf{r}_2$ which are both very close to the hyperplane $\mathbf{H}_1$, but located on opposite sides. Suppose there is a third point $\mathbf{q}$ that is close to both of these two. We ought to visit

both regions to correctly identify the nearest neighbors of $\mathbf{q}$. But if we choose another hyperplane, say $\mathbf{H}_2$, which is equivalent to rotate the reference points, all $\mathbf{r}_1, \mathbf{r}_2$ and $\mathbf{q}$ locates to the right side of $\mathbf{H}_2$. In this situation, greedy traversal is enough to find the true neighbors of $\mathbf{q}$. Our method is based on the sequential algorithm described in [19]. The basic idea is to avoid pruning: rotate the points randomly, build a KD-Tree, and search for nearest neighbors only at one leaf; iterate and combine the results from previous iteration till convergence. Algorithm 10 summarizes the whole procedures.

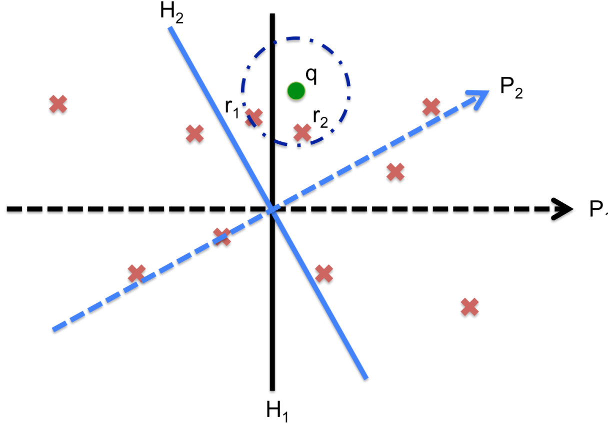


Fig. 2.7. *Illustration of random KD tree search*

Algorithm 10 RANDOMPROJECTIONTREESEARCH($\mathbf{q}, \mathbf{R}$)

1: $\mathcal{N}(\mathbf{q}) = \{\}$
2: **for** i = 1 : r **do**
3: Rotate Points from $\mathbf{X}_\mathcal{T}$ to $\mathbf{X}^i_\mathcal{T}$
4: DISTRIBUTEDNODESPLIT($\mathcal{T}_{root}$, $\mathbf{R}$, MPI_COMM_WORLD)
5: $\mathcal{N}^i(\mathbf{q})$ = GREEDYTRAVERSAL($\mathbf{q}$, $\mathcal{T}$)
6: Merge result of $\mathcal{N}(\mathbf{q})$ and $\mathcal{N}^i(\mathbf{q})$ to $\mathcal{N}(\mathbf{q})$
7: **end for**

Compared to the exact tree search, which is likely to visit many nodes, especially in high dimensions. The randomized `PKDT` only visit one leaf node at a time. Hence at most, the number of nodes a query point will visit would be r, where r is the number of iterations. It has been demonstrated that if data has a low intrinsic dimensional manifold structure, random projection could converge fast [9, 19].

3. Experimental results.

We present and analyze the performance and scalability results of each of our algorithms on several x86 clusters. We examine both single-node kernel performance, overall scalability, and accuracy of the `ANN` searches.

Platforms used. Our large strong- and weak-scaling results were obtained from runs on the *Jaguar* system at the National Center for Computational Sciences, *Kraken* platform at the National Institute for Computational Sciences, and the *Lonestar*, *Maverick* and *Stampede* clusters at the Texas Advanced Computing Center. Table 3.1 provides a summary of machine characteristics

	Jaguar	**Kraken**	**Lonestar**	**Stampede**	**Maverick**
nodes	18,688	9,408	1,888	6,400	132
cores	299,008	112,896	22,656	102,400	2640
GB/*node*	32	16	24	32	256
clock (GHz)	2.2	2.6	3.3	2.7	2.8
GFlops/*core*	8.8	10	13	21.6	22.4

TABLE 3.1
Machine characteristics.

All machines have dual-socket nodes. Kraken have AMD's Istanbul architecture connected to a Cray SeaStar2+ router, and the routers are interconnected in a 3-D torus topology. Each Jaguar compute node contains dual hex-core AMD Opteron 2435 (Istanbul) processors Jaguar uses a Gemini Torus topology. Lonestar is interconnected with QDR InfiniBand in a fat-tree topology and uses Intel Xeon X5680 processors for each socket. Maverick has Intel Xeon 2.8GHz E5-2680 v2 (Ivy Bridge) CPUs. Maverick has the Mellanox FDR InfiniBand interconnect. Stampede nodes has dual-soscket Intel Xeon 2.7GHz E5-2680 (Sandy Bridge) processors and 32GB RAM. The interconnect is the Mellanox FDR InfiniBand in a 2-level fat-tree topology. The libraries and executables were built using Intel compilers with MKL BLAS on Lonestar, Maverick and Stampede, and the Cray compilers on Jaguar an Kraken with vectorization and OpenMP.

Datasets. We test the performance on several datasets, both synthetic and real. The synthetic datasets are points sampled from a normal distribution. Furthermore, we also generate an embedded normal dataset, which we first generate a d-dimensional normal data, then expand them into D-dimensional space by padding zeros and rotation. In this way, the dataset could have an intrinsic dimensionality d. In the following, we denote the embedded normal data as "$d - D$ normal". We also use small datasets to test and compare the exact tree search algorithms. The first one is the US Census data from UCI Machine Learning Repository [22]. The second real dataset comes from the Gabor wavelet features of MRI cardiac images; The third one contains points from the phase space of a chaotic dynamical system. Finally, we test five large real datasets to show the accuracy and convergence of `PKDT`. They are "covtype", "susy", "higgs", "gas sensor" from UCI Machine Learning Repository [22], and the frequently used hand written digit dataset "mnist8m" [23]. Table 3.2 summarize all the datasets used in this paper.

3.1. Brute force direct `KNN` performance. For the 2d partitioned direct k-NN, we evaluate performance by conducting strong-scaling tests on 48 to 768 cores (8 to 128 sockets) on Kraken. Our runs were performed with a fixed number of nodes overall, $m = n = 100,000$, and with three values of dimension, $d = 8$, 512, and 8192. The results in Table 3.3 show nearly perfect linear strong scaling, indicating that the communication and node interdependency caused by the k-reduction does not produce a significant performance bottleneck. In Table 3.4, we demonstrate perfectly flat weak scaling up to 16,384 MPI processes.

For the cyclic direct `KNN`, we evaluate performance by conducting weak-scaling tests on 12 to 12,288 cores (1 to 1024 nodes) on Kraken. In this experiment, we use a single MPI process per core and disable multi-threading within processes. Even though we do achieve good performance with hybrid parallelism, this decision was made to enable measuring the scalability to a larger number of independent processes.

dataset	number of points (millions)	dimension	intrinsic dim
US census	2.45	68	-
dynamical system	10	81	-
gabor wavelets	10	276	-
covtype	0.5	54	-
gas sensor	6.7	16	-
susy	4.5	18	-
higgs	10.5	28	-
mnist8m	8	784	-
normal	12 (largest)	100	100
embedded normal	819 (largest)	2048	10

Table 3.2
Dataset characteristics. '-' means the intrinsic dimensionality is unknown. For the synthesized normal and embedded normal datasets, only the largest size are listed.

p	Time (s)			GFLOP / s		
	$d = 8$	512	$8,192$	8	512	$8,192$
4	1.68	4.33	44.04	0.101	23.7	36.8
8	0.59	1.91	22.09	2.86	53.6	74.2
16	0.37	1.04	11.22	4.57	98.8	146
32	0.15	0.48	5.856	11.7	212	280
64	0.08	0.26	2.932	20.3	400	559
128	0.05	0.14	1.595	33.8	729	1027

Table 3.3
Strong scaling of direct KNN with 2d partitioning. *The time required to complete a query and the floating-point performance of the* KNN *calculation on Kraken. A square partitioning is used and the total number of reference and query points is fixed at* $m = n = 100,000$. *One process per socket was used, with 6 OpenMP threads.*

	$p = 4$	16	4096	$16,384$
Time (s)	27.85	27.83	27.96	27.72
TFLOP / s	0.03	0.11	28.0	111

Table 3.4
Weak scaling of direct KNN with 2d partitioning. *The time required to complete a query and the floating-point performance of the* KNN *calculation on Kraken. A square partitioning is used and the number of points per block partition is fixed at* $m = n = 5000$ *reference and query points per MPI process. In all runs,* $d = 1000$. *One process per socket was used, with 6 OpenMP threads.*

We test the scalability of both an all nearest neighbors (the reference points and the query points are the same), and a query set that is significantly larger than the reference set.

The results of the weak scaling tests are summarized in Table 3.5. The algorithm exhibits nearly perfectly linear weak scaling, which indicates that even for a relatively small problem size per node, the cost of communication is hidden by overlapping it with computation. In fact, for the largest run, the code sustains roughly 40% of peak floating-point performance without any low-level optimization.

p	$m = n = 1000$		$m = 100000, n = 100$	
	Time (s)	TFLOP/s	Time (s)	TFLOP/s
96	4.58	0.44	49.1	0.38
192	9.30	0.80	97.7	0.76
384	18.4	1.61	195	1.53
768	36.9	3.23	403	2.96
1,536	73.9	6.45	815	5.85
3,072	151	12.6	—	—
6,144	303	25.2	—	—
12,288	604	50.5	—	—

TABLE 3.5
Weak scaling of direct KNN with cyclic algorithm. *The time required to complete a query and the floating-point performance of the* KNN *calculation on Kraken. In all runs we used $d = 100$. We use one process per core. Here m and n indicate the number of points per MPI process.*

3.2. KNN using PKDT. We have evaluated the KNN performance and scalability of our tree-based approach on the Kraken platform using different datasets.

We first compared $\texttt{PKDT}_H$ and $\texttt{PKDT}_C$. For the exact tree search, the most important evaluation is how many points (or leaf nodes) a query should search. In the worst case a query visits all leaf nodes and no pruning takes place. In the best case, perfect pruning, every query visits only one leaf node. To measure the efficiency of the pruning, we define the pruning percentage as

$$\text{prune}_r\% = \frac{N - n_r}{N - N/p} \tag{3.1}$$

where N is the total number of query points among all processes; n_r is the number of query points on a single process r; p is the number of processes.

dataset	hyperplane partition (100%)			clustering grouping (100%)		
	min	max	**avg**	min	max	**avg**
US census	45.90	99.98	**89.92**	32.87	99.87	**63.83**
dynamical system	97.76	99.97	**98.99**	0.6639	93.49	**15.39**
gabor wavelets	96.85	100.00	**99.01**	54.08	100.09	**88.24**
5d-100D normal	99.66	99.84	**99.75**	4.17	97.95	**57.74**
10d-100D normal	83.96	94.32	**89.82**	0	3.38	**0.5456**

TABLE 3.6
Pruning effect of exact tree search on different datasets. *The pruning is obtained using 5 different datasets. The US census run uses 9,100 reference points and 500 query points per process, and totally 256 processes. For the remaining four runs we use 10,000 reference points and 500 query points per process, for a total of 1,024 MPI processes. The maximum, minimum and average pruning percentage across all processes are reported.*

Table 3.6 shows the pruning percentage of both $\texttt{PKDT}_H$ and $\texttt{PKDT}_C$ on 5 different datasets. Generally speaking, all those 5 datasets has lower intrinsic dimensionality than their ambient dimensions. As a result, even the dimension is high, there are some pruning that could be obtained. We could find the clustering partition has less

pruning than the hyperplane partition on all the 5 datasets. On the other hand, unlike $\texttt{PKDT}_H$ which has a perfect load balance, it is very difficult to maintain the load balance for $\texttt{PKDT}_C$. Since clustering is impossible to geneate clusters that all have exact the same size, there is no guarantee then each kid has the same number of reference points. All in all we prefer the hyperplane partition strategy, and in the following scaling test, we only present the scalability of $\texttt{PKDT}_H$.

	$p =$	192	384	768	1536	3,072	6,144	12,288
5d	const. (s)	0.81	0.99	1.27	1.46	1.69	2.44	2.80
	query (s)	3.82	4.25	5.10	5.81	13.07	18.92	28.42
	% prune	98.86	99.40	99.68	99.83	99.91	99.95	99.97
10d	const. (s)	0.93	0.96	1.26	1.83	2.07	3.99	3.14
	query (s)	37.61	55.92	81.57	115.72	162.11	267.27	379.75
	% prune	75.04	82.41	87.99	91.96	94.74	96.54	97.56

TABLE 3.7
Weak scaling of $\texttt{PKDT}_H$ for KNN ($k = 2$) using the normal distribution data. *The time (in seconds) required for tree construction and query on Kraken with a fixed number of points per process. 10,000 reference points and 1000 query points per process were used for the 100-dimensional runs (points are sampled from a 5-dimensional normal distribution, and embedded into 100-dimensional space.) We use one MPI process per core.*

We evaluate the weak scaling performance of $\texttt{PKDT}_H$ for KNN problem on 128 to 24,576 cores (64 to 2048 nodes) on Kraken. In this experiment, we use a single MPI process per core and disable multi-threading within processes. Even through we do achieve good performance with hybrid parallelism, this decision was made to enable measuring the scalability to a larger number of independent processes. Performance is summarized in Table 3.7. We see that the construction time required to partition and redistribute the points grows very slowly as a function of problem size and process count. However, for the query points, there is some overlap in the bounding regions of the clusters, some number of query points will be replicated to multiple kids in the tree traversal. Since we test the weak scaling using data sampled from a normal distribution, as the process count increase, the density of data points becomes more and more large. Thus, it is likely that at the very beginning, there is a big overlap near the hyperplane. As going to deeper level, the overlap might be reduced. The query time increase quickly as the process count increases. This is mainly because as the process count increases, although the pruning percentage also is improved, the number of query points becomes larger on each leaf node, which means we should perform the direct nearest neighbor search kernel in a larges scale of data points. For example, when $p = 12288$, the maximum number of query points on a single leaf node is 417065, compared with 65951 for $p = 192$, which is 6.3 times larger. And to redistribute large amount of points also takes longer time. Generally speaking, the pruning degrades since the point density of the generated dataset increases rapidly as the problem size is increased.

3.3. ANN using randomized PKDT. In our first experiment, we study the overall performance of randomized PKDT in terms of accuracy, convergence, and wall-clock time. We use five real datasets in Table 3.8 to solve the all nearest neighbors problem, i.e., the reference and query set are the same. Such runs are typical in learning tasks as part of cross-validation (e.g., to decide how many neighbors to use for a supervised classification problem).

To measure accuracy of randomized PKDT, we use the *hit rate* and *mean relative error*. Hit rate, "hit", is defined as

$$\text{hit} = \frac{1}{nk}\sum_{i=1}^{n}\left|\left\{\mathcal{N}_i^{true}\right\}_{j=1}^{k} \cap \left\{\mathcal{N}_i^{found}\right\}_{j=1}^{k}\right| \tag{3.2}$$

where n is the number of points, k is the number of requested neighbors, $\mathcal{N}_i^{true}$ is the set of true neighbors of point i, and $\mathcal{N}_i^{found}$ is the set of neighbors found by the approximate approach.

$$\text{error} = \frac{1}{N}\sum_{i=1}^{N}\frac{\sum_{j=1}^{k}\left|\|\mathbf{q}_i - \mathbf{r}_j^{true}\| - \|\mathbf{q}_i - \mathbf{r}_j^{found}\|\right|}{\sum_{j=1}^{k}\|\mathbf{q}_i - \mathbf{r}_j^{true}\|} \tag{3.3}$$

The mean relative error is the mean of the relative difference between the distances returned from an approximate search and the distances to the exact nearest neighbors. The reason for using both metrics is that for a given query point q, an approximate search may not return the exact nearest neighbor, but it may return a point that is almost imperceptibly more distant from q than is the exact nearest neighbor. Because of the prohibitive cost of computing an exact KNN solution for large data sets, we compute both accuracy metrics for a random sample of $\mathcal{O}(\log n)$ points.

The maximum iteration number is 100 and the termination threshold of convergence is 99% for hit rate and 1E-4 for relative error. The leaf node size of PKDT is always $2k$. Usually the real data sets do have low intrinsic dimension. As a result, even if the data have a high ambient dimension, randomized PKDT can still converge fast. For the "covtype" and "gas sensor" sets, roughly 20 iterations are enough to converge to a hit rate of 99%. For the "gas sensor", $k = 2048$ PKDT only requires $0.9\%(15 \times 4096/6709412)$ evaluations compared to a direct search. (Notice that some of these evaluations are repeated, since leaf nodes across different iterations may overlap.) However, for the "higgs" (a high-energy physics dataset) and "mnist8m" (a handwritten character recognition dataset), 100 iterations can only converge to 90% hit rate. It is usually acceptable for a very large k, especially the relative errors is small (3 digits at least). Another interesting observation is that "mnist8m" has a much large ambient dimensionality than the "higgs" (although "higgs" is a larger set). However using the same number of iterations, "mnist8m" converges even faster an indication that the dimensinoality is much lower than the ambient dimension 784. One advantage of randomized PKDT over other algorithms is that there is no tree traversal for the all nearest neighbors search problem. At each iteration only one leaf node is visited, hence the time for different iterations is exactly the same. In Table 3.8, T_i is the time of each iteration, and T_{total} is the total time of all iterations. The reason that T_i's for different k are different is because once the distances are calculated, the complexity to find nearest neighbors for each query is $\mathcal{O}(n \log k)$; to merge the nearest neighbors with last iteration, all the nearest neighbors should be shuffled back to the original process where the query points locate, the message size of each query is $\mathcal{O}(k)$.

The strong scaling performance of randomized PKDT using "mnist8m" is illustrated in Table 3.9. We take 10 iterations and do not report the accuracy. The construction time of PKDT scales up to 8192 cores (1024 processes). For 16,384 cores, each process only has less $8.1 \times 10^6/2048 \approx 3955$ points and there is no enough parallelism during building the tree while the communication percentage increases with respect to grain size. Yet, for all the runs, the query time scales almost linearly with the local problem

dateset			kNN					
name	N	d	k	iter	hit rate	error	T_i	T_{total}
covtype	500,000	54	512	16	99.16%	6.4e-4	0.69	12
			1024	16	99.01%	4.2e-4	1.37	24
			2048	21	99.01%	1.7e-4	2.86	62
gas sensor	6,709,412	16	512	21	99.11%	5.4e-4	8.12	180
			1024	18	99.05%	6.1e-4	16.67	312
			2048	15	99.04%	7.2e-4	33.36	531
susy	4,500,000	18	512	60	99.15%	4.6e-4	5.27	320
			1024	55	99.01%	4.8e-4	10.48	584
			2048	51	99.03%	4.8e-4	21.62	1120
higgs	10,500,000	28	512	100	80.47%	1.0e-2	12.59	1264
			1024	100	83.90%	7.8e-3	24.98	2518
			2048	100	88.08%	5.3e-3	50.32	5054
mnist8m	8,100,000	784	512	100	86.37%	6.4e-3	18.93	1904
			1024	100	88.92%	4.8e-3	29.92	2999
			2048	100	91.21%	3.4e-3	53.63	5413

TABLE 3.8

Accuracy and time on several large real datasets of randomized PKDT. *All the experiments are run on Maverick using 32 nodes, one MPI process per socket. 'T_i' is the time for each iteration, 'T_{total}' is the total time spent. The maximum iteration number is set to be 100, and once the hit rate touches* 99%, *the iteration will terminate automatically. Also in all experiments, we construct the tree so that each leaf node has no more than* $2k$ *points, where* k *is the number of nearest neighbors. Notice that this gives an indication of the overall distance evaluations. For example for the* $k = 512$ *"mnist8m" run, we take 100* PKDT *iterations, which in turn means that we perform 102,400 distance evaluations per query point or 1.2% of evaluations required in a direct evaluation. For the* $k = 2046$ *this number increases to 5%. For the "gas sensor" dataset, we see that in all cases less than 1% evaluations end up giving almost exact solutions, and indication that the intrinsic dimensionality is really small.*

size $\mathcal{O}(nd/p)$.

cores		1,024	2,048	4,096	8,192	16,384
MNIST8M	construction	51.75	31.67	18.99	12.11	18.55
	query	542.71	257.48	132.70	71.67	48.11
	speedup	1	2.06	3.92	7.10	8.92

TABLE 3.9

Strong Scaling of PKDT on Maverick: *Strong scaling of 10 iterations of* PKDT *on Maverick for the "mnist8m" dataset. In this run, k is 2048, the maximum leaf node size is then 4096. 'construction' stands for tree construction time of the tree in total 10 iterations, and 'query' is the total querying time. The point-wise data exchange is applied in these runs.*

Next we study the weak scaling performance of randomized PKDT. First of all, we want to compare three difference data exchange mechanism discussed in §2.3.2. For the purpose of extensively comparison, we tested only the tree construction part on three different machines: Lonestar, Kraken and Jaguar. We use the metric *millions of cycles / point / core* to compare across platforms, i.e., the number of clock cycles that would be needed to perform the query for each point using only a single core.

On Lonestar (Table 3.10), the lower bandwidth of interconnect combined with its less-tuned MPI stack clearly shows the differences in the scalability of the three methods. the whole tree exchange performs well at small scales, but point-wise data

exchange shows the best overall scalability. Weak-scaling results for point-wise data exchange and the whole tree exchange on Kraken are presented in Table 3.11. Here too, the whole tree exchange performs well at small p but does not scale well as we increase the processes; however, point-wise exchange maintains very good efficiency up to 96K cores. Table 3.12 presents weak-scaling results for all three data exchange on Jaguar. On Jaguar, the differences between the three construction algorithms are less marked, with the exception of the fact that whole tree exchange also fails to scale to large p. Overall speaking, the whole tree exchange provides the best performance at small p but does not scale well at large p. Point-wise exchange exhibits slightly better performance and scalability than all-to-all exchange. It is not currently clear why point-wise exchange fails to maintain the same level of parallel efficiency on Jaguar that it exhibits on Kraken at comparable core counts. Further investigation is necessary to explain this behavior.

Tree: Lonestar weak scaling						
cores	192	384	768	1,536	3,072	6,144
ALL-TO-ALL EXCHANGE						
cycles	10.7	17.8	25.3	39.1	58.0	88.5
effic	100%	60%	42%	27%	18%	12%
POINT-WISE EXCHANGE						
cycles	6.9	8.4	9.8	11.3	13.0	15.2
effic	100%	83%	71%	62%	54%	46%
WHOLE TREE EXCHANGE						
cycles	5.7	7.3	8.4	10.2	12.6	17.6
effic	100%	78%	69%	56%	45%	32%

TABLE 3.10

Weak-scaling of Tree Construction on Lonestar. *This table shows the scalability of the three tree construction variants on Lonestar in terms of millions of cycles per point per core and the efficiency relative to the 192-core run. We use one process per socket with 6 OpenMP threads. We use 10K points per process in 2,048 dimensions.*

Tree: Kraken weak scaling							
cores	1,536	3,072	6,144	12,288	24,576	49,152	98,304
POINT-WISE EXCHANGE							
cycles	18.0	17.9	21.2	25.8	30.1	36.0	43.1
effic	100%	100%	85%	70%	60%	50%	42%
WHOLE TREE EXCHANGE							
cycles	14.0	12.5	20.7	31.7	59.3	-	-
effic	100%	112%	68%	44%	24%	-	-

TABLE 3.11

Weak-scaling of Tree Construction on Kraken. *This table shows the scalability of point-wise data exchange and the whole tree exchange on Kraken in terms of millions of cycles per point per core. The efficiency relative to the 1.5K core run. We use 10K points per process in 2,048 dimensions.*

Finally, we illustrate the weak scaling performance of the randomized `PKDT` on Stampede and Jaguar. The data exchange method is the point-wise exchange for all runs. On stampede, we use 1024 normal distribution data; each process has 50,000 points; one process per socket with 8 threads. We test $k = 2048$ in this run. In Figure 3.1, it is clear that the query part is the most expensive portion since selecting

Tree: Jaguar weak scaling							
cores	2,048	4,096	8,192	16,384	32,768	65,536	131,072
ALL-TO-ALL EXCHANGE							
cycles	11.1	16.4	20.3	27.3	33.5	46.1	51.1
effic	100%	68%	55%	41%	33%	27%	22%
POINT-WISE EXCHANGE							
cycles	9.8	12.5	15.8	22.9	32.4	38.8	49.8
effic	100%	78%	62%	43%	30%	25%	20%
WHOLE TREE EXCHANGE							
cycles	5.3	8.1	10.7	25.5	34.2	132.3	-
effic	100%	66%	50%	21%	16%	4%	-

TABLE 3.12
Weak-scaling of Tree Construction on Jaguar. *This table shows the scalability of the three tree construction variants on Jaguar in terms of millions of cycles per point per core and the efficiency relative to the 2048-core run. We use one process per socket with 8 OpenMP threads. We use 10K points per process in 2,048 dimensions.*

k neighbors requires maintaining a maximum heap of size k as well as merging k neighbors from the last iteration for each point. The construction part is linear with respect to the number of processes, which is consistent with the analysis in §2.3.2. The query part is almost constant because the local problem size is the same due to the load balanced data exchange mechanism. Overall if k is relative large, `PKDT` can get nearly constant weak scaling performance. For the small k case, we test the all nearest neighbors case on Jaguar using 2048 dimensional normal data; each process has 50,000 points; one process per socket with 8 threads. In the case of Figure 3.2 $k = 2$, the tree construction dominates the overall running time. Similarly, the query time is nearly constant but the construction time scales with respect to the number of processes.

We used the normalized cycles / points / core statistic to compare performances across different systems. For weak scaling, increasing numbers indicate communication overheads and load imbalance. Notice that algorithmically we show that the best-case complexity estimates include terms that grow with powers of $\log p$, linearly in p, or with factors that depend on $\log n$ that inherently limit the attainable efficiencies.

This is the first time that nearest neighbors solver has been scaled to this extent and a first attempt to characterize the parallel scalability of state-of-the-art approximate nearest neighbors methods in leadership architectures. However, there are some general conclusions that can be drawn. During the tree construction, the clear loser in three different data exchange technique is the whole tree exchange, but again there are regimes in which it performs well (small number of core counts). The interplay between latency, bandwidth, points per leaf node size suggests significant opportunities for performance tuning and optimization depending on the machine architecture, the dataset and its dimensionality.

3.4. Comparison with existing software packages. As we mentioned, no software offers the capabilities of our library. But just do indicate that even in the single node case, we outperform existing codes, we discuss a few examples. The first package we discuss is `MLPACK`, which is only supports exact searches, but does not have distributed memory parallelism. The larger dataset reported in [8] is has 1M

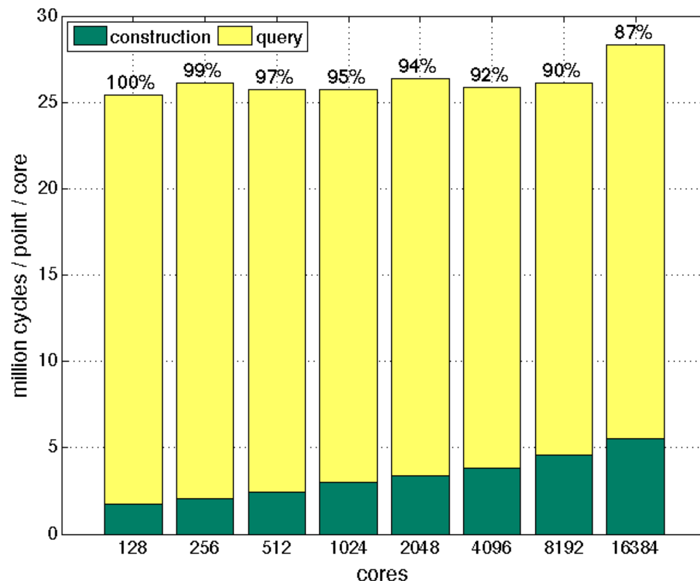


FIG. 3.1. Weak scaling of PKDTon Stampede: *Weak scaling of all nearest neighbors in 10 iterations of* PKDT *on Stampede. We use* 1024 *dimensional normal distribution data with* $k = 2048$*. 50,000 points per process are used; one process per socket with 8 threads. The percentages represent the relative efficiency of each run normalized to the 128-core run.*

points in 10 dimensions (the "randu" set) for KNN with $k = 3$. This test took 1020 secs on a 3.3 GHz AMD Phenom II X6 1100T processor (Table 1, [8]). Exact search with PKDT takes 18 secs on node of Maverick (with 99.9%) hit-rate accuracy. Even if we assume that the Phenom is 4× slower than the Ivy Bridge, PKDT is still over 10× faster than MLPACK. Notice that MLPACK significantly outperforms packages in existing high-level language packages like MATLAB, Scikit-Learn (Python) and WEKA (Java).

The second package we compare with is FLANN [30] a very popular approximate nearest neighbor algorithm also based on random KD-trees. The randomization is over the choice of coordinates to split, a different scheme than hours. FLANN supports multithreading. It also has rudimentary support for MPI by splitting the reference dataset across multiple MPI processes, performing the query for each subdataset and then merging the results sequentially. (Notice that the authors compared FLANN with the high-quality ANN library [27] used in the R package and found that FLANN is significantly faster). To compare FLANN we downloaded it and compiled it on the "Maverick" cluster using the same Intel compiler and flags as for our PKDT. We considered the all-KNN problem with a Gaussian distribution of points, with n =160K, $d = 32$, and $k = 32$. For FLANN we used the following parameters checks=2500, 8 trees and everything else to its default value. FLANN takes 155 seconds on the 20-core "Maverick" node to deliver 75% hit rate (estimated by testing 2000 points). For PKDT we used 16 iterations with 2000 points per leaf node and it required 20 secs, again for 75% hit rate (estimated by testing 2000 points). So PKDT is more than 7× faster than FLANN, without considering distributed memory parallelism. Also, notice that FLANN and MLPACK use single-precision arithmetic, whereas we use double-precision. So the actual speed-ups can be even higher. At any rate, none of the existing packages can

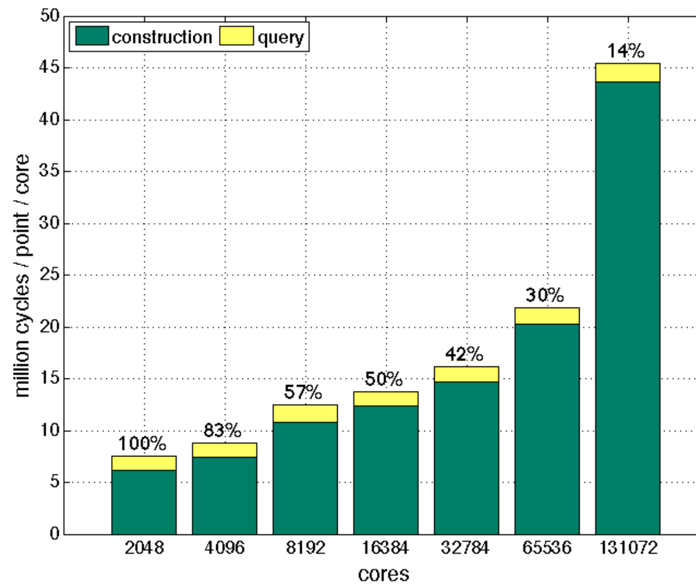


FIG. 3.2. Weak scaling of PKDTon Jaguar: *Weak scaling of all nearest neighbors in 8 iterations of* PKDT *on Jaguar. We use* 2048 *dimensional normal distribution data with* $k = 2048$. *50,000 points per process are used; one process per socket with 8 threads. The percentages represent the relative efficiency of each run normalized to the 2,048-core run.*

handle the large datasets one which we test PKDT.

4. Conclusion. We presented a set of algorithms for the parallel tree construction for points in high-dimensions, for exact nearest neighbor searches, and for approximate nearest-neighbor searches using greedy search on randomized KD-trees. We reported the accuracy and scalability of the scheme for synthetic and non-synthetic datasets across different clusters and problem sizes and we demonstrated unprecedented scalability. As mentioned in the introduction the software is at http://padas.ices.utexas.edu/matheme.

There are several things we did not have space to discuss. For example in [19] the idea of "supercharging" is discussed in which the greedy search is augmented by search on the nearest-neighbor graph. We implemented this in `PKDT` but we found, consistently, that in the distributed memory setting the procedure is not effective due to unpredictable memory loads and communication overheads: simply taking more iterations is faster and more robust. Also, we have implemented a parallel `LSH` algorithm. In a nutshell, similar good performance (in terms of accuracy and scalability) can be obtain with the LSH algorithm, however it requires significant parameter tuning for each dataset, whereas `PKDT` offers similar performance with no tuning parameters. The discussion of the comparison will be reported elsewhere.

Appendix A. Point Grouping Using kmeans Clustering with Randomized Seeds. In this section, we outline the point-grouping mechanism, the kmeans clustering. We use a standard kmeans clustering algorithm with the seeding introduced in Ostrovsky et al. [32]. Its main feature is that it reduces the number of iterations required to converge the kmeans algorithm and most importantly, removes the need for multiple invocation of kmeans with different seeds. For well-clusterable cases (e.g., mixtures of well-separated Gaussians), the new clustering can be orders of magnitude faster than uniformly randomly selecting seeds among the input points because it eliminates the need for repeated seed selection.

Parallelizing kmeans is straightforward [11]. Algorithm 11 describes the scheme. It is easy to see that the complexity per iteration is $\mathcal{O}(kd(\frac{n}{p} + \log p))$, where k is the number of clusters, n is the number of points, and p is the number of processors.

Algorithm 11 $\textsc{kMeans}(r, c, n, k)$

1: **for** $j = 1 \cdots n$ **do**
2: For each point r_j, assign cluster membership by finding the closest centers c_i
3: $c_i = \sum r_j, \quad r_j \in \text{cluster } i \ (V_i), \quad |V_i| = n_i$
4: $n_i = \textsc{Allreduce}(n_i)$
5: $c_i = \textsc{Allreduce}(c_i)$
6: $c_i = c_i / n_i$
7: **end for**

A.1. Seeding. To reduce the iterations of standard kmeans and obtain high quality clusters, seeds can be carefully selected. A choice of the initial centroids (*seeding*) with provable quality guarantees (under a quantitative assumption of clusterability) is discussed in [32]. Once the seeds have been computed, the ball-kmeans or the standard kmeans iteration can be used. The algorithm in [32] is based on the observation that k initial centroids that are far away from each other will belong to k different clusters. To find these points, we first oversample on probabilities based on the interpoint distances and then we eliminate *bad* seeds.

The oversampling step is summarized by Algorithm 12. First, we sample two

initial seeds (denote as s_1 and s_2) according to probabilities

$$p_j^1 = \frac{\left(d^2(r_j,\bar{r})n + \sum_j d^2(r_j,\bar{r})\right)}{\left(2n\sum_j d^2(r_j,\bar{r})\right)} \tag{A.1}$$

$$p_j^2 = \frac{d^2\left(r_j,s_1\right)}{\left(\sum_j d^2(r_j,\bar{r}) + nd^2(\bar{r},s_1)\right)} \tag{A.2}$$

where n is the number of points, $\bar{r}$ is the global mean of all points r_j, s_1 is the first selected seed, and $d(\cdot,\cdot)$ is the distance between two points.

Then, for the remaining points, the sampling probability is given by

$$p_j = \frac{d^2(r_j,s_*^j)}{\sum_{r_j} d^2(r_j,s_*^j)} \tag{A.3}$$

where $d(r_j,s_*^j)$ is the distance between point r_j and its nearest seed s_*^j, which has already been chosen. Finally, a randomly sampled point based on the probability from the unchosen set is used until a new seed is added. We repeat these steps until we have chosen k seeds.

Algorithm 12 ADDSEEDS(r,s,k)

1: **if** ISEMPTY(s) **then**
2: sample the first two seeds s_1 and s_2
3: **end if**
4: **if** $|s| \geq k$ **then** return
5: $d(r_j,s_*^j) = \min_{s_i} \mathrm{d}(s_i,r_j)$
6: compute sampling probabilities $p = \{p_j\}$ according to Eq. A.3
7: $s' =$ RANDOM_SAMPLE$(r/\{s\},p)$
8: $s \leftarrow s \cup s'$
9: ADDSEEDS$(r/\{s\},s,k)$

After oversampling k' seeds where $k' > k$, We eliminate these seeds one by one to exclude those which generate bad clustering quality, until finally we have k centers. The seed elimination step is described in Algorithm 13. In more detail, we evaluate the clustering quality of $k'-1$ seeds $\{s_i\}_{i=1}^{k'}\backslash s_j$ for each s_j, then delete the one with the lowest quality. The clustering quality is measured by the kmeans loss $\mathcal{L}$,

$$\mathcal{L}_k(X) = \frac{1}{|X|}\sum_{i=1}^{k}\sum_{x\in V_i}\|x-s_i\|^2 \tag{A.4}$$

where V_i is the Voronoi set of seed s_i. If the data is ϵ-separated, which implies $\mathcal{L}_k(X)/\mathcal{L}_{k-1}(X) \leq \epsilon^2$, [32] suggests oversampling $k' = \frac{2k}{1-5\sqrt{\epsilon}} + \frac{2\log(2/\sqrt{\epsilon})}{(1-5\sqrt{\epsilon})^2}$ points is sufficient to obtain good results. In practice, without any prior knowledge about the ϵ, it is possible to estimate it by using a small subset of the input data.

***k*-Means seeding performance.** We compare the performance of two types of seeding approaches for kmeans. The first one is the standard uniformly randomly

Algorithm 13 ELIMINATESEEDS(r, s, k)

1: **if** $|s| \leq k$ **then** return
2: **for** $i = 1 : |s|$ **do**
3: $n_i = |V_i|$ for s_i
4: $s^{(i)} \leftarrow s \backslash s_i$
5: compute the kmeans loss $\mathcal{L}^{(i)}$ of $s^{(i)}$
6: $T_i = n_i \mathcal{L}^{(i)}$
7: **end for**
8: $s^* = \min_s T_i$
9: $s \leftarrow s \backslash s^*$
10: **for** $i = 1 : |s|$ **do**
11: find V_i for each s_i
12: $s_i \leftarrow mean(V_i)$
13: **end for**
14: ELIMINATESEEDS(r, s, k)

selection of seeds among the input points; the other is the Ostrovsky seeding.

Experiments run on the synthetic data, a mixture of eight Gaussians with unit variance. Each center of these Gaussians is located on a vertex of a hypercube. Each run was repeated 100 times. Clustering quality is measured by the loss function Eqn. A.4 and the variance ratio $\mathcal{V}$

$$\mathcal{V}(X) = \max \frac{\sum_{i=1}^{k} p_i \|c_i - c\|^2}{\sum_{i=1}^{k} p_i \frac{1}{|V_i|} \sum_{x \in V_i} \|x - c_i\|^2}, \quad (A.5)$$

where $p_i = \frac{\|V_i\|}{\|X\|}$ and c is the center of the whole dataset X. The higher the variance ratio, the better clustering quality is.

D	k	**seeding**	$\mathcal{L}(X)$	$\mathcal{V}(X)$	**iters**	**accuracy**
3	4	random	4. 11±0.061	1.15±0.032	11.28±3.64	45.81%±0.88%
		Ostrovsky	4.10±0.046	1.16±0.024	11.01±4.33	45.61%±0.63%
	8	random	1.95±0.160	3.57±0.330	11.03±3.75	87.12%±5.65%
		Ostrovsky	1.97±0.171	3.51±0.361	8.02±3.57	86.30%±6.01%
6	4	random	11.20±0.421	1.60±0.083	7.91±2.85	49.91%±0.06%
		Ostrovsky	11.09±0.008	1.62±0.002	6.41±2.69	49.93%±0.04%
	8	random	3.55±1.216	8.34±3.528	7.44±2.29	91.25%±7.13%
		Ostrovsky	2.16±0.368	12.69±1.203	3.17±0.88	99.50%±2.16%

TABLE A.1

Clustering Quality: *The clustering quality is measured by the kmeans loss, variance ratio, number of kmeans iterations (***iters***), and the clustering accuracy (***accuracy***). We use a mixture of 8 Gaussians. Here D is the distance between each pair of Gaussian centers. k indicates the number of target clusters (the best k is eight).*

When the length of the hypercube edges is 6, the Gaussians are relatively well separated, it is clear that the Ostrovsky seeds results in a better clustering than random selected seeds. First, the loss function and variance ratio indicate better quality. Second, the standard deviations of these measures are much smaller than

those of random seeds, which implies the Ostrovsky seeds are more stable and can be expected to always produce good clustering results. Third, the Ostrovsky seeds require less iterations to converge. We also test the case of edge length 3. In this case, there is no significant difference between two types of seedings. Generally speaking, only the Ostrovsky seeding can be "safely" used without need for repeated seeding, which is expensive as it requires collective communication. On the other side, the Ostrovsky seeds could also be a good clusterability heuristic. The loss function and the variance ratio will vary slightly when k is close to the number of data's intrinsic groups. In Figure A.1, it is clear that the slope of the Ostrovsky seeding curve changes sharply when $k = 8$, which is the number of Gaussians we used. As a counterpart, it is less pronounced using uniformly random selected seeds.

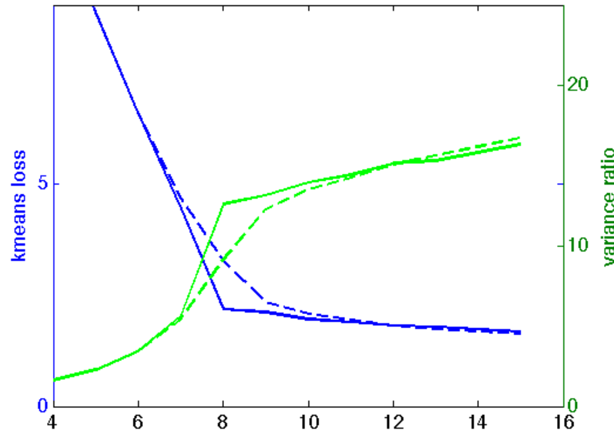


Fig. A.1. Clusterability: *The change in the value of the loss and the variance ratio is an indication for the intrinsic number of clusters existing in the data. The blue lines stand for the kmeans loss and the green lines stand for the variance ratio. The solid lines are the Ostrovsky seeds, the dashed lines indicate random seeds. The x-axis is the number of clusters.*

Appendix B. Clustering Tree. We briefly outline the construction of the clustering tree. The node splitting algorithm works as follows:

Let $\mathcal{T}$ be a tree node, $X_{\mathcal{T}}$ be all the points assigned in $\mathcal{T}$, $l_{\mathcal{T}}$ be the level of the node $\mathcal{T}$ in the tree, and $s_{\mathcal{T}}$ be the size of node $\mathcal{T}$'s communicator. Let i be a cluster of points with centroid c_i and radius R_i, $\mathcal{B}_{c,R}$ be a closed ball in $\mathbb{R}^d$ centered at c with radius R, $\mathcal{C}_i$ be the tree node that owns group i, and $\mathcal{N}_c$ be the number of clusters created in each split. Algorithm 14 describes the node-splitting procedure.

As outlined in Algorithm 14, we use a distributed top-down algorithm to construct the tree and, at each level, split a node to two children, each of which contain one or more groups of points. For assigning points to groups, we use our distributed kmeans routine. Except that, the remaining point repartition and communicator splitting procedures are similar to the `PKDT`.

Finally we sketch the complexity of the tree construction algorithm, assuming a uniform tree. Let l be the level of a node. Computing the clusters takes $\mathcal{O}(d(n_l/p_l + \log p_l))$ time, where $n_l = n/2^l$ are the points per node at the lth level and $p_l = p/2^l$ are the number of processors per node at the lth level. Furthermore, the collective to

Algorithm 14 NODESPLIT($\mathcal{T}$)

1: **if** $l_\mathcal{T}$ ==maxLevel || $|X_\mathcal{T}|$ < minNumofPoints **then**
2: return;
3: **else**
4: ADDSEEDS($X_\mathcal{T}, seeds, \mathcal{N}_c$)
5: ELIMINATESEEDS($X_\mathcal{T}, seeds, \mathcal{N}_c$)
6: centroids = KMEANS($X_\mathcal{T}, seeds, \mathcal{N}_c$)
7: **for** each point $x \in X_\mathcal{T}$ **do**
8: Assign x to cluster i such that $d(x, c_i)$ is minimized
9: **end for**
10: Assign clusters to children such that each child has roughly the same amount of points
11: Assign processes for each child $\mathcal{C}_i$
12: Split $\mathcal{T}$'s communicator, creating a new communicator for each $\mathcal{C}_i$
13: Distribute points in each cluster i to processes in $\mathcal{C}_i$
14: NODESPLIT($\mathcal{C}_{local}$)
15: **end if**

redistribute points within the node takes $\mathcal{O}(\frac{n_l}{p_l^2} p_l)$ time (ignoring latency). From these observations we can see that the overall complexity of the tree construction (assuming a uniform tree) is $\mathcal{O}(d\frac{n}{p}\log^2 p)$.

Appendix C. Distributed Select. To find the median value of an unordered array, we apply a *select* operation. SELECT(K) finds the kth smallest value in an array. For the median we use SELECT(N/2), where n is the total number of input elements. Algorithm 15 describes our distributed-memory select function. A more sophisticated

Algorithm 15 PARSELECT(arr, k)

1: N = ALLREDUCE($arr.size()$)
2: choose a pivot point x from arr at random
3: arr_less = [] arr_great = []
4: **for** each point i in input array arr **do**
5: **if** $arr[i] > x$ **then**
6: arr_less.insert($arr[i]$)
7: **else**
8: arr_great.insert($arr[i]$)
9: **end if**
10: **end for**
11: N_l = ALLREDUCE($arr_less.size()$)
12: N_g = ALLREDUCE($arr_great.size()$)
13: **if** $N == 1 \parallel N_l == k \parallel N_l == N \parallel N_g == N$ **then**
14: return x
15: **else if** $N_l > k$ **then**
16: PARSELECT(arr_less, k)
17: **else**
18: PARSELECT(arr_great, $k - N_l$)
19: **end if**

algorithm and more discussion can be also found in [43].

REFERENCES

[1] Mathematics of analysis of petascale data. Technical report, DOE Office of Science, 2008.

[2] Dror Aiger, Efi Kokiopoulou, and Ehud Rivlin. Random grids: Fast approximate nearest neighbors and range searching for image search. In *Computer Vision (ICCV), 2013 IEEE International Conference on*, pages 3471–3478. IEEE, 2013.

[3] 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(2):136–148, 2000.

[4] A. Andoni and P. Indyk. E2LSH 0.1 User manual, 2005. http://www.mit.edu/~andoni/LSH.

[5] A. Andoni and P. Indyk. Near-Optimal Hashing Algorithms for Approximate Nearest Neighbor in High Dimensions. *COMMUNICATIONS OF THE ACM*, 51(1):117, 2008.

[6] Alina Beygelzimer, Sham Kakade, and John Langford. Cover trees for nearest neighbor. In *ICML*, pages 97–104, 2006.

[7] P. Ciaccia, M. Patella, and P. Zezula. M-tree an efficient access method for similarity search in metric spaces. *Proceedings of the 23rd VLDB Conference*, pages 426–435, 1997.

[8] Ryan R. Curtin, James R. Cline, Neil P. Slagle, William B. March, P. Ram, Nishant A. Mehta, and Alexander G. Gray. MLPACK: A scalable C++ machine learning library. *Journal of Machine Learning Research*, 14:801–805, 2013.

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

[10] Karen D. Devine, Erik G. Boman, and George Karypis. Partitioning and load balancing for emerging parallel applications and architectures. In M. Heroux, A. Raghavan, and H. Simon, editors, *Frontiers of Scientific Computing*. SIAM, Philadelphia, 2006.

[11] I. Dhillon and D. Modha. A data-clustering algorithm on distributed memory multiprocessors. *Large-Scale Parallel Data Mining*, pages 802–802, 2000.

[12] J. H. Freidman, J. L. Bentley, and R. A. Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Trans. Math. Softw.*, 3:209–226, 1977.

[13] J. Friedman, T. Hastie, and R. Tibshirani. *The elements of statistical learning*, volume 1. Springer Series in Statistics, 2001.

[14] K. Fukunaga and P. M. Narendra. A brach and bound algorithm for computing k-nearest neighbors. *IEEE Trans. Comput.*, 24:750–753, 1975.

[15] B. Ganapathysubramanian and N. Zabaras. A non-linear dimension reduction methodology for generating data-driven stochastic input models. *Journal of Computational Physics*, 227(13):6612–6637, 2008.

[16] V. Garcia, E. Debreuve, and M. Barlaud. Fast k nearest neighbor search using GPU. 2008.

[17] Fabian Gieseke, Justin Heinermann, Cosmin Oancea, and Christian Igel. Buffer k-d trees: Processing massive nearest neighbor queries on GPUs. In *Proceedings of the 31st International Conference on Machine Learning (ICML)*, volume 32 of *JMLR W&CP*, pages 172–180. JMLR.org, 2014.

[18] A.G. Gray and A.W. Moore. N-Body'problems in statistical learning. *Advances in neural information processing systems*, pages 521–527, 2001.

[19] P.W. Jones, A. Osipov, and V. Rokhlin. Randomized approximate nearest neighbors algorithm. *Proceedings of the National Academy of Sciences*, 108(38):15679–15686, 2011.

[20] D. Karger and M. Ruhl. Finding nearest neighbors in growth restricted metrics. *Proceeding of STOC*, 2002.

[21] I. Lashuk, A. Chandramowlishwaran, T-A. Nguyen H. Langston, R. Sampath, A. Shringarpure, R. Vuduc, D. Zorin L. Ying, and G. Biros. A massively parallel adaptive fast-multipole method on heterogeneous architectures. In *SC '09: Proceedings of the 2009 ACM/IEEE conference on Supercomputing*, pages 1–12, Piscataway, NJ, USA, 2009. IEEE Press.

[22] M. Lichman. UCI machine learning repository, 2013.

[23] Gaelle Loosli, Stephane Canu, and Leon Bottou. Training invariant support vector machines using selective sampling. In *Large Scale Kernel Machines*, 2007.

[24] William B. March, Bo Xiao, and George Biros. ASKIT: Approximate skeletonization kernel-independent treecode in high dimensions. *SIAM Journal on Scientific Computing*, 37(2):1089–1110, 2015.

[25] 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 Computing Symposium, Hyderabad, India, May 2015. acceptance rate 107/495 (21.8%).

[26] G.L. Miller, S.H. Teng, W. Thurston, and S.A. Vavasis. Separators for sphere-packings and

nearest neighbor graphs. *Journal of the ACM (JACM)*, 44(1):1–29, 1997.

[27] D.M. Mount and S. Arya. Ann: A library for approximate nearest neighbor searching. In *CGC 2nd Annual Fall Workshop on Computational Geometry*, 1997. www.cs.umd.edu/~mount/ANN/.

[28] Marius Muja and David G. Lowe. Fast approximate nearest neighbors with automatic algorithm configuration. In *International Conference on Computer Vision Theory and Application VISSAPP'09)*, pages 331–340. INSTICC Press, 2009. www.cs.ubc.ca/~mariusm/index.php/FLANN/FLANN.

[29] Marius Muja and David G. Lowe. Fast approximate nearest neighbors with automatic algorithm configuration. In *In VISAPP International Conference on Computer Vision Theory and Applications*, pages 331–340, 2009.

[30] Marius Muja and David G Lowe. Scalable nearest neighbor algorithms for high dimensional data. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 36(11):2227–2240, 2014.

[31] S. M. Omohundro. Efficient algorithms with neural network behavior. *J. of Complex Systems*, 2:273–347, 1987.

[32] R. Ostrovsky, Y. Rabani, L.J. Schulman, and C. Swamy. The effectiveness of Lloyd-type methods for the k-means problem. In *Foundations of Computer Science, 2006. FOCS'06. 47th Annual IEEE Symposium on*, pages 165–176. IEEE, 2006.

[33] J. Pan, C. Lauterbach, and D. Manocha. Efficient nearest-neighbor computation for GPU-based motion planning. In *Intelligent Robots and Systems (IROS), 2010 IEEE/RSJ International Conference on*, pages 2243–2248. IEEE, 2010.

[34] Jia Pan and Dinesh Manocha. Fast gpu-based locality sensitive hashing for k-nearest neighbor computation. *ACM SIGSPATIAL GIS*, pages 211–220, 2011.

[35] Krauthgamer R. and J. Lee. Navigating nets: Simple algorithms for proximity search. *Proceedings of the 15th Annual Symposium on Discrete Algorithms (SODA)*, pages 791–801, 2004.

[36] A. Rahimian, I. Lashuk, S.K. Veerapaneni, A. Chandramowlishwaran, D. Malhotra, L. Moon, R. Sampath, A. Shringarpure, J. Vetter, R. Vuduc, D. Zorin, and G. Biros. Petascale direct numerical simulation of blood flow on 200k cores and heterogeneous architectures. In *SC '10: Proceedings of the 2010 ACM/IEEE conference on Supercomputing*, pages 1–12, Piscataway, NJ, USA, 2010. IEEE Press.

[37] P. Ram, D. Lee, W. March, and A. Gray. Linear-time algorithms for pairwise statistical problems. *Advances in Neural Information Processing Systems*, 23, 2009.

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

[39] S.T. Roweis and L.K. Saul. Nonlinear dimensionality reduction by locally linear embedding. *Science*, 290(5500):2323–2326, 2000.

[40] H. Samet. *Foundations of multidimensional and metric data structures*. Morgan Kaufmann, 2006.

[41] Bernhard Schölkopf and Alexander J Smola. *Learning with kernels: support vector machines, regularization, optimization, and beyond*. MIT press, 2002.

[42] N. Sismanis, N. Pitsianis, and X. Sun. Parallel search of k-nearest neighbors with synchronous operations. *IEEE Conference on High Performance Extreme Computing*, pages 1–6, 2012.

[43] Hari Sundar, Dhairya Malhotra, and George Biros. HykSort: A new variant of hypercube quicksort on distributed memory architectures. In *ICS'13: Proceedings of the International Conference on Supercomputing, 2013*, 2013.

[44] J. Tenenbaum, V. Silva, and J. Langford. A global geometric framework for nonlinear dimensionality reduction. *Science*, 290(5500):2319–2323, 2000.

[45] A. Torralba, R. Fergus, and W.T. Freeman. 80 million tiny images: A large data set for nonparametric object and scene recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pages 1958–1970, 2008.

[46] J. K. Uhlmann. Satisfying general proximity/similarity queries with metric trees. *Information Processing Letter*, 40:175–179, 1991.

[47] R. Weber, H.J. Schek, and S. Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *Proceedings of the International Conference on Very Large Data Bases*, pages 194–205. IEEE, 1998.