Parallel Algorithms for Constructing Range and Nearest-Neighbor Searching Data Structures*

Pankaj K. Agarwal Kyle Fox Kamesh Munagala
Abhinandan Nath
Duke University
{pankaj,kylefox,kamesh,abhinath}@cs.duke.edu

ABSTRACT

With the massive amounts of data available today, it is common to store and process data using multiple machines. Parallel programming platforms such as MapReduce and its variants are popular frameworks for handling such large data. We present the first provably efficient algorithms to compute, store, and query data structures for range queries and approximate nearest neighbor queries in a popular parallel computing abstraction that captures the salient features of MapReduce and other massively parallel communication (MPC) models. In particular, we describe algorithms for kd-trees, range trees, and BBD-trees that only require $O(1)$ rounds of communication for both preprocessing and querying while staying competitive in terms of running time and workload to their classical counterparts. Our algorithms are randomized, but they can be made deterministic at some increase in their running time and workload while keeping the number of rounds of communication to be constant.

1. INTRODUCTION

Rapid advances in sensing technologies as well as Internet applications such as social networks have led to unprecedented increase in the size and quantity of data sets. The term "big data" has become ubiquitous to describe data that cannot be stored or processed on a single machine. The MapReduce platform [18], its open source implementation Hadoop [43], and related platforms (such as Pregel [37], Spark [44], and Google Cloud Dataflow [7]) have emerged as dominant computing platforms for big-data processing. At a high level, these systems focus on computation local to the data stored on individual machines and have become popular due to their ability to abstract away the distributed nature of data storage and processing. The data itself is stored on disk in a distributed file system, for example, the Hadoop Distributed File System (HDFS) of the Hadoop platform [41]. A distributed file system can be treated as a cluster of machines, each with its own disk space where the data is stored. A high level programming language such as Pig [1] is used to write a program that is pushed to the machines where the data is stored and processed.

Motivated by social networks, target advertisements, and other applications, there has been a flurry of research activity on computing graph theoretic and machine learning primitives in MapReduce and other big data platforms. Moreover, advances in mapping and sensing technologies have led to the emergence of large geometric data sets as well. Even non-geometric data is often mapped to a set of points in a geometric space. One of the important problems in databases, GIS, and computational geometry is to answer various queries on such data. One could conceivably scan the entire data to answer each query, but since several queries are answered on the same data, it is desirable to preprocess data into a data structure so that a query can be answered quickly. A popular approach to cope with big data in the context of query processing is to work with a small summary of data such as random samples, coresets, sketches, *etc.* These methods are successful for answering aggregation queries, but they do not work well when queries involve analyzing local structure of data such as nearest-neighbor queries or range-reporting queries (especially for small ranges). In such instances, one has to work with the entire data. This difficulty raises the problem of constructing data structures on big data platforms. In this paper, we study data structures for range-reporting and nearest-neighbor queries – two very popular queries on geometric data – on big data platforms. In particular, we develop efficient algorithms for constructing some of the classical data structures such as kd-trees, BBD-trees, and range trees.

Our model. The technology for big data computation is in a constant state of flux. For instance, in MapReduce [18], the output data from a phase of computation is shuffled and stored on disk in the distributed file system. Spark [44] builds on MapReduce and attempts to keep data in memory to speed up machine learning applications that require multiple passes over the same data. Many of these platforms also incorporate streaming and real-time primitives.

To avoid dependence on specifics of a particular platform, we present our algorithms in a simple but popular abstract model called the (basic) *massively parallel communication* (MPC) model, originally proposed by Beame *et al.* [14] (see also Andoni *et al.* [9]).

*Work on this paper is supported by NSF under grants CCF-1161359, IIS-1408846, CCF-1408784, IIS-1447554, CCF-1546392, and CCF-1513816, by an ARO grant W911NF-15-1-0408, and by Grant 2012/229 from the U.S.-Israel Binational Science Foundation

PODS'16, June 26-July 01, 2016, San Francisco, CA, USA
© 2016 ACM. ISBN 978-1-4503-4191-2/16/06…$15.00
DOI: http://dx.doi.org/10.1145/2902251.2902303

Let n be the size of an input instance, and let I be a set of m machines. For simplicity, we assume I to be $\{0, \ldots, m-1\}$. Set $s = n/m$, and for simplicity, assume s is an integer.[1] Each machine has $O(s)$ memory (or space). As is standard, we assume $s \geq n^{\alpha}$ for some positive constant $\alpha < 1$. Computation proceeds in rounds. In each round, each machine reads its input, does some computation, and emits some output, where each output item is marked with the ID of the machine for which it is input in the next round. The size of the input to and output from any machine is bounded by $O(s)$ in each round. Communication across machines occurs only between rounds.

We explicitly allow data to persist on machines between rounds of computation and after all computation has been performed, as long as the total amount of data stored on each machine never exceeds $O(s)$. By considering data storage as we do, we are able to build and store data structures for massive geometric data. The explicit persistence of data between rounds and our hard requirements on the space of each machine are the main differences between our model and the MPC model as described by Beame *et al.* [14]. Andoni *et al.* [9] also limit the space on each machine, but they do not explicitly consider persistent storage. In fact, some work in similar models (*e.g.* Goodrich *et al.* [29]) require machines to communicate their own data to themselves in order for data to persist between rounds of computation.

Queries to our data structures behave as any other MPC computation. They simply take advantage of the distributed data structures already stored in the machines to reduce query time. When preprocessing a set of points P to build our data structures, we assume the points of P are distributed arbitrarily throughout the machines. Individual queries are sent to an arbitrary machine.

The efficiency of an algorithm is measured using three metrics: the number of rounds of computation R, the running time T, and the total work W. The first is obvious; we describe the latter two below. For machine $\beta \in I$, let $t_{\beta r}$ denote the computation time spent by this machine in round r. The *running time* is defined as

$$T = \sum_{r=1}^{R} \max_{\beta \in I} t_{\beta r}.$$

Note that this definition does not count the time it takes to communicate between the machines, which is accounted for separately by the quantity R.

The *total work* is defined as

$$W = \sum_{r=1}^{R} \sum_{\beta \in I} t_{\beta r}.$$

In order to make guarantees about our total work, we assume a machine performs zero work in a single round of computation if it does not receive any input or communication at the beginning of that round. This assumption can be interpreted as each machine "sleeping" through a round of computation unless it is given a reason to wake up, and this makes it possible for W to be significantly smaller than $m \cdot T$.

While most prior work on the MPC model focuses on minimizing the number of computation rounds R of an algorithm, e.g., [14, 23, 31], we must consider all three metrics in order to say anything interesting about geometric data structures. Consider processing a set of points $P \subseteq \mathbb{R}^2$ so that one may report the points lying within an axis-aligned query rectangle. If number of rounds were our only concern, we would build a trivial data structure in one round of computation by doing nothing. A query for rectangle $\square_q$ would then be performed in one round of computation as well by having each machine individually check which of its points lie inside $\square_q$. However, queries will take $O(s)$ time and require $O(n)$ work. We could slightly improve our data structure by building *kd*-trees for each machine's set of points using one round, $O(s \log s)$ time, and $O(n \log n)$ total work (see Section 3). The runtime of a query would improve to $O(\sqrt{s})$ in the worst case, but queries would still require $\Omega(m)$ work in the best case and $O(\sqrt{s}m) = O(n/\sqrt{s})$ work in the worst case. Unless s is nearly as large as n, these worst-case workloads are significantly higher than what is possible using classical sequential data structures. Therefore, we will focus on minimizing all three performance metrics.

Our results. Let P be a set of n points in $\mathbb{R}^d$, where d is a constant. We present efficient algorithms for building and querying *kd*-trees, range trees, and BBD-trees on P. The first two are used for range queries and the last one is used for approximate nearest-neighbor (NN) queries. They are formally defined in later sections; see [12, 17] for details. A *kd*-tree uses linear space and $O(n^{1-1/d} + k)$ query time to answer an orthogonal range query (*i.e.*, reporting all k points of P lying in a query axis-parallel rectangle). It can be computed in $O(n \log n)$ time. In contrast, a range tree uses $O(n \log^{d-1} n)$ space and $O(\log^{d-1} n + k)$ query time; range trees with slightly smaller space are also known [5]. A BBD-tree can answer an ε-approximate NN query in $O(c_{d,\varepsilon} \log n)$ time using $O(n)$ space where $c_{d,\varepsilon}$ is a constant dependent on d and ε.

For all of the data structures we consider, we will show a bound of $R = \text{poly}(\log_s n) = O(1)$ on the number of rounds of computation required both to build the data structure and to perform queries. Our running times T for building our data structures and performing queries will be comparable to the best sequential algorithms when performed on point sets of size s and our total work W will be comparable with the best sequential algorithms for point sets of size n. More precisely, we obtain the following results; see Table 1 for a summary.

- A *kd*-tree on P can be built in $O(1)$ rounds, $O(s \log s)$ time, and $O(n \log n)$ work with probability at least $1 - \frac{1}{n^{\Omega(1)}}$.[2] Queries can be answered using $O(1)$ rounds and $O(n^{1-1/d} + k)$ work where k is the output size; the running time is $O(s^{1-1/d} + k')$ where k' is the maximum number of points reported by a machine.
- A BBD-tree on P can be built in $O(1)$ rounds, $O(s \log s)$ time, and $O(n \log n)$ work with probability at least $1 - \frac{1}{n^{\Omega(1)}}$. Queries can be answered in $O(1)$ rounds, $O\left(\frac{1}{\varepsilon^d} \log\left(\frac{1}{\varepsilon}\right) \log n\right)$ time, and $\frac{1}{\varepsilon^{O(d)}} \log n$ work.
- A range tree on P can be built in $O(1)$ rounds, $O(s \log^d s)$

[1]For simplicity, we will ignore floor and ceiling operators throughout this paper and assume appropriate choice of parameters so that the ratios are integers when needed.

[2]The *kd*-tree we build is a slight variant of the standard *kd*-tree in that unlike the latter, our tree is not exactly balanced in partitioning the points. Nevertheless, we show that the space, query, and preprocessing requirements are asymptotically identical.

time, and $O(n \log^d n)$ work.[3] Queries can be answered in $O(\log^d n + k)$ time and work in $O(1)$ rounds of computation, where k is the number of points reported.

The algorithm for kd-trees can be extended to many variants of kd-trees such as hB-trees [36], hB^π-trees [24], and box trees [4]. It can also be extended to construct a partition tree [15] that is used for answering simplex range searching queries, *i.e.*, reporting all points that lie inside a query simplex.

The algorithms for building kd-trees and BBD-trees use randomization. However, they can be derandomized while keeping the number of computation rounds constant by increasing the running time – we obtain a tradeoff between the number of rounds and the running time.

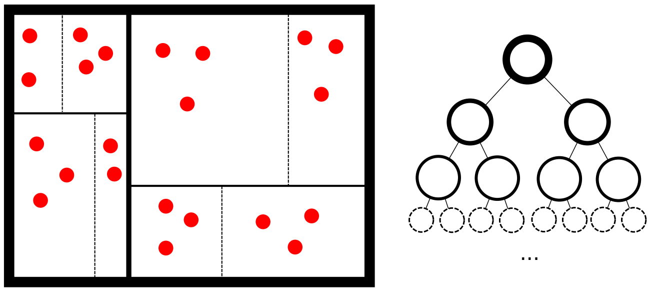

Figure 1. A kd-tree partitions space into a hierarchy of boxes. Each box is represented by a node in a tree, and nesting boxes are represented by ancestor-descendant relationships within the tree.

We remark that there is extensive work on I/O-efficient data structures in databases. For example, the kdB-tree is an I/O-efficient version of a kd-tree [10]. Similarly I/O-efficient range trees have been proposed [10]. Our algorithms can be modified easily to construct I/O-efficient versions of the data structures.

Our techniques. Our algorithms for constructing data structures depend upon the concept of ε-approximations defined formally in Section 2.1. Intuitively, an ε-approximation is a small subset of points $S \subset P$ such that any low complexity region $\mathcal{R}$ of $\mathbb{R}^d$ contains about the same fraction of points of S as that of P. One property common to most of the data structures we build is that they induce a hierarchical decomposition of $\mathbb{R}^d$ represented by a balanced tree. See Figure 1. In a constant number of rounds of computation, we sample an ε-approximation S of P. We choose S to be small enough to fit on a single machine, so we use a sequential algorithm to build the first few layers of the tree data structure using only points in S. The regions of $\mathbb{R}^d$ associated with the leaves of the partial tree partition S into approximately equal sized subsets. Because S is an ε-approximation of P, these same regions partition P into almost equal sized subsets as well. We recursively build the lower levels of the data structure on each piece of the partition.

Our algorithms are simple, and our analysis shows that a simple sampling based technique gives essentially the best possible running time in the MPC model, without using additional features that some modern data processing techniques may enable.

Sampling and the similar concept of filtering have been used before for the design of efficient algorithms for models based on MapReduce [22, 23, 32, 33]. However, these algorithms complete their work on a single machine after acquiring a small set of samples. In contrast, the partial trees our algorithms compute using their sample sets are not an entire solution; the algorithms must continue processing the remaining points in the input set using the partial trees to aid in the remaining computation.

Related work. A common concern when designing data structures for massive amounts of (geometric) data is the cost of reading and writing from disk, called an I/O operation. Aggarwal and Vitter [6] described the two-level I/O-memory model in an attempt to understand those costs. Extensive work has been done on developing range-searching data structures in this model. See the survey by Arge [10]. Agarwal *et al.* [3] describe I/O-optimal algorithms for constructing kd-trees and BBD-trees; however their techniques are not easily parallelizable.

An alternative to the two-level I/O model described above is the cache-oblivious model introduced by Frigo *et al.* [25]. In essence, algorithms for the traditional RAM model are evaluated in terms of performance using the two-level I/O model with *unknown* internal memory size and unknown block size. The analysis assumes transfers between internal and external memory are done using an optimal caching strategy. Again, there is extensive study of range-searching data structures in this model; see Arge *et al.* [11] for a survey.

While the I/O models above are designed with the storage of massive data in mind, they are still inherently *sequential.* In contrast, the Parallel Random Access Machine (PRAM) model ignores I/O complexities and instead models parallel processors sharing a common memory pool. Many efficient PRAM algorithms have been described; see Goodrich [28] for a survey.

The Bulk Synchronous Parallel (BSP) model [42] of Valiant models parallel processing, communication, and synchronization. A specialization of this model to multiprocessors is termed coarse grained parallelism (CGP) [19]. This model is similar to MPC; there are $p \leq n^\epsilon$ processors (where n is the input size and $\epsilon \leq 0.5$), each of which is allowed $O(n/p)$ communication between rounds, and arbitrary sequential computation. The goal is to simultaneously optimize the number of rounds, as well as the sequential computation done per processor. Dehne *et al.* [19] present optimal algorithms for several geometric problems. Simple deterministic partitioning of the point sets, however, suffice for the problems they consider. Furthermore, they do not focus on the more challenging problem of constructing data structures.

In order to theoretically model modern distributed programming frameworks, many authors have focused on variations of the MapReduce model [9, 14, 26, 29, 31, 33]. There is extensive work in database systems on developing algorithms under MapReduce and its variants, e.g., algorithms for large graph processing, join operations, query processing etc. See e.g. [30, 34, 40].

From a theoretical perspective, Goodrich *et al.* [29] describe a connection between MapReduce and the BSP model mentioned above, which leads to efficient implementation of some geometric algorithms. Andoni *et al.* [9] develop geometric approximation algorithms under MapReduce; see

[3]In order to have sufficient space just to store the range tree, we slightly amend our model so the memory and per-round input and output size of each machine is $O(s \log^{d-1} n)$.

Performance metric		*kd*-tree	BBD-tree	Range tree
Pre-processing	Rounds	$O(1)$	$O(1)$	$O(1)$
	Time	$O(s \log s)$	$O(s \log s)$	$O(s \log^d s)$
	Work	$O(n \log n)$	$O(n \log n)$	$O(n \log^d n)$
Query	Rounds	$O(1)$	$O(1)$	$O(1)$
	Time	$O(s^{1-1/d} + k')$	$O(\log n)$	$O(\log n + k)$
	Work	$O(n^{1-1/d} + k)$	$O(\log n)$	$O(\log^d n + k)$

Table 1. Our results.

also [19]. However the partitioning scheme used in [9] is based on randomly shifted quadtrees, hence largely independent of the input point set. On the other hand, we use a small sample of the input points in a clever way to partition the points. There is some work on similarity search in high dimensions in the distributed setting [13]. However, their focus is on reducing the amount of communication per round and is based on distributed locality-sensitive hashing, whereas our approach is quite different and is tailored for low dimensions. The most closely related works to this paper is MapReduce implementations for analyzing and querying spatial and geometric data, see [2, 8, 20–22] and references there in. For example, SpatialHadoop [22] is a full-fledged MapReduce framework that adapts traditional spatial index structures like R-tree and R+-tree to form a two-level spatial index. It is also equipped with other operations, including range query, k-nearest neighbors, and spatial join. However, we are not aware of any work on constructing range-searching and other geometric query data structures with *provable bounds* on performance, which is the main focus of this paper.

The rest of the paper is organized as follows. We describe some primitive operations useful for building geometric data structures in Section 2. We discuss *kd*-trees and an extension of our techniques to partition trees [15] in Section 3. We discuss BBD-trees and range trees in Sections 4 and 5, respectively. Finally, we conclude in Section 6 by mentioning some directions for future research.

2. MPC PRIMITIVES

Before describing our MPC primitives, we introduce the notion of an ε-approximation, which will be constructed by one of the primitives and which will be used by many of our algorithms.

2.1 Range spaces and ε-approximations

A *range space* Σ is a pair $(X, \mathcal{R})$, where X is a ground set and $\mathcal{R}$ is a family of subsets (*ranges*) of X. For example, X is a set of points in $\mathbb{R}^2$ and

$$\mathcal{R} = \{X \cap \square \mid \square \text{ is a rectangle in } \mathbb{R}^2\}.$$

A subset $X' \subseteq X$ is *shattered* by Σ if $\{X' \cap R \mid R \in \mathcal{R}\} = 2^{X'}$. The *VC dimension* of Σ is the size of the largest subset of X shattered by Σ. If there are arbitrarily large shattered subsets, the VC dimension of Σ is set to ∞.

Given range spaces $\Sigma_1 = (X, \mathcal{R}_1)$ and $\Sigma_2 = (X, \mathcal{R}_2)$ with VC dimensions δ_1 and δ_2 respectively, the union of Σ_1 and Σ_2 is defined as

$$(X, \mathcal{R}) = (X, \{R_1 \cup R_2 \mid R_1 \in \mathcal{R}_1, R_2 \in \mathcal{R}_2\}),$$

and has VC dimension $O(\delta_1 + \delta_2)$. The complement of Σ is defined as $(X, \overline{\mathcal{R}}) = (X, \{X \setminus R \mid R \in \mathcal{R}\})$ and has the same VC dimension as Σ. See Chazelle [16] for details.

Given a range space $\Sigma = (X, \mathcal{R})$ and $0 \le \varepsilon \le 1$, a subset $X' \subseteq X$ is called an *ε-approximation* of Σ if for any range $R \in \mathcal{R}$, we have

$$\left| \frac{|X' \cap R|}{|X'|} - \frac{|R|}{|X|} \right| \le \varepsilon.$$

The following bound on an ε-approximation was proved by Li *et al.* [35].

THEOREM 2.1 ([35]). *There is a positive constant c such that if $\Sigma = (X, \mathcal{R})$ is a range space with VC dimension δ, then a random subset of X of size $\frac{c}{\varepsilon^2}\left(\delta + \log \frac{1}{\psi}\right)$ is an ε-approximation of X with probability at least $1 - \psi$.*

For range spaces with constant VC dimension, a random subset of size $O\left(\frac{1}{\varepsilon^2} \log n\right)$ is an ε-approximation with probability at least $1 - 1/n^{\Omega(1)}$, where n is the size of the ground set.[4] The ranges that we consider in this paper are induced by axis-aligned boxes, simplices, and rectilinear regions defined by constant number of rectangles. Since these regions can be formed by taking a Boolean combination of a constant number of halfspaces, the range spaces induced by these regions have constant VC dimension.

2.2 Geometric primitives and techniques

We define a few primitive operations that we use to build our distributed data structures :

`PrefixSum`$(P, I, \text{rank}, \text{value})$**:** Given a set of points P stored on a contiguous subset of machines $I = \{i_0, i_0{+}1, \dots\}$, a rank function rank : $P \to \mathbb{Z}^+$, and a value function value : $P \to \mathbb{R}$, compute the prefix sum of values for each point $p \in P$ where rank(p) is the rank of point p in some sorted order.

`Broadcast`$(\mathcal{S}, I, \beta)$**:** Given a set of words $\mathcal{S}$, a contiguous set of machines $I = \{i_0, i_0 + 1, \dots\}$, and a machine β storing $\mathcal{S}$, copy $\mathcal{S}$ to all machines in I. We require $\mathcal{S}$ has size $O(s^{1/2})$.

`Partition`(P, I, Π, β)**:** Given a spatial subdivision Π of $\mathbb{R}^d$ into simple regions (e.g., rectangles, simplices), reorganize the points of P so that all points lying in a cell π of Π lie on a distinct contiguous subset of machines

[4]Better bounds for the size of ε-approximations exist for range spaces with finite VC dimension [16], but the bounds given here suffice for our purposes.

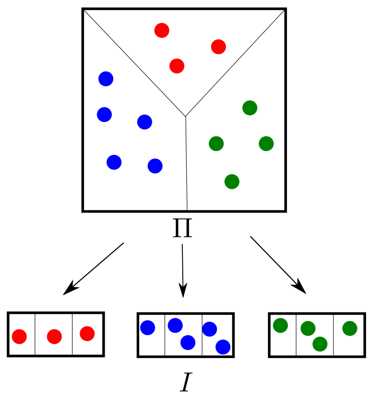

Figure 2. `Parition`(Π, I, β) reorganizes the points stored on I as dictated by the partition Π.

$I_\pi \subseteq I$. See Figure 2. We assume I itself is also a contiguous subset of machines $\{i_0, i_0+1, \dots\}$. We require that $|\Pi| = O(s^{1/2})$, $|\Pi| \le |I|$, and each cell π of Π contains $O(|P|/|\Pi|)$ points.

`Sample`(P, I, r, β)**:** Given a set of points P stored on a contiguous subset of machines $I = \{i_0, i_0+1, \dots\}$, compute a $(1/r)$-approximation $S \subseteq P$ of size $O(r^2 \log n)$ and send it to machine β. We require $|S| = O(s^{1/2})$.[5]

Before describing how to implement these primitives efficiently, we describe a procedure that will be used by these primitives.

To facilitate transfer of information between a given contiguous subset of machines $I = \{i_0, i_0+1, \dots\}$, we construct a tree T, a *complete* $s^{1/2}$*-ary tree* with $|I|$ leaves. Each node v of T is associated with a single machine $\beta(v) \in I$ such that each machine is used at most once per level of T. We let r_T denote the root of T and set $\beta(r_T)$ appropriately depending on the primitive. Let λ_T denote the depth of T. We have $\lambda_T = O(\log_{s^{1/2}} |I|) = O(1)$. Goodrich *et al.* [29] use a similar tree to compute prefix sums. A number of simple operations can be completed in a constant number of rounds of computation guided by T, by passing information between machines belonging to parent and child nodes. For example, it is straightforward to compute the max and sum of several numbers stored across the machines. While sending information from parent to children, the parent node's machine sends $O(s^{1/2})$ data to each child node's machine. If information is sent from children to parent, each child's machine sends $O(s^{1/2})$-size data to the parent's machine. Since each node has $s^{1/2}$ children, the input and output size for each machine is bounded by $O(s)$ in one round.
We now describe an implementation of above primitives.

`PrefixSum`$(P, I, \text{rank}, \text{value})$**:** It can be implemented in $O(1)$ rounds with linear time and work using an algorithm of Goodrich *et al.* [29]. Without giving details here, we state the bounds :

LEMMA 2.2. `PrefixSum`$(P, I, \text{rank}, \text{value})$ *can be implemented by a deterministic algorithm in* $O(1)$ *rounds,* $O(s)$ *time, and* $O(n)$ *total work.*

[5]Strictly speaking, the algorithm for computing a $(1/r)$-approximation depends on the underlying range space. In our applications, the range space associated with P will be clear from the context and will have constant VC dimension.

`Broadcast`$(\mathcal{S}, I, \beta)$**:** We use the tree T over the machines in I. Our broadcast procedure takes exactly $\lambda_T + 1 = O(1)$ rounds of computation. In round i, each machine $\beta(v)$ for a node v at level $i-1$ receives a copy of $\mathcal{S}$. If it has not done so already, $\beta(v)$ stores a copy of $\mathcal{S}$. When the round of computation ends, $\beta(v)$ sends a copy of $\mathcal{S}$ to each machine $\beta(v')$ where v' is a child of v in T.

LEMMA 2.3. `Broadcast`$(\mathcal{S}, I, \beta)$ *can be implemented by a deterministic algorithm using* $O(1)$ *rounds of computation,* $O(s)$ *time, and* $O(n)$ *total work.*

`Partition`(P, I, Π, β)**:** Order the cells of Π in an arbitrary way, and let $\{\pi_0, \pi_1, \dots\}$ be the cells of Π. We let $I_{\pi_k} = \{i_0 + k(|I|/|\Pi|), \dots, i_0 + (k+1)(|I|/|\Pi|) - 1\}$. We begin by running `Broadcast`(Π, I, β) to store Π on all the machines. In order to distribute the points of each partition cell π evenly across I_π, we use the `PrefixSum` primitive to count for each machine-cell pair (i, k) the points that are either in cells $\{\pi_0, \dots, \pi_{k-1}\}$ or in cell π_k and stored on machines $\{i_0, \dots, i-1\}$. Using these counts, we can then quickly finish the partitioning procedure.

Each machine $i \in I$ creates a set of $|\Pi|$ *counting points* C_i to be given to the `PrefixSum` primitive. The point $q_{i,k} \in C_i$ is associated with the cell π_k. Let $\text{rank}(q_{i,k}) = k|I| + i - i_0 + 1$, and let $\text{value}(q_{i,k})$ be equal to the number of points on machine i in cell π_k. Let $C = \bigcup_{i \in I} C_i$. We run `PrefixSum`$(C, I, \text{rank}, \text{value})$. The prefix sum of counting point $q_{i,k}$ is the number of points in cell π_k stored on machines $\{i_0, \dots, i-1\}$ plus the number of points in cells $\{\pi_0, \dots, \pi_{k-1}\}$. Now, let $\{p_0, p_1, \dots\} \subseteq P$ be a set of points in arbitrary order belonging to some machine i and partition cell π_k. Let $q_{|I|+1,k} = q_{|I|,k} + \text{value}(|I|, k)$. Each value $q_{i+1,k}$ can be sent to machine i in one round of communication. Machine i then sends point p_j to machine $i_0 + k(|I|/|\Pi|) + \lfloor (q_{i+1,k} - q_{i,k} + j)/(|P|/|\Pi|) \rfloor$.

LEMMA 2.4. `Partition`(P, Π, I, β) *can be implemented by a deterministic algorithm using* $O(1)$ *rounds of computation,* $O(s \log s)$ *time, and* $O(n \log n)$ *total work.*

`Sample`(P, I, r, β)**:** We first describe a simple Las Vegas algorithm for computing the sample $S \subseteq P$. Each machine i selects a subset of its points S_i by selecting points independently, each with probability $p = c(r^2/|P|) \ln n$ for some constant c that depends on the VC dimension of the underlying range space. We then use the tree T to compute the total *number* of selected points in $\lambda_T + 1 = O(1)$ rounds of computation by summing the values $|S_i|$ across each machine. If this number is more than $2cr^2 \ln n$ or less than $(c/2)r^2 \ln n$, an incorrect number of points have been selected and the procedure restarts. Otherwise, all the selected points from each machine are sent to β to form the set S. A standard Chernoff bound [39] guarantees the probability of restarting even once is at most $\exp(-(c/8)r^2 \ln n) \le 1/n^2$ for large enough c. Each sample of size $|S|$ is chosen with equal probability, so S is an $(1/r)$-approximation with probability at least $1 - 1/n^2$ as well by Theorem 2.1.

We can also choose S deterministically as follows. We use the tree T and let $\beta(r_T) = \beta$. Our sampling procedure still takes $\lambda_T + 1 = O(1)$ rounds of computation. In round i, each machine $\beta(v)$ for a node v at depth $\lambda_T - i + 1$ receives a set S_v of $O(s)$ points from its children that may be used in the $(1/r)$-approximation. If v is a leaf of T, then $\beta(v)$ simply uses the members of P initially found at $\beta(v)$. Machine $\beta(v)$

computes a (c/r)-approximation $S'_v \subseteq S_v$, for some sufficiently small constant c, of size $O(r^2 \log n) = O(s^{1/2})$ using the deterministic algorithm of Matoušek [38] in $O(s(r^2 \log r)^\delta)$ time. If $v = r_T$, then S'_v is the final $(1/r)$-approximation desired by the algorithm. When the round of computation ends, $\beta(v)$ sends set S'_v to $\beta(p(v))$, the machine corresponding to the parent of v.

LEMMA 2.5. *The above procedure computes a $(1/r)$-approximation of P.*

PROOF. Let P_v be the subset of points contained in the subtree of T rooted at v, and let λ_v be the depth of v. Fix a node v, and assume inductively that $S'_{v'}$ is a $(\gamma^{\lambda_{v'} - \lambda_T} c/r)$-approximation of $P_{v'}$ for each descendant v' of v for some constant γ. The children of v can be divided into three groups: the children where every descendant leaf has depth λ_T, children where every descendant leaf has depth $\lambda_T - 1$, and the solitary child with descendant leaves of both depths. Let V_1, V_2, and V_3 be these respective groups of child nodes. For any pair of children v_1, v_2 in a single group, there exist constants c' and c'' such that $c'|P_{v_2}| \le |P_{v_1}| \le c''|P_{v_2}|$ and $c'|S'_{v_2}| \le |S'_{v_1}| \le c''|S'_{v_2}|$. Let v' be an arbitrary node of V_i, and let $x_i = |S'_{v'}|$ and $y_i = |P_{v'}|$. Let R be an arbitrary range in our range space. We have

$$\begin{aligned}
&\left| \frac{|S_v \cap R|}{|S_v|} - \frac{|P_v \cap R|}{|P_v|} \right| \\
&\quad \le \sum_{i=1}^{3} \left| \frac{|(\bigcup_{v' \in V_i} S'_{v'}) \cap R|}{|\bigcup_{v' \in V_i} S'_{v'}|} - \frac{|(\bigcup_{v' \in V_i} P_{v'}) \cap R|}{|\bigcup_{v' \in V_i} P_{v'}|} \right| \\
&\quad \le \sum_{i=1}^{3} \left| \frac{|(\bigcup_{v' \in V_i} S'_{v'}) \cap R|}{c'|V_i| x_i} - \frac{|(\bigcup_{v' \in V_i} P_{v'}) \cap R|}{c''|V_i| y_i} \right| \\
&\quad \le \sum_{i=1}^{3} \frac{1}{|V_i|} \sum_{v' \in V_i} c''' \left(\left| \frac{|S'_{v'} \cap R|}{|S'_{v'}|} - \frac{|P_{v'} \cap R|}{|P_{v'}|} \right| \right) \\
&\quad \le \gamma^{\lambda_v - \lambda_T} c/2r
\end{aligned}$$

where c''' is a constant.

Therefore, S_v is a $(\gamma^{\lambda_v - \lambda_T} c/2r)$-approximation of P_v. Set S'_v is a $(\gamma^{\lambda_v - \lambda_T} c/r)$-approximation [38, Observation 4.3]. T has depth $O(1)$, so S'_{r_T} is a $(1/r)$-approximation of $P_{r_T} = P$ when c is sufficiently small.[6] □

LEMMA 2.6. *(i) `Sample`(P, I, r, β) can be implemented by a Las Vegas algorithm using $O(1)$ rounds of computation, $O(s)$ time, and $O(|P|)$ total work, with probability at least $1 - 1/n^{\Omega(1)}$.*
(ii) The procedure can be implemented by a deterministic algorithm using $O(1)$ rounds of computation, $O(s(r^2 \log r)^\delta)$ time, and $O(|P|(r^2 \log r)^\delta)$ total work.

3. *KD*-TREE

Given a set of n points $P \in \mathbb{R}^d$, a kd-tree on P can answer an orthogonal range-reporting query in $O(n^{1-1/d} + k)$ time using $O(n)$ space, where k is the number of points lying inside the query rectangle. Each node v of the tree is associated with a d-dimensional rectangle $\square_v$, called the cell of v, with the root cell being large enough to contain the entire set P. Let $P_v = P \cap \square_v$. If $|P_v| \le 1$, v is a leaf and P_v is stored at v. If $|P_v| > 1$, then $\square_v$ is split into two cells $\square_w$ and $\square_z$ by an axis-parallel hyperplane h_v such that the interior of each cell contains at most $|P_v|/2$ points of P_v. If the depth of v is i, then h_v is parallel to the $((i \bmod d) + 1)$-th axis. The cell $\square_w$ (resp. $\square_z$) is associated with the child w (resp. z) of v. Since the partitions are balanced, the tree has height $O(\log n)$. The size of the tree is $O(n)$. It is well known that the tree can be constructed in $O(n \log n)$ time by first sorting P along each axis and then constructing it level by level in a top-down manner, spending $O(n)$ time at each level. See [17] for details.

Queries are performed recursively in a top-down manner starting with the root node. Given a query rectangle ρ and a node v of the tree, there are three cases : (a) $\square_v \subseteq \rho$, then all points of P_v are reported , (b) $\square_v \cap \rho = \varnothing$, there is nothing to be done, (c) $\square_v \cap \partial\rho \neq \varnothing$, we recurse on the children of v.

[6]Our procedure and its proof require a fairly small constant c. One can increase the constant by guaranteeing each set $P_{v'}$ has equal size and each set $S_{v'}$ has equal size by choosing T carefully and adding $O(n)$ additional points. While possible, doing so would be very tedious in our model.

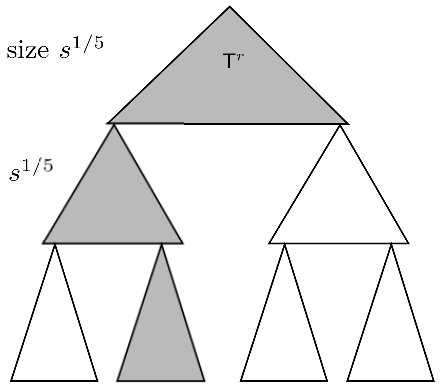

Figure 3. T^r is stored on a single machine. The leaves of T^r partition P into smaller subsets, each represented by their own recursively constructed kd-trees. Shaded subtrees may be stored on a single machine; however the space used on a single machine is still $O(s)$.

3.1 An MPC algorithm

We now describe an algorithm for constructing the kd-tree, denoted by $\mathsf{T} := \mathsf{T}(P)$, of linear size that has $O(s^{1-1/d} + k)$ query time. Our algorithm uses only $O(1)$ rounds of computation to build the tree. T is constructed and stored recursively in a distributed fashion across all machines. If $|P| = O(s)$, T is stored on a single machine and constructed using the sequential algorithm mentioned above. Otherwise we choose a parameter r, and the top subtree T^r of T containing $\log_2 r$ levels and $\Theta(r)$ leaves is built and stored on one machine, say β. The leaves of T^r partition P into subsets, each subset P_i associated with leaf l_i residing on its own consecutive set of machines I_i (the sets of machines being pairwise disjoint). For each such set P_i, $\mathsf{T}(P_i)$ is built and stored recursively using the machines in I_i. A machine may store multiple subtrees, one from each level of the recursion. However, the space used on a single machine is still $O(s)$. See Figure 3.

The key to our $O(1)$ round algorithm is the use of an ε-approximation to build the top subtree T^r. While the size of all $P'_i s$, the subsets associated with the leaves of T^r, will not be *exactly* the same, as for the standard *kd*-tree, the usage of ε-approximation will ensure that $|P_i| \leq 2|P|/r$, which in turn will guarantee that the height of T is at most $\log_2 n + O(1)$.

We now describe the recursive procedure `Build-kd-tree`$(S, \square, I)$, which for a rectangle $\square$, builds the tree $T(S)$ on $S = P \cap \square$ using a contiguous subset I of $\frac{|S|}{n}m$ machines, say $\{\beta, \beta+1, \ldots, \beta + \frac{|S|}{n}m - 1\}$. If $|I| = 1$, i.e., $|S| = O(s)$, then $\mathsf{T}(S)$ is constructed on machine β using the sequential algorithm. So assume $|I| > 1$.

Set $r = s^{1/5}$. Let $\Sigma = (S, \mathcal{R})$ be the range space where the ranges are induced by rectangles, i.e., $\mathcal{R} = \{S \cap \square \mid \square \text{ is a rectangle}\}$. We compute a $(1/r)$-approximation R of Σ of size $cr^2 \ln n$, for some constant $c > 0$, by calling the (randomized) procedure `Sample`(S, r, β). We compute on machine β the top subtree T^r of the standard *kd*-tree on R so that $|\mathsf{R}_v| \leq cr \ln n$ for every leaf v of T^r. The height of T^r is $\log_2 r$. Let `Partial-kd-tree`(R, r) denote this procedure. We assume this procedure returns the tree T^r as well as the partition Π' of $\square$ induced by the rectangles associated with the leaves of T^r. By calling `Partition`(S, I, Π', β), we partition S so that for each rectangle $\square_v$ of Π', $S_v = S \cap \square_v$ lies in a contiguous subset I_v of machines of I; $|I_v| = \frac{|S_v|}{n} \cdot m$. If $|S_v| > \frac{2|S|}{r}$ for some leaf v of T^r, we discard R and T^r, and repeat the above step. Otherwise, we recursively compute $(S_v, \square_v, I_v)$ for all leaves v of T^r. Algorithm 1 describes the pseudocode.

Algorithm 1 `Build-kd-tree`$(S, \square, I)$

```
1: If $|I| = 1$, compute $\mathsf{T}(S)$ sequentially.
2: $r = s^{1/5}$.
3: $\mathsf{R} \leftarrow$ Sample$(S, r, \beta)$.
4: $\mathsf{T}^r, \Pi' \leftarrow$ Partial-kd-tree$(\mathsf{R}, r)$.
5: Partition$(S, I, \Pi', \beta)$.
6: for all $\square_v \in \Pi'$ in parallel do
7:     Build-kd-tree$(S_v, \square_v, I_v)$
8: end for
```

LEMMA 3.1. *T^r has $\Theta(r)$ leaves. If R is a $(1/r)$-approximation of Σ then $|S_v| \leq \frac{2|S|}{r}$ for all leaves of T^r.*

PROOF. Since the depth of T^r is $\log_2 r$, it has $\Theta(r)$ leaves. For any leaf $v \in \mathsf{T}^r$, let $\mathsf{R}_v = \mathsf{R} \cap \square_v$. If R is a $(1/r)$-approximation of Σ, then

$$\left| \frac{|S_v|}{|S|} - \frac{|\mathsf{R}_v|}{|\mathsf{R}|} \right| \leq \frac{1}{r}.$$

Therefore,

$$|S_v| \leq |S| \left(\frac{1}{r} + \frac{|\mathsf{R}_v|}{|\mathsf{R}|} \right) \leq |S| \left(\frac{1}{r} + \frac{cr \ln n}{cr^2 \ln n} \right) = \frac{2|S|}{r}.$$

$\square$

Since R is a $(1/r)$-approximation with probability at least $1 - 1/n^{\Omega(1)}$, the algorithm succeeds in one attempt with probability at least $1 - 1/n^{\Omega(1)}$. The depth of recursion is $O(\log_r n)$. Since $r = s^{1/5}$ and we assume $s = n^\alpha$ for some constant $\alpha > 0$, the depth of recursion is $O(1)$. By Lemmas 2.4 and 2.6 (i), the running time of the algorithm is $O(s \log s)$ and the total work performed is $O(n \log n)$ with probabillity at least $1 - 1/n^{\Omega(1)}$.

LEMMA 3.2. *The height of the tree constructed by the algorithm is at most $\log_2 n + O(1)$.*

PROOF. The base case, i.e. $|I| = 1$, constructs a subtree of height $\log_2 s$. `Partial-kd-tree` constructs a subtree of height at most $\log_2 r$. Since the depth of the recursion is at most $\log_{r/2} \left(\frac{n}{s}\right)$, the total height of the tree is at most

$$\begin{aligned}
&\log_2 s + \log_{r/2} \left(\frac{n}{s}\right) \cdot \log_2 r \\
\leq\ & \log_2 s + \log_{r/2} \left(\frac{n}{s}\right) \left(1 + \log_2 \left(\frac{r}{2}\right)\right) \\
\leq\ & \log_2 s + \log_2 \left(\frac{n}{s}\right) + \left(\frac{\log_2(n/s)}{\log_2 r - 1}\right) \\
\leq\ & \log_2 n + \frac{(1-\alpha)\log_2 n}{(\alpha/5)\log_2 n - 1} \\
=\ & \log_2 n + O(1).
\end{aligned}$$

$\square$

Alternatively, the deterministic version of `Sample`(S, r, β) can be used to construct a $(1/r)$-approximation R. Since the deterministic procedure is more expensive, we choose $r = s^\beta$ where $\beta \leq 1/5$ is a sufficiently small constant. The depth of recursion remains $O(\log_r n) = O(1/\beta)$. By Lemma 2.6 (ii), the running time of the algorithm is $s^{1+O(\beta)}$ and the total work peformed by the algorithm is $n^{1+O(\beta)}$.

3.2 Query procedure

Given a query rectangle ρ and the set of machines I containing T, in the first round we perform the query locally on the machine β containing the top subtree T^r. This gives us a set of leaves of T^r whose squares intersect with ρ. For each such leaf v, the query is performed recursively in parallel on the subtree rooted at v contained in the machines I_v.

The number of levels of recursion is the same as that of the construction algorithm, i.e., $O(1)$. This is also the number of rounds of computation required. The following lemma bounds the query procedure's work.

LEMMA 3.3. *The total work performed by the query procedure is $O(n^{1-1/d} + k)$ where k is the number of points in the query rectangle.*

PROOF. For simplicity, we prove the lemma for $d = 2$; the proof is similar for higher values of d. Let ρ be a query rectangle. The total work performed by the query procedure is $O(k)$ plus the number of nodes v in$\mathsf{T}$ such that $\partial\rho$ intersects $\square_v$. Fix an edge e of ρ. Since the splitting line alternates between being horizontal and vertical, e intersects the cells associated with at most two grandchildren of a node v; see [17]. Let $\varphi(h)$ denote the number of nodes in a subtree of height h that intersect e. We obtain the following recurrence:

$$\varphi(h) \leq 2\varphi(h-2) + 3.$$

The solution to the above recurrence is $\varphi(h) = O(2^{h/2})$. By Lemma 3.2, the height of T is at most $\log_2 n + O(1)$. We obtain that e intersects the cells associated with $O(\sqrt{n})$ nodes of T. Hence, $\partial\rho$ intersects $O(\sqrt{n})$ cells of T. This completes the proof of the lemma. $\square$

The time required by the query procedure can be similarly bounded by $O(s^{1-1/d}+k')$, where k' is the maximum number of points reported by a single machine. We thus have the following.

THEOREM 3.4. *Let P be a set of n points in $\mathbb{R}^d$. A kd-tree on P can be constructed in the MPC model so that an orthogonal range-reporting query can be answered using $O(1)$ rounds of computation and $O(n^{1-1/d}+k)$ work, where k is the output size; the running time is $O(s^{1-1/d}+k')$ where k' is the maximum number of points reported by a machine. The tree can be built in $O(1)$ rounds, $O(n\log n)$ work, and $O(s\log s)$ time with probability $1-1/n^{\Omega(1)}$. Alternatively, for a parameter $\beta \le 1/5$, it can be constructed deterministically in $O(1/\beta)$ rounds, $s^{1+O(\beta)}$ time, and $n^{1+O(\beta)}$ work.*

3.3 Partition trees

Given a set of n points P in $\mathbb{R}^d$, Chan [15] described a partition tree that can answer simplex range-reporting queries (*i.e.,* reporting points lying in a simplex) in $O(n^{1-1/d}+k)$ time using $O(n)$ space. The expected time to construct the partition tree is $O(n\log n)$. Like *kd*-trees, partition trees represent a hierarcical decomposition of space into cells; however each cell is a simplex. The number of points within a cell decreases by a constant factor at every level, so the tree has height $O(\log n)$. Any arbitrary hyperplane intersects at most $O(n^{1-1/d})$ cells in the tree, hence a simplicial range-reporting query can be answered in $O(n^{1-1/d}+k)$ time, where k is the number of points reported.

Our algorithm for constructing a *kd*-tree can be extended to build a partition tree in the MPC model in $O(1)$ rounds of computation. Omitting all the details, we conclude the following:

THEOREM 3.5. *Let P be a set of n points in $\mathbb{R}^d$. A partition tree on P can be constructed in the MPC model so that a simplicial range-reporting query can be answered using $O(1)$ rounds of computation and $O(n^{1-1/d}+k)$ work, where k is the output size; the running time is $O(s^{1-1/d}+k')$ where k' is the maximum number of points reported by a machine. The tree can be built in $O(1)$ rounds in expected time $O(s\log s)$ and $O(n\log n)$ expected work. Alternatively, for a parameter $\beta \le 1/5$, it can be constructed deterministically in $O(1/\beta)$ rounds, $s^{1+O(\beta)}$ time, and $n^{1+O(\beta)}$ work.*

4. NN SEARCHING AND BBD-TREE

Given a set of points P and a query point q in $\mathbb{R}^d$, a $(1+\varepsilon)$-nearest neighbor (ε-NN) of q is a point in P whose distance from q is within a factor of $(1+\varepsilon)$ of the distance between q and its closest point in P. The goal is to process P into a data structure so that for a query point q, an ε-NN of q in P can be reported quickly. Arya et al. [12] proposed the *balanced-box decomposition* (BBD) tree for answering ε-NN queries. This section describes building a BBD-tree in the MPC model.

Given n points $P \in \mathbb{R}^d$, the BBD-tree of P, denoted by $\mathsf{B}(P) := \mathsf{B}$, is a binary tree of height $O(\log n)$. A node $v \in \mathsf{B}$ stores a cell $\square_v$ and a representative point $p_{\square_v} \in P$ lying inside $\square_v$. The cell $\square_v$ is either a d-dimensional rectangle or the region between two nested rectangles. All rectangles have *aspect ratio* (the ratio between the longest and the shortest side) bounded by 3. The root cell is large enough to contain the entire set P. The cell $\square_v$ (with possibly a hole inside) is split into two cells $\square_w$ and $\square_z$ in one of two ways:

(a) by an axis-parallel hyperplane not intersecting the hole (if any), or

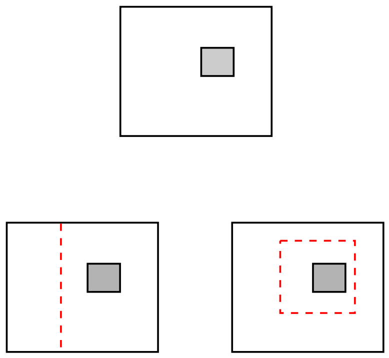

Figure 4. Two ways to divide a cell (with one hole) into two cells in a BBD tree.

(b) by an axis-parallel rectangle containing the hole (if any).

See Figure 4. The cell $\square_w$ (resp. $\square_z$) is associated with the child w (resp. z) of v. Like a *kd*-tree, a BBD-tree induces a hierarchical partition of $\mathbb{R}^d$. The *size* of a cell is the length of its longest side. Arya et al. show that cells at each node can be split in a way so that the number of points inside the cells reduces by at least a factor of 2/3 every 4 levels of B, and the size of cells decreases by at least a factor of 2/3 every $4d$ levels of the tree. Each leaf cell contains at most one input point. The crucial observation is that the range space induced by the cells in a BBD-tree has constant VC-dimension (see Section 2.1), and we can adapt the *kd*-tree construction algorithm for building the BBD-tree. The only difference is that instead of building a *kd*-tree locally in the procedure `Partial-kd-tree`, we build a local BBD-tree. The tree is also stored in a manner similar to the *kd*-tree, i.e., $\mathsf{B}(P)$ is stored in a distributed fashion across all machines.

Query procedure. We first describe a sequential query procedure that is slightly different from the one in [12], and then show how to implement it in the MPC model.

Given a query point q, the query procedure proceeds top-down, level by level. At each step, it maintains an estimate r_{curr} of distance from q to its nearest neighbor, and it stores the set of active nodes in a queue Q. Algorithm 2 summarizes the query procedure.

Algorithm 2 `NN-Query`(q)

```
 1: $r_{curr} \leftarrow ||qp_{root}||$, $p_{curr} \leftarrow p_{root}$.
 2: $Q \leftarrow \{root\}$.
 3: while $Q \neq \varnothing$ do
 4:     $v \leftarrow$ Dequeue$(Q)$.
 5:     if $||qp_v|| < r_{curr}$ then
 6:         $r_{curr} \leftarrow ||qp_v||$, $p_{curr} \leftarrow p_v$.
 7:     end if
 8:     if dist$(q, \square_v) < \frac{r_{curr}}{1+\varepsilon}$ and $v$ not a leaf then
 9:         Enqueue$(w, Q)$ for all child $w$ of $v$.
10:     end if
11: end while
12: Return $p_{curr}$.
```

LEMMA 4.1. *The above query procedure returns an ε-NN of q.*

PROOF. Let p^* be the actual nearest neighbor of q. We show that at the end of the procedure $r_{curr} \leq (1+\varepsilon)||qp^*||$.

Note that the value of r_{curr} is non-increasing throughout the procedure. Let v be the last node examined by the procedure such that $p^* \in \square_v$. If v is a leaf, then $r_{curr} = ||qp^*||$. Otherwise, v was discarded, in which case $\mathrm{dist}(q, \square_v) \geq \frac{r_{curr}}{1+\varepsilon}$. However, $||qp^*|| \geq \mathrm{dist}(q, \square_v)$ and hence $r_{curr} \leq (1+\varepsilon)||qp^*||$. □

We use the following fact, proved as Lemma 4 in [12], to bound the number of cells examined at each level.

FACT 4.2. *Given a BBD-tree for a set of data points in $\mathbb{R}^d$, the number of cells of size at least $\Delta > 0$ that intersect a ball of radius r is at most $\lceil 1 + 6r/\Delta \rceil^d$.*

The next lemma is then a simple variant of Lemma 5 in [12].

LEMMA 4.3. *The query procedure visits at most $\lceil 1 + 6d/\varepsilon \rceil^d$ cells at any level.*

The MPC implementation of the query procedure is obtained by modifying Algorithm 2 as follows. Let q be a query point, and let I be the set of machines that store B. The first round runs Algorithm 2 on the machine β containing the top subtree B^r, having $O(\log r)$ levels. When the procedure finishes traversing B^r, by Lemma 4.3, Q has $O(\frac{1}{\varepsilon^d})$ leaves of B^r. For each such leaf v, we pass the values p_{curr} and r_{curr} to the machines I_v containing the subtree B_v rooted at v, and Algorithm 2 is then run recursively in parallel on all B_v for $v \in Q$. At the end, we return the point that is nearest to q among all the points returned by these subtrees.

The query procedure performs $O(1)$ rounds of computation. The number of recursive subproblems increases by a factor of $O\left(\frac{1}{\varepsilon^d}\right)$ at every level of recursion. The subproblems at each level of recursion run in parallel and take time $O\left(\frac{1}{\varepsilon^d}\log\left(\frac{1}{\varepsilon}\right)\log n\right)$. Since the number of recursive subproblems is $\frac{1}{\varepsilon^{O(d)}}$, the total work done is $\frac{1}{\varepsilon^{O(d)}}\log n$. We thus have the following.

THEOREM 4.4. *Given a set P of n points in $\mathbb{R}^d$, a BBD-tree on P can be built in the MPC model that can answer $(1+\varepsilon)$-nearest neighbor queries in $O(1)$ computation rounds, $O\left(\frac{1}{\varepsilon^d}\log\left(\frac{1}{\varepsilon}\right)\log n\right)$ time and $\frac{1}{\varepsilon^{O(d)}}\log n$ work, for some constant $c \geq 2$. The tree can be built in $O(1)$ rounds of computation, $O(n\log n)$ work, and $O(s\log s)$ time with probabillity $1 - 1/n^{\Omega(1)}$. Alternatively, for a parameter $\beta \leq 1/5$, it can be constructed deterministically in $O(1/\beta)$ rounds, $s^{1+O(\beta)}$ time, and $n^{1+O(\beta)}$ work.*

5. RANGE TREE

Given a set P of n points in $\mathbb{R}^d$, a d-dimensional range tree on P, denoted by $\mathsf{T} := \mathsf{T}(P)$, can answer orthogonal range queries in $O(\log^{d-1} n + k)$ time using $O(n\log^{d-1} n)$ space, where k is the number of points reported [17]. T is defined recursively. For $d = 1$, T is a sorted array or a balanced binary search tree. For $d > 1$, T consists of a primary tree $\mathsf{T}_0 := \mathsf{T}_0(P)$, and each node of T_0 stores a $(d-1)$-dimensional range tree as a secondary structure. T_0 is a balanced binary search tree on the x_1-coordinates of P. Each node $v \in \mathsf{T}_0$ stores an interval δ_v. Let P_v be the points whose x_1-coordinates lie in δ_v (the root has the set P and the interval spanning all of P). The interval δ_v is split into δ_w and δ_z so that $|P_w|, |P_z| \leq \lceil |P_v|/2 \rceil$, where w and z are the children of v. Let $P_v^\perp$ denote the projection of P_v onto the hyperplane $x_1 = 0$. Node v also has a secondary data structure $\mathsf{T}(P_v^\perp)$, a $(d-1)$-dimensional range tree on $P_v^\perp$. A simple recursive argument shows that the size of T is $O(n\log^{d-1} n)$, and that it can be constructed in $O(n\log^{d-1} n)$ time after sorting P along each of its coordinates.

Let $\rho = [a_1, b_1] \times \ldots \times [a_d, b_d]$ be a query rectangle, and let $\rho^\perp = [a_2, b_2] \times \ldots \times [a_d, b_d]$. Let w_a (resp. w_b) be the leaf of the primary tree such that $a_1 \in \delta_{w_a}$ (resp. $b_1 \in \delta_{w_b}$), and let v^* be the lowest common ancestor of w_a and w_b. Let V_ρ be the set of nodes v such that either v is the right child of its parent and the left sibling of v lies on the path from v^* to w_a or v is the left child of its parent and its right sibling lies on the path from v^* to w_b. It is known [17] that $P \cap \rho = \bigcup_{v \in V_\rho}(P_v \cap \rho)$ and a point $p \in P_v \cap \rho$, for $v \in V_\rho$, if and only if $p^\perp \in P_v^\perp \cap \rho^\perp$. Hence, we recursively query $P_v^\perp$ with $\rho^\perp$ for all $v \in V_\rho$.

5.1 An MPC algorithm

Since the total space required by T is $O(n\log^{d-1} n)$, we slightly amend our model so that the memory and I/O size for each machine per round is $O(s\log^{d-1} n)$. We still have m machines with $s = n/m$ and $s \geq n^\alpha$ for some positive constant $\alpha < 1$. For each $\ell \in \{1, \ldots, d\}$, let $s_\ell = s\log^{\ell-1} n$. We assume that the input points are initially distributed so that each machine contains $O(s_1) = O(s)$ points. This assumption can be guaranteed by redistributing the points using an algorithm similar to the `Partition` procedure described in Section 2.2. Our construction procedure recursively builds primary trees for each coordinate x_ℓ using $O(s_\ell)$ memory and I/O size per machine.

T is stored in a distributed fashion. Suppose we are building an ℓth level structure of the range tree (initially, $\ell = 1$). The primary tree T_0 for this level is built and stored in a manner similar to the *kd*-tree, using a procedure we call `Build-primary-tree`(P, I, ℓ), which is nearly the same as the one used to build the *kd*-tree. The only difference is that the range space is induced by intervals over the real line, whose VC-dimension is a constant. Briefly, if $|P| = O(s_\ell)$, then T_0 is stored on a single machine and constructed using the sequential algorithm. Otherwise, we choose a parameter r, and the top subtree T_0^r of T_0 containing $\log_2 r$ levels and $\Theta(r)$ leaves is built and stored on a machine β. The leaves of T_0^r again partition P, and the subtrees of T_0 rooted at these leaves are built and stored recursively using disjoint sets of machines. We can also build T_0 deterministically in the same amount of time, work, and rounds by using a simpler deterministic `Sample` procedure since our ranges are just intervals over the real line. We omit details for the deterministic procedure.

Each node v of T_0 has a pointer to a secondary structure. Each bottom-most subtree of T_0 of size $O(s_\ell)$ stored on a single machine has all the secondary structures of its nodes stored on the same machine. These secondary structures are built using the sequential algorithm in $O(s_\ell \log^{d-\ell} n)$ time and space each. The secondary structures for the remaining nodes are stored on disjoint subsets of I. See Figure 5.

To build the secondary structures, each point p is copied $\Theta(\log |P|)$ times, sending the copies to the disjoint sets of machines that will store $\mathsf{T}(P_v^\perp)$ for all P_v containing p. We name our copying procedure Copy-points(P, I, T_0). Each machine of $I = \{i_0, i_0 + 1, \ldots\}$ contains several points lying in the leaves of T_0. To facilitate our copying procedure, we inform each machine about the nodes of T_0 that lie above its points. Let β be the machine storing T_0^r. We run the procedure Broadcast$(\mathsf{T}_0^r, I, \beta)$ and then recursively repeat the broadcast procedure with the child subtrees of T_0^r and their disjoint sets of machines. In $O(1)$ rounds, each machine will receive the ancestor subtrees for its points.

Let $\lambda = \Theta(\log |P|)$ be the depth of T_0. Intuitively, the set I is divided into λ equal-sized sets of size $|I|/\lambda$, one for each level of T_0. Sets P_v are then distributed among disjoint subsets of machines from those machines set aside for level λ. Now, consider a node v of T_0. Let λ_v be the depth of node v in T_0, $\{v_0, v_1, \ldots\}$ be the nodes of T_0 lying at depth λ_v, and $v_j = v$. Let $I_v = \{i_v, i_v + 1, \ldots, \}$ be the set of machines storing the points P_v. Let $p \in P_v$, and let i be the machine storing p. Finally, let c be a sufficiently small constant. Machine i sends a copy of p to machine $i_0 + \lambda_v \cdot cm/\lambda + cmj/(2^{\lambda_v}\lambda) + \lfloor c(i - i_v)/\lambda \rfloor - 1$. Let I'_v be the set of machines that receive the points of P_v, the set that is used to construct the secondary structure for v. Each point is copied $\Theta(\log |P|)$ times, so the total communication out of i while copying points is $O(s_\ell \log |P|)$.

Our construction algorithm concludes by recursively building the secondary structures (at the $(\ell + 1)$st level) for each node v on the set of machines I'_v. We describe our construction procedure in Algorithm 3. The procedure Build-range-tree takes ℓ as one of its parameters to account for the storage required when building each level of the data structure.

Algorithm 3 Build-range-tree(P, I, ℓ)

```
1: If $|I| = 1$, build $\mathsf{T}(P)$ sequentially.
2: $\mathsf{T}_0(P) \leftarrow$ Build-primary-tree$(P, I, \ell)$.
3: Copy-points$(P, I, \mathsf{T}_0)$.
4: for all $v \in \mathsf{T}_0$ do
5:     Build-range-tree$(P, I'_v, k + 1)$
6: end for
```

Lemma 5.1. *Algorithm* Build-range-tree$(P, I, 1)$ *takes $O(1)$ rounds of computation, $O(n \log^d n)$ work, and $O(s \log^d n)$ time.*

Proof. Consider running Build-range-tree(P, I, ℓ) for an arbitrary $\ell \in \{1, \ldots, d\}$. Step 1 can be done locally in $O(s_\ell \log^{d-\ell} n)$ time. Step 2 can be done in $O(1)$ rounds using $O(s_\ell \log n)$ time and $O(m \cdot s_\ell \log n)$ work. Copying points can also be done in one round using $O(s_\ell \log n)$ time and $O(m \cdot s_\ell \log n)$ work. There are $O(1)$ levels in the data structure, and each is built in $O(1)$ rounds, so the entire construction procedure takes $O(1)$ rounds of computation. The running time $T(\ell)$ for building ℓth and lower levels of a d-dimensional range tree with $O(s_\ell)$ points initially stored on each machine can be expressed using the recurrence $T(\ell) = O(s_\ell \log n) + T(\ell+1)$ with a base case of $T(d) = O(s_d \log n) = O(s \log^d n)$. This recurrence solves to $T(\ell) = O(s \log^d n)$. Similarly, the total work is $O(n \log^d n)$. $\square$

5.2 Query procedure

Given a query ρ, we run it through the primary structure T_0 to get the $O(\log n)$ nodes whose secondary structures

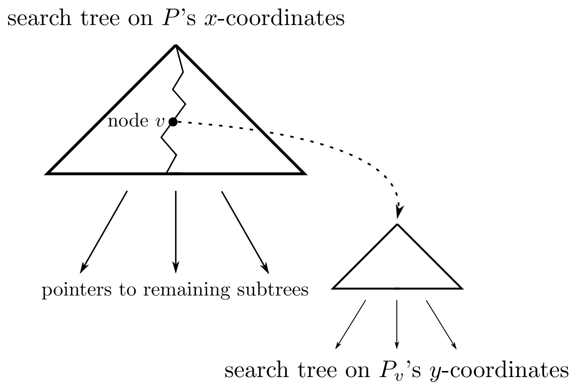
search tree on P's x-coordinates
node v
pointers to remaining subtrees
search tree on P_v's y-coordinates

Figure 5. A distributed range tree in 2 dimensions. A balanced binary search tree over the x-coordinates is stored in a recursive manner. Each node points to a distributed binary search tree over the y-coordinates.

have to be probed further (as described in the beginning of Section 5). This takes $O(\log n)$ time and work and $O(1)$ rounds. Each secondary structure is then probed recursively.

In total, we take $O(\log n)$ time and $O(\log^{d-1})$ work over $O(1)$ rounds finding secondary structures to probe that are stored in the distributed manner described above. The actual time and work bottleneck is the $O(\log^d n + k)$ time and work required to query the bottom-most subtrees stored entirely within individual machines.

We thus have the following theorem.

Theorem 5.2. *Given a set P of n points in $\mathbb{R}^d$, a range tree $\mathsf{T}(P)$ on P of size $O(n \log^{d-1} n)$ can be constructed that can answer an orthogonal range-reporting query in $O(\log^d n + k)$ time and work in the MPC model in $O(1)$ rounds of computation, where k is the number of points reported. The tree can be built in $O(1)$ rounds, $O(n \log^d n)$ work, and $O(s \log^d s)$ time.*

6. CONCLUSIONS

We presented efficient algorithms to build and query *kd*-trees, range trees, and BBD-trees in the MPC model. Our algorithms were based on recursively partitioning a set of points in $\mathbb{R}^d$ by first sampling a small set of points and then computing the partition based on the sample. We believe our framework may be useful in designing efficient data structures in other domains or in the direct analysis of data.

We leave several questions open for future research. The one most closely related to our current work is whether our algorithms for *kd*-trees and BBD-trees can be made deterministic while remaining work-optimal. Our algorithms use poly$(\log_s n) = O(1)$ rounds; it will be interesting to see if this can be improved to $O(\log_s n)$. A lower bound result from [27] says that $\Omega(\log_s n)$ rounds are needed to construct the data structures. Further afield, we ask what other data structures can be constructed efficiently in massively parallel models like MPC; can we efficiently build geometric data structures with more direct GIS applications such as those supporting fast point location queries? Can we efficiently parallelize geometric and topological data analysis methods such as persistent homology? Finally, can our techniques be used outside the geometric domain? In particular, it would be interesting to see if similar hierarchical data structures can be used to efficiently answer graph connectivity and related queries after some preprocessing.

7. REFERENCES

[1] http://pig.apache.org/.

[2] F. N. Afrati, A. D. Sarma, S. Salihoglu, and J. D. Ullman. Upper and lower bounds on the cost of a MapReduce computation. *Proc. VLDB Endow.*, 6(4):277–288, 2013.

[3] P. K. Agarwal, L. Arge, O. Procopiuc, and J. S. Vitter. A framework for index bulk loading and dynamization. In *Proc. 28th Int. Colloq. Automata, Lang. and Program.*, pages 115–127, 2001.

[4] P. K. Agarwal, M. de Berg, J. Gudmundsson, M. Hammar, and H. J. Haverkort. Box-trees and R-trees with near-optimal query time. *Disc. Comput. Geom.*, 28(3):291–312, 2002.

[5] P. K. Agarwal and J. Erickson. Geometric range searching and its relatives. In B. Chazelle, J. Goodman, and R. Pollack, editors, *Advances in Discrete and Computational Geometry*, pages 1–56. American Mathematical Society, 1998.

[6] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Comm. ACM*, 31(9):1116–1127, 1988.

[7] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernandez-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, and S. Whittle. The Dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proc. VLDB Endow.*, 8(12):1792–1803, 2015.

[8] L. Alarabi, A. Eldawy, R. Alghamdi, and M. F. Mokbel. TAREEG: a MapReduce-based system for extracting spatial data from OpenStreetMap. In *Proc. 22nd ACM SIGSPATIAL Int. Conf. Advances Geo. Inf. Systems*, pages 83–92, 2014.

[9] A. Andoni, A. Nikolov, K. Onak, and G. Yaroslavtsev. Parallel algorithms for geometric graph problems. In *Proc. 46th ACM Annu. Symp. Theory Comput.*, pages 574–583, 2014.

[10] L. Arge. External memory data structures. In J. Abello, P. M. Pardalos, and M. G. Resende, editors, *Handbook of Massive Data Sets*, pages 313–357. Springer, 2002.

[11] L. Arge, G. S. Brodal, and R. Fagerberg. Cache-oblivious data structures. In D. Mehta and S. Sahni, editors, *Handbook of Data Structures and Applications*. CRC Press, 2005.

[12] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Y. Wu. An optimal algorithm for approximate nearest neighbor searching in fixed dimensions. *J. ACM*, 45(6):891–923, 1998.

[13] B. Bahmani, A. Goel, and R. Shinde. Efficient distributed locality sensitive hashing. In *Proc. 21st ACM Int. Conf. Info. Know. Manage.*, pages 2174–2178, 2012.

[14] P. Beame, P. Koutris, and D. Suciu. Communication steps for parallel query processing. In *Proc. 32nd Symp. Princ. Database Systems*, pages 273–284, 2013.

[15] T. M. Chan. Optimal partition trees. *Disc. Comput. Geom.*, 47(4):661–690, 2012.

[16] B. Chazelle. *The Discrepancy Method - Randomness and Complexity.* Cambridge University Press, 2001.

[17] M. de Berg, M. van Kreveld, M. Overmars, and O. Cheong. *Computational geometry.* Springer, 3rd edition, 2000.

[18] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Comm. ACM*, 51(1):107–113, 2008.

[19] F. Dehne, A. Fabri, and A. Rau-Chaplin. Scalable parallel geometric algorithms for coarse grained multicomputers. In *Proc. 9th Annu. Symp. Comput. Geom.*, pages 298–307, 1993.

[20] A. Eldawy, L. Alarabi, and M. F. Mokbel. Spatial partitioning techniques in SpatialHadoop. *Proc. VLDB Endow.*, 8(12):1602–1613, 2015.

[21] A. Eldawy, Y. Li, M. F. Mokbel, and R. Janardan. CG_Hadoop: computational geometry in MapReduce. In *Proc. 21st ACM SIGSPATIAL Int. Conf. Advances Geo. Inf. Systems*, pages 284–293, 2013.

[22] A. Eldawy and M. F. Mokbel. SpatialHadoop: A MapReduce framework for spatial data. In *Proc. 31st IEEE Int. Conf. Data Eng.*, pages 1352–1363, 2015.

[23] A. Ene, S. Im, and B. Moseley. Fast clustering using MapReduce. In *Proc. 17th ACM SIGKDD Int. Conf. Know. Discovery and Data Mining*, pages 681–689, 2011.

[24] G. Evangelidis, D. B. Lomet, and B. Salzberg. The hB-Pi-tree: A multi-attribute index supporting concurrency, recovery and node consolidation. *VLDB J.*, 6(1):1–25, 1997.

[25] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Proc. 40th IEEE Annu. Symp. Found. Comp. Sci.*, pages 285–298, 1999.

[26] A. Goel and K. Munagala. Complexity measures for MapReduce, and comparison to parallel computing. *CoRR*, abs/1211.6526, 2012.

[27] M. T. Goodrich. Communication-efficient parallel sorting. *SIAM J. Comput.*, 29(2):416–432, 1999.

[28] M. T. Goodrich. Parallel algorithms in geometry. In *Handbook of Discrete and Computational Geometry, Second Edition.*, pages 953–967. 2004.

[29] M. T. Goodrich, N. Sitchinava, and Q. Zhang. Sorting, searching, and simulation in the MapReduce framework. In *Proc. 22nd Int. Symp. Alg. Comput.*, pages 374–383, 2011.

[30] E. Jahani, M. J. Cafarella, and C. Ré. Automatic optimization for MapReduce programs. *Proc. VLDB Endow.*, 4(6):385–396, 2011.

[31] H. Karloff, S. Suri, and S. Vassilvitskii. A model of computation for MapReduce. In *Proc. 21st ACM-SIAM Annu. Symp. Disc. Alg.*, pages 938–948, 2010.

[32] R. Kumar, B. Moseley, S. Vassilvitskii, and A. Vattani. Fast greedy algorithms in MapReduce and streaming. *ACM Trans. Parallel Comput.*, 2(3):14:1–14:22, 2015.

[33] S. Lattanzi, B. Moseley, S. Suri, and S. Vassilvitskii. Filtering: A method for solving graph problems in MapReduce. In *Proc. 23rd ACM Annu. Symp. Parall. Alg. Arch.*, pages 85–94, 2011.

[34] K.-H. Lee, Y.-J. Lee, H. Choi, Y. D. Chung, and

B. Moon. Parallel data processing with MapReduce: A survey. *SIGMOD Rec.*, 40(4):11–20, 2012.

[35] Y. Li, P. M. Long, and A. Srinivasan. Improved bounds on the sample complexity of learning. *J. Comp. Syst. Sci.*, 62(3):516–527, 2001.

[36] D. B. Lomet and B. Salzberg. The hb-tree: A multiattribute indexing method with good guaranteed performance. *ACM Trans. Database Syst.*, 15(4):625–658, 1990.

[37] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *Proc. 2010 ACM SIGMOD Int. Conf. Manage. Data*, pages 135–146, 2010.

[38] J. Matoušek. Approximations and optimal geometric divide-and-conquer. *J. Comp. Syst. Sci.*, 50(2):203–208, 1995.

[39] M. Mitzenmacher and E. Upfal. *Probability and Computing - Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, 2005.

[40] L. Qin, J. X. Yu, L. Chang, H. Cheng, C. Zhang, and X. Lin. Scalable big graph processing in MapReduce. In *Proc. 2014 ACM SIGMOD Int. Conf. Manage. Data*, pages 827–838, 2014.

[41] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *Proc. 26th IEEE Symp. Mass Storage Syst. Tech.*, pages 1–10, 2010.

[42] L. G. Valiant. A bridging model for parallel computation. *Comm. ACM*, 33(8):103–111, 1990.

[43] T. White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 2012.

[44] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *Proc. 2nd USENIX Conf. Hot Topics Cloud Comput.*, pages 10–10, 2010.