Parallel \( k \) Nearest Neighbor Graph Construction Using Tree-Based Data Structures

Nazneen Rajani  Kate McArdle

Abstract

Construction of a nearest neighbor graph is often a necessary step in many machine learning applications. However, constructing such a graph is computationally expensive, especially when the data is of high dimensionality. In this work, we focus on the use of two tree structures, \( k \)-d trees and ball trees, to implement nearest neighbor graph construction. We present parallel implementations of nearest neighbor graph construction using such tree structures, with parallelism provided by OpenMP and the Galois framework. Our results show that \( k \)-d trees are faster when the number of dimensions is small (\( N \gg 2^d \)), ball trees on the other hand scale well with the number of dimensions. Our Galois implementation with maxspread was faster than OpenMP for data that satisfied the \( k \)-d-tree requirement, irrespective of number of threads. OpenMP ball trees were faster than Galois \( k \)-d-tree for datasets with large dimension, irrespective of number of threads.

1 Introduction

Construction of a nearest neighbor graph is often a necessary step in data mining, computer vision, robotics, machine learning, and other research fields. The nearest neighbor or, in general, the \( k \) nearest neighbor (\( k \)NN) graph of a data set is obtained by connecting each instance in the data set to its \( k \) closest instances from the data set, where a distance metric defines closeness. However, this important step is often computationally expensive, especially when the data is of high dimensionality. Indeed, often one of the most computationally expensive parts of computer vision algorithms is finding the most similar matches to high-dimensional vector representations of images [12].

To compute a nearest neighbor graph of a large data set with high-dimensional features, known exact nearest-neighbor search algorithms usually do not provide acceptable performance [12]. To speed up the performance, many applications must settle for approximate \( k \) nearest neighbor graphs: construction of such graphs is designed to take less time but still provide a graph that is close to the exact solution. Many approximate nearest neighbor graph construction algorithms result in graphs with over 95% of the correct neighbors and in practice are two or more orders of magnitude faster than a brute force search [12].

In this paper, we focus on \( k \)-d trees and ball trees, popular tools used to construct both exact and approximate nearest neighbor graphs. \( k \)-d trees are data structures that organize points in a \( d \) dimensional space by dividing the space into several partitions [3]. Each \( d \)-dimensional point in a data set is represented by a node in the \( k \)-d tree, and every level of the tree splits the space along one of the \( d \) dimensions. Thus every node that is not a leaf node implicitly generates a hyperplane perpendicular to the dimension on which its level splits, and all nodes in the node’s left subtree fall to the left of the hyperplane, while all nodes in the node’s right subtree fall to the right of the hyperplane.

1While dimensionality regarding \( k \)-d trees is usually denoted by the letter \( k \), we use the letter \( d \) in this work, to remove potential confusion with \( k \) nearest neighbors.
A ball tree is a binary tree data structure that organizes points in multidimensional space. Each node owns a set of points such that the root node has full set of points in the dataset while each leaf node has some maximum points called leaf size. A non leaf node does not explicitly contain any points but it points to two children nodes such that \( \text{child1.points} \cap \text{child2.points} = \phi \) and \( \text{child1.points} \cup \text{child2.points} = \text{node.points} \). Each node has a pivot and a radius that determine how the points are split among the children nodes [11].

To use a \( k \)-d tree or a ball tree for nearest neighbor search, the tree must first be built, and then searched. One popular implementation of nearest neighbor search with such trees is found in Scikit-learn, an open-source machine learning library in Python\(^2\) \( k \)-d trees are commonly used for nearest neighbor search because \( O(\log N) \) operations are needed in the case of randomly distributed \( N \) points. However, in high-dimensional spaces, the curse of dimensionality causes the algorithm to need to visit many more branches than in lower-dimensional spaces. In particular, when the number of points is only slightly higher than the number of dimensions, the algorithm is only slightly better than a linear search of all of the points. If exact results are not needed from the nearest neighbor search, \( k \)-d trees can be used for an approximate search, by stopping the search early, for example after a given number of leaf nodes in the tree have been visited. The complexity for searching a ball tree depends on the leaf size of the tree. If the leaf size is big a lot of time will be spent doing linear search over the leaf nodes, however if the leaf size is small building the tree will take more time. So there is a tradeoff between building the tree time and linear search over leaf nodes.

In this work, we present implementations of exact nearest neighbor search using \( k \)-d trees and ball trees in parallel settings, namely OpenMP\(^3\) and the Galois system\(^4\), a graph-based framework that provides data-parallelism. With these implementations, users can take advantage of parallel \( k \)-NN graph construction as a first step in their machine learning work. Before describing in more detail our implementation in Section 4, we first provide an overview of related nearest neighbor algorithms in Section 2 and a more detailed background of \( k \)-d trees, ball trees, and the Galois framework in Section 3. In Section 5 we present our experimental results.

2 Related Work

Nearest neighbor graph construction has been widely studied and many variations of the search algorithm have been proposed. We discuss some of the related work in this section that is relevant to our research.

The \( k \)-d tree [3] is a well-known nearest neighbor search algorithm, as it is very effective in low dimensionality spaces. One popular implementation of approximate \( k \)NN search using \( k \)-d trees is found in Scikit-learn; this Python library has been used for machine learning applications such as advertisement blocking [4] and image classification using bag-of-visual-words features [7].

However, the use of \( k \)-d trees for \( k \)NN search suffers a decrease in performance for high dimensional data. As such, Scikit-learn’s popularity often drops off as data grows in dimension (as well as size, as Python is single-threaded), and there is not much literature on the use of this package for large and high-dimensional datasets. Some methods have been proposed to remedy this performance loss while still using \( k \)-d trees. One way of approximating nearest neighbor search is by limiting the time spent during search, or “time bound” approximate search, as proposed by [2]. In this approach, search through the \( k \)-d tree is stopped early after examining a fixed number of leaf nodes.

\( K \)-d trees do not scale well with dimension [11], hence ball trees which are similar to \( k \)-d trees in the way they organize points spatially have been widely studied. Several algorithms have been proposed for efficient construction of ball trees on large data. Andrew Moore in his paper [11] uses the idea of anchors instead of balls and using triangle inequality efficiently builds a ball tree that prunes nodes which would not belong to the current child. Kumar and others in their paper [8] do a comprehensive survey of tree based algorithms for nearest neighbor search.

Multiple randomized \( k \)-d trees (\( k \)-d forest) are proposed in [19] as a means to speed up the approximate nearest neighbor search; this is one of the most effective methods for matching high dimen-

\(^{2}\)http://scikit-learn.org/
\(^{3}\)http://openmp.org
\(^{4}\)http://iss.ices.utexas.edu/?p=projects/galois
sional data [13]. This approach builds multiple k-d trees that are searched in parallel. While the classic k-d tree construction splits data on the dimension with the highest variance, the randomized forest approach chooses the split dimension randomly from the top \( N_d \) dimensions with the highest variance, where \( N_d \) is selected by the user. When searching the randomized k-d forest in parallel, a single priority queue is maintained across all the randomized trees.

Apart from the use of k-d trees, another promising approach to approximate kNN graph construction for high dimensional data is the divide and conquer approach proposed by Chen et al. in [5]. The authors discuss two divide implementations, overlap and glue, that use recursive Lanczos bisection to split a set of data points into subsets (possibly with overlaps); the approach recursively computes the approximate kNN graphs for the subsets, then conquer the results into a final kNN graph. In the conquer step, if a data point belongs to multiple subsets, then the algorithm simply selects its \( k \) nearest neighbors from the union of its neighbors in each subset. Chen’s approach achieves a time complexity of \( O(dn^t) \), where \( t \) is a function of an internal parameter such that \( 1 < t < 2 \). The authors show that their approach’s accuracy (when the approximate kNN graph is compared to the exact kNN graph) is often above 95\%, even with small \( t \).

In work by Plaku and Kavraki [17], an implementation of kNN graph construction in an distributed environment is proposed. Message passing is used to communicate between processors in a cluster and efficiently distribute the computation of kNN graphs. The authors show that nearly linear speedup can be obtained with over one hundred processors.

3 Background

The focus of this work is to provide implementations of kNN graph construction using k-d trees and ball trees in the Galois framework. As such, we now provide the necessary background on k-d trees, ball trees, and the Galois framework.

3.1 k-d Trees

Multidimensional binary search trees, better known as k-d trees, were first proposed by Bentley in 1975 [3] as an efficient means to store and retrieve information. They have since become very popular for multidimensional search; one recent paper states that Bentley’s work is one of the most cited papers in computational geometry [15]. The k-d tree is an extension of the binary search tree for multidimensional data, in which each \( d \)-dimensional point in a data set is represented by a node in the k-d tree. Every node in the tree splits the data in its subtrees along one of the \( d \) dimensions, such that all descendants of the node whose value at the splitting dimension is less than that of the node are found in the node’s left subtree, and all descendants of the node whose value at the splitting dimension is greater than that of the node are found in the node’s right subtree. For descendants of the node whose value at the splitting dimension is equal to that of the node, the choice of left or right subtree is arbitrary, as presented in Bentley’s original paper; the optimal rule may depend on the purpose of the k-d tree.

3.1.1 Tree Construction

There are several ways to construct a k-d tree, with the “best” approach depending on the use of the tree and the dataset. One may randomly insert nodes one at a time; this approach is useful when the set of data points will vary over time. For our purposes of using k-d trees for nearest neighbor search, we focus instead on a standard top-down approach presented in [6], in which the set of data points remains constant and is known \( a \) priori, and the resulting k-d tree is balanced.

The top-down recursive algorithm that our approach uses is presented in Figures 1 and 2. Given an array of the entire set of data points, the algorithm finds the dimension with the maximum spread in the data. It then sorts the array in place, ordered by each point’s value at that dimension. The point at the median of this now-sorted array is assigned as the tree’s root. Using the recursive buildSubtree

\[ \text{We note that an alternative approach to constructing a k-d tree is to simply begin with the first dimension as the splitting dimension for the root and increase the splitting dimension by one for each level down in the subtree. This is the approach we used for the OpenMP implementation described in the next section. While this implementation results in a faster time to build the k-d tree, it often results in a longer time to perform} \]
function, all points to the left of the median in the array are passed to the root’s left subtree, while all points to the right of the median in the array are passed to the root’s right subtree. The \texttt{buildSubtree} function similarly sorts the given portion of the array on the dimension with the most spread, assigns the median within this portion to the parent node’s left or right child accordingly, and passes the two halves of its portion of the array to the child’s subtrees. It is important to note that for a given level of the tree, the portions of the array being sorted never overlap.

**Algorithm 1** \texttt{KDTree(points, N, d)}

1: find the dimension with most spread, store as \texttt{dim}
2: sort in place all \texttt{points} in increasing order of each’s value at \texttt{dim}
3: \texttt{median} = \(N \div 2\)
4: \texttt{tree.root.point} = \texttt{points[median]}
5: \texttt{tree.root.dim} = \texttt{dim}
6: \texttt{buildSubtree(root, “left”, points, 0, median)};
7: \texttt{buildSubtree(root, “right”, points, median + 1, N)};

**Algorithm 2** \texttt{buildSubtree(parent, childType, points, startIndex, endIndex)}

1: if \texttt{endIndex} == \texttt{startIndex} then
2: return
3: end if
4: if \texttt{endIndex} − \texttt{startIndex} == 1 then
5: if \texttt{child} == “left” then
6: \texttt{parent.leftChild.point} = \texttt{points[startIndex]}
7: else if \texttt{child} == “right” then
8: \texttt{parent.rightChild.point} = \texttt{points[endIndex]}
9: end if
10: return
11: end if
12: find the dimension with most spread, store as \texttt{dim}
13: sort in place \texttt{points} between \texttt{startIndex} and \texttt{endIndex} in increasing order of each’s value at \texttt{dim}
14: \texttt{median} = \texttt{startIndex} + (\texttt{endIndex} − \texttt{startIndex}) \div 2
15: if \texttt{child} == ”left” then
16: \texttt{parent.leftChild.point} = \texttt{points[median]}
17: \texttt{parent.leftChild.dim} = \texttt{dim}
18: \texttt{buildSubtree(parent.leftChild, “left”, points, startIndex, median)};
19: \texttt{buildSubtree(parent.leftChild, “right”, points, median + 1, endIndex)};
20: else if \texttt{child} == ”right” then
21: \texttt{parent.rightChild.point} = \texttt{points[median]}
22: \texttt{parent.rightChild.dim} = \texttt{dim}
23: \texttt{buildSubtree(parent.rightChild, “left”, points, startIndex, median)};
24: \texttt{buildSubtree(parent.rightChild, “right”, points, median + 1, endIndex)};
25: end if

3.1.2 \textit{kNN Graph Construction}

Given a \textit{k-d} tree constructed for a set of data points, we now wish to construct a \textit{kNN} graph, such that every data point in the dataset corresponds to a node in the graph, and there is an edge from \textit{node1} to \textit{node2} if \textit{node2} is among the \textit{k} nearest neighbors of \textit{node1}. To construct the \textit{kNN} graph from the \textit{k-d} tree, we use a standard approach such as the one presented in [20].

The \textit{kNN} search algorithm we use is presented in Figures 3 and 4. This algorithm is performed for every point in the dataset. We use a priority queue to store the \textit{k} best neighbors of the point in question (which we call the \textit{keyPoint}), prioritized on each neighbor’s distance to the \textit{keyPoint}.
The algorithm calls the recursive searchKDSubtree function. When it returns, the points stored in the priority queue correspond to the $k$ best neighbors for the keyPoint, and edges are added to the $k$-NN graph from the keyPoint to each neighbor. The recursive searchKDSubtree function considers a node in the $k$-d tree. If the priority queue has fewer than $k$ elements in it or if the distance from the keyPoint to the point corresponding to the given node (which we call currentPoint) is less than the distance to the farthest neighbor in the priority queue, then the algorithm adds the currentPoint to the priority queue, popping the farthest neighbor if necessary. It then searches the current node’s left subtree (if it exists) if the keyPoint value at the dimension on which this node splits (in the $k$-d tree) is less than that of the current node; otherwise, it searches the current node’s right subtree (if it exists). After this recursive subtree search returns, the algorithm considers if the current node’s other subtree must be searched: if fewer than $k$ neighbors have been found or if the distance between the keyPoint value at the current node’s splitting dimension and that of the currentPoint is less than the overall distance to the farthest neighbor in the priority queue, the other subtree is searched.

**Algorithm 3** findKNN(keyPoint, kdtree, $k$)

1: $pq = \text{new priority queue of potential neighbors, prioritized on their distance to keyPoint}$
2: searchKDSubtree($pq$, root, keyPoint, $k$)
3: for each neighborPoint in $pq$ do
4: add edge from keyPoint to neighborPoint
5: end for

**Algorithm 4** searchKDSubtree($pq$, currentNode, keyPoint, $k$)

1: $currentPoint = currentNode.point$
2: $dim = currentNode.dim$
3: $dist = \text{distance from keyPoint to currentPoint}$
4: if number of points in $pq < k$ then
5: push currentPoint onto $pq$ with priority $dist$
6: else if $dist < \text{priority of max-priority element in } pq$ then
7: pop max-priority element from $pq$
8: push currentPoint onto $pq$ with priority $dist$
9: end if
10: if $keyPoint[dim] < currentPoint[dim]$ then
11: if currentNode has a left child then
12: searchKDSubtree($pq$, currentNode.leftChild, keyPoint, $k$)
13: otherChild = currentNode.rightChild
14: end if
15: else
16: if currentNode has a right child then
17: searchKDSubtree($pq$, currentNode.rightChild, keyPoint, $k$)
18: otherChild = currentNode.leftChild
19: end if
20: end if
21: if number of points in $pq < k$ OR $|keyPoint[dim] - currentPoint[dim]| < \text{max priority in } pq$ then
22: searchKDSubtree($pq$, otherChild, keyPoint, $k$)
23: end if

### 3.2 Ball Trees

Ball trees are very similar to $k$-d trees, spatially organizing points in multiple dimensions. However, unlike $k$-d trees, which split the points parallel to the axis, ball trees split such that points closer to each other go to one child while the other set of nearby points go to the other child. Figure 1 shows the split on an example dataset by the ball tree and the $k$-d tree (figure from [8]). $K$-d trees are very effective in low dimension, however as the number of dimensions increase, the property of $k$-d tree to collate points nearby in the same leaf node disappears.
For a ball tree, it is easy to show that a split will, with very high probability, put points belonging to one class into one child and points belonging to another class into the other child of the node. The results is that for a nearest neighbor algorithm, a search will only need to visit half the datapoints in a ball tree but many more in a k-d tree [11].

3.2.1 Tree Construction

Many fast algorithms for ball tree construction have been proposed. We follow the approach described by Andrew Moore in his paper [11]. Algorithm 5 describes the recursive method that builds the ball tree top-down starting from the root node. As mentioned earlier, each node called ball node of the tree owns a set of points. It has a pivot, which in our implementation is the centroid of the points owned by that node, and the radius of the ball node is the distance between the pivot and the furthest point in the node. Depending on the implementation, the centroid can also be chosen to be one of the points.

Algorithm 5 recursiveBuildSubtree(parent, N, points, D, leafSize)

1: parent.data = points
2: parent.pivot = centroid(parent.data)
3: parent.radius = Max(distance(parent.pivot, parent.points))
4: parent.child1 = point farthest from parent.pivot
5: parent.child2 = point farthest from parent.child1
6: for point ∈ parent.points do
7: if dist(point, child1) <= dist(point, child2) then
8: child1.points ← point
9: else
10: child2.points ← point
11: end if
12: end for
13: if child1.points.size > leafSize then
14: recursiveBuildSubtree(parent.child1, N, points, D, leafSize);
15: end if
16: if child2.points.size > leafSize then
17: recursiveBuildSubtree(parent.child2, N, points, D, leafSize);
18: end if

Figure 2 displays the recursive steps for ball tree construction by splitting on a set of points (figure taken from [10]).

3.2.2 kNN Graph Construction

The ball tree kNN graph construction is very similar to the k-d tree's. It examines nodes in depth first order, starting from the root. During the search, the algorithm maintains a priority queue, which in our case is a max-heap. At each step the heap is maintained such that it contains the k nearest neighbors obtained so far. Algorithm 6 describes the recursive search in a ball tree for kNN graph construction.
Algorithm 6 searchBalltree(pq, currentNode, keyPoint, k)

1: if \( \text{distance}(\text{keyPoint}, \text{currentNode.pivot}) \geq \text{distance}(\text{keyPoint}, \text{pq.first}) \) then
2: return pq
3: else if currentNode is a leaf node then
4: for point \( \in \text{currentNode.points} \) do
5: if \( \text{distance}(\text{keyPoint}, \text{point}) < \text{distance}(\text{keyPoint}, \text{pq.first}) \) then
6: push currentPoint onto pq with priority dist
7: if \( \text{size}(pq) > k \) then
8: pop max-priority element from pq
9: end if
10: end if
11: end for
12: else
13: let child1 be the node closest to currentNode
14: let child2 be the node furthest from currentNode
15: searchBalltree(pq, child1, keyPoint, k)
16: searchBalltree(pq, child2, keyPoint, k)
17: end if
3.3 Galois Framework

The Galois framework is a parallel graph analytics infrastructure recently presented by Nguyen et al. [14]. Written in C++, Galois provides the user with parallel graph data structures. The user in turn writes sequential code for applications to manipulate and maneuver around the graphs, to accomplish his particular goal. Thus, the user does not need to manage the program’s concurrency directly; it is built-in to the Galois data structures. Galois has recently been shown to perform very well for graph algorithms on large datasets [18]. However, it does not currently contain an implementation of nearest neighbor search, which motivates our decision to use this framework and develop such an implementation.

4 Implementation

We extended the general sequential approaches to \(k\)NN graph construction via \(k\)-d trees and ball trees as described in Section 3 to two parallel settings, OpenMP and Galois. Here we discuss our specific parallelization strategies. For all implementations, we found that constructing the underlying tree frequently took only a small fraction of the total time (underlying tree construction plus \(k\)NN search and graph construction), and in some cases the overhead of parallelizing tree construction caused the parallel construction to be slower than sequential construction. As such, all implementations construct the underlying tree sequentially; nearest neighbor search is parallelized.

4.1 OpenMP

Extending sequential nearest neighbor search to a parallel setting using OpenMP is straightforward. In order to construct a \(k\)NN graph, we must find the \(k\) nearest neighbors of every node in the dataset. However, these searches are completely independent of one another and do not modify the underlying tree structure. Additionally, the result of a single search is a list of neighbors for the given node, which is stored as one row of a two-dimensional array; but each thread will be searching the neighbors for a unique node, corresponding to a unique row in the shared array. Therefore, the searches can be done in parallel without any concern of data races. We use an OpenMP parallel for loop over the nodes, to find each node’s \(k\) nearest neighbors.

4.2 Galois

Extending sequential nearest neighbor search to the Galois framework required shifting the sequential data structures for both the underlying tree and the \(k\)NN to Galois graphs. We used the FirstGraph object from Galois, which allows us to mutate the graph structure (namely, the edges between nodes) as we progress through our algorithms. We note that due to time restrictions, we did not implement the ball-tree-based nearest neighbor search in Galois, so this discussion focuses on the \(k\)-d tree approach.

To construct the \(k\)-d tree as a Galois FirstGraph object, we define a custom node class KDNode that contains the data point associated with the node, the index of the data point in the original data set, the dimension on which the node splits its descendants, and a flag indicating whether it is a left child of its parent. We create a KDNode for each data point and insert all of these nodes (without any edges) into the graph, which we call kdtree. As in the sequential version, we find the dimension of maximum spread for all of the data points, and we find the median point on this dimension and call it the root. We note the index of the root for later use, as the graph itself does not explicitly contain this information. We sort the data points by the dimension of maximum spread. We then use the Galois concept of a worklist for iterating through the remaining data points. Each item in the worklist implicitly corresponds to a subtree in the kdtree: it contains the index of the subtree’s root’s parent node, the portion of the data set that the subtree contains, and a flag denoting whether the subtree’s root is the left or right child of the parent. We add the left and right subtrees of the root to the worklist, and we use a Galois for_each iterator over the worklist, using an operator we define to recursively add the appropriate edges in the kdtree. This operator highly resembles the buildSubtree function from the sequential approach: it finds the dimension of maximum spread, sorts the subtree’s data points along that dimension, and finds the corresponding median data point. The KDNode corresponding to this median data point will be the root of the subtree, so
the KDNode’s dimension and left child flag fields are updated accordingly. An edge is added to the kdtree from the subtree’s parent node to this KDNode. The operator then adds the left and right subtrees to the worklist to be processed. The tree is complete when the worklist is empty.

We then build the kNN graph as a Galois FirstGraph object, again using KDNodes as the underlying node in the graph (although we can ignore the dimension and left child fields for nodes in the kNN graph). We refer to our kNN graph as knngraph. We create a KDNode for each data point and insert all of these nodes (without any edges) into the knngraph. We then use a Galois do all iterator over all nodes in the knngraph, using an operator we define to traverse the kdtree in search of the given node’s k nearest neighbors. This operator highly resembles the findKNN and searchKDSubtree functions from the sequential approach. Once the kdtree has been searched for a given node’s neighbors, edges are added to the knngraph from the node to each of its k nearest neighbors, and the value of the edge is equal to the distance from the node to the neighbor.

We have a very similar approach for ball trees in Galois since we build both using max spread among dimensions. The kNN graph for ball trees is build similar to its OpenMP approach, maintaining a priority queue for each node.

5 Experimental Results

In this section we discuss our experimental results. We compare the performance of the following kNN graph construction implementations:

- OpenMP using k-d trees, splitting on increasing dimension (our implementation)
- OpenMP using ball trees (our implementation)
- Galois using k-d trees, splitting on increasing dimension (our implementation)
- Galois using k-d trees, splitting on dimension of maximum spread (our implementation)
- Python’s Scikit-learn library’s NearestNeighbors module: k-d tree, ball tree, and brute force approaches

We note that all approaches except for Scikit-learn are implemented in C/C++.

5.1 Data Sets

We performed our experiments on several data set:

- MNIST: database of handwritten digit images
- Poker: Poker Hand data set of possible suit and rank five-card hands
- RNA: Non-coding RNA data
- Housing: Economic covariates of housing prices in California
- Covtype: Cartographic data used to predict forest cover type

Table 1: Image data sets

<table>
<thead>
<tr>
<th></th>
<th>MNIST test set</th>
<th>MNIST training set</th>
<th>Covtype</th>
<th>Poker</th>
<th>RNA</th>
<th>Housing</th>
</tr>
</thead>
<tbody>
<tr>
<td># data points (N)</td>
<td>10,000</td>
<td>60,000</td>
<td>581,012</td>
<td>1,000,000</td>
<td>271,617</td>
<td>20,640</td>
</tr>
<tr>
<td># dimensions (d)</td>
<td>784</td>
<td>784</td>
<td>54</td>
<td>10</td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>

Table 1 provides some characteristics of the data. It is important to note that the two MNIST datasets have a very large number of dimensions, Covtype has a moderate number of dimensions, and the remaining datasets have relatively few dimensions. The dimensionality, and importantly the ratio of the number of data points to the dimensionality, is a strong influencer on the performance of k-d trees and ball trees, as our results indicate.

*While original sources for the datasets are provided in the paper, we wish to recognize that the data files themselves were obtained from the LIBSVM collection: http://www.csie.ntu.edu.tw/ cjlin/libsvmtools/datasets
(a) MNIST test dataset: \( N = 10000, d = 784 \)

(b) MNIST train dataset: \( N = 60000, d = 784 \)

(c) Covtype dataset: \( N = 581012, d = 54 \)

Figure 3: Performance Results
5.2 Experimental Setup

All results presented here were gathered from experiments conducted on a supercomputer known as Stampede at the Texas Advanced Computing Center\footnote{https://www.tacc.utexas.edu/stampede}. The Stampede system is a 10 PFLOPS Dell
Linux Cluster with Dell PowerEdge server nodes, each of which has two Intel Xeon E5 processors. Each node has 32 gigabytes of memory.

5.3 Results

For each data set, we measured runtime for each of the approaches outlined above. For our parallel implementations, we measured runtime for 1, 4, 8, and 16 threads. Results are provided in Figures 3 and 4. We note that for the RNA, Poker, and Covtype datasets, the Scikit-learn brute force implementation encountered a MemoryError. Additionally, we did not obtain results for the Poker and Covtype datasets for our k-d tree OpenMP implementation, as the runtimes were too long. Finally, we present results for two Galois k-d tree implementations: one splitting every node on the dimension with maximum spread, as described in Section 4, and one splitting on one dimension higher than the previous level, as the OpenMP implementation takes this approach. Since ball trees are faster than KD Trees for high dimensional data, we compare ball tree implementations on MNIST datasets using OpenMP, Galois and ScikitLearn. Results for runtime and scalability are provided in Figure 5.

6 Discussion

Our results indicate that for high-dimensional data, such as the MNIST datasets, ball trees outperform k-d trees. This is as expected, as it is well-known that k-d trees suffer from the curse of dimensionality. For lower dimensional data, k-d trees perform the best. We also see that the Galois k-d trees outperform the OpenMP k-d trees, although the largest gains in k-d tree performance are a result of splitting each level on the dimension with maximum spread.
We also find that for high-dimensional data, the Galois implementation of \( k \)-d trees that uses maximum spread does not scale well with an increase in the number of threads beyond 8. The other parallel implementations provide increasing scalability for high-dimensional data as the number of threads increases. For the lower-dimensional data, the OpenMP \( k \)-d tree implementation provides the best scalability.

We observe that on MNIST datasets, ball trees using Galois and OpenMP for leaf size 1 have very similar performance as number of threads increases. For 16 threads, Galois beats OpenMP on both datasets. Also in terms of scalability, Galois is almost linear with number of threads. OpenMP for ball trees does not scale well when compared to Galois for MNIST datasets.

An important note is that brute force fails with Out of Memory exception on Stampede for all datasets that have number of points \( N > 60,000 \). That is the reason why we don’t have brute force comparisons for all datasets.

References


