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

Nazneen Rajani
Dept. of Computer Science
University of Texas at Austin
Austin, TX 78712-1188, USA
nrajani@cs.utexas.edu

Kate McArdle
The Center for Advanced Research in Software Engineering
University of Texas at Austin
Austin, TX 78712-1188, USA
kate.mca@utexas.edu

Inderjit S. Dhillon
Dept. of Computer Science
University of Texas at Austin
Austin, TX 78712-1188, USA
inderjit@cs.utexas.edu

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 high dimensional. Python's open source machine learning library Scikit-learn uses k-d trees and ball trees to implement nearest neighbor graph construction. However, this implementation is inefficient for large datasets. In this work, we focus on exploiting these underlying tree-based data structures to optimize parallel execution of the nearest neighbor algorithm. We present parallel implementations of nearest neighbor graph construction using such tree structures, with parallelism provided by the OpenMP and the Galois framework. We empirically show that our parallel and exact approach is efficient as well as scalable, compared to the Scikit-learn implementation. We present the first implementation of k-d trees and ball trees using Galois. Our results show that k-d trees are faster when the number of dimensions is small ($2^d \ll N$); ball trees on the other hand scale well with the number of dimensions. Our implementation of ball trees in Galois has almost linear speedup on a number of datasets irrespective of the size and dimensionality of the data.

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 (kNN) 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. 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. To speed up the performance, many applications must settle for *approximate* k nearest neighbor graphs.

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. A ball tree, like the k-d tree, is also a binary tree data structure that organizes points in multidimensional space. Each node owns the set of points in its subtree. Thus the root node has the full set of points in the dataset and each leaf node has some maximum number of points, called *leaf size*. A non-leaf node does not explicitly contain any points, but it points to two child nodes such that $child1.points \cap child2.points = \phi$ and $child1.points \cup child2.points = node.points$. Each node has a pivot and a radius that determine how the points are split among the child nodes [11].

To use a k-d tree or a ball tree for nearest neighbor search, the tree must first be built, and then it is searched. One popular implementation of nearest neighbor search with such trees is found in Scikit-learn [15], an open-source machine learning library in Python[1]. K-d trees are commonly used for nearest neighbor search because for small D (< 20), approximately $O(DlogN)$ operations are needed in the case of randomly distributed N points in D dimensions. However, in high-dimensional spaces, the *curse of dimensionality* causes the algorithm to need to visit many more branches than in lower-dimensional spaces and the number of operations increases to $O(DN)$. In particular, when the number of points is only slightly higher than the number of dimensions, the algorithm is only slightly better than a brute force 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 on the other hand grows as $O(DlogN)$ even in high dimensionality. The complexity for k-d trees and ball trees also depends on the leaf size. With a bigger leaf size, more time is spent doing linear search within the leaf nodes; however, if the leaf size is smaller, building the tree takes more time than building the same tree with a bigger leaf size. Thus, there is a tradeoff between the tree building time and linear search time over leaf nodes. We note that the construction of a k-d tree is not affected as much by the leaf size as the construction of a ball tree due to their underlying structures[15].

[1] http://scikit-learn.org/

In this work, we present implementations of exact nearest neighbor search using k-d trees and ball trees in parallel settings, namely OpenMP[2] and the Galois system[3], a graph-based framework that provides data-parallelism. Our principal contribution is the implementation of k-d trees and ball trees in Galois and comparing our approaches to Scikit-learn's. The principal bottleneck for parallelization is the parallel tree construction while ensuring full resource utilization. Galois however overcomes it in the way it activates nodes, Section 3. We provide efficient implementations for both k-d trees and ball trees. Before describing our implementation in more detail 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

The k-d tree [3] is a well-known nearest neighbor search algorithm, as it is very effective in low-dimensional spaces. One popular implementation of exact kNN search using k-d trees is found in Scikit-learn; this Python library has been used for several machine learning applications [5]. However, the use of k-d trees for kNN search suffers a decrease in performance for high dimensional data. As such, Scikit-learn's performance often drops off as data grows in dimension or in size. The current version of Scikit-learn does not natively support parallelism for its k-d tree and ball tree implementations and there is not much literature on its use 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, 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. Moore et al. uses the idea of anchors instead of balls and uses the triangle inequality to efficiently build a ball tree that prunes nodes which would not belong to the current child [11]. Kumar et al. do a comprehensive survey of tree-based algorithms for nearest neighbor search [8]. Multiple, randomized k-d trees (a 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 dimensional data [12]. 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.

In [17], an implementation of kNN graph construction in a 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. Note the distinction between "multicore" and "distributed" approaches. Distributed is across machines while multicore is shared memory abstraction using multiple cores on the same machine. While [17] use a distributed approach, the focus of this paper is on a multicore approach using Galois.

[2]http://openmp.org

[3]http://iss.ices.utexas.edu/?p=projects/galois

3. BACKGROUND

The focus of this work is to provide implementations of kNN graph construction using k-d trees and ball trees in OpenMP 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. 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.

Algorithm 1 KDTree(*points*, N, d)

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

3.1.1 Tree Construction

There are several ways to construct a k-d tree, with the "best" approach depending on the application 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 [4], 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 Algorithms 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* 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 *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 impor-

tant to note that for a given level of the tree, the portions of the array being sorted never overlap. 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 nearest neighbor search on the tree.

Algorithm 2 buildSubtree(*parent*, *childType*, *points*, *startIndex*, *endIndex*)

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

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

```
1: pq = new priority queue of potential neighbors, priori-
   tized 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
```

3.1.2 kNN Graph Construction

Given a k-d tree constructed for a set of data points, we now wish to construct a kNN graph, such that every data point in the dataset corresponds to a node in the graph, and there is an edge from *node*1 to *node*2 if *node*2 is among the k nearest neighbors of *node*1. To construct the kNN graph from the k-d tree, we use a standard approach such as the one presented in [20].

The kNN search algorithm we use is presented in Algorithms 3 and 4. The algorithm in 3 is performed for every point in the dataset. We use a priority queue to store the k best neighbors of the point in question (which we call the *keyPoint*), prioritized on each neighbor's distance to the *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 kNN graph from the *keyPoint* to each neighbor. The recursive *searchKDSubtree* function operates on 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 4 searchKDSubtree(*pq*, *currentNode*, *keyPoint*, k)

```
1: currentPoint = currentNode.point
2: dim = currentNode.dim
3: dist = 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 < 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] |< 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 Cartesian axes, ball trees split data in a series of hyperspheres such that points closer to each other go to one child while the other set of nearby points goes to the other child. The structure of the ball tree overcomes the inefficiencies of the k-d tree

in high dimensions. 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]. The number of points for a nearest neighbor search is reduced through use of the triangle inequality.

3.2.1 Tree Construction

Many fast algorithms for ball tree construction have been proposed. We follow the approach described in [11]. Algorithm 5 describes the recursive method that builds the ball tree top-down starting from the root node. As mentioned earlier, each node of the tree (called a *ball node*) owns a set of points. It has a pivot, which in our implementation is the centroid of the points owned by that node, and a radius, which 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 data points.

Algorithm 5 buildSubtree($parent$, N, $points$, D, $leaf_Size$)

1: $parent.data = points$
2: $parent.pivot = centroid(parent.data)$
3: $parent.radius = Max(distance(parent.pivot, parent.points))$
4: $parent.child1 = pointfarthestfromparent.pivot$
5: $parent.child2 = pointfarthestfromparent.child1$
6: **for** $point \in parent.points$ **do**
7: **if** $dist(point, child1) <= dist(point, child2)$ **then**
8: $child1.points \leftarrow point$
9: **else**
10: $child2.points \leftarrow point$
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**

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. Figure 1 displays the recursive steps for ball tree construction by splitting on a set of points(figure taken from [10]).

3.3 Galois Framework

In their work on parallelism in algorithms, Pingali et al. [16] propose a data-centric formulation of algorithms, called the *operator formulation*, to replace the traditional program-centric abstraction. They argue that a program-centric abstraction is not adequate for data mining algorithms such as the k nearest neighbor graph construction. They go on to prove that a generalized form of data-parallelism exists in all algorithms and, depending on the algorithm's structure, this parallelism may be exploited by the program. They demonstrate their work with the Galois programming model [13].

The Galois framework is a parallel graph analytics infrastructure recently presented by Nguyen et al. [13]. It is a

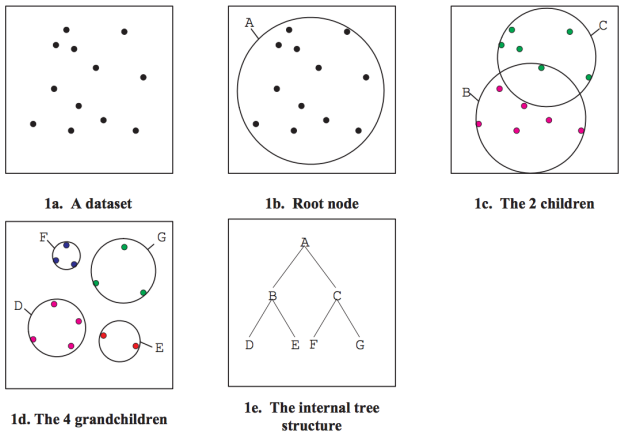


Figure 1: Ball Tree construction on a set of points (from [10])

work-item-based parallelization framework and provides a rich programming model that can handle graph computations. Since Galois provides its own schedulers and scalable data structures with coordinated and autonomous scheduling, the user does not need to manage the program's concurrency directly. Written in C++, Galois does not impose a particular graph partitioning scheme; it may be edge or vertex based, depending on how the computation is expressed. The user writes sequential code for applications to manipulate and maneuver around the graphs to accomplish his particular goal. Galois has recently been shown to perform very well for graph algorithms on large datasets [18]. Kulkarni et al. use Galois to do an in-depth research on the extent of data-parallelism for irregular algorithms such as clustering which uses the nearest neighbor approach [7]. However, it does not currently contain an implementation of nearest neighbor search using tree-based structures, which motivates our decision to use this framework and develop such an implementation.

Algorithm 6 searchBallSubtree(pq, $currentNode$, $keyPoint$, k)

1: **if** $distance(keyPoint, currentNode.pivot) >= distance(keyPoint, pq.first)$ **then**
2: $returnpq$
3: **else if** $currentNodeisaleafnode$ **then**
4: **for** $point \in currentNode.points$ **do**
5: **if** $distance(keyPoint, point) < distance(keyPoint, pq.first)$ **then**
6: push $currentPoint$ onto pq with priority $dist$
7: **if** $size(pq) > k$ **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**

The authors of Galois emphasize that a parallel program comprises an operator, schedule and parallel data structure and can be informally stated as: Parallel Program = Oper-

ator + Schedule + Parallel Data Structure [13]. A Galois operator is a user-defined function that operates on a graph node. Operators acting on different nodes can be executed in parallel and Galois ensures that no two neighboring nodes are operated on simultaneously, thus avoiding conflicts and ensuring serializability. In addition, the order in which nodes are operated are provided via a schedule or a worklist. Galois provides several implementations for schedules and parallel data structures, while the user provides the operator and picks a schedule based on the application.

For our implementation of nearest neighbor, we use two Galois operators, one for building the underlying tree and one for the actual nearest neighbor search. The tree construction operator builds the k-d tree or ball tree as a Galois graph. The first step is to split the data points on dimension of maximum spread. Thereafter the operator adds points in each split to a parallel worklist. Each node in the worklist can be operated on in parallel. The nearest neighbor graph construction operator has all the data points in its parallel worklist for execution, and it operates on a parallel graph data structure. For each data point it adds edges in the kNN graph using the constructed tree to find nearest neighbors.

Pingali et al. state that exploiting the structure is the most important factor for efficient parallel implementation and thus encourage tao analysis of algorithms [16]. Tao analysis classifies algorithms based on their topology, node activation strategy and operator type. Because both k-d trees and ball trees have an underlying tree structure in their algorithms, the topology is semi-structured based on their definition [16]. Both the tree construction algorithm and the kNN graph construction algorithm for both k-d trees and ball trees are classified as data driven, unordered, morph.

4. PARALLEL ALGORITHMS

We extended the general sequential approaches to kNN 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. It is valuable to notice that the underlying tree construction as well as the kNN graph construction can be both done in parallel one after the other. However, for all implementations, we found that constructing the underlying tree took only a small fraction of the total time (underlying tree construction plus kNN search and graph construction) and in some cases the overhead of parallelizing tree construction caused the parallel construction to be slower than sequential construction. For these reasons, all implementations construct the underlying tree sequentially and only the k nearest neighbor search is parallelized. We now discuss extensions of the sequential versions, described in Section 3 for each of the two parallel settings separately. We note that all implementations in the Section 4 are exact extensions of the sequential algorithms as opposed to the approximate extensions discussed in Section 2.

4.1 OpenMP

Extending sequential nearest neighbor search to a parallel setting using OpenMP is straightforward. In order to construct a kNN 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 for data races. We use an OpenMP `parallel for` loop over the nodes, to find each node's k nearest neighbors.

4.2 Galois

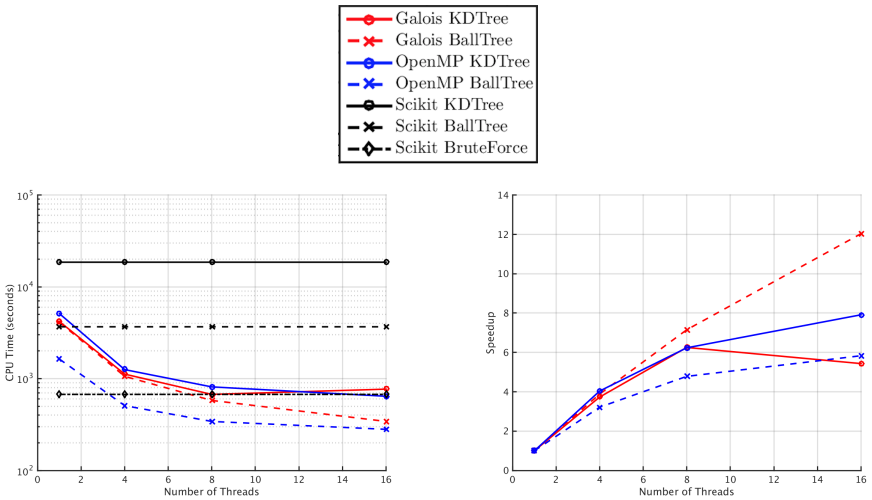
Extending sequential nearest neighbor search to the Galois framework required shifting the sequential data structures for both the underlying tree and the kNN to Galois graphs. We used the `FirstGraph` object from Galois, which allows us to mutate the graph structure (namely, the edges between nodes) as we progress through our algorithms. We now discuss the underlying tree construction and the kNN search implementation in Galois. 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 if 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.

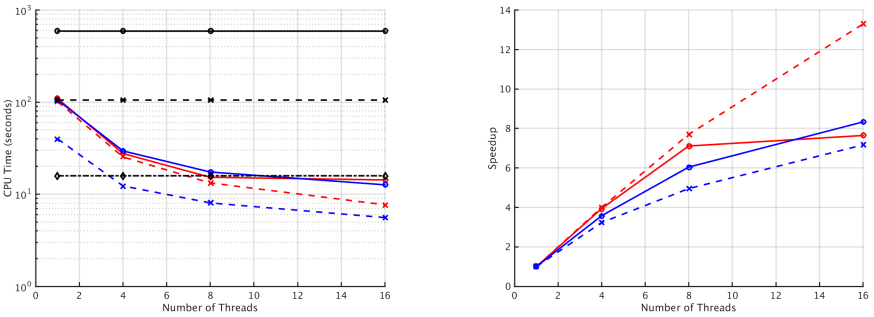
We follow a very similar approach for ball trees in Galois

	MNIST test set	MNIST training set	Covtype	Poker	RNA	Housing
# data points (N)	10,000	60,000	581,012	1,000,000	271,617	20,640
# dimensions (d)	784	784	54	10	8	8

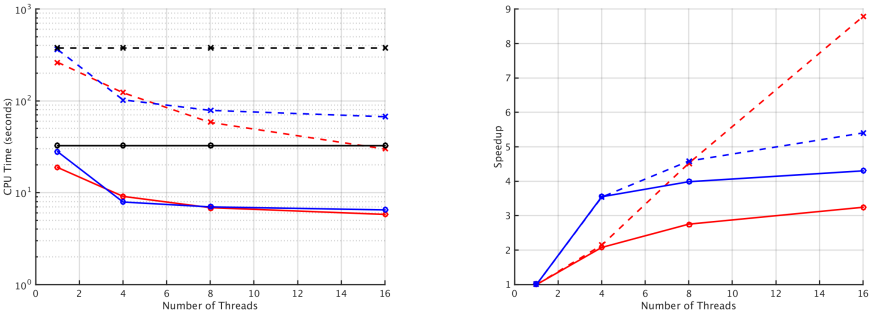
Table 1: Description of data sets used in our experiments



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



(a) MNIST test dataset: N = 10000, d = 784



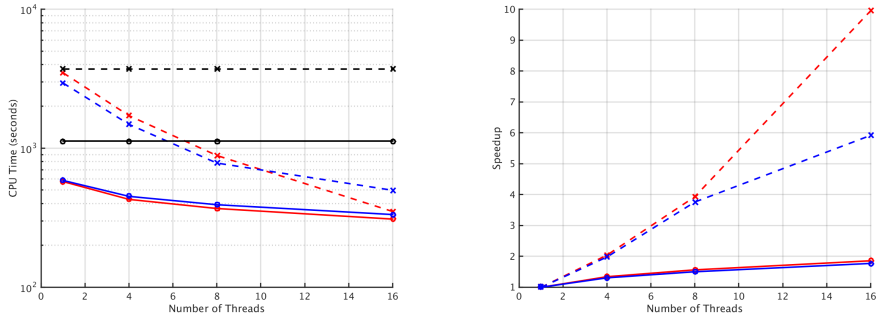
(c) Covtype dataset: N = 581012, d = 54. Brute force runs out of memory

Figure 2: Runtime and scalability results on datasets that do not satisfy $2^d << N$ property

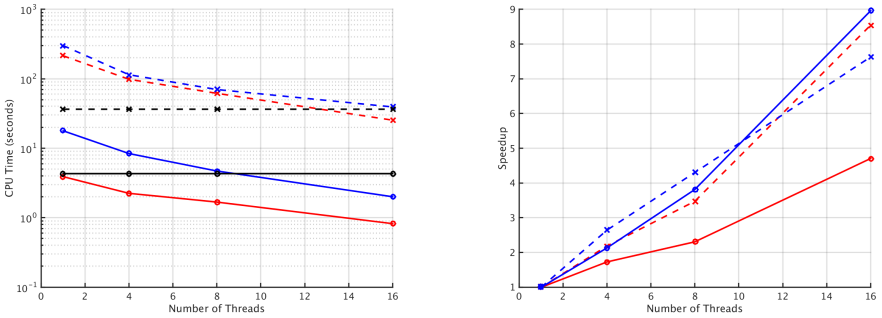
and skip the details due to space constraints. Both the k-d tree and ball tree use max spread among dimensions for splitting and thus have a very similar approach for building the underlying tree. The ball tree construction terminates if the number of data points the child nodes have is less than or equal to the leaf size. The kNN graph for ball trees is built similarly to the k-d tree using a `FirstGraph` object. The operator definition for the ball tree implementation of `knngraph` follows the approach discussed in the kNN graph construction description in Section 3.2.2. Similar to the k-d tree implementation in Galois, the ball tree implementation also maintains a priority queue for searching the nearest neighbors of a given node and adds edges to the `knngraph` from the node to each of its k nearest neighbors.

5. EXPERIMENTAL RESULTS

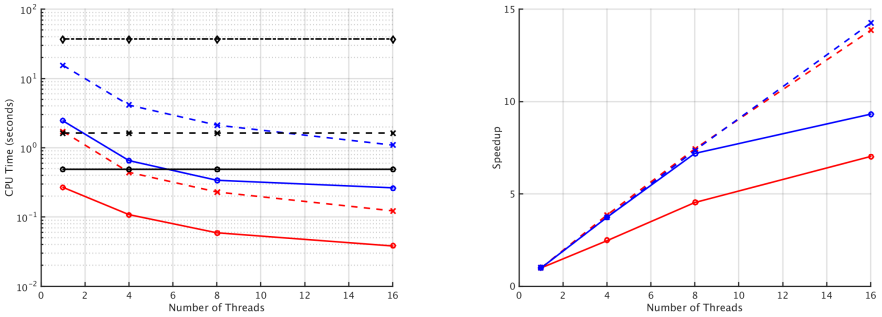
In this section we discuss our experimental results. We compare the performance of the following kNN graph construction implementations: OpenMP using k-d trees, OpenMP using ball trees, Galois using k-d trees, Galois using ball trees and Python's Scikit-learn library's Near-



(a) Poker dataset: N = 1000000, d = 10. Brute force runs out of memory



(b) RNA dataset: N = 271617, d = 8. Brute force runs out of memory



(c) Housing dataset: N = 20640, d = 8

Figure 3: Runtime and scalability results on datasets that satisfy $2^d << N$ property. Legend same as Figure 2.

estNeighbors module: k-d tree, ball tree, and brute force approaches. We note that all approaches except for Scikit-learn are implemented by us in C++. The default leaf size for k-d trees and ball trees used by Scikit-learn is 30. The reason provided for this choice is that for $N < 30$, $log(N)$ is comparable to N and a brute force approach can be more efficient [15]. To have a fair comparison, we also use 30 as the leaf size for our implementations of k-d trees and ball trees. As discussed earlier, the tree construction time is insignificant compared to the total time (underlying tree construction plus kNN search), and we only report the total time in all our experiments.

5.1 Data Sets

We performed our experiments on several data sets: MNIST: database of handwritten digit images [9], Poker: Poker Hand data set of possible suit and rank five-card hands [1], RNA: Non-coding RNA data [21], Housing: Economic covariates of housing prices in California [14], Covtype: Cartographic data used to predict forest cover type [1]. Table 1 provides some characteristics of the data. It is important to note that among all our datasets, the two MNIST datasets have the largest 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 has a strong influence on the performance of k-d trees and ball trees, as our results indicate.

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[4]. The supercomputer 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. The brute force approach on all datasets was conducted on large memory nodes. These nodes have 32 cores/node and 1TB of memory for data-intense applications.

5.3 Results and Discussion

For each data set, we measured CPU runtime for each of

[4]https://www.tacc.utexas.edu/stampede

the approaches outlined above. For our parallel implementations, we measured runtime for 1, 4, 8, and 16 threads as well as speedup on each of the datasets. For measuring speedup, we use Amdahl's law for multiple threads. Past research on nearest neighbor search suggests that the performance of k-d trees is best when $2^d << N$ [6]. However, there has been little empirical evidence for this claim. We therefore choose some datasets that satisfy this property and others that do not, for comparison, and we measured the performance of all approaches on both of these dataset categories. Performance results provided in Figure 2 are for datasets that do not satisfy the $2^d << N$ property while performance results in Figure 3 are for datasets that do. We note that for datasets with $N > 60K$, the Scikit-learn brute force implementation encountered a MemoryError even on Stampede's large memory nodes.

Our results indicate that for high-dimensional data, such as the MNIST datasets, ball trees outperform k-d trees. Although the OpenMP parallel version of ball trees is faster than the Galois ball tree implementation, the Galois ball tree implementation is better in terms of scalability than OpenMP for high dimensional data because of its almost linear speedup with the number of threads. Scikit-learn's brute force is faster than the k-d tree and ball tree approach on MNIST datasets; however, our parallel implementation with 8 and 16 threads outperforms the brute force approach, as shown in Figure 2. For moderate-dimensional data, such as the Covtype dataset, k-d trees are faster than ball trees. Our results suggest that OpenMP and Galois have almost the same performance in moderate-dimensional data for both k-d trees and ball trees. Our parallel implementations of k-d tree outperform Scikit-learn's even with one thread for the Covtype data, as shown in Figure 2.

K-d trees are known to perform really well on low-dimensional data, and our results in Figure 3 demonstrate such good performance. The k-d tree in Galois outperforms all other approaches, even with 1 thread. For ball trees in low-dimensional data, Galois is faster than OpenMP for all number of threads. Although k-d trees are faster than ball trees in low dimensionality, our results show that ball trees scale better than k-d trees for their corresponding implementation. Overall we observe that Galois ball trees have almost linear speedup on all datasets irrespective of dimensions. We may thus conclude that Galois provides a very efficient framework for parallel algorithms to scale on larger datasets. On the larger Poker dataset, our parallel implementations outperform Scikit-learn's by a large margin. Since all the nearest neighbor approaches we experimented with are exact, we do not compare them in terms of accuracy.

6. CONCLUSION

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. We present parallel implementations of nearest neighbor graph construction using k-d trees and ball trees, with parallelism provided by OpenMP and the Galois framework. We empirically show that our parallel approach is efficient as well as scalable, compared to the Scikit-learn implementation for datasets of various size and dimensions.

Our results indicate that k-d trees outperform ball trees when the number of dimensions is small ($2^d << N$) or moderate; ball trees on the other hand scale well will the number of dimensions. Overall, Galois ball trees have almost linear speedup on all datasets irrespective of the size and dimensionality of the data. Thus we conclude that data driven parallelism as implemented in Galois for k-d trees and ball trees provides a fast and scalable framework for big data applications.

7. REFERENCES

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

[2] J. S. Beis and D. G. Lowe. Shape indexing using approximate nearest-neighbour search in high-dimensional spaces. In *Computer Vision and Pattern Recognition, 1997. Proceedings, 1997 IEEE Computer Society Conference on*, pages 1000–1006. IEEE, 1997.

[3] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

[4] J. Friedman, J. Bentley, and R. Finkel. An algorithm for finding best matches in logarithmic expected time. *Transactions on Mathematical Software*, 3(3):209–226, 1977.

[5] C. Hentschel and H. Sack. Does one size really fit all? evaluating classifiers in bag-of-visual-words classification. In *i-KNOW Proc. 14th Intl. Conf. Knowledge Technologies and Data-driven Business*. ACM, 2014.

[6] P. Indyk. Nearest neighbors in high-dimensional spaces, 2004.

[7] M. Kulkarni, M. Burtscher, R. Inkulu, K. Pingali, and C. Casçaval. How much parallelism is there in irregular applications? In *ACM sigplan notices*, volume 44, pages 3–14. ACM, 2009.

[8] N. Kumar, L. Zhang, and S. Nayar. What is a good nearest neighbors algorithm for finding similar patches in images? In *Computer Vision–ECCV 2008*, pages 364–378. Springer, 2008.

[9] Y. LeCun and C. Cortes. The mnist database of handwritten digits, 1998.

[10] T. Liu, A. W. Moore, and A. Gray. New algorithms for efficient high-dimensional nonparametric classification. *The Journal of Machine Learning Research*, 7:1135–1158, 2006.

[11] A. W. Moore. The anchors hierarchy: Using the triangle inequality to survive high dimensional data. In *Proceedings of the Sixteenth conference on Uncertainty in artificial intelligence*, pages 397–405. Morgan Kaufmann Publishers Inc., 2000.

[12] M. Muja and D. G. Lowe. Fast approximate nearest neighbors with automatic algorithm configuration. In *VISAPP (1)*, pages 331–340, 2009.

[13] D. Nguyen, A. Lenharth, and K. Pingali. A lightweight infrastructure for graph analytics. In *Proc. 24th Symposium on Operating Systems Principles*, pages 456–471. ACM, 2013.

[14] Pace and Barry. Sparse spatial autoregressions. *Statistics and Probability Letters*, 1997.

[15] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[16] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo, et al. The tao of parallelism in algorithms. *ACM Sigplan Notices*, 46(6):12–25, 2011.

[17] E. Plaku and L. E. Kavraki. Distributed computation of the k nn graph for large high-dimensional point sets. *Journal of parallel and distributed computing*, 67(3):346–359, 2007.

[18] N. Satish, N. Sundaram, M. A. Patwary, J. Seo, J. Park, M. A. Hassaan, S. Sengupta, Z. Yin, and P. Dubey. Navigating the maze of graph analytics frameworks using massive graph datasets. In *Proc. 2014 ACM SIGMOD Int'l. Conf. on Management of Data*, 2014.

[19] C. Silpa-Anan and R. Hartley. Optimised kd-trees for fast image descriptor matching. In *Computer Vision and Pattern Recognition, 2008. CVPR 2008. IEEE Conference on*, pages 1–8. IEEE, 2008.

[20] R. Sproull. Refinements to nearest-neighbor searching in k-dimensional trees. *Algorithmica*, 6(4):579–589, 1991.

[21] A. Uzilov, J. Keegan, and D. Mathews. Detection of non-coding rnas on the basis of predicted secondary structure formation free energy change. *BMC Bioinformatics*, 7, 2006.