# Fast construction of $k$-Nearest Neighbor Graphs for Point Clouds

Michael Connor, Piyush Kumar

**Abstract**—We present a parallel algorithm for $k$-nearest neighbor graph construction that uses Morton ordering. Experiments show that our approach has the following advantages over existing methods: (1) Faster construction of k-nearest neighbor graphs in practice on multi-core machines. (2) Less space usage. (3) Better cache efficiency. (4) Ability to handle large data sets. (5) Ease of parallelization and implementation. If the point set has a bounded expansion constant, our algorithm requires one comparison based parallel sort of points according to Morton order plus near linear additional steps to output the $k$-nearest neighbor graph.

**Index Terms**—Nearest neighbor searching, point based graphics, $k$-nearest neighbor graphics, Morton Ordering, parallel algorithms.

## 1 Introduction

This paper presents the design and implementation of a simple, fine-grain parallel and cache-efficient algorithm to solve the $k$-nearest neighbor graph computation problem for cache-coherent shared memory multi-processors. Given a point set of size $n$ it uses only $\mathcal{O}(n)$ space. We show that with a bounded expansion constant $\gamma$ (as described in [15]), using $\mathbf{p}$ threads (assuming $\mathbf{p}$ cores are available for computation) we can compute the $k$-NNG in $\mathcal{O}(\lceil \frac{n}{\mathbf{p}} \rceil k \log k)$ unit steps, plus one parallel sort on the input. We also present extensive experimental results on a variety of architectures.

The $k$-nearest neighbor graph problem is defined as follows: given a point cloud $P$ of $n$ points in $\mathbb{R}^d$ and a positive integer $k \leq n-1$, compute the $k$-nearest neighbors of each point in $P$. More formally, let $P = \{p_1, p_2, \ldots, p_n\}$ be a point cloud in $\mathbb{R}^d$ where $d \leq 3$. For each $p_i \in P$, let $\mathcal{N}_i^k$ be the $k$ points in $P$, closest to $p_i$. The *$k$-nearest neighbor graph* ($k$-NNG) is a graph with vertex set $\{p_1, p_2, \ldots, p_n\}$ and edge set $E = \{(p_i, p_j) : p_i \in \mathcal{N}_j^k \text{ or } p_j \in \mathcal{N}_i^k\}$. The well known all-nearest-neighbor problem corresponds to the $k = 1$ case. For the purpose of this paper we are constraining ourselves to Euclidean distance, as well as low dimensions.

The problem of computing k-nearest neighbor graph computation arises in many applications and areas including computer graphics, visualization, pattern recognition, computational geometry and geographic information systems. In graphics and visualization, computation of $k$-NNG forms a basic building block in solving many important problems including normal estimation [18], surface simplification [24], finite element modeling [8], shape modeling [25], watermarking [11], virtual walkthroughs [7] and surface reconstruction [1], [13]. With the growing sizes of point clouds, the emergence of multi-core processors in mainstream computing and the increasing disparity between processor and memory speed; it is only natural to ask if one can gain from the use of parallelism for the $k$-NNG construction problem.

The naive approach to solve the $k$-NNG construction problem uses $\mathcal{O}(n^2)$ time and $\mathcal{O}(nk)$ space. Theoretically, the $k$-NNG can be computed in $\mathcal{O}(n \log n + nk)$ [4]. The method is not only theoretically optimal and elegant but also parallelizable. Unfortunately, in practice, most practitioners choose to use variants of kd-tree implementations [18], [23], [8] because of the high constants involved in theoretical algorithms [29], [4], [9], [12]. In low dimensions, one of the best kd-tree implementations is by Arya et al. [2]. Their kd-tree implementation is very carefully optimized both for memory access and speed, and hence has been the choice of practitioners for many years to solve the $k$-NNG problem in point based graphics [18], [25], [8]. In our experiments we use this implementation as a basis of comparison, and results indicate that for $k$-NNG construction our algorithm has a distinct advantage.

Our method uses Morton order or Z-order of points, a space filling curve that has been used previously for many related problems. Tropf and Herzog [27] present a precursor to many nearest neighbor algorithms. Their method uses one, unshifted Morton order point set to conduct *range queries*. The main drawbacks of their method were: (1) It does not allow use of non-integer keys. (2) It does not offer a rigourous proof of worst case or expected run time, in fact it leaves these as an open problem. (3) It offers no scheme to easily parallelize their algorithm. Orenstein and Merrett [22] described another data structures for range searching using Morton order on integer keys [19]. Bern [3] described an algorithm using $2^d$ shifted copies of a point set in Morton order to compute an approximate solution to the $k$-nearest

• *Michael Connor and Piyush Kumar are with the Department of Computer Science, Florida State University, Tallahassee, FL 32306.* *This research was funded by NSF through CAREER Award CCF-0643593.*
*E-mail:* {miconnor,piyush}@cs.fsu.edu. *Software available at : http://compgeom.com/∼stann.*

*Manuscript received Oct 19, 2008; revised January 11, 2007.*

neighbor problem. This paper avoids a case where two points lying close together in space are far away in the one-dimensional order, in expectation. In all these algorithms, the Morton order was determined using explicit interleaving of the coordinate bits.

Liao et al. [16] used Hilbert curves to compute an approximate nearest neighbor solution using only $d+1$ shifted curves. Chan [5] refined this approach, and later presented an algorithm that used only one copy of the data, while still guaranteeing a correct approximation result for the nearest neighbor problem [6]. It is worth noting that in practice, these methods for $k$-nearest neighbor computation are less efficient than state of the art kd-tree based algorithms [6]. However, in this paper, we show that a single shift based algorithm, that is also exact, cache friendly and multi-core aware, can compete with a kd-tree algorithm in both theory and practice.

Recently, the $k$-NNG construction problem has also been studied in the external memory setting or using the disk access model [26]. The design and implementation of our algorithm is more tuned toward the cache-oblivious model [14] and hence differs significantly from [26]. While our algorithm was not specifically designed for external memory, experiments have shown that through the use of large amounts of swap space it can handle very large data sets. Parallel to this work, the same group has announced a GPU based $k$-NNG algorithm [17]. The paper is mostly concentrated on solving the similarity join problem efficiently, which is a different problem than the $k$-NNG problem. We believe that our rigorous analysis on the running time for getting exact answers, as well as the use of floating point coordinates in Morton ordering (Algorithm 1) could help the GPU implementation of similarity joins.

Among the various multi-processor/core architectures that have been proposed in recent times, a new tightly coupled multi-core multi-processor architecture is gaining a central place in high performance computing. These new architectures have a shared address programming model with physically distributed memory and coherent replication (either in caches or main memory). These architectures usually comprise of one or more multi-core or chip multi-processor CPUs, and are emerging as the dominant computing platform. This class of systems differs a lot from the traditional message passing machines, clusters or vector supercomputers.

Our algorithm mainly consists of the following three high level components:

- **Preprocessing Phase:** In this step, we sort the input points $P$ using Morton ordering (a space filling curve).
- **Sliding Window Phase:** For each point $p$ in the sorted array of points, we compute its approximate $k$-nearest neighbors by scanning $\mathcal{O}(k)$ points to the left and right of $p$. Another way to think of this step is, to slide a window of length $\mathcal{O}(k)$ on the sorted array and find the $k$-nearest neighbors restricted to this window.
- **Search Phase:** We refine the answers of the last phase by zooming inside the constant factor approximate $k$-nearest neighbor balls using properties of the Morton order.

The first phase is implemented using parallel Quick Sort [28]. In section 2 we describe Morton ordering and a parallel implementation of the final two phases. Section 2 also presents the analysis of the running time of our algorithm. Section 3 describes the experimental setup we use. Section 4 presents our experimental results. Section 5 concludes the paper.

## 2 METHODOLOGY

In this section we describe our algorithm in detail. Before we start the description, we need to describe the Morton order on which our algorithm is based.

### 2.1 Morton Ordering

Morton order or Z-order is a space filling curve with good locality preserving behavior. It is often used in data structures for mapping multi-dimensional data to one dimension. The Z-value of a point in multiple dimensions can be calculated by interleaving the binary representations of its coordinate values. Our preprocessing phase consists of sorting the input data using their Z-values without explicitly computing the Z-value itself. The Morton order curve can be conceptually achieved

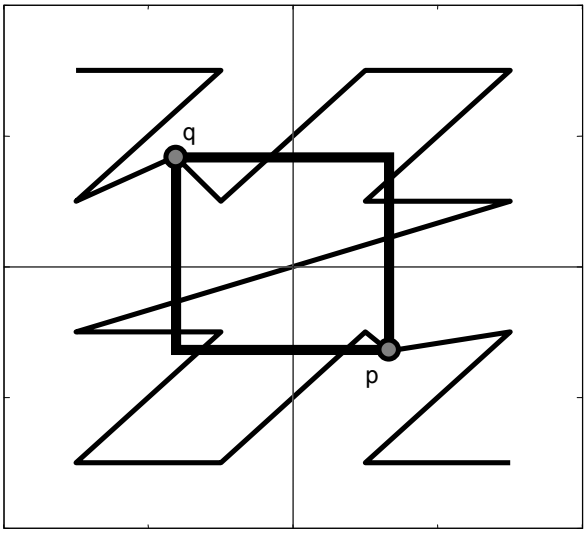



Fig. 1: The Morton order curve preceding the upper left corner, and following the lower right corner of a box, will never intersect the box.

by recursively dividing a $d$-dimensional cube into $2^d$ cubes, then ordering those cubes, until at most 1 point resides in each cube. In 2 and 3 dimensions, the resulting

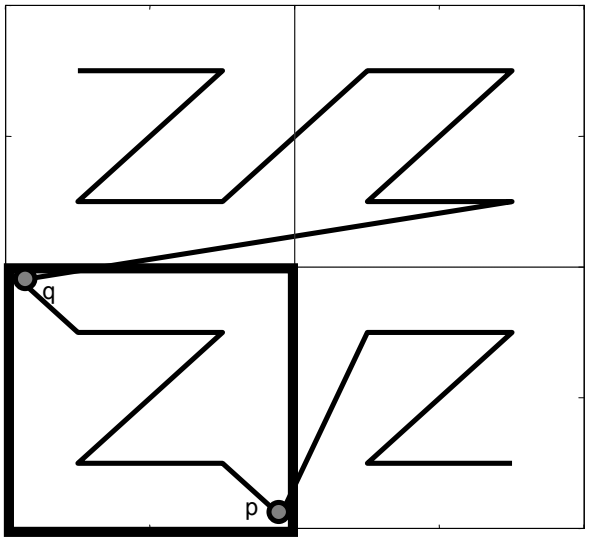



Fig. 2: The smallest quadtree box containing two points will also contain all points lying between the two in Morton order.

**Algorithm 1** Floating Point Morton Order Algorithm

**Require:** $d$-dimensional points $p$ and $q$
**Ensure:** **true** if $p < q$ in Morton order

```
1:  procedure COMPARE(point p , point q)
2:      x ← 0; dim ← 0
3:      for all j = 0 to d do
4:          y ← XORMSB(p_(j), q_(j))
5:          if x < y then
6:              x ← y; dim ← j
7:          end if
8:      end for
9:      return p_(dim) < q_(dim)
10: end procedure
11: procedure XORMSB(double a , double b)
12:     x ←EXPONENT(a); y ←EXPONENT(b)
13:     if x = y then
14:         z ← MSDB(MANTISSA(a),MANTISSA(b))
15:         x ← x − z
16:         return x
17:     end if
18:     if y < x then return x
19:     else return y
20: end procedure
```

trees are sometimes referred to as quadtrees and octrees respectively. Although, we use quadtrees to explain our algorithm, the algorithm itself extends easily to higher dimensions. Two simple properties of Morton order, shown in Figure 1 and Figure 2, will be used to prune points in our $k$-NNG construction algorithm.

Chan [6] showed that the relative Morton order of two integer points can be easily calculated, by determining which pair of coordinates have the first differing bit in binary notation in the largest place. He further showed that this can be accomplished using a few binary operations. Our preliminary experiments with sorting points in Z-order showed that Chan's trick was faster than explicitly interleaving the binary representation of the coordinate values.

While Chan's method only applies to integer types, it can be extended to floating point types as shown in Algorithm 1. The algorithm takes two points with floating point coordinates. The relative order of the two points is determined by the pair of coordinates who have the first differing bit with the highest exponent. The XORMSB function computes this by first comparing the exponents of the coordinates, then comparing the bits in the mantissa, if the exponents are equal. Note that the MSDB function on line 14 returns the most significant differing bit of two integer arguments. This is calculated by first XORing the two values, then shifting until we reach the most significant bit. The EXPONENT and MANTISSA functions return those parts of the floating point number in integer format. This algorithm allows the relative Morton order of points with floating point coordinates to be found in $\mathcal{O}(d)$ time and space complexity.

### 2.2 Notation

In general, lower case Roman letters denote scalar values. $p$ and $q$ are specifically reserved to refer to points. $P$ is reserved to refer to a point set. $n$ is reserved to refer to the number of points in $P$.

We write $p < q$ iff the Z-value of $p$ is less than $q$ ($>$ is used similarly). We use $p^s$ to denote the shifted point $p + (s, s, \ldots, s)$. $P^s = \{p^s | p \in P\}$. $dist(p, q)$ denotes the Euclidean distance between $p$ and $q$.

$p_i$ is the $i$-th point in the sorted Morton ordering of the point set. We use $p_{(j)}$ to denote the $j$-th coordinate of the point $p$. The Morton ordering also defines a quadtree on the point cloud. $box_Q(p_i, p_j)$ refers to the smallest quadtree box that contains the points $p_i$ and $p_j$. We use $box(c, r)$ to denote a box with center $c$ and radius $r$. The radius of a box is the radius of the inscribed sphere of the box.

$k$ is reserved to refer to the number of nearest neighbors to be found for every point. $d$ is reserved to refer to the dimension.

In general, upper case Roman letters ($B$) refer to a bounding box. Bounding boxes with a subscript Q ($B_Q$) refer to a quadtree box. and $dist(p, B)$ is defined as the minimum distance from point $p$ to box (or quadtree box) $B$. $E[\,]$ is reserved to refer to an expected value. $\mathcal{E}$ refers to an event. $P(\mathcal{E})$ is reserved to refer to the probability of an event $\mathcal{E}$. $A_i$ is reserved to refer to the current $k$ nearest neighbor solution for point $p_i$, which may still need to be refined to find the exact nearest neighbors. $nn^k(p, \{\})$ defines a function that returns the $k$ nearest neighbors to $p$ from a set. The bounding box of $A_i$ refers

to the minimum enclosing box for the ball defined by $A_i$. Finally, $rad(p, \{\})$ returns the distance from point $p$ to the farthest point in a set. $rad(A_i) = rad(p_i, A_i)$.

### 2.3 The k-Nearest Neighbor Graph Construction Algorithm

Algorithm 2 describes our $k$-NNG algorithm in more detail. The first step of the construction is to sort the input point set $P$, using the Morton order comparison operator. In our implementation, this is done using a parallel Quick Sort [28].

Once $P$ has been sorted, a partial solution is found for each point $p_i$ by finding the $k$ nearest neighbors from the set of points $\{p_{i-ck} ... p_{i+ck}\}$ for some constant $c \geq \frac{1}{2}$. This is done via a linear scan of the range of points (corresponding to Line 4). The actual best value of $c$ is platform dependant; we chose a value of 1 based on tuning, and in general $c$ should not be too large. Once this partial nearest neighbor solution is found, its correctness can be checked using the property of Morton ordering shown in Figure 1. If the corners of the bounding box for our current solution lie within the range we have already searched, then the partial solution we have calculated is in fact the true solution (see Figure 3). This check is performed in lines 5 and 10. If the current approximate nearest neighbor ball is not bounded by the lower and upper points already checked, lines 7 and 8 (also 12 and 13), use a binary search to find the location of the lower and upper corner of the bounding box of the current approximate nearest neighbor ball in the Morton ordered point set. This defines the range that needs to be searched by the CSEARCH function.

For each point $p_i$ for which the solution was not found in the previous step, the partial solution must be refined to find the actual solution. This is done using a recursive algorithm. Given a range of points $\{p_a ... p_b\}$, we first check if the distance $r$ from $p_i$ to $box_Q(p_a, p_b)$ is greater than the radius of $A_i$ (line 24). If it is, then the current solution does not intersect this range. Otherwise, we update $A_i$ with $p_{a+b/2}$. We then repeat the procedure for the ranges $\{p_a ... p_{a+b/2-1}\}$ and $\{p_{a+b/2+1} ... p_b\}$. One important observation is the property used as a check in the scan portion of the algorithm still holds (lines 5 and 10), and one of these two new ranges of points may be eliminated by comparing the bounding box of $A_i$ with $p_{a+b/2}$ (lines 27 and 30). If the length of a range is less than $\nu$, a fixed constant, we do a linear scan of the range instead of recursing further. Since a good value for $\nu$ is platform dependent, we used a self tuning program to experimentally determine its value. On most architectures, $\nu = 4$ worked well.

### 2.4 Parallel Construction

Parallel implementation of this algorithm happens in three phases. For the first phase, a parallel Quick Sort is used in place of a standard sorting routine. Second, the sorted array is split into $\mathbf{p}$ chunks (assuming $\mathbf{p}$ threads to be used), with each thread computing the initial approximate nearest neighbor ball for one chunk independently. Finally, each thread performs the recursive step of the algorithm on each point in its chunk.

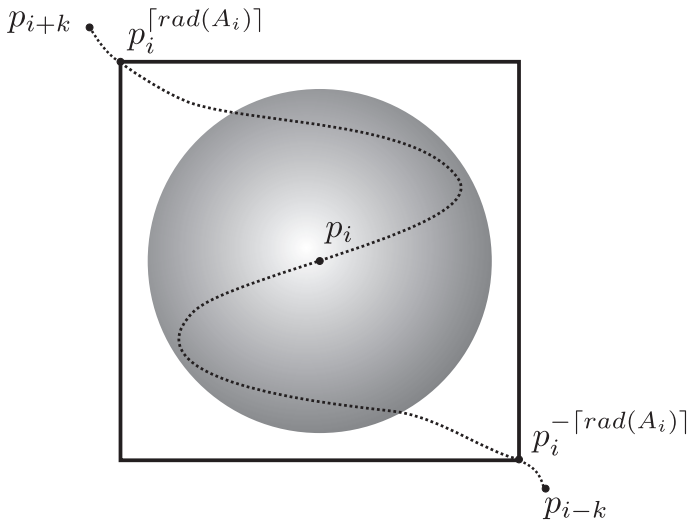



Fig. 3: Pictorial representation of Algorithm 2, Line 5. Since all the points inside the approximate nearest neighbor ball of $p_i$ have been scanned, we must have found the nearest neighbor. This happens because $p_i^{\lceil rad(A_i) \rceil}$ is the largest point with respect to the Morton ordering compared to any point inside the box. Hence any point greater than $p_{i+k}$ in Morton ordering cannot intersect the box shown. A similar argument holds for $p_i^{-\lceil rad(A_i) \rceil}$ and $p_{i-k}$.

### 2.5 Handling large data sets

Many applications of $k$-NNG construction require large point clouds to be handled that do not fit in memory. One way to handle this problem is to make disk-based data structures [26]. We use an alternative solution by simply increasing the swap space of the operating system and running the same implementation that we did in internal memory. Many operating systems allow on the fly creation and deletion of temporary swap files (Windows, Linux), which can be used to run our code on very large data sets (100 million or more points). Unfortunately, we were unable to compare our results to the previously mentioned disk-based methods [26] directly (their code was not available). However, we were able to calculate $k$-NNG for much larger data sets (up to 285 million points as seen in Table 1).

In Linux, new user space memory allocations (using `new` or `malloc`) of large sizes are handled automatically using `mmap` which is indeed a fast way to do IO from disks. Once the data is memory mapped to disk, both

**Algorithm 2** KNN Graph Construction Algorithm

**Require:** Randomly shifted point set $P$ of size $n$. Morton order compare operators: $<$,$>$. (COMPARE= $<$).
**Ensure:** $A_i$ contains $k$ nearest neighbors of $p_i$ in $P$.

```
1: procedure CONSTRUCT(P , int k)
2:     P ← ParallelQSort(P, <)
3:     parallel for all p_i in P
4:         A_i ← nn^k(p_i, {p_{i-k}, ..., p_{i+k}})
5:         if p_i^{⌈rad(A_i)⌉} < p_{i+k} then u ← i
6:         else
7:           I ← 1; while p_i^{⌈rad(A_i)⌉} < p_{i+2^I} do: ++I
8:           u ← min(i + 2^I, n)
9:         end if
10:        if p_i^{-⌈rad(A_i)⌉} > p_{i-k} then l ← i
11:        else
12:          I ← 1; while p_i^{-⌈rad(A_i)⌉} > p_{i-2^I} do: ++I
13:          l ← max(i - 2^I, 1)
14:        end if
15:        if l ≠ u then CSEARCH(p_i, l, u)
16: end procedure
17: procedure CSEARCH(point p_i , int l , int h)
18:    if (h - l) < ν then
19:        A_i ← nn^k(p_i, A_i ∪ {p_l ... p_h})
20:        return
21:    end if
22:    m ← (h + l)/2
23:    A_i ← nn^k(p_i, A_i ∪ p_m)
24:    if dist(p_i, box(p_l, p_h)) ≥ rad(A_i) then return
25:    if p_i < p_m then
26:        CSEARCH(p_i, l, m - 1)
27:        if p_m < p_i^{⌈r(A_i)⌉} then CSEARCH(p_i, m + 1, h)
28:    else
29:        CSEARCH(p_i, m + 1, h)
30:        if p_i^{-⌈r(A_i)⌉} < p_m then CSEARCH(p_i, l,m - 1)
31:    end if
32: end procedure
```

sorting and scanning preserve locality of access in our algorithm and hence are not only cache friendly but also disk friendly. The last phase of our algorithm is designed to be disk friendly as well. Once an answer is computed for point $p_i$ by a single thread, the next point in the Morton order uses the locality of access from the previous point and hence causes very few page faults in practice.

### 2.6 Analysis of the Algorithm

In this section, we show that Algorithm 2 runs in expected $\mathcal{O}(\lceil \frac{n}{\mathbf{p}} \rceil k \log k)$ unit operations plus a parallel Quick Sort call for point clouds with bounded expansion constant $\gamma = \mathcal{O}(1)$. Except for the storage of the input and the output, our algorithm needs only an $\mathcal{O}(\mathbf{p}k)$ extra space.

Let $P$ be a finite set of points in $\mathbb{R}^d$ such that $|P| = n \gg k \geq 1$. Let $\mu$ be a counting measure on $P$. Let the measure of a box, $\mu(B)$ be defined as the number of points in $B \cap P$. $P$ is said to have expansion constant $\gamma$ if $\forall p_i \in P$ and for all $k \in (1, n)$:

$$\mu(box(p_i, 2 \times \text{rad}(p_i, \mathcal{N}_i^k))) \leq \gamma k$$

This is a similar restriction to the doubling metric restriction on metric spaces [10] and has been used before [15]. Throughout the analysis, we will assume that $P$ has an expansion constant $\gamma = \mathcal{O}(1)$.

The first phase of the algorithm is sorting. The second phase is a linear scan. The dominating factor in the running time will be from the third phase; the recursive CSEARCH function. The running time of this phase will be bounded by showing that the smallest quadtree box containing the actual nearest neighbor ball for a point is, in expectation, only a constant factor smaller than the quadtree ball containing the approximate solution found in phase two. Given the distribution stated above, this implies there are only $\mathcal{O}(k)$ additional points that need to be compared to refine the approximation to the actual solution. We do not concern ourselves with the actual running time of the CSEARCH function, since it is upper bounded by the time it would take to simply scan the $\mathcal{O}(k)$ points.

To prove the running time of our algorithm, we will first need the solution to the following game: In a room tiled or paved with equal square tiles (created using equidistant parallel lines in the plane), a coin is thrown upwards. If the coin rests cleanly within a tile, the length of the square tile is noted down and the game is over. Otherwise, the side length of the square tiles in the room are doubled in size and the same coin is tossed again. This process is repeated till the coin rests cleanly inside a square tile.

Note that in our problem, the square tiles come from quadtrees defined by Morton order, and the coin is defined by the optimal $k$-nearest neighbor ball of $p_i \in P$. What we are interested in bounding, is the number of points inside the smallest quadtree box that contains $box(p_i, rad(p_i, \mathcal{N}_i^k))$. This leads us to the following lemma:

**Lemma 2.1.** *Let $B$ be the smallest box, centered at $p_i \in P$ containing $\mathcal{N}_i^k$ and with side length $2^h$ (where $h$ is assumed w.l.o.g to be an integer $> 0$) which is randomly placed in a quadtree $Q$. If the event $\mathcal{E}_j$ is defined as $B$ being contained in a quadtree box $B_Q$ with side length $2^{h+j}$, and $B_Q$ is the smallest such quadtree box, then*

$$P(\mathcal{E}_j) \leq \left(1 - \frac{1}{2^j}\right)^d \frac{d^{j-1}}{2^{\frac{j^2-j}{2}}}$$

*Proof:* From Figure 4, we can infer that in order for $B$ to be contained in $B_Q$, the total number of candidate boxes where the upper left corner of $B$ can lie, is $(2^j - 1)$ along each dimension. The total number of candidate grey boxes is therefore $(2^j - 1)^d$. The probability that the upper left corner lies in a particular grey box is $2^{hd}/2^{(h+j)d}$. Thus the probability that $B$ is contained in

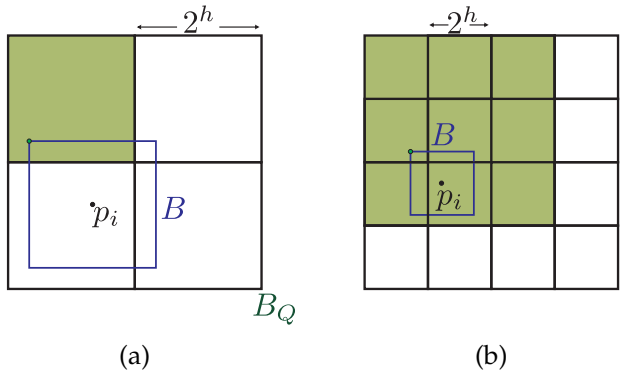



Fig. 4: (a) $B$ lands cleanly on the quadtree box $B_Q$ twice its size. (b) $B$ lands cleanly on a quadtree box $2^2$ times its size. In both figures, if the upper left corner of $B$ lies in the shaded area, the box $B$ does not intersect the boundary of $B_Q$. Obviously, (a) happens with probability $\frac{1}{4}$ (probability $(\frac{1}{2})^d$ in general dimension) and (b) happens with probability $(\frac{2^2-1}{2^2})^2 = \frac{9}{16}$ (probability $(\frac{2^2-1}{2^2})^d$ in general dimension).

$B_Q$ is $((2^j-1)/2^j)^d$. If $B_Q$ is the smallest quadtree box housing $B$, then all quadtree boxes with side lengths $2^{h+1}$,$2^{h+2}$,...,$2^{h+j-1}$ cannot contain $B_Q$. This probability is given by:

$$\prod_{l=1}^{j-1}\left(1-\left(\frac{2^l-1}{2^l}\right)^d\right) = \prod_{l=1}^{j-1}\frac{(2^l)^d-(2^l-1)^d}{(2^l)^d}$$

The probability of $B_Q$ containing $B$ is therefore

$$P(\mathcal{E}_j) = \frac{(2^j-1)^d}{(2^j)^d}\prod_{l=1}^{j-1}\frac{(2^l)^d-(2^l-1)^d}{(2^l)^d}$$

We now use the following inequality: Given $v$ such that $0 < v < 1$; $(1-v)^d \leq 1 - dv + d(d-1)\frac{v^2}{2!}$, which can be easily proved using induction or alternating series estimation theorem. Putting $v = 1/2^l$ we get:

$$\begin{aligned}(1-v)^d \leq & \quad 1 - dv(1+v/2) + d^2v^2/2 \\ \leq & \quad 1 - \tfrac{d}{2^l}\left(1+1/2^{l+1}\right) + d^2/2^{2l+1}\end{aligned}$$

To simplify the sum, we will use a Taylor series with $v = \frac{1}{2^l}$.

$$\begin{aligned}(1-v)^d \leq & \\ & 1 - dv + d(d-1)\frac{v^2}{2!} - d(d-1)(d-2)\frac{v^3}{3!} + \ldots \\ \leq & \quad 1 - dv + (d^2-d)\frac{v^2}{2!} \\ \leq & \quad 1 - dv + \frac{d^2v^2}{2} - \frac{dv^2}{2} \\ \leq & \quad 1 - dv\left(1+\frac{v}{2}\right) + \frac{d^2v^2}{2} \\ \leq & \quad 1 - \frac{d}{2^l}\left(1+\frac{1}{2^{l+1}}\right) + \frac{d^2}{2^{2l+1}}\end{aligned}$$

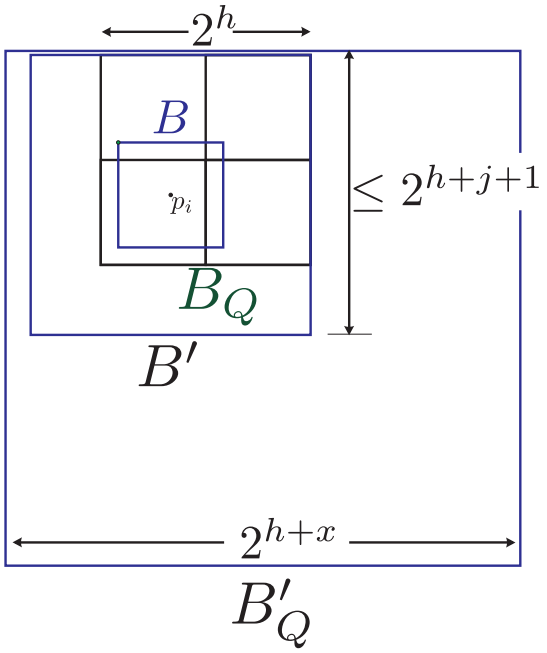



Fig. 5: All boxes referred in Lemma 2.3.

Then, by substituting and simplifying

$$\begin{aligned}P(\mathcal{E}_j) \quad \leq & \quad \left(1-\frac{1}{2^j}\right)^d \prod_{l=1}^{j-1}\left[\frac{d}{2^l}\left(1+\frac{1}{2^{l+1}}\right) - \frac{d^2}{2^{2l+1}}\right] \\ \leq & \quad \left(1-\frac{1}{2^j}\right)^d \prod_{l=1}^{j-1}\frac{d}{2^l}\left[1+\frac{d}{2^{2l+1}} - \frac{d}{2^{2l+1}}\right] \\ \leq & \quad \left(1-\frac{1}{2^j}\right)^d \prod_{l=1}^{j-1}\frac{d}{2^l} \\ \leq & \quad \left(1-\frac{1}{2^j}\right)^d \frac{d^{j-1}}{2^{\frac{j^2-j}{2}}}\end{aligned}$$

□

**Lemma 2.2.** *The linear scan phase of the algorithm produces an approximate $k$-nearest neighbor box $B'$ centered at $p_i$ with radius at most the side length of $B_Q$. Here $B_Q$ is the smallest quadtree box containing $B$, the $k$-nearest neighbor box of $p_i$.*

*Proof:* Our algorithm scans at least $p_{i-k}\ldots p_{i+k}$, and picks the top $k$ nearest neighbors to $p_i$ among these candidates. Let $a$ be the number of points between $p_i$ and the largest Morton order point in $B$. Similarly, let $b$ be the number of points between $p_i$ and the smallest Morton order point in $B$. Clearly, $a+b \geq k$. Note that $B$ is contained inside $B_Q$, hence $\mu(B_Q) \geq k$. Now, $p_i \ldots p_{i+k}$ must contain $a$ points inside $B_Q$. Similarly, $p_{i-k}\ldots p_i$ must contain at least $b$ points from $B_Q$. Since we have collected at least $k$ points from $B_Q$, the radius of $B'$ is upper bounded by the side length of $B_Q$. □

**Lemma 2.3.** *The smallest quadtree box containing $B'$, $B'_Q$,*

*has only a constant number of points more than $k$ in expectation.*

*Proof:* Let there be $k'$ points in $B$. Clearly $k' = \mathcal{O}(k)$ given $\gamma = \mathcal{O}(1)$. The expected number of points in $B'_Q$ is at most $E[\gamma^x k']$, where $x$ is such that if the side length of $B$ is $2^h$ then the side length of $B'_Q$ is $2^{h+x}$. Let the event ${\mathcal{E}'}_x$ be defined as this occurring for some fixed value of $x$.

Recall from Lemma 2.1 that $j$ is such that the side length of $B_Q$ is $2^{h+j}$. The probability for the event $\mathcal{E}_j$ is

$$\begin{aligned} P(\mathcal{E}_j) &= \frac{(2^j-1)^d}{(2^j)^d}\prod_{l=1}^{j-1}\frac{(2^l)^d-(2^l-1)^d}{(2^l)^d} \\ &\leq (1-\frac{1}{2^j})^d\frac{d^{j-1}}{2^{\frac{j^2-j}{2}}} \end{aligned}$$

From Lemma 2.2 $B'$ has a side length of at most $2^{h+j+1} = 2^{h'}$. Let ${\mathcal{E}''}_{j'}$ be the event that, for some fixed $h'$, $B'$ is contained in $B'_Q$ with side length $2^{h'+j'}$. Note that ${\mathcal{E}''}_{j'}$ has the same probability mass function as $\mathcal{E}_j$, and that the value of $j'$ is independent of $j$. Given this, $P({\mathcal{E}'}_x) = \sum_{j=1}^{x-1} P(\mathcal{E}_j)P({\mathcal{E}''}_{j'})$. From this, $E[\gamma^x k']$ follows

$$\begin{aligned} &E[\gamma^x k] \leq \\ &\quad \sum_{x=2}^{\infty}\gamma^x k P({\mathcal{E}'}_x) \\ &= \sum_{x=2}^{\infty}\gamma^x k\sum_{j=1}^{x-1}P(\mathcal{E}_j)P({\mathcal{E}''}_{j'}) \\ &\leq \sum_{x=2}^{\infty}\gamma^x k\sum_{j=1}^{x-1}(1-\frac{1}{2^j})^d\frac{d^{j-1}}{2^{\frac{j^2-j}{2}}}(1-\frac{1}{2^{x-j}})^d\frac{d^{x-j-1}}{2^{\frac{(x-j)^2-(x-j)}{2}}} \\ &\leq \sum_{x=2}^{\infty}\gamma^x k\sum_{j=1}^{x-1}(1-\frac{1}{2^j})^d(1-\frac{1}{2^{x-j}})^d\frac{d^{x-2}}{2^{\frac{x^2-(1+2j)x+2j^2}{2}}} \end{aligned}$$

We now use the fact that, $\forall j \in \{1,2,\ldots,x-1\}$:

$$(1-2^{-j})^d(1-2^{j-x})^d \leq (1-2^{-x/2})^{2d}$$

which can be proved by showing:

$$\begin{aligned} (1-2^{-j})(1-2^{j-x}) &\leq (1-2^{-x/2})^2 \\ \text{or } 2^{-j}+2^{j-x} &\geq 2^{-x/2+1} \end{aligned}$$

which is true because $\frac{a+b}{2} \geq \sqrt{ab}$. Putting this simplified upper bound back in the expectation calculation we get:

$$\begin{aligned} &E[\gamma^x k] = \\ &\leq \sum_{x=2}^{\infty}\gamma^x k\sum_{j=1}^{x-1}\left(1-\frac{1}{2^{x/2}}\right)^{2d}\frac{d^{x-2}}{2^{\frac{x^2-(1+2j)x+2j^2}{2}}} \\ &\leq k\sum_{x=2}^{\infty}\gamma^x\left(1-\frac{1}{2^{x/2}}\right)^{2d}d^{x-2}\sum_{j=1}^{x-1}2^{-\frac{x^2-(1+2j)x+2j^2}{2}} \\ &\leq k\sum_{x=2}^{\infty}\gamma^x\left(1-\frac{1}{2^{x/2}}\right)^{2d}d^{x-2}2^{(x-x^2)/2}\sum_{j=1}^{x-1}2^{j(x-j)} \end{aligned}$$

It is easy to show that $j(x-j) \leq x^2/4\ \forall j \in \{1,\ldots,x-1\}$: Let $a' = x/2$ and $b' = j - x/2$. Then $(a'+b')(a'-b') = a'^2 - b'^2$ where the LHS is $j(x-j)$ and RHS is $x^2/4 - b'^2$. Hence $j(x-j) \leq x^2/4$. Using the fact that $2^{j(x-j)} \leq 2^{(x/2)^2}$ in the expectation calculation, we have:

$$\begin{aligned} &\leq k\sum_{x=2}^{\infty}\gamma^x\left(1-\frac{1}{2^{x/2}}\right)^{2d}d^{x-2}2^{(x-x^2)/2}x2^{x^2/4} \\ &\leq k\sum_{x=2}^{\infty}x(d\gamma\sqrt{2})^x\left(1-\frac{1}{2^{x/2}}\right)^{2d}2^{-x^2/4} \end{aligned}$$

Putting $y = x/2$ and $c = 2(d\gamma)^2$, we get:

$$\leq 2k\sum_{y=1}^{\infty}y\sqrt{c}^{2y}2^{-y^2}\left(1-\frac{1}{2^y}\right)^{2d}$$

Using the Taylor's approximation:

$$\left(1-\frac{1}{2^y}\right)^{2d} \leq \left(1-d2^{-y}\left(2+2^{-y}\right)+d^2 2^{-2y+1}\right)$$

which when substituted:

$$\leq 2k\int_{y=0}^{\infty}y\left(\frac{c}{2^y}\right)^y\left(1-d2^{-y}\left(2+2^{-y}\right)+d^2 2^{-2y+1}\right)dy$$

Integrating and simplifying using the facts that the *error function*, *erf*$(x)$ encountered in integrating the normal distribution follows *erf*$(x) \leq 1$, and $c = 2(d\gamma)^2 = \mathcal{O}(1)$, we have:

$$\begin{aligned} &\leq k\left(\frac{1}{\ln 2}+\left(\sqrt{\pi\ln 2}\right)e^{\frac{(\ln(c))^2}{4\ln(2)}}\right) \\ &= \mathcal{O}(k) \end{aligned}$$

□

Now we are ready to prove our main theorem on the running time of our algorithm:

**Theorem 2.4.** *For a given point set $P$ of size $n$, with a bounded expansion constant and a fixed dimension, in a constrained CREW PRAM model, assuming $\mathbf{p}$ threads, the $k$-nearest neighbor graph can be found in one comparison based sort plus $\mathcal{O}(\lceil\frac{n}{\mathbf{p}}\rceil k\log k)$ expected time.*

*Proof:* Once $B'$ is established in the first part of the algorithm, $B'_Q$ can be found in $\mathcal{O}(\log k)$ time by using a binary search outward from $p_i$ (this corresponds to lines 5 to 14 in Algorithm 2). Once $B'_Q$ is found, it takes at most another $\mathcal{O}(k)$ steps to report the solution. There is an additional $\mathcal{O}(\log k)$ cost for each point update to maintain a priority queue of the $k$ nearest neighbors of $p_i$. Since the results for each point are independent, the neighbors for each point can be computed by an independent thread. Note that our algorithm reduces the problem of computing the $k$-NNG to a sorting problem (which can be solved optimally) when $k = \mathcal{O}(1)$, which is the case for many graphics and visualization applications. Also the expectation in the running time is independent of the input distribution and is valid for arbitrary point clouds. □

## 3 Experimental Setup

To demonstrate the practicality of our algorithm, we ran our implementation on a number of different data sizes. The source code that was used in these tests is available at http://www.compgeom.com/~stann.

The algorithm was tested on three different architecture setups, each detailed in its own section below.

ANN [20] had to be modified to allow a fair comparison. Nearest neighbor graph construction using ANN is done in two phases. The preprocessing stage is the creation of a kd-tree using the input data set. Then, a nearest neighbor query is made for each point in the input data set. For our experiments, we modified the source code for the query to allow multiple threads to query the same data structure simultaneously. We did not modify the kd-tree construction to use a parallel algorithm. However, it is worth noting that even if a parallel kd-tree construction algorithm was implemented, it would almost certainly still be slower than parallel sorting (the preprocessing step in our algorithm). In the interests of a fair comparison, the empirical results section includes several examples of $k$-NNG construction where only one thread was used (Figures 6c, 7c, 8c).

Except where noted, random data sets were generated from 3-dimensional points uniformly distributed between $(0, 1]$, stored as 32 bit floats. Graphs using random data sets were generated using five sets of data, and averaging the result. Results from random data sets with different distributions (such as Gaussian, clustered Gaussian, and spherical) were not significantly different from the uniform distribution. Also included were several non-random data sets, consisting of surface scans of objects. In all graphs, our algorithm will be labeled 'knng(float)'. 'knng(long)' means that the data was scaled to a 64 bit integer grid, and stored as a 64 bit integer. This improves the running time of our algorithm dramatically, and can be done without loss of precision in most applications.

## 4 Experimental Results

### 4.1 Intel Architecture

The system that we experimented on is equipped with dual Quad-core 2.66GHz Intel Xeon CPUs, and a total of 4 GB of DDR memory. Each core has 2 MB of total cache. SUSE Linux with kernel 2.6.22.17-0.1-default was running on the system. We used `gcc` version 4.3.2 for compilation of all our code (with `-O3`).

#### 4.1.1 Construction Time Results

As shown in Figure 6a and Table 1, our algorithm performs very favorably against $k$-NNG construction using ANN. Table 1 shows timings of $k$-NNG construction on very large point sets, where ANN was unable to complete a graph due to memory issues. Construction times improve dramatically when floating points are scaled to an integer grid. Other random distributions had similar construction times to these cases, and so more graphs were not included. Figure 6b shows that as $k$ increases, the advantage runs increasingly toward our implementation. Finally, Figure 6c shows the speedup gained by increasing the number of threads. In these and all other graphs shown, standard deviation was very small; less than 2% of the mean value.

| Dataset | Size (points) | ANN (s) | knng(long) (s) | knng(float) (s) |
|---|---|---|---|---|
| Screw | 27152 | .06 | .04 | .06 |
| Dinosaur | 56194 | .11 | .07 | .11 |
| Ball | 137602 | .31 | .14 | .21 |
| Isis | 187644 | .46 | .18 | .27 |
| Blade | 861240 | 2.9 | .86 | 1.3 |
| Awakening | 2057930 | 8.6 | 2.1 | 3.2 |
| David | 3614098 | 16.7 | 3.7 | 5.6 |
| Night | 11050083 | 62.2 | 12.4 | 18.6 |
| Atlas | 182786009 | - | 1564 | 2275 |
| Tallahassee | 285000000 | - | 2789 | 4235 |

TABLE 1: Construction times for $k = 1$-nearest neighbor graphs constructed on non-random 3-dimensional data sets. Each graph was constructed using 8 threads. All timings are in seconds.

#### 4.1.2 Memory Usage and Cache Efficiency Results

Figure 9 shows the estimated memory usage, per point, for both algorithms. Memory usage was determined using valgrind [21]. As shown in Figure 10, our algorithm has greater cache efficiency than ANN's kd-tree implementation. This should allow for better scaling in our algorithm as processing power increases.

### 4.2 AMD Architecture

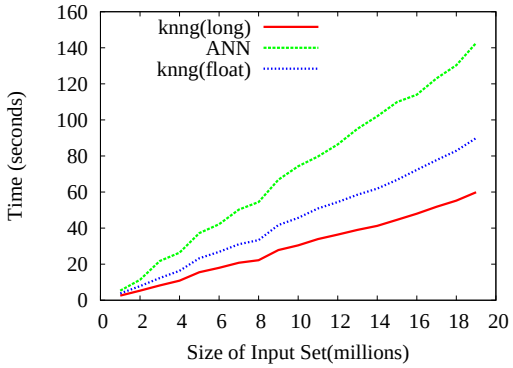
This machine is equipped with 8 Dual Core 2.6GHz AMD Opteron™Processor 885, for a total of 16 cores. Each processor has 128 KB L1 Cache, 2048 KB L2 cache and they share a total of 64 GB of memory. We used `gcc` version 4.3.2 for compilation of all our code (with `-O3`).

As can be seen in Figures 7a, 7b and 7c, the knng algorithm performs well despite the change in architecture. ANN fared particularly poorly on this architecture.

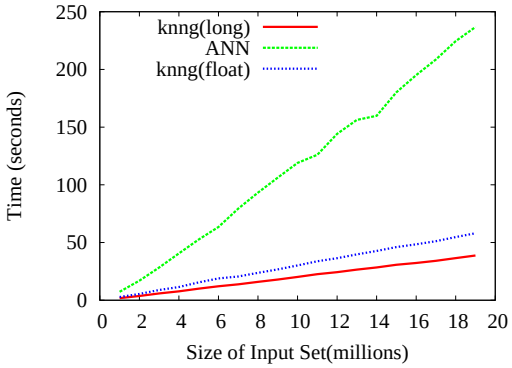
### 4.3 Sun Architecture

This machine is a Sun T5120 server with a eight-core, T2 OpenSparc processor and 32 GB of memory. Overall, this was a slower machine compared to the others that were used, however it was capable of running 64 threads simultaneously. We used `gcc` version 4.3.2 for compilation of all our code (with `-O3`).

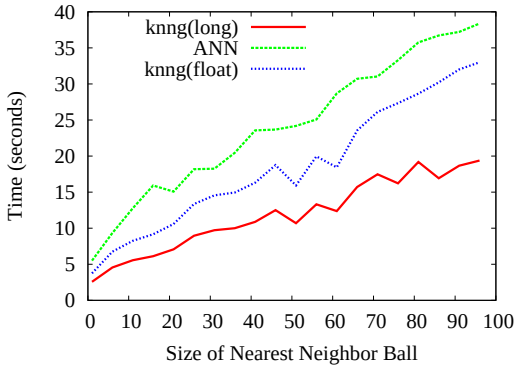
As can be seen in Figure 8a, results for construction were similar to the previous experiments. One unexpected result was ANN's performance as $k$ increased, as seen in Figure 8b. Since ANN was developed on a Sun platform, we believe that the improvements we see as $k$ increases are due to platform specific tuning. In Figure 8c, we observe how both algorithms behave with a large number of threads. Both ANN and the two versions of knng level out as the number of threads increases (processing power is limited to eight cores).



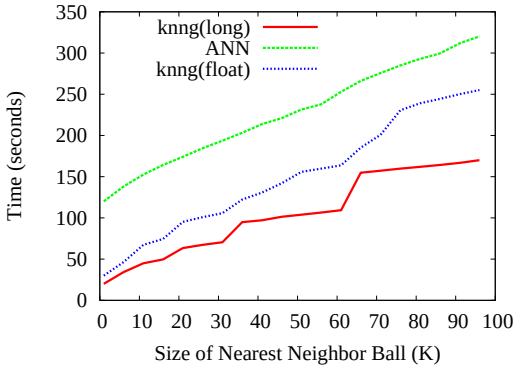

(a) Graph of 1-NN graph construction time vs. number of data points on the Intel architecture. Each algorithm was run using 8 threads in parallel.



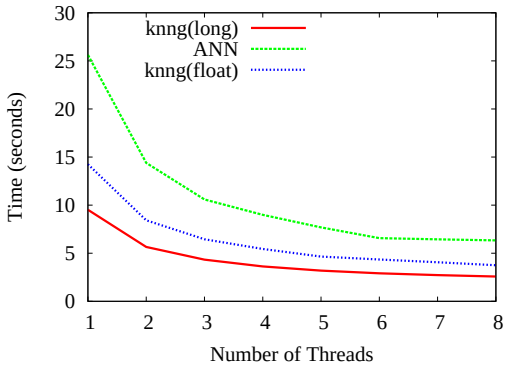

(a) Graph of 1-NN graph construction time vs. number of data points on AMD architecture. Each algorithm was run using 16 threads in parallel.




(b) Graph of $k$-NN graph construction time for varying $k$ on the Intel architecture. Each algorithm was run using 8 threads in parallel. Data sets contained one million points.

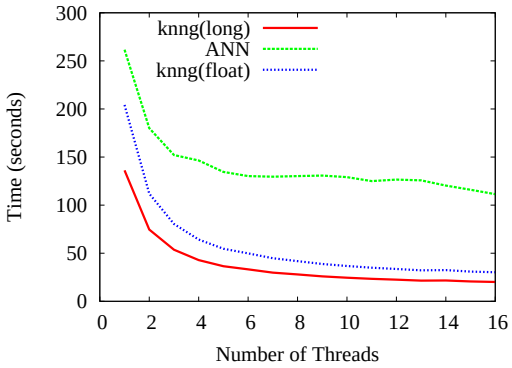



(b) Graph of $k$-NN graph construction time for varying $k$ on AMD architecture. Each algorithm was run using 16 threads in parallel. Data sets contained ten million points.



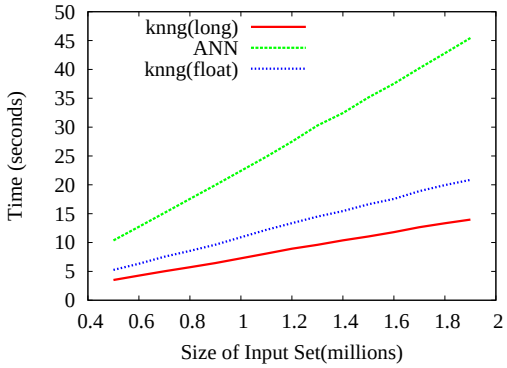

(c) Graph of 1-NN graph construction time for varying number of threads on Intel architecture. Data sets contained ten million points.

Fig. 6: Intel Results

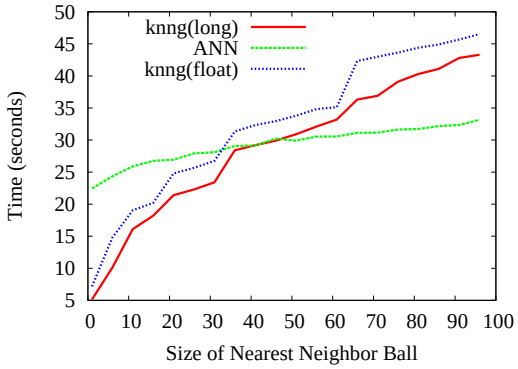



(c) Graph of 1-NN graph construction time for varying number of threads on AMD architecture. Data sets contained ten million points.

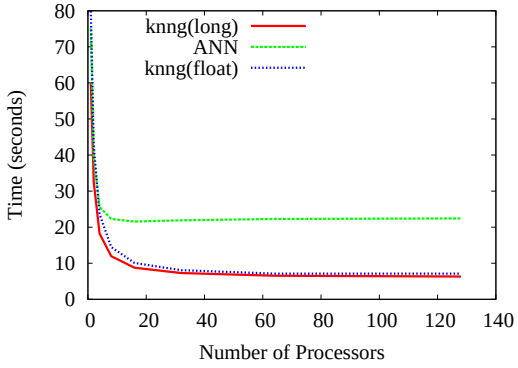
Fig. 7: AMD Results




(a) Graph of 1-NN graph construction time vs. number of data points on Sun architecture. Each algorithm was run using 128 threads in parallel.




(b) Graph of $k$-NN graph construction time for varying $k$ on Sun architecture. Each algorithm was run using 128 threads in parallel. Data sets contained ten million points.




(c) Graph of 1-NN graph construction time for varying number of threads on Sun architecture. Data sets contained ten million points.

Fig. 8: Sun Results

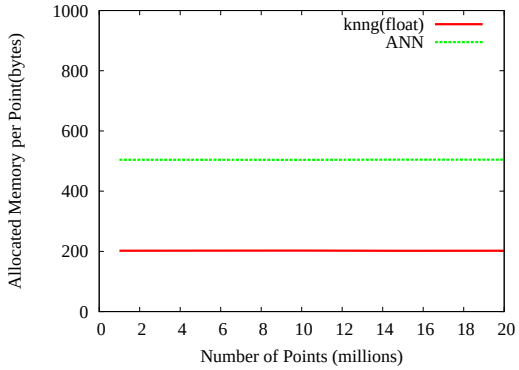



Fig. 9: Graph of memory usage per point vs. data size. Memory usage was determined using valgrind.

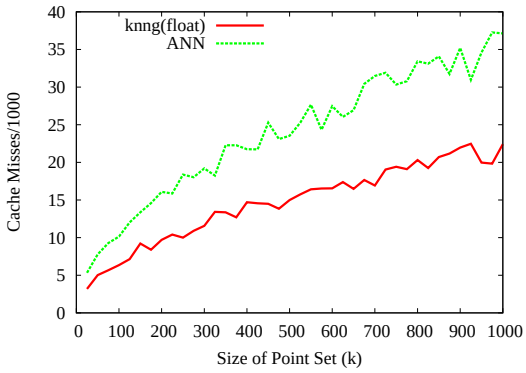



Fig. 10: Graph of cache misses vs. data set size. All data sets were uniformly random 3-dimensional data sets. Cache misses were determined using valgrind which simulated a 2 MB L1 cache.

## 5 Conclusions

We have presented an efficient $k$-nearest neighbor construction algorithm which takes advantage of multiple threads. While the algorithm performs best on point sets that use integer coordinates, it is clear from experimentation that the algorithm is still viable using floating point coordinates. Further, the algorithm scales well in practice as $k$ increases, as well as for data sets that are too large to reside in internal memory. Finally, the cache efficiency of the algorithm should allow it to scale well as more processing power becomes available.

### Acknowledgements

We would like to thank Samidh Chatterjee for his comments on the paper and Sariel Har-Peled for useful discussions.

## REFERENCES


[1] M. Alexa, J. Behr, D. Cohen-Or, S. Fleishman, D. Levin, and C. T. Silva. Point set surfaces. In *VIS '01: Proceedings of the conference on Visualization '01*, pages 21–28, Washington, DC, USA, 2001. IEEE Computer Society.

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

[3] M. Bern. Approximate closest-point queries in high dimensions. *Inf. Process. Lett.*, 45(2):95–99, 1993.

[4] P. B. Callahan and S. R. Kosaraju. A decomposition of multidimensional point sets with applications to k-nearest-neighbors and n-body potential fields. *J. ACM*, 42(1):67–90, 1995.

[5] T. M. Chan. Approximate nearest neighbor queries revisited. In *SCG '97: Proceedings of the thirteenth annual symposium on Computational geometry*, pages 352–358, New York, NY, USA, 1997. ACM.

[6] T. M. Chan. Manuscript: A minimalist's implementation of an approximate nearest neighbor algorithm in fixed dimensions, 2006.

[7] J. Chhugani, B. Purnomo, S. Krishnan, J. Cohen, S. Venkatasubramanian, and D. S. Johnson. vlod: High-fidelity walkthrough of large virtual environments. *IEEE Transactions on Visualization and Computer Graphics*, 11(1):35–47, 2005.

[8] U. Clarenz, M. Rumpf, and A. Telea. Finite elements on point based surfaces. In *Proc. EG Symposium of Point Based Graphics (SPBG 2004)*, pages 201–211, 2004.

[9] K. L. Clarkson. Fast algorithms for the all nearest neighbors problem. In *FOCS '83: Proceedings of the Twenty-fourth Symposium on Foundations of Computer Science*, Tucson, AZ, November 1983. Included in PhD Thesis.

[10] K. L. Clarkson. Nearest-neighbor searching and metric space dimensions. In G. Shakhnarovich, T. Darrell, and P. Indyk, editors, *Nearest-Neighbor Methods for Learning and Vision: Theory and Practice*, pages 15–59. MIT Press, 2006.

[11] D. Cotting, T. Weyrich, M. Pauly, and M. Gross. Robust watermarking of point-sampled geometry. In *SMI '04: Proceedings of the Shape Modeling International 2004*, pages 233–242, Washington, DC, USA, 2004. IEEE Computer Society.

[12] M. T. Dickerson and D. Eppstein. Algorithms for proximity problems in higher dimensions. *Computational Geometry Theory & Applications*, 5(5):277–291, January 1996.

[13] S. Fleishman, D. Cohen-Or, and C. T. Silva. Robust moving least-squares fitting with sharp features. *ACM Trans. Graph.*, 24(3):544–552, 2005.

[14] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *FOCS '99: Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, pages 285–298, Washington, DC, USA, 1999. IEEE Computer Society.

[15] D. R. Karger and M. Ruhl. Finding nearest neighbors in growth-restricted metrics. In *STOC '02: Proceedings of the thirty-fourth annual ACM symposium on Theory of computing*, pages 741–750, New York, NY, USA, 2002. ACM.

[16] Swanwa Liao, Mario A. Lopez, and Scott T. Leutenegger. High dimensional similarity search with space filling curves. In *Proceedings of the 17th International Conference on Data Engineering*, pages 615–622, Washington, DC, USA, 2001. IEEE Computer Society.

[17] Michael D. Lieberman, Jagan Sankaranarayanan, and Hanan Samet. A fast similarity join algorithm using graphics processing units. In *ICDE '08: Proceedings of the 2008 IEEE 24th International Conference on Data Engineering*, pages 1111–1120, Washington, DC, USA, 2008. IEEE Computer Society.

[18] N. J. Mitra and A. Nguyen. Estimating surface normals in noisy point cloud data. In *SCG '03: Proceedings of the nineteenth annual symposium on Computational geometry*, pages 322–328, New York, NY, USA, 2003. ACM.

[19] G. M. Morton. A computer oriented geodetic data base and a new technique in file sequencing. In *Technical Report,IBM Ltd*, 1966.

[20] D. Mount. ANN: Library for Approximate Nearest Neighbor Searching, 1998. `http://www.cs.umd.edu/~mount/ANN/`.

[21] N. Nethercote and J. Seward. Valgrind: a framework for heavy-weight dynamic binary instrumentation. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 89–100, New York, NY, USA, 2007. ACM.

[22] J. A. Orenstein and T. H. Merrett. A class of data structures for associative searching. In *PODS '84: Proceedings of the 3rd ACM SIGACT-SIGMOD symposium on Principles of database systems*, pages 181–190, New York, NY, USA, 1984. ACM.

[23] R. Pajarola. Stream-processing points. In *Proceedings IEEE Visualization, 2005, Online.*, pages 239–246. Computer Society Press, 2005.

[24] M. Pauly, M. Gross, and L. P. Kobbelt. Efficient simplification of point-sampled surfaces. In *VIS '02: Proceedings of the conference on Visualization '02*, pages 163–170, Washington, DC, USA, 2002. IEEE Computer Society.

[25] M. Pauly, R. Keiser, L. P. Kobbelt, and M. Gross. Shape modeling with point-sampled geometry. *ACM Trans. Graph.*, 22(3):641–650, 2003.

[26] J. Sankaranarayanan, H. Samet, and A. Varshney. A fast all nearest neighbor algorithm for applications involving large point-clouds. *Comput. Graph.*, 31(2):157–174, 2007.

[27] H. Tropf and H. Herzog. Multidimensional range search in dynamically balanced trees. *Angewandte Informatik*, 2:71–77, 1981.

[28] P. Tsigas and Y. Zhang. A simple, fast parallel implementation of quicksort and its performance evaluation on SUN Enterprise 10000. *Eleventh Euromicro Conference on Parallel, Distributed and Network-Based Processing*, 00:372, 2003.

[29] P. M. Vaidya. An $O(n \log n)$ algorithm for the all-nearest-neighbors problem. *Discrete Comput. Geom.*, 4(2):101–115, 1989.


**Michael Connor** received a MS in Computer Science from Florida State University in 2007, where he is currently a PhD student. His primary research interests are Computational Geometry and Parallel Algorithms.

**Piyush Kumar** received his PhD from Stony Brook University in 2004. His primary research interests are in Algorithms. He is currently an Assistant Professor at the Department of Computer Science at Florida State University. He was awarded the FSU First Year Assistant Professor Award in 2005, the NSF CAREER Award in 2007, and the FSU Innovator Award in 2008. His research is funded by grants from NSF, FSU, and AMD.