Accelerating Nearest Neighbor Search on Manycore Systems

Lawrence Cayton
Max Planck Institute
Tübingen, Germany
and Microsoft Corp.
Email: work@lcayton.com

***Abstract*—We develop methods for accelerating metric similarity search that are effective on modern hardware. Our algorithms factor into easily parallelizable components, making them simple to deploy and efficient on multicore CPUs and GPUs. Despite the simple structure of our algorithms, their search performance is provably sublinear in the size of the database, with a factor dependent only on its *intrinsic* dimensionality. We demonstrate that our methods provide substantial speedups on a range of datasets and hardware platforms. In particular, we present results on a 48-core server machine, on graphics hardware, and on a multicore desktop.**

***Keywords*-similarity search; metric spaces; parallel algorithms; manycore**

I. Introduction

We study methods to accelerate nearest neighbor search on modern parallel systems. We work in the standard metric nearest neighbor (NN) setting: given a database X, return the closest point to any query q, where closeness is measured with some fixed metric. Though the problem setting is by now well-studied, there are two recent developments that provide a different focus for this work. The first development comes from studies in data analysis and advances in theoretical computer science, where the notion of intrinsic dimensionality has been refined and profitably exploited to manage high-dimensional data. The second development comes from the computer hardware industry: CPUs are virtually all multicore now and GPUs have rapidly evolved into powerful secondary devices for numerical computation, forcing a renewed interest in parallelism.

High-dimensional data challenges nearly all methods for accelerating NN search. Unfortunately, complicated, high-dimensional data is the norm in many domains that rely on NN search, such as computer vision [1], bioinformatics [2], and data analysis in general [3]. Because such data must be dealt with, researchers have explored data properties that might be exploited to accelerate search. A compelling property is the intrinsic dimensionality of a data set; the idea is that often data only *appears* high-dimensional (i.e., each element has many features), but is actually governed by only a few parameters. In recent years, this type of data has been studied extensively and is now believed to be widespread; in machine learning, for example, several methods have been developed to reveal the intrinsic structure of data [4], [5]. In research on accelerated NN retrieval, a renewed focus on intrinsic dimensionality in recent years has yielded methods with strong theoretical guarantees [6], [7] and state-of-the-art empirical performance [8].

The end goal of all of this work on NN is, of course, to make NN search run fast. While properties of data are important for computational efficiency, equally important is the machine hardware on which search actually runs. Hardware properties are especially important now, as the machines in use for everyday data analysis and database operations are fundamentally different than they were even a decade ago. In particular, standard computer chips are multicore with vector (SIMD) units, and GPUs have become popular as secondary devices for data-intensive processing. These modern processors have very impressive computational capabilities, but fully exploiting it requires considerable parallelism.

The shift towards parallelism in hardware necessitates a refocusing in algorithm design and software engineering methods [9]. At the level of algorithm design, advances in algorithms and data structures may not be useful unless they can be effectively parallelized. At the level of software development, developing efficient implementations on parallel systems is challenging, so parallel primitives and design patterns are required to ease the burden on programmers. For the case of NN search, the modern search algorithms discussed were developed for sequential systems, and seem quite difficult to deploy on modern hardware. This is a major practical limitation.

In this work, we develop a new approach to NN search that is both provably sensitive to intrinsic structure *and* that is effective on modern CPUs and GPUs. Our algorithms are based on fundamental ideas from metric similarity search, and are designed carefully for two important goals. First, our design choices allow us to rigorously bound the search complexity of our methods. Equally important, these choices allow us to factorize our algorithms into a basic primitive that is simple to parallelize, making them effective and relatively easy to implement on different systems.

Our methods are of immediate practical use. We demonstrate their performance benefits on a range of modern platforms: a 48-core server machine, a GPU, and a multicore desktop.

II. Related Work

The focus of the present work is unique; as far as we know, it is the only work to simultaneously make use of modern algorithmic developments and modern hardware. Still, the ideas behind our methods are based on a substantial amount of previous work: the algorithmic ideas are based on techniques from metric similarity search; the results on intrinsic dimensionality are related to ideas developed mostly in the theory and data analysis communities; and the motivation behind the project comes from recent trends in computer hardware, especially as related to databases.

The data structure and algorithm in this work are based on two fundamentals of algorithms for metric data: space decomposition and the triangle inequality. These pillars are used in virtually all work on metric NN search; see the surveys of Chávez et al. and Clarkson for detailed overviews [10], [11]. Two of the most empirically effective structures are AESA [12] and metric ball trees [13], [14], both of which have spawned many relatives.

A long-standing problem in similarity search is the difficulty of dealing with high-dimensional data; e.g. [15]–[17]. The basic challenge is that space-decomposition structures that reduce the work for NN retrieval seem to have performance that scales exponentially with the dimensionality of the data, rendering them useless to all but the smallest problems. Recently, there have been two promising directions of work that attempt to deal with this challenge.

The first is called Locality Sensitive Hashing (LSH) [18]. LSH has retrieval performance that is provably sublinear, independent of the underlying dimensionality. This was a major theoretical breakthrough and the data structure has been successfully deployed on some tasks (e.g. [2]). However, LSH has some limitations: it can only provide approximate answers, it is defined only for particular distance measures (not at the generality of metrics), and setting the parameters correctly can be complex [19].

The second line of work, upon which we build, is based on the notion of *intrinsic dimensionality*. The basic idea here is that many data sets only appear high-dimensional, but are actually governed by a small number of parameters. Within data analysis and machine learning, the idea of low-dimensional intrinsic structure has become extremely popular and such structure is believed to be common in many data sets of interests [4], [5].

In the context of NN search, the focus on intrinsic dimensionality has been profitable theoretically and empirically. In particular, the *expansion rate* is a notion of metric space dimensionality that is central to a series of algorithms with strong runtime bounds and excellent performance in practice [6]–[8], [20].

Perhaps the two methods most similar to the present work are the Cover Tree [8] and the GNAT [16]; let us distinguish this research from the present work. The GNAT uses a simple space decomposition based on representatives from the database, much as we do, and also discusses the idea of intrinsic dimensionality. However, the relationship of the GNAT's search performance and the intrinsic dimensionality is only discussed in an informal, heuristic way, whereas we give rigorous runtime guarantees. These rigorous bounds require a search algorithm that is different than that of the GNAT. Additionally, parallelization is not discussed in [16].

The Cover Tree has rigorous guarantees on the query time dependent on the expansion rate and has empirical performance that is state of the art. Even so, our algorithms, data structure, and theory are substantially different; in particular, the Cover Tree is a deep tree and is explored in a conditional way that seems quite difficult to parallelize. We compare our method with the Cover Tree in the experiments.

Next, we touch on the major inspiration for this paper: the use of hardware to accelerate data-intensive processes. Impelled by the sudden ubiquity of multicore CPUs and the development of GPUs for general-purpose computation, this area of research has exploded in the last decade; let us provide a couple of inspiring examples. A relatively early work develops methods to off-load expensive database operations onto the GPU [21]. A very recent piece of work tunes basic tree search algorithms (such as for index lookup) to be effective on modern multicore CPUs and GPUs [22].

Finally, there have been a few papers related to NN and manycore. One paper suggests simply running brute force search on a GPU to accelerate NN search [23]; this simple approach provides a surprising amount of acceleration over computation on sequential CPUs [24]. A very recent paper studies accelerated NN specifically for a multicore CPU platform, but this method offers no runtime guarantees [25]; in particular, its running time is not sublinear in the database size.

III. Challenges of manycore

In this section, we describe the challenges of algorithm design for manycore systems.

Manycore simply means a system with a high-core count, such as a GPU or 48-core system. From the perspective of algorithm design, the characteristic feature of a manycore system is not so much a certain minimum core count as it is the requirement that algorithms decompose into hundreds or thousands of pieces—i.e., algorithms must be *fundamentally parallel*. As a concrete example, consider the difference between designing an algorithm for a dual-core chip and for a GPU. Effectively utilizing a dual-core chip might be possible by re-engineering a sequential algorithm; however, fully utilizing a modern GPU requires an algorithm that decomposes into *thousands* of threads, which is a vastly different regime than the sequential one.

The main challenge of manycore algorithm is simply finding enough parallelism is an a problem. Beyond this basic challenge, there are a variety of performance pitfalls

to be avoided. First, there can be a significant penalty for synchronization (and more generally, communication) among threads. In particular, computational resources can be wasted as some threads sit idle waiting for others, and the cost of exchanging messages between threads can be non-trivial. Second, load-balancing can be a major challenge; not only does a problem have to be decomposed into thousands of pieces, those pieces need to be roughly the same size. Third, conditional computation can be inefficient. This is especially true on GPUs, on account of their wide vector units and limited branch prediction ability. Finally, access to memory can be a bottleneck. Though this bottleneck exists on single-core systems, it is exacerbated by a multiplicity of threads accessing the memory system, and, in the case of GPUs, primitive caching. See e.g. [26] for more details on these challenges.

As an illustrating example of these principles, and to further motivate the present work, let us consider porting a standard sequential NN data structure over to a manycore system. We use metric trees as an example [13]. This data structure is popular, effective, and closely related to the state-of-the-art Cover Tree. Moreover, its basic structure is quite similar to many NN schemes.

The data structure is a deep tree, with nodes that are associated with regions of space that contain a subset of the database. Querying this data structure requires a depth-first exploration of the tree. This exploration involves an interleaved series of distance computations, bound computations, and distance comparisons. These computations and comparisons guide the exploration process, resulting in computational structure that is conditional. Put differently, the specific distance calculations made at one step depend on the comparisons from the previous step.

To implement the search algorithm on a manycore system, we must first decide how to decompose the search algorithm into subtasks. The two natural parallelization schemes are distributing different nodes of the tree across threads and distributing queries across threads.[1] Unfortunately, both have problems. The node-parallel scheme can lead to significant thread imbalance since many nodes will be not be accessed during a search. This scheme also requires significant communication between threads. The other scheme, parallelizing over queries, is only appropriate in the case when many queries are run at once. Even when this scheme is appropriate, it can lead to a work imbalance depending on the location of queries and the distribution of the database. Another drawback of this scheme is that it makes inefficient use of the memory bus: if multiple queries all need to access a particular portion of the database, it is much more efficient to do one load from memory rather than many loads. Hence both of the natural parallelization schemes are possible, but come with major drawbacks.

An additional complication is the conditional nature of the computation, which seems endemic to sequential metric search algorithms. This conditional structure—in which the database elements that are retrieved and examined depend on the results of inequality evaluations—requires a complicated series of memory accesses which are difficult to hide the latency of; as a result it is difficult to keep the (many) processing elements busy.

In addition to performance considerations, there is the major issue of program complexity: developing a manycore implementation of a deep tree structure with the requisite synchronization is highly non-trivial. This difficulty is compounded because the goal of the implementation is to accelerate NN search, meaning that many performance considerations must be examined carefully.

Though some of these difficulties could perhaps be overcome, the basic structure of the metric tree search algorithm is not a natural fit for a manycore architecture: it does not seem to decompose easily into many independent pieces. In sharp contrast, brute force search parallelizes trivially, as we describe in the next section.

IV. The Brute Force Primitive

In the last section, we explored the possibility of fitting a metric tree onto a manycore system and encountered many difficulties. In this section, we describe the brute force search approach to NN search, which is simple to parallelize. But though it parallelizes easily, brute force search requires much more total work than an intelligent tree-based search. Hence we introduce brute force here not as a complete solution to NN search, but as a useful *primitive* to build intelligent search algorithms on top of. By using brute force as a primitive, our search algorithms—developed in the coming sections—are able to reduce the work for NN search *and* parallelize effectively.

Given a set of queries Q and a database X with n elements, finding the NNs for all $q \in Q$ can be achieved by a series of linear scans. For each query q, the distance between q and each $x \in X$ is computed, the distances are compared, and the database point that is closest is returned. We denote this subroutine as $\mathbf{BF}(Q, X)$. If L is some set of IDs (i.e. $L \subset \{1, \ldots, n\}$), then brute force search between Q and this subset of the database is denoted $\mathbf{BF}(Q, X[L])$.

The work required for $\mathbf{BF}(Q, X)$ is $O(n)$ per query; we later prove that our accelerated search algorithms have work only roughly $O(\sqrt{n})$, which is performed in two brute force calls $\mathbf{BF}(Q, X[L_1])$ and $\mathbf{BF}(Q, X[L_2])$, where lists L_1 and L_2 are determined by our algorithm.

Parallelizing $\mathbf{BF}(Q, X)$ is relatively straightforward; we provide a brief overview here, then go into detail in section VIII. We break the procedure down into two steps: a distance computation step, and a comparison step. In the distance computation step, all pairs of distances are computed. This

[1] Note that simply distributing the database elements across threads is a special case of distributing the nodes across threads.

has virtually the same structure as matrix-matrix multiply, and hence block decomposition approaches are effective. In the case where there is only a single query presented at a time (e.g. a stream of queries), the distance computation step of $\mathbf{BF}(q, X)$ has the structure of a matrix-vector multiplication. In both cases, the parallelization is extremely well-studied.[2]

The second step is the comparison: for each query, the distances must be compared, and the nearest database element returned. This is simple to do in parallel systems as well; the problem can simply be plugged into the standard parallel-reduce paradigm where comparisons are made according to an inverted binary tree.

The simple structure of **BF** makes it relatively simple to manage the challenges of manycore implementation described the in the last section. In contrast, accelerated NN routines seem to depend on a complex computational structure. A major contribution of the present work is in demonstrating that a much simpler structure can be substituted without significant loss.

With the brute force primitive in place, we proceed to discuss our data structure and algorithms, all of which will be built from this primitive.

V. Data structure

We discuss the data structure underlying our methods in this section, which we call the *Random Ball Cover* (RBC). This is a simple, single-level cover of the underlying metric space. The basic idea is to use a random subset of the database as representatives for the rest of the DB.

The database is denoted $X = \{x_1, \ldots, x_n\}$ and the metric in use is denoted $\rho(\cdot, \cdot)$. The data structure consists of a random subset of the database, which will be of size about $O(\sqrt{n})$; we make this precise later. This set of random representatives will be denoted R. It is built by choosing each element of the database independently at random with probability n_r/n, where the exact value of n_r is discussed in the theory section. In expectation, there will be n_r representatives chosen—one can think of the symbol n_r as shorthand for **n**umber of **r**epresentatives.

Each representative *owns* some subset of the database. The list of points that a representative r owns is denoted L_r, which we will at times refer to as an *ownership list*. In principle, L_r could contain any subset of the database, but for our algorithms L_r will contain points near to r.

Finally, for each representative r, a radius is stored. This radius is defined as the distance from r to the furthest point that it owns:

$$\psi_r = \max_{x \in L_r} \rho(x, r).$$

To summarize, the RBC data structure is simply a set of representatives R, along with an ownership list L_r and radius ψ_r for each $r \in R$.

We introduce two search algorithms which use the RBC in the next section, called the *exact* and *one-shot* algorithms. For the exact search algorithm, the ownership list L_r contains all $x \in X$ for which r is the nearest neighbor among R. For the one-shot algorithm, L_r contains a suitably sized set of r's NNs among X.

The focus of this paper is on algorithms that are simple to parallelize; this is achieved by folding the computational work into brute force calls. The building routines for the RBC demonstrate this principle concisely. The building routine for the exact search algorithm must find the NN for each $x \in X$ among the representatives R. Thus this routine is simply a call to $\mathbf{BF}(X, R)$. Similarly, the building routine for the one-shot algorithm must find the NNs for each representative R among the database elements X; thus this procedure is simply a call to $\mathbf{BF}(R, X)$. Both parallelize easily.

With the data structure and notation in place, we proceed to describe the search algorithms.

VI. Search Algorithms

In this section, we describe two search algorithms which use the RBC data structure. The theoretical analysis of these algorithms appears in the next section. Both of these algorithms build up from the brute force search primitive.

We first describe the *one-shot* algorithm, then the *exact* search algorithm; both are quite simple. Both algorithms rely on a randomized data structure, and so are probabilistic. In the one-shot algorithm, the solution itself is randomized: the data structure returns a correct result with high probability. In the exact algorithm, the solution is guaranteed to be correct; only the running time is probabilistic.[3] Hence when an exact answer is required, the exact algorithm is appropriate; if a small amount of error can be tolerated, the one-shot algorithm is simpler and often faster, as we show in the experiments.

For simplicity, we focus on the problem of 1-NN search, but mention the minor differences for k-NN where appropriate.

A. One-shot search

First, let us review the RBC data structure. The representatives R are chosen at independently at random, and each ownership list L_r contains the s closest database elements to r. Depending on the setting of s, points will typically belong to more than one representative. We discuss these parameters further in the theory section.

On a query q, the algorithm proceeds as follows. It first computes the NN to q among the representatives using a

[2]See, for example, the workshop series *Parallel Matrix Algorithms and Applications*.

[3]This algorithm can be easily modified so that it only guarantees an *approximate* nearest neighbor, which reduces search time.

simple linear scan (brute force search), call it r. It then scans the ownership list L_r, computing the distance from q to each listed database point. The nearest one is returned as the nearest neighbor. In the case of k-NN search, the nearest k are returned from L_r.

We restate the algorithm in terms of the brute force primitive. The algorithm first calls $\mathbf{BF}(q, R)$, which returns a representative r. It then calls $\mathbf{BF}(q, X[L_r])$ and returns the answer.

The one-shot algorithm is almost absurdly simple. Yet, as we show rigorously in the theory section, it provides a massive speedup; in particular, it reduces the work from $O(n)$ (required for a full brute force search) to roughly $O(\sqrt{n})$. Moreover, it is very fast empirically, as we show in the experiments section.

B. Exact search

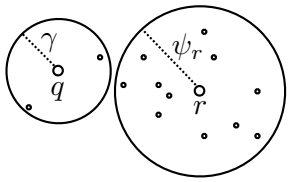
Whereas the one-shot algorithm does not use the triangle inequality (though the analysis requires it), the exact search algorithm explicitly prunes portions of the space using it; it is thus reminiscent of classic branch-and-bound techniques.

Like the one-shot algorithm, the exact search algorithm uses the RBC data structure. However, the ownership list is slightly different: each representative r owns the database elements for which it is the nearest representative. The data structure is constructed algorithmically as follows. The build algorithm calls $\mathbf{BF}(X, R)$, then adds each $x \in X$ to the ownership list of its returned NN in R. The radii ψ_r are set to $\max_{x \in L_r} \rho(x, r)$ as before.

We now detail the search algorithm. First, the closest point to q among all $r \in R$ is computed; call it r_q, and let $\gamma = \rho(q, r_q)$. This distance is an upper bound on the distance to q's NN (since $r_q \in X$), and so the algorithm can use it to discard some of the database from consideration. Recall that the radius of each representative r is stored as ψ_r—*i.e.* each $x \in L_r$ satisfies $\rho(x, r) \leq \psi_r$. Because γ is an upper bound on the distance to the NN, any point belonging to an r satisfying

$$\rho(q, r) > \gamma + \psi_r \tag{1}$$

can be discarded. The following sketch illustrates (1); since there is a point within distance γ of q, no points within distance ψ_r of r can be q's NN.



Hence the only points that need to be considered belong to a list L_r, where r violates (1).

The algorithm simultaneously checks a second bound in hopes of pruning out more of the representatives. As Lemma 1 (see section VII) shows, if r_q is q's NN among the representatives, any representative that owns q's NN must satisfy

$$\rho(q, r) \leq 3 \cdot \rho(q, r_q). \tag{2}$$

Hence any representative violating this inequality is pruned out by the search algorithm.

In the case of k-NN search, inequalities (1) and (2) are evaluated with respect to the kth-nearest representative instead of the nearest.

Once the pruning stage is complete, the search algorithm computes the distance to all points belonging to one of the remaining representatives, and returns the nearest.

We restate the algorithm in terms of our primitive. It first computes $\mathbf{BF}(q, R)$, much like the one-shot algorithm. In this case, however, the distances must be retained so that inequalities (1) and (2) can be checked. Once the inequalities are checked, some representatives will still remain, with lists $L_1, \ldots, L_t$. Next the search algorithm performs another brute force search, namely $\mathbf{BF}(q, X[L_1 \cup L_2 \cup \ldots \cup L_t])$, and returns the answer.

We will see that the size of each of the brute force calls is about $O(\sqrt{n})$, providing major time savings over a full brute force search.

We emphasize that the computation structure of both search algorithms is quite different from tree-based search, in which bounds are incrementally refined, and distance computations are interleaved with bound evaluations. In both cases, this structure makes the algorithm extremely simple to implement and effective to parallelize. It is rather surprising that such simple algorithms can effectively reduce the work required for NN search, but that is exactly what we show both theoretically and empirically in the following sections.

VII. Theory

In this section, we present rigorous runtime guarantees for our search algorithms. Though the algorithms are intuitive and simple, bounding the runtime is non-trivial. Indeed, many methods for accelerated NN search lack performance guarantees.

As we described in the background section, all methods that reduce the work for NN search have some dependence on the dimensionality of the database. Much of the success of metric indexing methods is commonly ascribed to their dependence only on the intrinsic dimensionality of data. In this section, we prove that the RBC search algorithms scale with the *expansion rate*, which is a useful notion of intrinsic dimensionality.

Definition 1. *Let $B(x, r)$ denote the closed ball of radius r around x—i.e. the set $\{y : \rho(x, y) \leq r\}$—and $|B(x, r)|$ denote its cardinality. A finite metric space M has expansion rate c if for all $r > 0$ and $x \in M$*

$$|B(x, 2r)| \leq c \cdot |B(x, r)|.$$

The expansion rate is fairly well-establish quantity; see [27] for mathematical background on it, and [7], [8], [11], [20] for details specifically related to NN search. We emphasize that the ideas in this section build off of ideas in the aforementioned papers.

To gain some intuition for this measure, consider a grid of points in $\mathbb{R}^d$ under the ℓ_1 metric

$$\rho(x, y) = \sum_{i=1}^{d} |x_i - y_i|.$$

The expansion rate c in this case is 2^d, hence $\log c$ corresponds to the dimensionality of the data [6].

Before proceeding, we note two important properties of the expansion rate. First, the expansion rate is defined only in terms of the metric, not in terms of the representation of data; in this sense, the rate captures the intrinsic structure of the metric space. Second, the expansion rate is defined for arbitrary metric spaces, so is a meaningful quantity for many problem settings outside of the standard Euclidean one.

Throughout we assume that $X \cup Q$ has expansion rate c, and we prove bounds dependent on this expansion rate and n. We note that work on lower bounds in related settings suggests that such a dependence on c is necessary [6].

The exact search algorithm and analysis rely on the following lemma, which is known [11]; we push the proof to the Appendix to streamline the exposition.

Lemma 1. *Let $R \subset X$ and assign each $x \in X$ to its nearest $r \in R$. Let $\gamma = \min_{r \in R} \rho(q, r)$ (i.e., γ is the distance to q's NN in R). Then, if some $r^* \in R$ owns the nearest neighbor to q in X, it must satisfy*

$$\rho(q, r^*) \leq 3\gamma.$$

We now analyze the search algorithms. Throughout, we assume R is a random subset of X, built by picking each element of X independently at random with probability $p = n_r/n$. Recall that the ownership list of $r \in R$ is denoted L_r and the radius of this list (i.e. $\max_{x \in L_r} \rho(x, r)$) is denoted ψ_r. Finally, n_r is the expected number of representatives and n is the cardinality of the database.

A. Exact Search

First, let us consider the exact search algorithm. The search algorithm performs two steps: in the first step, the algorithm performs brute force search from the queries to R; and in the second step, it performs a brute force search from the queries to the database elements belonging to ownership lists of un-pruned representatives. The first step clearly has work complexity $O(n_r)$ per query, where n_r is the expected number of representatives; the following analysis bounds the complexity of the second step. In particular, we show that the expected number of distance evaluations is $c^3 n/n_r$. Hence if $n_r \approx c^{3/2}\sqrt{n}$, the expected number of distance evaluations in the second step is $O(c^{3/2}\sqrt{n})$, the same as the first step. We call $n_r = O(c^{3/2}\sqrt{n})$ the *standard parameter setting*.

In the first step of the algorithm, the nearest point to q in R is found; call this point r_q. How many database points are likely to be closer to q than r_q?

Claim 2. *Let γ be the distance from q to its nearest neighbor in R, r_q. The expected number of points in $B(q, \gamma)$ is n/n_r, which is $O(\sqrt{n}/c^{3/2})$ for the standard parameter setting.*

Proof: Form a list $L = [x_1, x_2, \ldots, x_n]$ by ordering the database points $x \in X$ by their distance to q. Some subset of X also belongs to R; let x_t be the first representative appearing in L (i.e. the closest representative to q). Then the expected number of points in $B(q, \gamma)$ is equal to $t - 1$.

A slightly different way to view the process is that the L is fixed, then x_1 is chosen as a representative with probability n_r/n, then x_2 is chosen as a representative with probability n_r/n, and so on. We wish to know the expected time before the first x_i is chosen as a representative. That is given precisely by the geometric distribution: the number of Bernoulli trials needed to get one success. The mean of a Bernoulli distribution with parameter $p = n_r/n$ is $1/p = n/n_r$. Hence $\mathbb{E}|B(q, \gamma)| = n/n_r$, which is $O(\sqrt{n}/c^{3/2})$ for the standard parameter setting. ∎

We note that a high-probability version of the above claim follows easily from standard concentration bounds. We also point out that the expectation is over randomness in the algorithm; we are not making any distributional assumptions on the database.

After computing the nearest neighbor to the query q among the representatives, the exact search algorithm uses γ ($\equiv \rho(q, r_q)$) as an upper bound on the distance to q's NN to prune out some representative sets. In particular, any representative r with radius ψ_r satisfying

$$\rho(q, r) > \gamma + \psi_r \tag{3}$$

cannot possibly own q's NN. Additionally, the algorithm can safely prune any representative r such that

$$\rho(q, r) > 3\gamma. \tag{4}$$

This property follows from Lemma 1. In the following we only work with inequality (4). The simultaneous use of both inequalities improved the empirical performance, but it is not necessary for the following theory.

The second step of the algorithm examines only points with a representative violating (4). We now show that we can actually restrict this set further, and subsequently bound the cardinality of all points considered.

Claim 3. *Suppose that x is the nearest neighbor of q, and that r_x is the representative owning x. Then $\rho(x, r_x) \leq 4\gamma$.*

Proof: From the triangle inequality, $\rho(x, r_x) \leq \rho(x, q) + \rho(q, r_x)$. But since x is q's NN, and since $r_x \in X$,

$\rho(x,q) \leq \gamma$. The other term is bounded by 3γ on account of (4). Hence the nearest neighbor x must lie within 4γ of its representative. ■

We have shown that the search algorithm only needs to compute distances from q to points x that are within distance 4γ of their representative. Hence, if the lists L_r are stored in sorted order according to the distance to r, the search algorithm can simply ignore all points x more than distance 4γ from their representative.[4]

Theorem 4. *The expected number of points examined in the second stage of the exact search algorithm is $c^3 n/n_r$, which is $O(c^{3/2}\sqrt{n})$ for the standard parameter setting.*

Proof: By the previous claim, any point x that the algorithm examines satisfies $\rho(x, r_x) \leq 4\gamma$, where r_x is the representative of x. From inequality (4), we also know that $\rho(q, r_x) \leq 3\gamma$. Putting these two facts together with the triangle inequality, we have that any point x examined by the algorithm satisfies $\rho(q,x) \leq 7\gamma$. In other words, all points examined lie in $B(q, 7\gamma)$, which we now bound the cardinality of.

Applying the expansion rate condition, we have that

$$|B(q,7\gamma)| \leq |B(q,8\gamma)| \leq c^3 |B(q,\gamma)|. \tag{5}$$

From Claim 2, $\mathbb{E}|B(q,\gamma)| = n/n_r$, which we can plug into (5). The resulting bound is $\mathbb{E}|B(q,7\gamma)| \leq c^3 n/n_r$, which is $O(c^{3/2}\sqrt{n})$ for the standard parameter setting. ■

As each x only appears on one list L_r, each x is only compared to q once, implying that (5) bounds not only the *cardinality* of the examined set points, but also the number of computations (in the second step). Since the time for the first brute force step was also $O(c^{3/2}\sqrt{n})$, we have shown that the expected runtime of the exact search algorithm is $O(c^{3/2}\sqrt{n})$.

Finally, we add that Theorem 4 can be extended to the k-NN case, with a bound of $O(c^{3/2}\sqrt{kn})$. We omit the extra details because of space restrictions.

B. One-Shot Search

The one-shot search algorithm is considerably simpler than the exact search algorithm, and also uses a slightly different data structure configuration. In particular, the algorithm searches only a single representative list per query, and the ownership lists of the RBC will usually overlap. Unlike the exact search algorithm, the one-shot algorithm only returns the NN with high probability.

With these differences in mind, the resulting time complexity bound is actually quite similar to the bound in Theorem 4. Recall that there are two parameters governing its run time: n_r, the number of representatives; and s, the number of points assigned to each representative. Hence the time complexity of the one-shot search algorithm is $O(n_r + s)$. Thus we need to prove that a certain setting of these parameters guarantees a high probability of success.

We begin with a simple known lemma and then prove the main theorem for the general k-NN case. This analysis requires some different ideas than the analysis for the exact algorithm and as a result is more technically involved.

Lemma 5. *Suppose that $\rho(q,r) = \gamma$. Then*

$$B(q,\gamma) \subset B(r, 2\gamma) \subset B(q, 4\gamma).$$

Proof: Let $x \in B(q,\gamma)$. Then $\rho(x,r) \leq \rho(x,q) + \rho(q,r) \leq 2\gamma$, proving the first inequality. For the second, let $x \in B(r, 2\gamma)$. Then $\rho(x,q) \leq \rho(x,r) + \rho(r,q) \leq 2\gamma + \gamma \leq 4\gamma$. ■

Theorem 6. *Set the parameters*

$$n_r \;=\; s \;=\; O\left(c\sqrt{kn}\cdot\sqrt{\ln\frac{1}{\delta}}\right).$$

Then the one-shot algorithm returns the k-NNs with probability at least $1-\delta$.

Proof: Recall that the one-shot algorithm computes the nearest representative to the query q, and returns the k-closest points owned by this representative. Let r_1 be the nearest representative to q, and let $\gamma_1 \equiv \rho(q, r_1)$. Similarly, let r_k be the k-closest representative and $\gamma_k \equiv \rho(q, r_k)$.

First we show that the algorithm succeeds if $\gamma_k < \psi_{r_1}/2$, where ψ_{r_1} is the radius of the ball associated with representative r_1. Since the representatives are themselves database points, γ_k is an upper bound on the distance to q's kth NN. Hence the k-NNs are contained in $B(q,\gamma_k)$; moreover if r_1 owns this entire set, the one-shot algorithm is guaranteed to return the correct k points. So we show that $B(q,\gamma_k) \subset B(r_1,\gamma_1)$, given the assumption that $\gamma_k < \psi_{r_1}/2$. Let $x \in B(q,\gamma_k)$. Then,

$$\begin{aligned}\rho(x,r_1) &\leq \rho(x,q) + \rho(q,r_1)\\ &\leq \gamma_k + \gamma_1\\ &\leq \gamma_k + \gamma_k\\ &\leq \psi_{r_1},\end{aligned}$$

as desired. Thus the algorithm fails only if $\gamma_k > \psi_{r_1}/2$.

We bound the probability that $\gamma_k > \psi_{r_1}/2$. By Lemma 5 $B(r_1,\psi_{r_1}) \subset B(r, 2\gamma_1)$ and $B(r_1, 2\gamma_1) \subset B(q, 4\gamma_1)$. Then by the expansion condition,

$$|B(r_1,\psi_{r_1})| \leq |B(q,4\gamma_1)| \leq c^2 |B(q,\gamma_1)|.$$

Moreover, $B(q,\gamma_1) \subset B(q,\gamma_k)$, since $\gamma_k > \gamma_1$. Rearranging, we have that $|B(q,\gamma_k)| > 1/c^2 |B(r_1,\psi_{r_1})|$. Of course, $B(r_1,\psi_{r_1})$ contains exactly the points owned by r_1, so $|B(r_1,\psi_{r_1})| = s$. Thus there are at least s/c^2 points closer

[4] We found that cutting off the search after all points within distance 4γ were explored had little effect empirically; hence we opted for the simpler exposition in the algorithms section.

to q than r_k. What is the probability that at most k were chosen as representatives?

The probability is given by the CDF of the binomial distribution $B(t,p)$ with parameter $t = s/c^2$, $p = n_r/n$. The lower tail of the CDF of $B(t,p)$ can be bounded by

$$\exp\left[-\frac{1}{2p}\frac{(tp-k)^2}{t}\right]$$

using Chernoff's inequality. Plugging in the parameters:

$$\exp\left[-\frac{1}{2}\frac{nc^2}{n_r s}\left(\frac{n_r s}{nc^2}-k\right)^2\right].$$

Set $n_r = s = \sqrt{\eta k c^2 n}$. Then $nc^2/(n_r s) = nc^2/(\eta k c^2 n) = 1/(\eta k)$. This simplifies the previous expression to

$$\begin{aligned}\exp\left[-\frac{1}{2}\frac{1}{\eta k}(\eta k-k)^2\right] &= \exp\left[-\frac{1}{2}\frac{1}{\eta k}(\eta-1)^2k^2\right]\\ &\le \exp\left[-\frac{1}{2}\frac{(\eta-1)^2}{\eta}\right]\\ &\le \exp\left[-\frac{1}{8}\eta\right] \quad \text{for } \eta > 2.\end{aligned}$$

Setting δ equal to the last line and re-arranging yields the theorem. ∎

VIII. Parallel Implementation

In this section, we detail the implementation of the RBC on a manycore system. We have previously described the RBC algorithms abstractly; this section provides a systems-level view of the methods. It serves to emphasize the natural parallelism of our methods and provides details behind the experiments in the following section.

A. Hardware overview

The algorithms of this paper are designed for a manycore system, by which we simply mean a shared-memory system with a high core-count. At present, the two dominant forms of manycore systems are GPUs and CPUs. Though these two types of processing units are at present quite different, our methods are simple and general enough to use either effectively. Importantly, we do not make use of features present in one but not the other, such as complex branching abilities, or specialized graphics functionality.

A CPU consists of multiple cores, each of which is essentially an independent processor.[5] Each core has a single instruction, multiple data (SIMD) unit, also called a vector unit. The vector unit is capable of executing an instruction on multiple memory locations simultaneously, and hence is useful for operations on vectors. The vector unit on most contemporary CPUs has four to eight single-precision slots. At the software level, a thread is generally associated with a core, not with individual slots of the vector unit. Each core has a small amount of on-chip memory, called a cache.

[5]We note that for the purposes of this paper, the cores need not be on the same chip.

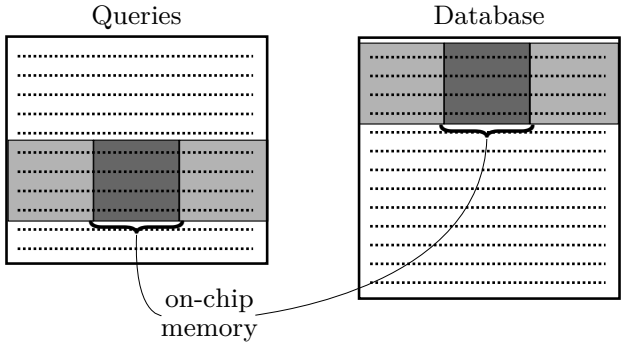


Figure 1. Computing distances. Each core is assigned a contiguous segment of queries and database elements to compute distances between (highlighted rows). During a stage of the distance computation, a portion of these elements are loaded into the on-chip memory and partial distances are computed (boxed area within the highlighted rows).

A GPU has roughly similar components. It consists of multiple cores (also called multiprocessors), each of which is essentially a wide vector unit with a pool of on-chip memory. This on-chip memory is further subdivided into registers, shared memory, and a cache. The number of cores and the width of the vector unit varies across GPU models. The GPU we experiment with (NVIDIA Tesla c2050) has 14 cores, each of which has 32 single-precision slots. Other high-end GPUs have roughly similar configurations. Unlike on the CPU, threads are generally associated with a slot of the vector unit (at least in the CUDA programming model).

B. Implementing BF

In this section, we describe our implementation of the brute force primitive, in which virtually all of the work of our search algorithms takes place.

The most basic implementation design challenge is how to decompose the problem into enough subcomponents to keep all of the processing elements busy. Perhaps the most straightforward way to parallelize **BF** is a query-level or database-level parallelism; i.e., each query can be processed in parallel, or each database element can be processed in parallel. Our strategy is more subtle and takes into account the two levels of parallelism in the hardware, namely the core level and the vector element level. At the core level, it parallelizes over subsets of the queries and database elements; at the vector unit level, it parallelizes over the features of each element.

This implementation approach is a type of *block* strategy [28]. This strategy exposes the necessary parallelism to keep the processing elements busy. It has an additional major benefit over a simpler decomposition: it minimizes memory accesses by making intelligent use of the on-chip memory. Memory accesses can be a major bottleneck on manycore

systems, hence reducing global memory traffic is crucial for performance.

We proceed to describe the implementation in more detail. Since the programming models for CPUs and GPUs differ, we describe the implementation at the hardware level. Let X be the database elements that must be examined, and Q a set of queries. Note that whether X is the entire database or just some subset is unimportant here.

There are $|X| \cdot |Q|$ distances to compute; the naive approach is to distribute these computations across the cores uniformly. Instead, each core is assigned a contiguous sub-block of X and of Q, and must compute all pairwise distances within these sub-blocks. Suppose that a particular core is assigned sub-blocks X_i and Q_j. Naively, each $x \in X_i$ must be loaded from memory $|Q_j|$ times, and each $q \in Q_j$ must be loaded $|X_i|$ times. Instead, the implementation loads all elements of X_i and all elements of Q_j into the on-chip memory, and then performs all necessary computations through the vector units. In general, X_i and Q_j will be too large to fit into the on-chip memory. Let k be the largest integer such that $k|X_i| + k|Q_j|$ values fit into the on-chip memory. Then the implementation loads the first k features of all elements of X_i and Q_j into the on-chip memory, computes partial distances, then loads the next k features, updates the partial distances, and so on, until reaching the end of all elements. See Figure 1 for a diagram of this process.

In the case where $|Q| = 1$—i.e., a single query is processed at a time—we are no longer able to re-use loads from memory, but the parallelism is still designed in the same way. Thus the procedure is relatively more efficient when there are many queries to process at once, in much the same way that matrix-matrix multiplication is generally more efficient than a series of matrix-vector multiplications.

After the distances between each element of Q and X are computed, a parallel reduce is used for each $q \in Q$ to determine the minimum [28]. To avoid storing all the distances, these parallel reduces can be performed in an interleaved fashion with the distance computations.

IX. Experiments

We perform several sets of experiments to demonstrate the effectiveness of our methods. The first, and probably most important, set of experiments demonstrate the performance benefit of the RBC on a 48-core machine, as compared to a brute force implementation (§IX-B). These experiments show that the RBC significantly reduces the work required for NN search and that it parallelizes effectively.

The second set of experiments demonstrates that the RBC is effective on graphics hardware (§IX-C). It is challenging to deploy data structures on such hardware, but very important because of the ubiquity of GPUs in scientific and database systems.

Name	Num pts	Dim
Bio	200k	74
Covertype	500k	54
Physics	100k	78
Robot	2M	21
TinyIm	10M	4-32

Table I
Overview of data sets.

The final set of experiments compares the performance of the RBC to the Cover Tree on a desktop machine (§IX-D). These experiments demonstrate that the exact search algorithm is competitive with the state-of-the-art even on a machine with a low degree of parallelism.

A. Experimental setup

Our CPU implementation of the RBC was written in C and parallelized with OpenMP. Our GPU implementation was written in C and CUDA. Both are available for download from the author's website.

In very low-dimensional spaces, basic data structures like kd-trees are extremely effective, hence the challenging cases are data that is somewhat higher dimensional. We experiment on several different data sets over a range of dimensionalities. Table I provides a quick overview; we describe a few details next.

The Bio, Covertype, and Physics data sets are standard benchmark data sets used in machine learning and are available from the UCI repository [29]. They have been used to benchmark NN search previously [8], [30]. The Robot data was generated from a Barret WAM robotic arm; see [31]. The TinyIm data set is taken from the Tiny Images database, which is used for computer vision research [1]. We took the image descriptors and reduced the dimensionality using the method of random projections.[6] We experimented with dimensionalities of 4, 8, 16, 32. For all experiments, we measured distance with the ℓ_2-norm (i.e. standard Euclidean distance), which is appropriate for this data.

We perform the first set of CPU experiments on a 48-core/4-chip AMD server machine. Each chip is a 12-core AMD 6176SE processor, and is divided into two 6-core segments. This machine has a high core count, so it is a good system to test the scalability of our algorithms. The comparison to the Cover Tree is performed on a quad-core Intel Core i5 machine, which is a reasonable representative for a mid-range desktop. Our GPU experiments are run on a NVIDIA Tesla c2050 graphics card, which is designed for general purpose computation.[7] We previously described the details of our GPU code in a workshop paper [34].

[6]This dimensionality reduction technique approximately preserves the lengths of vectors, and hence is a useful preprocessor for NN search; see e.g. [32]. The technique is formally justified by the Johnson-Lindenstrauss Lemma [33].

[7]We note that, in limited experiments, our methods were effective on (much cheaper) consumer-grade GPUs as well.

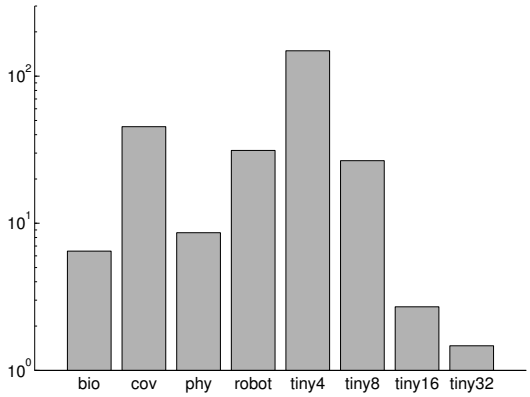


Figure 2. Speedup of exact search over brute force.

B. 48-core experiments

We compare the performance of our methods to brute force search on the 48-core machine. As far as we know, there is no readily available accelerated NN method for such a machine. Furthermore, brute force is already quite fast because of the raw computational power.

First, we look at the performance of the exact search algorithm, which is guaranteed to return the exact NN. Figure 2 shows the results. We are getting a strong speedup of up to two orders of magnitude, despite the challenging hardware setting and the reasonably high data dimensionality.

Next we consider the one-shot search algorithm. As developed in the theory section, we set n_r (the number of representatives) and s (the number of points owned per representative) equal to one another. The parameter allows one to trade-off between the quality of the solution and time required; we scan over this parameter to show the trade-off. This algorithm is not guaranteed to return a nearest neighbor, so we must evaluate the quality of the solution. A standard error measure is the *rank* of the returned point: i.e., the number of database points closer to the query than the returned point [30]. A rank of 0 denotes the exact NN, and rank of 1 denotes the second NN, and so on.

Figure 3 shows the results. The speedup achieved in these experiments is quite significant; even with a rank around 10^{-1} (very close to exact search), the *worst* speedup is an order of magnitude. In applications where a small amount of error is tolerable, the one-shot search algorithm can provide a massive speedup, even better than our exact search algorithm in some cases. In many applications, e.g. in data mining, there is uncertainty associated with the data, so a small amount of error in the NN retrieval is not important.

C. GPU experiments

GPUs have impressive brute force search performance, but the GPU architecture makes efficient data structure design quite difficult. In particular, GPUs are vector-style processors with limited branching ability; hence conditional computation can lead to serious under-utilization.

Data	Speedup
Bio	38.1
Covertype	94.6
Physics	19.0
Robot	53.2
TinyIm4	188.4

Table II
GPU RESULTS: SPEEDUP OF THE ONE-SHOT ALGORITHM OVER BRUTE FORCE SEARCH (BOTH ON THE GPU).

We show that our RBC one-shot algorithm provides a substantial speedup over the already-fast brute force search on a GPU. Table II shows the results. We show only the speedups, as the error rate is the same as that of the CPU experiments. The parameter was set to achieve an error rate of roughly 10^{-1} (refer back to Figure 3). Our method is clearly very effective in this setting; despite the challenging hardware design, it provides a one-to-two order of magnitude speedup on all datasets.

D. Cover Tree comparison

Finally, we compare the performance of the RBC to the state-of-the-art sequential algorithm for NN, the Cover Tree. As discussed previously, algorithms for manycore systems (or parallel systems in general) are burdened with major additional design constraints as compared to sequential algorithms. But, if a sequential algorithm is far superior to a parallel one, then it may not be worth using the parallel one at all. Here we show that the RBC is algorithmically competitive with the state-of-the-art: even without leveraging massive parallelism, its performance is comparable to the Cover Tree's and is even superior on the higher-dimensional data sets.

We compare the exact RBC algorithm to the Cover Tree on a quad-core desktop, using code from [8]. The Cover Tree has the advantage that it can use the full architectural advantages of a modern CPU (in particular branching) and that the algorithm need not scale to many cores; the RBC has the advantage that it can use all the cores (though there are only four). Both algorithms are designed for exact nearest neighbor search under the same notion of intrinsic dimensionality.

Table III shows the results.[8] The RBC is competitive on all of the datasets, and significantly outperforms the Cover Tree on the three largest datasets. Again, these results are surprising, as our exact search algorithm is much simpler than the Cover Tree search algorithm, and since our methods have the additional (significant) constraint that they must work on highly parallel systems.

[8]We were unable to get the Cover Tree software to run on the full TinyIm data sets, so we reduced the database size to 1M.

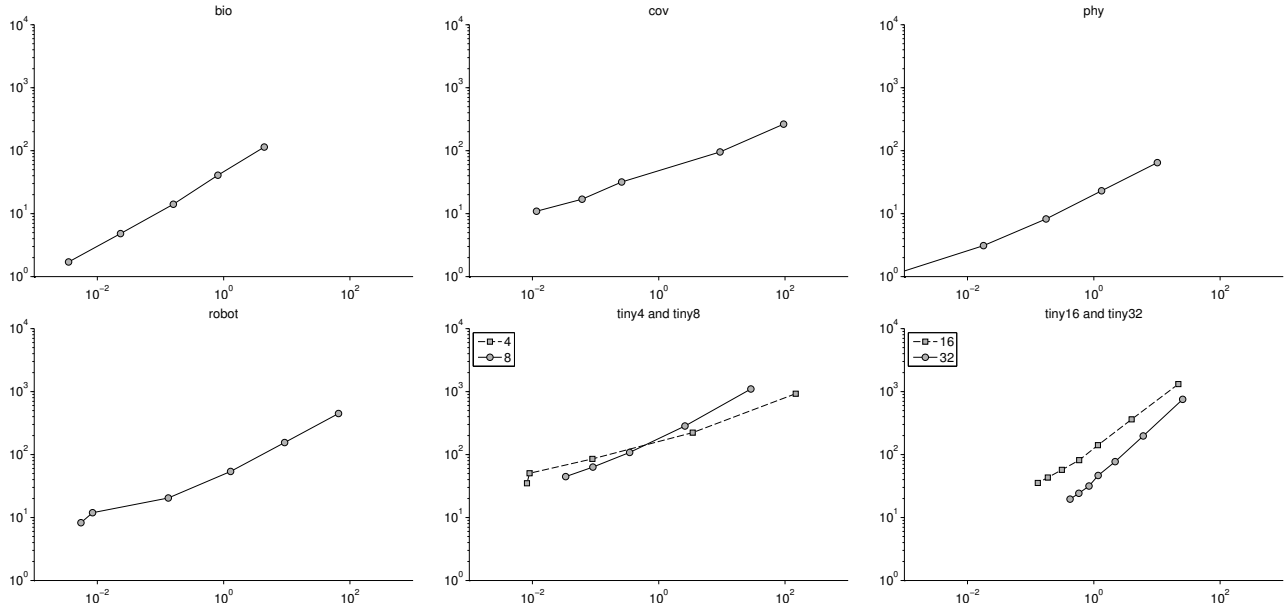


Figure 3. Results of the one-shot algorithm. This is a log-log plot of the speedup as a function of the error rate. The x-axis is logarithmic and runs from 10^{-3} to 10^{3} and signifies the average (over queries) rank of the returned result. For example, a rank of 10^{0} indicates that on average the algorithm returns the 2nd NN. The y-axis is also logarithmic and runs from 10^{0} (no speedup) to 10^{4} (10000x speedup).

Data	Cover Tree	RBC
Bio	18.9	6.4
Covertype	0.4	1.1
Physics	1.9	1.7
Robot	4.6	5.1
Tiny4	0.5	1.2
Tiny8	14.6	3.3
Tiny16	178.9	25.1
Tiny32	387.0	67.9

Table III
COMPARISON OF THE COVER TREE AND THE EXACT RBC ALGORITHM ON A QUAD-CORE DESKTOP MACHINE. TIMES SHOWN ARE THE TOTAL QUERY TIME IN SECONDS FOR 10K QUERIES.

We note that the RBC has a significantly lower theoretical dependence on the dimensionality than the Cover Tree ($O(c^{3/2})$ versus $O(c^{12})$). This is reflected in the experiments; the two datasets that the Cover Tree significantly outperforms the RBC on are very low-dimensional: the Tiny4 data set is four-dimensional, and the Covertype dataset has low intrinsic dimensionality [8]. This reduced dependence on dimensionality appears to be another advantage of the RBC and would be interesting to explore further.

X. CONCLUSION

In this paper, we introduced techniques for metric similarity search on parallel systems. In particular, we demonstrated that the RBC search algorithms significantly reduce the work required for NN retrieval, while being structured in such a way that can be easily implemented on parallel systems. Our experiments show that these techniques are practical on a range of modern hardware. The theory behind the RBC shows that the data structure is broadly effective.

Our code is available for download. This code supplies additional implementation details for the RBC. Moreover, the implementations are practical tools for many NN problems.

An interesting direction for future work is to explore the performance of the RBC in a distributed or multi-GPU environment. The RBC data structure suggests a simple distribution of the database according to the representatives that could be quite effective in such environments. There are many interesting details for study here, such as I/O and communication costs, and the connection to distributed programming paradigms. Furthermore, a distributed implementation would be a useful toolkit item.

REFERENCES

[1] A. Torralba, R. Fergus, and W. T. Freeman, "80 million tiny images: a large dataset for non-parametric object and scene recognition," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2008.

[2] J. Buhler, "Efficient large-scale sequence comparison by locality-sensitive hashing," *Bioinformatics*, vol. 17, no. 5, pp. 419–28, 2001.

[3] D. L. Donoho, "High-dimensional data analysis: The curses and blessings of dimensionality," AMS Conf. on Math Challenges of the 21st century, 2000.

[4] S. T. Roweis and L. K. Saul, "Nonlinear dimensionality reduction by locally linear embedding," *Science*, 2000.

[5] J. B. Tenenbaum, V. de Silva, and J. C. Langford, "A global geometric framework for nonlinear dimensionality reduction," *Science*, 2000.

[6] D. R. Karger and M. Ruhl, "Finding nearest neighbors in growth-restricted metrics," in *Proc. Symposium on Theory of Computing (STOC)*, 2002.

[7] R. Krauthgamer and J. R. Lee, "Navigating nets: simple algorithms for proximity search," in *Proc. Symposium on Discrete Algorithms (SODA)*, 2004.

[8] A. Beygelzimer, S. Kakade, and J. Langford, "Cover trees for nearest neighbor," in *Proceedings of the International Conference on Machine Learning*, 2006.

[9] D. Patterson, "The trouble with multicore," *IEEE Spectrum*, July 2010.

[10] E. Chávez, G. Navarro, R. Baeza-Yates, and J. L. Marroquín, "Searching in metric spaces," *ACM Comput. Surv.*, vol. 33, no. 3, pp. 273–321, 2001.

[11] K. L. Clarkson, "Nearest-neighbor searching and metric space dimensions," in *Nearest-Neighbor Methods for Learning and Vision: Theory and Practice*, 2006.

[12] E. Vidal, "An algorithm for finding nearest neighbours in (approximately) constant average time," *Pattern Recognition Letters*, vol. 4, pp. 145–157, 1986.

[13] S. Omohundro, "Five balltree construction algorithms," ICSI, Tech. Rep., 1989.

[14] P. N. Yianilos, "Data structures and algorithms for nearest neighbor search in general metric spaces," in *Symposium on Discrete Algorithms*, 1993.

[15] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Wu, "An optimal algorithm for approximate nearest neighbor searching," *Journal of the ACM*, vol. 45(6), pp. 891–923, 1998.

[16] S. Brin, "Near neighbor search in large metric spaces," in *Proc. Very Large Data Bases*, 1995.

[17] R. Weber, H.-J. Schek, and S. Blott, "A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces," in *Proc. Very Large Data Bases*, 1998.

[18] P. Indyk and R. Motwani, "Approximate nearest neighbors: towards removing the curse of dimensionality," in *Proc. Symposium on Theory of Computing (STOC)*, 1998.

[19] W. Dong, Z. Wang, W. Josephson, M. Charikar, and K. Li, "Modeling LSH for performance tuning," in *Proc. ACM conference on Information and Knowledge Management (CIKM)*, 2008.

[20] K. L. Clarkson, "Nearest neighbor queries in metric spaces," *Discrete and Computational Geometry*, vol. 22, pp. 63–93, 1999.

[21] N. K. Govindaraju, B. Lloyd, W. Wang, M. Lin, and D. Manocha, "Fast computation of database operations using graphics processors," in *Proceedings of ACM SIGMOD*. ACM Press, 2004, pp. 215–226.

[22] C. Kim, J. Chhugani, N. Satish, E. Sedlar, A. D. Nguyen, T. Kaldewey, V. W. Lee, S. A. Brandt, and P. Dubey, "FAST: fast architecture sensitive tree search on modern CPUs and GPUs," in *Proceedings of ACM SIGMOD*, 2010.

[23] B. Bustos, O. Deussen, S. Hiller, and D. Keim, "A graphics hardware accelerated algorithm for nearest neighbor search," in *Computational Science ICCS, volume 3994 of LNCS*. Springer, 2006, pp. 196–199.

[24] V. Garcia, E. Debreuve, and M. Barlaud, "Fast k nearest neighbor search using GPU," in *CVPR Workshop on Computer Vision on GPU (CVGPU)*, 2008.

[25] M. Al Hasan, H. Yildirim, and A. Chakraborty, "Sonnet: Efficient approximate nearest neighbor using multi-core," in *International Conference on Data Mining (ICDM)*, 2010.

[26] N. Bell and M. Garland, "Implementing sparse matrix-vector multiplication on throughput-oriented processors," in *Proceedings of Supercomputing*, 2009.

[27] J. Heinonen, *Lectures on Analysis on Metric Spaces*. Springer, 2000.

[28] S. Roosta, *Parallel Processing and Parallel Algorithms: Theory and Computation*. Springer, 1999.

[29] A. Frank and A. Asuncion, "UCI machine learning repository," 2010. [Online]. Available: http://archive.ics.uci.edu/ml

[30] P. Ram, D. Lee, H. Ouyang, and A. Gray, "Rank-approximate nearest neighbor search: Retaining meaning and speed in high dimensions," in *Advances in Neural Information Processing Systems 22*, 2009.

[31] D. Nguyen-Tuong and J. Peters, "Using model knowledge for learning inverse dynamics," in *Proc. IEEE International Conference on Robotics and Automation*, 2010.

[32] T. Liu, A. W. Moore, A. Gray, and K. Yang, "An investigation of practical approximate neighbor algorithms," in *Advances in Neural Information Processing Systems*, 2004.

[33] W. B. Johnson and J. Lindenstrauss, "Extensions of lipschitz mappings into a hilbert space," *Contemporary Mathematics*, 1984.

[34] L. Cayton, "A nearest neighbor data structure for graphics hardware," in *First International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures*, 2010.

APPENDIX

We prove Lemma 1.

Proof: Suppose that x is q's NN in X—i.e. $\rho(q,x) \leq \rho(q,y)$ for all $y \in X$—and that r^* owns x (r^* is x's NN among R). Furthermore, let r be q's NN in R. Since $R \subset X$ and $\rho(q,r) = \gamma$, γ gives an upper bound on the distance to q's NN; hence $\rho(q,x) \leq \gamma$. Using this bound along with the triangle inequality gives $\rho(x,r) \leq 2\gamma$, but since $\rho(x,r^*) \leq \rho(x,r)$, we have $\rho(x,r^*) \leq 2\gamma$ as well. Applying the triangle inequality to the bounds on $\rho(x,r^*)$ and $\rho(q,x)$ yields the lemma. ■