

# Parallel Construction of $k$ -Nearest Neighbor Graphs for Point Clouds <sup>†</sup>

M. Connor<sup>1</sup> and P. Kumar<sup>1</sup>

<sup>1</sup>Department of Computer Science  
Florida State University

## 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.

Categories and Subject Descriptors (according to ACM CCS): I.3.5 [Computer Graphics]:  $k$ -NN Graphs

## 1. Introduction

In this paper we present a practical algorithm for the following problem: 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  $P = \{p_1, p_2, \dots, p_n\}$  be a point cloud in  $\mathbb{R}^d$  where  $d \leq 3$ . For each  $p_i \in P$ , let  $NN_k(p_i)$  be the  $k$  points in  $P$  closest to  $p_i$ . The  $k$ -nearest neighbor graph ( $k$ -NNG) is a graph with vertex set  $\{p_1, p_2, \dots, p_n\}$  and edge set  $E = \{(p_i, p_j) : p_i \in NN_k(p_j) \text{ or } p_j \in NN_k(p_i)\}$ . 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.

In point based graphics, computation of  $k$ -NNG forms a basic building block in solving many important problems including normal estimation [MN03], surface simplification [PGK02], finite element modeling [CRT04], shape modeling [PKKG03], watermarking [CWMG04] and surface reconstruction [ABCO\*01, FCOS05]. 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; its 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  $\mathcal{O}(n \log n + nk)$  [CK95]. 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 [MN03, Paj05, CRT04] because of the high constants involved in theoretical algorithms [Vai89, CK95, Cla83, DE96]. In low dimensions, one of the best kd-tree implementations is by Arya et.al. [AMN\*98]. 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 [MN03, PKKG03, CRT04]. 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.

The  $k$ -NNG construction problem has also been studied in the external memory setting or using the disk access model recently [SSV07]. The design and implementation of our algorithm is more tuned toward the cache-oblivious model [FLPR99] and hence differs significantly from [SSV07]. While our algorithm was not specifically designed for external memory, through the use of large amounts of swap space experiments have shown it can handle very large data sets.

<sup>†</sup> The source code associated with this paper is available from <http://compgeom.com/~stann>. Work on this paper was partially supported by the NSF CAREER award CCF-0643593.

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.

In order to take advantage of the widespread availability of multi-core machines, the algorithm was designed to be easily parallelizable and cache efficient. The first phase is implemented using parallel distribution sort. The second and third phase are run in loops where multiple cores process multiple searches simultaneously.

The remainder of the paper is organized as follows. In the next section we describe our algorithm in more detail. Section 3 describes the experimental setup we use. Section 4 presents our experimental results. Section 5 concludes the paper.

## 2. Methodology

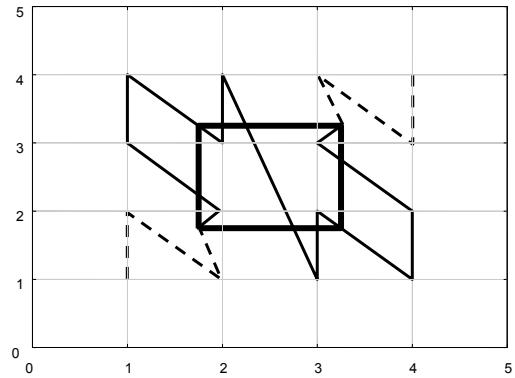
In this section we describe our algorithm in detail. Before we start the description, we need to describe the Morton order on which our algorithm is based on.

### 2.1. Morton Ordering

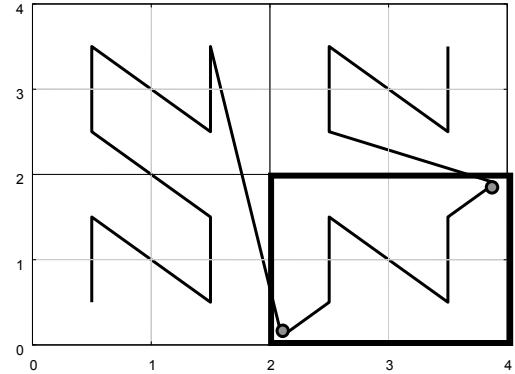
Morton-order or Z-order is a space-filling curve with good locality preserving behavior. It is often used in data structures for mapping multidimensional data to one dimension. The z-value of a point in multiple dimensions can be calculated by interleaving the binary representations of its coordinate 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 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 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.

Timothy Chan showed that the relative Morton order of two integer points can be easily calculated by determining



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



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

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 only a few binary operations [Cha06]. While this method only applies to integer types, it can be extended to floating point types as shown in Algorithm 1. The algorithm takes two points whose coordinates are a floating point type. 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 13 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 EX-

PONENT and MANTISSA functions return those parts of the floating point number in integer format. This algorithm allows the relative Morton order of floating points to be found with only a constant factor more of work.

---

**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:    $x \leftarrow 0$ ;  $dim \leftarrow 0$ 
3:   for all  $i = 0$  to  $d$  do
4:      $y \leftarrow \text{XORMSB}(p_i, q_i)$ 
5:     if  $x < y$  then
6:        $x \leftarrow y$ ;  $dim \leftarrow i$ 
7:     end if
8:   end for
9: end procedure
10: procedure XORMSB(double  $a$  , double  $b$ )
11:    $x \leftarrow \text{EXPONENT}(a)$ ;  $y \leftarrow \text{EXPONENT}(b)$ 
12:   if  $x = y$  then
13:      $z \leftarrow \text{MSDB}(\text{MANTISSA}(a), \text{MANTISSA}(b))$ 
14:      $x \leftarrow x - \text{EXPONENT}(z)$ 
15:     return  $x$ 
16:   end if
17:   if  $y < x$  then return  $x$ 
18:   else return  $y$ 
19: end procedure
```

---

## 2.2. The $k$ -NNG Construction Algorithm

The  $k$ -NNG construction algorithm proceeds as follows. The point set  $P$  is sorted according to the Morton order comparison operator. In our implementation this is done using a parallel distribution sort. An approximate nearest neighbor ball is then found for each point by scanning a small number of adjacent points in the sorted array. We then refine this approximate ball by recursively searching the implicit quad-tree. Quadtree boxes can be defined by a range of points in the sorted array. The recursive search has two stop conditions. For the first condition, the minimum enclosing quadtree box is calculated for the entire range. If this box does not intersect the current nearest neighbor ball, then no further recursion is needed within the range. For the second condition, the midpoint in the range is compared to the upper and lower corner of the box which circumscribes the current nearest neighbor ball. If the midpoint in the range lies outside the box, either the upper or lower half of that range can be eliminated.

The function CONSTRUCT, described in Algorithm 2, takes as input a point set  $P$  and an integer  $k$ , and returns a matrix  $A$ , where where each element  $a_{ij} \in A$  is the index of the  $j$ -th nearest neighbor of  $p_i \in P$ . We will indicate the  $i$ -th row of  $A$  as  $a_i$ . The function  $nn$  (Line 3, Algorithm 2) computes the distances from  $p_i \in P$  to every point in a range. The

answer  $a_i$  is updated as needed. The function  $r(p_i)$  (Line 13) returns the maximum distance of  $p_i \in P$  to the elements of  $a_i$ . The recursive function CSearch is based on the space filling curve algorithm described by Chan [Cha06, LL]. It refines an approximate nearest neighbor solution for a point  $p_i \in P$  to an exact answer. Lines 7 to 10 take advantage of the fact that for small ranges, it is more efficient to do a linear scan of the points instead of recursing. The constant  $v$  depends on the cache line length of the system using the algorithm, and should be calculated as the maximum number of points that reside in one cache line. For large ranges, first the middle point in the range is compared to the current nearest neighbor ball and added to the answer queue if needed. This corresponds to line 12 in Algorithm 2. Finally, the two previously mentioned stop conditions are checked in lines 13 to 22 in Algorithm 2. Function  $B(i, j)$  computes the minimum enclosing quadtree box for two points  $i$  and  $j$ . Since the quadtree box divisions occur on powers of 2, the quadtree box is determined by  $s$ , the place of the highest order differing bit between the coordinate pairs of  $i$  and  $j$ . In each dimension, the two edges of the quadtree box are located at position  $2^s$  and  $2^{s+1}$ . Function  $d$  computes the distance from point  $q$  to  $B(i, j)$  as the sum of the square distances between the coordinates of  $q$  and the nearest edge of the quadtree box.

---

**Algorithm 2** KNN Graph Construction Algorithm

---

**Require:** Point set  $P$  of size  $n$ . Morton order compare operator COMPARE  
**Ensure:** **for all**  $p_i$  in  $P$ ,  $a_i$  contains  $k$  points from  $P$  with minimum distance to  $p_i$

```

1: procedure CONSTRUCT( $P$  , int  $k$ )
2:    $P \leftarrow \text{Sort}(P, \text{COMPARE})$ 
3:   for all  $p_i$  in  $P$  do:  $a_i \leftarrow nn(p_i, p_{i-k \dots i+k})$ 
4:   for all  $p_i$  in  $P$  do: CSEARCH( $p_i, 1, n$ )
5: end procedure
6: procedure CSEARCH(point  $p_i$  , int  $l$  , int  $h$ )
7:   if  $(h - l) < v$  then
8:      $a_i \leftarrow nn(p_i, p_l \dots p_h)$ 
9:     return
10:   end if
11:    $m \leftarrow (h + l)/2$ 
12:    $a_i \leftarrow nn(p_i, p_m)$ 
13:   if  $d(p_i, B(p_l, p_h)) \geq r(p_i)$  then return
14:   if COMPARE( $p_i, p_m$ ) then
15:     CSEARCH( $p_i, l, m - 1$ )
16:     if COMPARE( $p_i^{[r(p_i)]}, p_m$ )
17:       then CSEARCH( $p_i, m + 1, h$ )
18:     else
19:       CSEARCH( $p_i, m + 1, h$ )
20:       if COMPARE( $p_i^{-[r(p_i)]}, p_m$ )
21:         then CSEARCH( $p_i, l, m - 1$ )
22:     end if
23:   end procedure
```

---

### 2.3. Parallel Construction

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

### 2.4. 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 [SSV07]. 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 [SSV07] directly (their code was not available). However, we were able to calculate  $k$ -NNG for much larger data sets (up to 182 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 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 processor, 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.

## 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.compgem.com/~stann>.

The system that we experimented on is equipped with 2 Quad-core 2.66Ghz Intel Xeon CPUs, and a total of 4GB of DDR memory. Each core has 2MB of total cache. SUSE Linux with kernel 2.6.22.17-0.1-default was running on the system. We used `gcc` version 4.2.1 for compilation of all our code (with `-O3`). The machine has a 150GB Western digital Raptor drive on which we created a temporary swap partition of 40GB.

ANN [Mou98] 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 with this preprocessing time removed, as well as examples where only one processor was used.

Except where noted, random data sets were generated from 3-dimensional points uniformly distributed between  $(0, 1]$ , and scaled to a  $(0, 2^{24})$  integer grid. 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’. A hyphenated entry indicates the number of threads that were used.

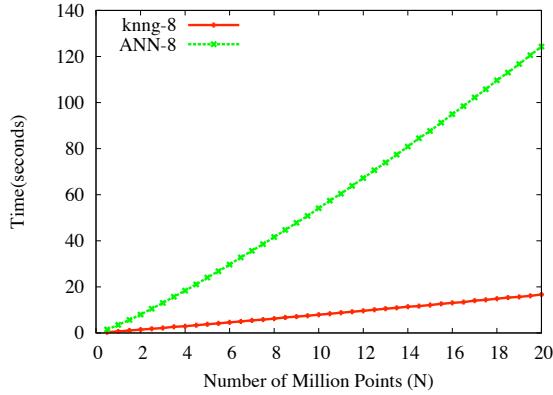
Dataset	Size (points)	ANN (s)	STANN (s)
Screw	27152	.06	.04
Dinosaur	56194	.11	.07
Ball	137602	.31	.14
Isis	187644	.46	.18
Blade	861240	2.9	.86
Awakening	2057930	8.6	2.1
David	3614098	16.7	3.7
Night	11050083	62.2	12.4

**Table 1:** Construction times for  $k = 1$ -nearest neighbor graphs constructed on non-random 3-dimensional data sets with integer coordinates. Each graph was constructed using 8 processors. All timings are in seconds.

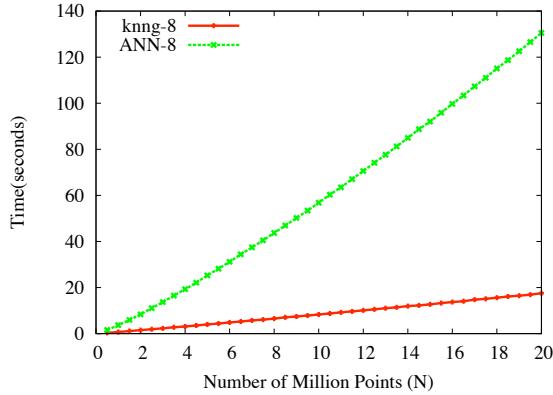
## 4. Experimental Results

### 4.1. Construction Time Results

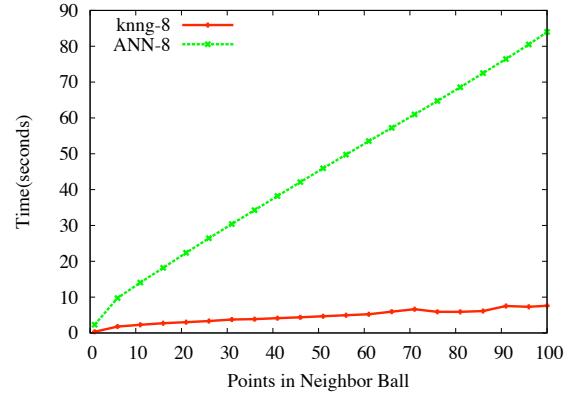
As shown in Figure 3 and Table 1, our algorithm performs very favorably against  $k$ -NNG construction using ANN. Figure 4 is an example of construction time for a random, non-uniform distribution. Other distributions were also similar to these cases, and so more graphs were not included. Figure 5 shows that as  $k$  increases, the advantage runs increasingly toward our implementation. Figure 6 shows the performance with points using 64 bit doubles for coordinates.



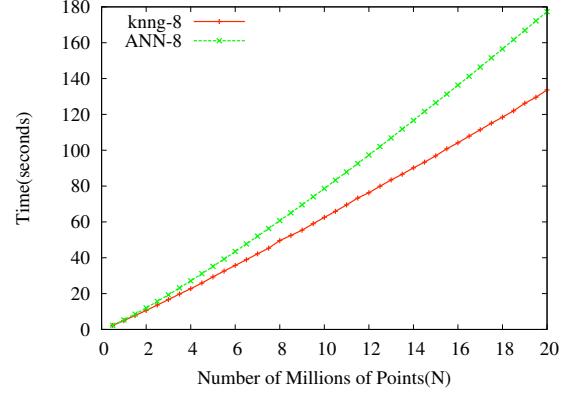
**Figure 3:** Graph of 1-NN graph construction time vs. number of data points. Each algorithms were run using 8 processors in parallel. Data files were uniformly random 3-dimensional data sets with integer coordinates.



**Figure 4:** Graph of 1-NN graph construction time vs. number of data points. Each algorithms were run using 8 processors in parallel. Data files were 3-dimensional random with gaussian distribution and integer coordinates.



**Figure 5:** Graph of  $k$ -NN graph construction time for varying  $k$ . Each algorithms were run using 8 processors in parallel. Data files were uniformly random 3-dimensional data sets with integer coordinates.



**Figure 6:** Graph of 1-NN graph construction time vs. number of data points, with the preprocessing time removed. Each algorithms were run using 1 processor. Data files were 3-dimensional random with uniform distribution and doubles for coordinates. The graph using floats for coordinates is similar to this graph.

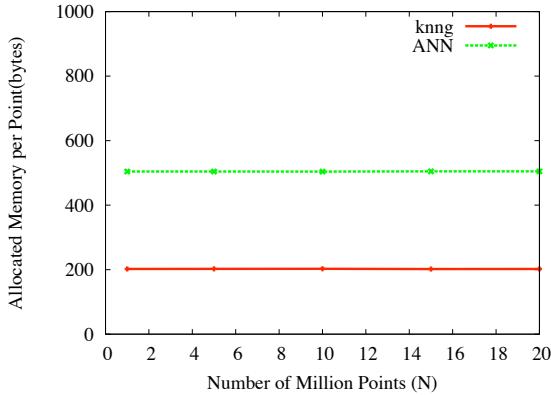
#### 4.2. Memory Usage and Cache Efficiency Results

Figure 7 shows the estimated memory usage, per point, for both algorithms. Memory usage was determined using valgrind [NS07]. As shown in Figure 8, our algorithm has greater cache efficiency than ANN's kd-tree implementation. This implies that we will derive greater benefit from more available processing power. This is demonstrated in Table 2. Construction was done using one data set, while varying the number of processors. In the first pair of columns we see the improvement in total construction time as the number of processors is varied. We can see that moving from one to eight processors improves our construction time by 68%, while only improving ANN by 30%. Since ANN's kd-tree construction is not done in parallel, if we remove prepro-

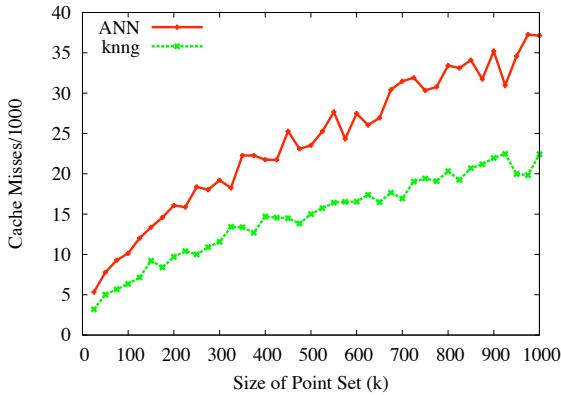
cessing time and only time the actual queries, we see in the second pair of columns that ANN's query time is improved by 71%, while ours is improved by 88%. Figure 9 shows this behavior for data sets of varying size.

#### 5. Conclusions

We have presented an efficient  $k$ -nearest neighbor construction algorithm which takes advantage of multiple processors. 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 behaves nicely in practice as  $k$  increases,



**Figure 7:** Graph of memory usage per point vs. data size. All data sets were uniformly random 3-dimensional data sets with integer coordinates. Memory usage was determined using valgrind.

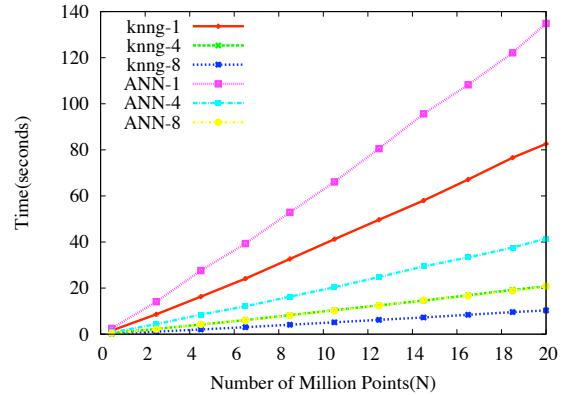


**Figure 8:** Graph of cache misses vs. data set size. All data sets were uniformly random 3-dimensional data sets. Cache misses were determined using valgrind which simulated a 2MB L1 cache.

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.

## References

- [ABCO\*01] ALEXA M., BEHR J., COHEN-OR D., FLEISHMAN S., LEVIN D., SILVA C. T.: Point set surfaces. In *VIS '01: Proceedings of the conference on Visualization '01* (Washington, DC, USA, 2001), IEEE Computer Society, pp. 21–28.
- [AMN\*98] ARYA S., MOUNT D. M., NETANYAHU N. S., SILVERMAN R., WU A.: An optimal algorithm for approximate nearest neighbor searching in fixed dimensions. *J. ACM* 45 (1998), 891–923.
- [Cha06] CHAN T. M.: Manuscript: A minimalist’s implementation of an approximate nearest neighbor algorithm in fixed dimensions, 2006.
- [CK95] CALLAHAN P. B., KOSARAJU S. R.: A decomposition of multidimensional point sets with applications to k-nearest-neighbors and n-body potential fields. *J. ACM* 42, 1 (1995), 67–90.
- [Cla83] CLARKSON K. L.: 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.



**Figure 9:** Graph of 1-NN graph construction time vs. number of data points, with the preprocessing time removed. Graphs are listed by algorithm name and the number of processors used. Note that the knng-4 and ANN-8 graphs lie on top of each other. Data files were uniformly random 3-dimensional data sets with integer coordinates.

Processors	ANN		STANN	
	(s)	(s)	(s)	(s)
1	88.0	28.4	36.1	21.7
2	71.0	18.0	24.1	10.9
4	64.0	11.6	16.0	5.4
8	61.0	8.9	10.6	2.7

**Table 2:** Construction time for 1-nearest neighbor graph of the Night data set using varying number of processors. The first pair of columns are total construction time, the second pair have preprocessing time (which ANN does not do in parallel) removed.

- [CRT04] CLARENZ U., RUMPF M., TELEA A.: Finite elements on point based surfaces. In *Proc. EG Symposium of Point Based Graphics (SPBG 2004)* (2004).
- [CWMG04] COTTING D., WEYRICH T., M.PAULY, GROSS M.: Robust watermarking of point-sampled geometry. In *SMI '04: Proceedings of the Shape Modeling International 2004* (Washington, DC, USA, 2004), IEEE Computer Society, pp. 233–242.
- [DE96] DICKERSON M. T., EPPSTEIN D.: Algorithms for proximity problems in higher dimensions. *Computational Geometry Theory & Applications* 5, 5 (January 1996), 277–291.
- [FCOS05] FLEISHMAN S., COHEN-OR D., SILVA C. T.: Robust moving least-squares fitting with sharp features. *ACM Trans. Graph.* 24, 3 (2005), 544–552.
- [FLPR99] FRIGO M., LEISERSON C. E., PROKOP H., RAMACHANDRAN S.: Cache-oblivious algorithms. In *FOCS '99: Proceedings of the 40th Annual Symposium on Foundations of Computer Science* (Washington, DC, USA, 1999), IEEE Computer Society, p. 285.
- [LL] LOPEZ M. A., LIAO S.: 197 session c6.1 12th canadian conference on computational geometry finding k-closest-pairs efficiently for high dimensional data.
- [MN03] MITRA N. J., NGUYEN A.: Estimating surface normals in noisy point cloud data. In *SCG '03: Proceedings of the nineteenth annual symposium on Computational geometry* (New York, NY, USA, 2003), ACM, pp. 322–328.
- [Mou98] MOUNT D.: ANN: Library for Approximate Nearest Neighbor Searching, 1998. <http://www.cs.umd.edu/~mount/ANN/>.
- [NS07] NETHERCOTE N., SEWARD J.: Valgrind: a framework for heavyweight dynamic binary instrumentation. In *In PLDI* (2007), pp. 89–100.
- [Paj05] PAJAROLA R.: Stream-processing points. In *Proceedings IEEE Visualization, 2005, Online.* (2005), Computer Society Press.
- [PGK02] PAULY M., GROSS M., KOBBELT L. P.: Efficient simplification of point-sampled surfaces. In *VIS '02: Proceedings of the conference on Visualization '02* (Washington, DC, USA, 2002), IEEE Computer Society, pp. 163–170.
- [PKKG03] PAULY M., KEISER R., KOBBELT L. P., GROSS M.: Shape modeling with point-sampled geometry. *ACM Trans. Graph.* 22, 3 (2003), 641–650.
- [SSV07] SANKARANARAYANAN J., SAMET H., VARSHNEY A.: A fast all nearest neighbor algorithm for applications involving large point-clouds. *Comput. Graph.* 31, 2 (2007), 157–174.
- [Vai89] VAIDYA P. M.: An o(n log n) algorithm for the all-nearest-neighbors problem. *Discrete Comput. Geom.* 4, 2 (1989), 101–115.