Accelerating kd-tree searches for all k-nearest neighbours

Bruce Merry, James Gain and Patrick Marais

January 2013

Abstract

Finding the k nearest neighbours of each point in a point cloud forms an integral part of many point-cloud processing tasks. One common approach is to build a kd-tree over the points and then iteratively query the k nearest neighbors of each point. We introduce a simple modification to the queries to exploit the coherence between successive points; no changes are required to the kd-tree data structure. The path from the root to the appropriate leaf is updated incrementally, and backtracking is done bottom-up. We show that this can reduce the time to compute the neighbourhood graph of a 3D point cloud by over 10%, and by up to 24% when $k = 1$. The gains scale with the depth of the kd-tree, and the method is suitable for parallel implementation.

1 Introduction

The "all k-nearest neighbours problem" (AKNN) can be defined as follows: given a set of points in space and a number k, find the k nearest points to each of the given points. It is a special case of the k-nearest neighbours (KNN) problem, where the input point cloud is also the set of query points. AKNN is a standard tool in point-cloud processing tasks, including density estimation, normal estimation, smoothing, surface reconstruction and others [Connor and Kumar, 2010]. The brute force approach takes $O(dN^2)$ time in d dimensions, which is prohibitively expensive for typical point clouds. Even when using more efficient algorithms, it is computationally intensive, and often dominates the execution time of point-cloud processing tasks [Sankaranarayanan et al., 2007]. We review previous work on the AKNN problem in Section 3.

One popular data structure for accelerating spatial queries is the kd-tree [Bentley, 1975], which is a multi-dimensional extension of a binary tree. Section 2 describes kd-trees in more detail, along with our implementation. A kd-tree can be used to solve the AKNN problem by making N independent KNN queries. Our contribution, presented in Section 4, is a simple modification to this approach. We process the points in a spatially-coherent order, which allows some information computed in each query to be re-used for the following query. The modification is general and can be combined with other variations of the problem, such as finding approximate nearest neighbours. The results show a significant reduction in the total time.

In section 5 we show that our algorithm is suitable for parallel execution on modern multi-core CPUs, and additionally describe how we build the kd-tree in parallel.

2 kd-Trees

A kd-tree is a tree structure where each node corresponds to a *rectangle*: in d-dimensional space, a rectangle is the product of d closed intervals on the coordinate axes. Each internal node has an axis-aligned hyperplane that splits the rectangle; the two sub-rectangles thus formed are associated with the two child nodes. Each point in a point cloud is stored in a leaf whose rectangle contains it. The set of points in a leaf is also known as a *bucket*, and the bucket size is normally bounded by some small constant. Figure 1 shows an example of a kd-tree in two dimensions. As there is a large body of literature on kd-trees, we will not attempt to review it here. The interested reader is referred to Elseberg et al. [2012] for a comparison of several kd-tree implementations.

A kd-tree can be used to accelerate k-nearest neighbour queries Friedman et al. [1977], using ball-rectangle intersection tests. Given a query point p and k candidate neighbours, we can be sure that the true k-neighbourhood will be found inside a ball centred on p and passing through the current kth-nearest candidate. When searching for better candidates, a node which does not intersect this ball can be skipped without considering any of its children. Figure 1 shows a 2D example, where $k = 2$. While searching for a neighbour for p, we have identified n_1 and n_2 as candidates. We can ignore any points that lie outside the circle shown,

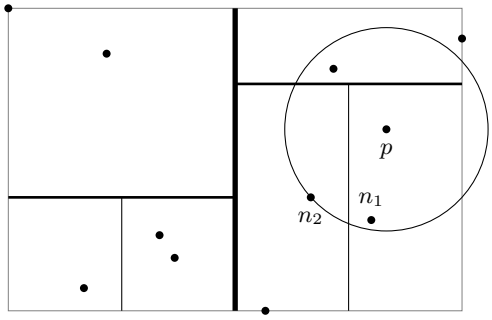

Figure 1: Search for neighbours in a kd-tree.

allowing the left subtree of the root to be pruned.

We first describe how kd-trees are built in Section 2.1 and return to query algorithms in Section 2.2.

2.1 Constructing the tree

A kd-tree is constructed recursively, by repeatedly splitting a node into two children and distributing corresponding points to the appropriate children. A splitting rule determines the axis and position of the splitting plane for each internal node. We have used the *sliding midpoint* rule, which has good theoretical and practical performance for nearest-neighbour searches [Maneewongvatana and Mount, 1999]. The splitting hyperplane is aligned perpendicular to the longest side of the rectangle. It is positioned to split the rectangle into two equally-sized sub-rectangles, unless all the points would then fall on the same side of the split. In this case, the splitting hyperplane "slides" the minimum distance to ensure that both sub-rectangles contain points.

Another design choice is the representation of nodes in memory. Elseberg et al. [2012] have obtained high performance in their kd-tree library (`libnabo`) by keeping the nodes as small as possible, to minimize pressure on the memory hierarchy. We have closely followed their design: each internal node is represented in 8 bytes, with $\lfloor \log_2 d \rfloor + 1$ bits to indicate the split axis, 32 bits to store the split coordinate, and the remaining bits to store the index of the right child. Leaf nodes also contain 8 bytes, with the split axis replaced by a sentinel value to indicate a leaf, the split coordinate replaced by the index of the first point in the bucket, and the child index replaced by the bucket size. The nodes are stored as a flat array, ordered by a pre-order walk (see Figure 2), so it is not necessary to store a pointer to the left child as it will always be adjacent in memory.

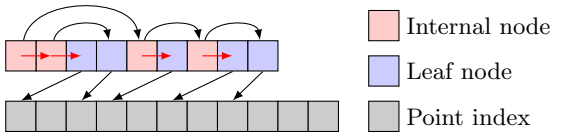

Figure 2: A flattened kd-tree. The red arrows are implicit pointers to the left child, while other arrows are explicitly encoded array indices. Each leaf node additionally encodes the size of its bucket.

Our implementation diverges from `libnabo` in a few respects. We fix the dimension at compile time, which removes dynamic memory overheads; we eliminate a redundant pointer from each bucket entry to the coordinates for the corresponding point; and we set the maximum bucket size to 16 (rather than 8), as we found that this gave slightly better performance across a range of neighbourhood sizes. Presumably as a result of these changes, our implementation out-performs `libnabo` in 3D.

2.2 Finding nearest neighbours

Arya and Mount [1993b] introduced a number of refinements to accelerate nearest neighbour searches. One that we take advantage of is *incremental distance computation*, which reduces the number of operations needed to compute the minimum distance between the query point and the rectangles, particularly in high dimensions. While processing a node N, they store the minimum squared distance between p and N, as well the portion of this squared distance along each axis. When moving to the children of N, there are two cases. For the closer child N_1, the closest point to p is the same as for N, and so no update is required. For the more distant child N_2, only the distance along the split axis changes, so the total squared distance d can be updated by subtracting the old value along this axis and adding the new value, as shown in Algorithm 1.

Arya and Mount also describe two tree traversal orders. The *standard* search is recursive, with the closer subtree visited prior to visiting the more distant subtree. The *priority* search visits leaves in increasing order of distance from p, but incurs additional overhead in maintaining a priority queue. We briefly experimented with priority search but found that the overhead exceeded the gains from examining fewer leaves, so we have used standard search instead.

While searching for k nearest neighbours with

Algorithm 1: `FindKNN` with incremental distance computation

Input: Query point p
Input: Subtree root node N
Input: Per-axis squared distances a
Input: $d = \sum a$
Input: Squared distance to the kth-nearest candidate D
if N *is a leaf* **then**
  foreach *point* q *in* N **do**
    if $\|p - q\|^2 < D$ **then**
      Add q as a candidate;
      Update D;
    end
  end
else
  Let N_1, N_2 be children of N, with N_1 closer to p;
  `FindKNN` (p, N_1, a, d, D);
  $u \leftarrow (p_{N.\mathsf{axis}} - N.\mathsf{split})^2$;
  $d \leftarrow d - a_{N.\mathsf{axis}} + u$;
  $a_{N.\mathsf{axis}} \leftarrow u$;
  if $d < D$ **then**
    `FindKNN`(p, N_2, a, d, D);
  end
end

$k > 1$, it is also necessary to have some data structure for maintaining the k best candidates encountered. We have used just a simple array: Elseberg et al. [2012] report that this is faster for k up to about 30, which is the typical range for many point-cloud processing tasks.

3 Related work

In this section we will focus on the AKNN problem, as the whole field of nearest-neighbour techniques is too broad to be reviewed here. However, we will mention one technique that forms the basis for our contribution. Nüchter et al. [2007] use kd-trees to answer nearest-neighbour queries in the Iterated Closest Point (ICP) algorithm. They note that each query is typically quite close to the corresponding query from the previous iteration of the algorithm. To exploit this, they modify the query procedure to return both the closest point and the leaf node that contains it. On the next query, the search is started in this leaf node, and backtracking is implemented explicitly using pointers in the tree to parent nodes, rather than top-down using recursion.

Connor and Kumar [2010] sort the input points along a space-filling curve, which places points close to many of their neighbours. For each point, they obtain a candidate neighbourhood by testing $O(k)$ elements to either side in the sorted list. The candidate is then refined by finding a conservative range to search and recursively subdividing it, pruning sub-ranges when they provably contain no nearest neighbours. Their implementation also parallelises the queries.

The AKNN problem has also been studied in the context of databases, where it is generalised to the "k-nearest neighbour join": for each point in one set, find the k nearest neighbours in another set. Böhm and Krebs [2004] use an R-tree-like structure, and they find the neighbourhoods of all points from a disk page concurrently. Heuristics are used to prioritise pairs of buckets to be examined. Xia et al. [2004] also consider the problem from the perspective of I/O scheduling, but use principal component analysis to project high-dimensional data into a lower-dimension space, which is divided into a grid of blocks and sorted lexicographically. As before, the neighbourhoods for all the points in one block are searched concon-currently.

Although these algorithms are optimized to answer multiple queries in terms of I/O scheduling and pruning, each neighbourhood is still computed independently. Sankaranarayanan et al. [2007] improve the search by using the neighbourhood of the previous point as an initial candidate for the neighbourhood of the current point. This gives an upper bound on the search radius, and they further modify the search to avoid re-searching the space occupied by the previous neighbourhood. This idea of using information from the previous point is similar in principle to our technique, although we exploit the coherence in a different way.

4 Exploiting coherence

Once a kd-tree has been constructed, a naïve approach to solving the all k-nearest neighbours problem is to perform an independent search for each point against the tree. This is sub-optimal, because it discards information determined for one point which can be reused for nearby points. Our approach is based on the work of Nüchter et al. [2007], but adapted to the all k-nearest neighbours problem. In particular, our approach does not store parent pointers in the kd-tree.

Nüchter et al. exploit coherence in time: a query is moved some distance by an iteration of the ICP algorithm. We instead exploit coherence in space:

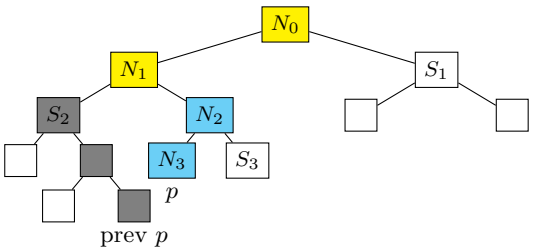

Figure 3: Primary path update. The gray nodes belong only to the old primary path, and are popped (bottom-up). The blue nodes are then pushed (top-down) until reaching N_m.

we perform the queries in an order that maximises spatial coherence. Fortunately, the kd-tree construction has already computed such an order, namely the order the points are encountered in a recursive walk of the tree. In our implementation the buckets are stored in exactly this order, so it is possible to do this walk iteratively rather than recursively.

Given a query point p, let $P = \{N_0, \ldots, N_m\}$ be the path through the kd-tree from the root (N_0) to the leaf containing p (N_m), as shown in Figure 3. We will call this the *primary path*. Let S_i be the sibling of N_i. If we expand the recursive calls from Algorithm 1 that follow the primary path, we find that we first visit N_m and then the subtrees rooted at $S_m, S_{m-1}, \ldots, S_1$. This corresponds to lines 10–21 in Algorithm 2.

We refine this approach in two ways. Firstly, we need to actually compute the primary path, which we do incrementally starting with the primary path for the previous query. We pop nodes that do not contain p until we are left with a prefix of the primary path, and then we complete this partial path by walking down the tree as usual (Figure 3). This is shown in lines 1–8. Since each node is pushed once and popped once, this takes amortized $O(1)$ time per query. The second refinement is that we can exit the loop early if the rectangle associated with node N_i completely contains the ball centred at p and passing through the kth-nearest candidate — the so-called "ball-within-bounds" test [Friedman et al., 1977] (line 13).

The ball-within-bounds test could also be applied to the naïve search. We found that applying it there reduced performance, while it increased performance in our incremental implementation. The test itself is applied exactly as often, so we assume the difference is that each naïve query computes the rectangles top-down from the root to the leaf, rather than bottom-up only to the point where the test succeeds.

Algorithm 2: Bottom-up backtracking. Refer to 1 for the definitions of `FindKNN`, a, d and D.

Input: Query point p
Input: Leaf L containing p
Input: Primary path P of some point
Output: Neighbourhood of p
Output: Primary path of p

```
1  while L is not a descendant of P.back do
2  |  P.pop();
3  end
4  while P.back ≠ L do
5  |  Find child C of P.back containing L;
6  |  Compute rectangle and node range of C;
7  |  P.push(C);
8  end
   // P now the primary path of p
9  a ← 0;
10 FindKNN(P.back, a, 0);
11 foreach N_i in P except the root do //
   bottom-up
12 |  if N_i completely contains candidate ball
   |  then
13 |  |  break;
14 |  end
15 |  axis ← N_{i-1}.axis;
16 |  a_axis ← (N_{i-1}.split − p_axis)^2;
17 |  d ← a_axis if d < D then
18 |  |  FindKNN(s(N_i), a, d);
19 |  end
20 |  a_axis ← 0;
21 end
```

To implement these operations efficiently, we need to associate some extra fields with each node: the range of node indices for the descendants of the node, and the rectangle corresponding to the node. It is not necessary to store these fields in the kd-tree itself: they are maintained only for the primary path.

5 Parallel execution

Modern CPUs typically contain multiple cores, and parallel programming is important in obtaining high performance. If our modified search procedure were unsuitable for parallel implementation it would be of limited practical value. We now show that our algorithm can very easily be adapted for parallel execution on a multi-core CPU. We have not considered massively parallel processors such as GPUs: in principle the same methods can

be used, but we have not measured the impact on performance.

Our modification introduces a serial dependency, because each query uses information from the previous query. However, this information is only used as a hint for acceleration, and should it be expedient we can find the neighbourhood of a point with no prior information. To break the serial dependencies, we divide the nodes into equally-sized sequential chunks, and process all the points in each chunk serially. The first point in a chunk is processed with no prior information, while the rest are processed incrementally. Chunks do not always take a uniform amount of time to process, so we use OpenMP [OpenMP Architecture Review Board, 2008] scheduling for load-balancing. We found that a chunk size of 1024 gives good load-balancing without excessive scheduling overheads.

If only the k-nearest neighbour search is optimised, the total running time could come to be dominated by the time to construct the kd-tree, particularly for small values of k. To avoid this, we also parallelise the construction of the kd-tree. Since each subtree is built independently, this should be simple to parallelise. Unfortunately, the way nodes are stored in a flat array introduces a serial dependency, as one needs to know the number of nodes in the left subtree to determine where to place nodes from the right subtree.

To facilitate parallelisation, we build the tree in phases. In the first phase, we build a "hybrid" tree where the top few layers use a different node representation, labelled "super-nodes" in Figure 4. The construction is recursive. To process a subtree, we pick a splitting plane, then partition the points into the left and right sides. If the tree contains at least $C = \frac{N}{256}$ points then its root becomes a super-node with pointers to its children, and the children are generated in parallel. Since the children are written to separately allocated arrays, there are no conflicts. On the other hand, if the tree contains fewer than C points then it is generated into a flat array, and the children are processed serially. The choice of C is design to provide sufficient parallelism to saturate the CPU without creating an excessive number of super-nodes.

After the hybrid tree is constructed, we make additional passes to flatten it into a single array. First, we recursively (and in parallel) compute the size of each subtree. Next, we use these sizes to determine where each of the separate flat arrays shown in Figure 4 will be placed within the final array. Finally, we copy the nodes from the hybrid tree into this array.

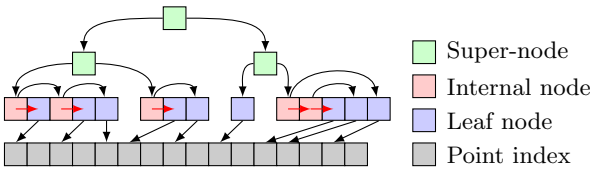

Figure 4: Hybrid kd-tree. Each green *super-node* is allocated separately. Each connected piece of internal and leaf nodes is a separate array that is produced as a serial task.

6 Results

6.1 Experimental setup

Experiments were carried out on an Intel Core i7-2600 (4 cores, 3.4 GHz) with 16 GiB of RAM running Ubuntu 12.04. The code was written in C++ and compiled with GCC 4.6.

Table 1 lists the data sets we used. The data sets marked with a (*) are range-scanned point clouds, while the others are polygon meshes from which only the vertices are used. Although the latter are not necessarily representative of a typical point cloud, we have used them to facilitate comparison with previous work [Sankaranarayanan et al., 2007, Connor and Kumar, 2010].

6.2 Performance comparison

Table 1 shows the reduction in total time (including time taken to build the tree) due to our modified search. For comparison, we also show the time taken by the integer version of STANN 0.74 [Connor and Kumar, 2010]. STANN was not able to process the Pisa data set as our test machine has insufficient virtual memory.

Figure 5 shows the improvement against the number of vertices. In general, larger point clouds benefit more. Large clouds have deeper kd-trees, and so gain the most from eliminating the top-down computation of the primary path. It is also interesting to note that the Armadillo data set, which is far noisier than the other data sets, benefits more than might be expected.

We have not implemented the scheme of Sankaranarayanan et al. [2007], but a comparison with their reported results suggest that their scheme is not competitive for in-core use. For example, they report 2657.9 seconds to compute 8-neighborhoods for the Lucy model on a quad-CPU system, of which only about 10 seconds is I/O time.

Table 1: Data sets and search times with $k = 8$ and 8 threads. The data sets marked with a (*) are range-scanned point clouds, while the other models are reconstructed meshes with the connectivity information removed. *Build* is the time to construct the kd-tree. *Naïve* and *Backtrack* are the times for independent queries and for our method respectively, and *Reduction* is the difference between them. Values in parentheses exclude the build time. *STANN* is the time taken by the STANN library [Connor and Kumar, 2010]. The Armadillo data contains a significant amount of noise and background, while the other range-scanned data sets contain clean data. We also dropped scans from the Armadillo data set that had no registration information.

Data set	Points ($\times 10^6$)	Build (s)	Naïve (s)	Backtrack (s)	Reduction (%)	STANN (s)
Bunny (*)	0.36	0.02	0.10 (0.08)	0.09 (0.08)	4.6 (5.4)	0.82
Happy Buddha	0.54	0.02	0.14 (0.11)	0.13 (0.10)	6.3 (7.8)	0.97
Turbine Blade	0.88	0.04	0.23 (0.19)	0.21 (0.17)	6.2 (7.5)	1.94
Armadillo (*)	2.93	0.20	0.98 (0.78)	0.88 (0.69)	9.4 (11.7)	6.65
David (2mm)	3.61	0.20	0.99 (0.78)	0.90 (0.70)	8.6 (10.9)	6.89
Lucy	14.03	0.85	3.90 (3.05)	3.47 (2.62)	10.9 (13.9)	27.23
David (1mm)	28.18	1.60	7.67 (6.07)	6.78 (5.18)	11.6 (14.7)	56.42
Pisa (*)	157.43	10.02	48.07 (38.05)	41.81 (31.80)	13.0 (16.4)	—

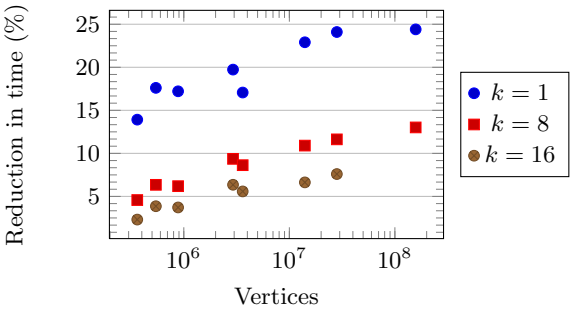

Figure 5: Improvement over the naïve implementation, for the models listed in Table 1 (including build time). Pisa with $k = 16$ is not shown because the neighbourhood graph exceeds the available memory.

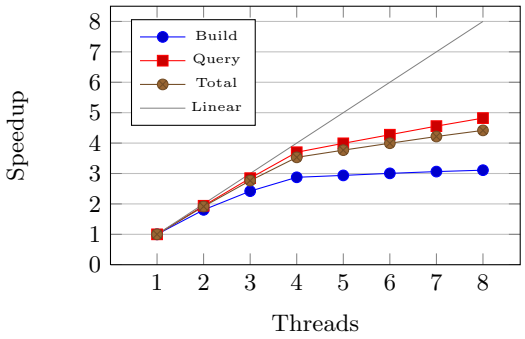

Figure 6: Speedup with increasing number of threads, for the David model with $k = 8$. The results are almost identical for other models and other values of k. With larger values of k the queries take a higher proportion of total time and so overall speedup is slightly improved.

6.3 Parallel performance

Figure 6 shows the effectiveness of our parallelisation. The kd-tree construction is accelerated, but the speedup is sub-linear, achieving less than 3× speedup with 4 threads. However, it is the k-nearest neighbour searches that dominate running time, and here we achieve close to linear speedup with 4 threads. Since our CPU has 8 hardware threads but only 4 cores, the sub-linear speedup with more threads is as expected.

7 Conclusions

We have presented a modification to a standard kd-tree search that accelerates queries in the all

k-nearest neighbours problem, with the greatest improvements when k is small and the point cloud is large. The modification is very simple to implement: Algorithm 2 hides very little detail. It also requires no modifications to the tree structure itself, so it can be implemented on top of an existing kd-tree library provided that the library exposes the internal data structures. There are a number of variations of nearest-neighbour searches that we have not discussed: searching for approximate nearest neighbours [Arya and Mount, 1993a], searches within a radius bound, non-Euclidean metrics, user-provided predicates to exclude certain neighbours, and so on. Provided that these do not modify the search order, they should trivially integrate with our modification. Our implementation also supports higher dimensions, but the impact of dimension on performance is unknown.

We have also shown that in spite of the incremental nature of the algorithm, it is straightforward to parallelise the queries, and we achieve nearly linear speedup with four cores.

8 Acknowledgements

The data sets are courtesy of the Stanford 3D scanning repository, the Georgia Institute of Technology Large Geometric Models Archive and the Digital Michelangelo project. Funding was provided by the South African Centre for High Performance Computing.

References

Sunil Arya and David M. Mount. Approximate nearest neighbor queries in fixed dimensions. In *Proceedings of the fourth annual ACM-SIAM Symposium on Discrete algorithms*, SODA '93, pages 271–280, Philadelphia, PA, USA, 1993a. Society for Industrial and Applied Mathematics. ISBN 0-89871-313-7.

Sunil Arya and David M. Mount. Algorithms for fast vector quantization. In *Data Compression Conference, 1993. DCC '93.*, pages 381–390, 1993b.

Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, Sep 1975. ISSN 0001-0782.

Christian Böhm and Florian Krebs. The k-nearest neighbour join: Turbo charging the KDD process. *Knowl. Inf. Syst.*, 6(6):728–749, November 2004. ISSN 0219-1377.

M. Connor and P. Kumar. Fast construction of k-nearest neighbor graphs for point clouds. *IEEE Transactions on Visualization and Computer Graphics*, 16(4):599–608, July–Aug 2010. ISSN 1077-2626.

Jan Elseberg, Stéphane Magnenat, Roland Siegwart, and Andreas Nüchter. Comparison of nearest-neighbor-search strategies and implementations for efficient shape registration. *Journal of Software Engineering for Robotics*, 3(1), Feb 2012.

Jerome H. Friedman, Jon Louis Bentley, and Raphael Ari Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Trans. Math. Softw.*, 3(3):209–226, Sep 1977. ISSN 0098-3500.

Songrit Maneewongvatana and David M. Mount. It's okay to be skinny, if your friends are fat. In *Center for Geometric Computing 4th Annual Workshop on Computational Geometry*, 1999.

Andreas Nüchter, Kai Lingemann, and Joachim Hertzberg. Cached k-d tree search for ICP algorithms. In *Proceedings of the 6th IEEE International Conference on Recent Advances in 3D Digital Imaging and Modeling (3DIM '07)*, pages 419–426. IEEE Computer Society Press, August 2007. ISBN ISBN 0-7695-2939-9.

OpenMP Architecture Review Board. OpenMP application program interface version 3.0, May 2008. `http://www.openmp.org/mp-documents/spec30.pdf`.

Jagan Sankaranarayanan, Hanan Samet, and Amitabh Varshney. A fast all nearest neighbor algorithm for applications involving large point-clouds. *Comput. Graph.*, 31(2):157–174, April 2007. ISSN 0097-8493.

Chenyi Xia, Hongjun Lu, Beng Chin Ooi, and Jing Hu. GORDER: an efficient method for KNN join processing. In *Proceedings of the Thirtieth international conference on Very large data bases*, VLDB '04, pages 756–767. VLDB Endowment, 2004. ISBN 0-12-088469-0.