EG UK Theory and Practice of Computer Graphics (2012)
Hamish Carr and Silvester Czanner (Editors)

Fast scalable k-NN computation for very large point clouds

S.Spina[1] and K.Debattista[1] and K.Bugeja[1] and A.Chalmers[1]

[1]International Digital Lab, University of Warwick

Abstract

The process of reconstructing virtual representations of large real-world sites is traditionally carried out through the use of laser scanning technology. Recent advances in these technologies led to improvements in precision and accuracy and higher sampling rates. State of the art laser scanners are capable of acquiring around a million points per second, generating enormous point cloud data sets. These data sets are usually cleaned through the application of numerous post-processing algorithms, like normal determination, clustering and noise removal. A common factor in these algorithms is the recurring need for the computation of point neighborhoods, usually by applying algorithms to compute the k-nearest neighbours of each point. The majority of these algorithms work under the assumption that the data sets operated on can fit in main memory, while others take into account the size of the data sets and are thus designed to keep data on disk. We present a hybrid approach which exploits the spatial locality of point clusters in the point cloud and loads them in system memory on demand by taking advantage of paged virtual memory in modern operating systems. In this way, we maximize processor utilization while keeping I/O overheads to a minimum. We evaluate our approach on point cloud sizes ranging from 50K to 333M points on machines with 1GB, 2GB, 4GB and 8GB of system memory.

Categories and Subject Descriptors (according to ACM CCS): I.3.6 [Computer Graphics]: Methodology and Techniques—Graphics data structures and data types

1. Introduction

The acquisition of 3D point clouds of objects and environments has become common place in fields like robotics and cultural heritage. The raw data resulting from the acquisition process usually needs to be processed in order for important topological information to be extracted. For instance, in the case of robot navigation, this processing might be required in order to determine the location of a particular object in the environment and guide the robot around it. In some cases, the size of the data set acquired is so large that it does not fit in main memory. This is particularly true of outdoor cultural heritage sites (e.g [Rut10]) acquired using professional grade 3D scanners capable of generating highly accurate data at sampling rates of close to a million points per second [EBN11].

Post processing operations, like normal determination, clusters and noise removal, all require the computation of the k-nearest neighbours (k-NN) for each point in the point cloud. When the size of the point cloud is very large, a considerable amount of time is spent searching for the k-NN of each point. Moreover, many of these post-processing operations (e.g. noise removal) are usually applied on the same data set more than once using different input parameters. This is especially true in the case of clustering or labelling algorithms where input parameters may produce widely varying results. Even if these operations are usually carried out offline, execution time is still an important factor to take in consideration. Optimal performance results are achieved when k-NN computation is carried out in-core, i.e. when both points and acceleration structure are stored in main memory. On the other hand out-of-core techniques take into account the size of the points but are much slower due to overheads related to disk I/O. In this paper, we present a novel external memory algorithm using a hybrid of spatial subdivision techniques for out-of-core fast k-nearest neighbour searches on point cloud data.

2. Related Work

The development of algorithms for the efficient determination of the k-nearest neighbours of points in a point cloud has been an active area of research for many years (e.g. [Cla83], [Vai89], [SSV07]). In most cases memory-based space subdivision data structures are used to help quickly determine

DOI: 10.2312/LocalChapterEvents/TPCG/TPCG12/085-092

neighbouring points. One such accelleration structure is the k-d tree [FBF77] which is used in many prominent libraries [e.g. [ML09], [RC11]] to provide a spatial subvision over the input point cloud. Search algorithms, mostly based on either depth-first search (DFS) or best-first search (BFS) are then used to efficiently compute neighbours. These search algorithms can either compute the exact nearest neighbours or else the approximate nearest neighbours (ANN). In the case of ANN, an error threshold ε is used to speed up the computation of neighbours at the expense of correctness. Significant speedup can be achieved when the data set consists of higher dimensional data points [AMN*94]. In the case of 3D scanned point cloud data, the difference in preformance between ANN and k-NN is minimal. In our approach both approximate and exact nearest neighbours can be computed. In the results presented here only the exact k-NN are computed.

An important consideration which is addressed by Sankaranarayanan et al. [SSV07], is the size of these point clouds. As size increases, search algorithms based on in-core data structures, such as k-d trees [FBF77], are limited by the amount of memory present in the computer on which they are deployed. Sankaranarayanan et al. [SSV07], describes an all nearest neighbour algorithm for applications involving large point clouds. Their algorithm makes use of disk-based out-of-core data structures and is thus not limited by the amount of system memory available. They first determine localities, for blocks of points, which are then used to decrease the range of candidate neighbour points to search. Even though their algorithm is desgined to work with multi-dimensional data sets, evaluation is carried out only on 3D point clouds and report significant improvements over previous methods with respect to the time it takes to compute k-NN. For example, when using a data set of 50 million points, 7999 neighbourhood/s are computed on a machine with 1GB of system memory. In our case we adopt a hybrid approach which takes advantage of all system memory available but never exceeding it. On a 1GB machine with similar specifications, our approach achieves approximately 100,000 neighbourhoods/s using a data set of 166 million points.

3. Preliminaries

In order to design a fast k-NN computation procedure, we take advantage of two important concepts, namely, spatial subdivision and memory mapped files. The first is used to reduce the time complexity of the nearest neighbour algorithm, whilst the second is used to maximise the use of available memory.

3.1. Spatial Subdivision

Regular grids subdivide 3-space into regions of equal volume where each region can be uniquely addressed by an index (i, j, k). If the regions operated on are known, one doesn't need to be concerned with the whole grid, but can concentrate instead on the said regions. The straightforward subdivision afforded by regular grids allows us to maximize memory utilization by loading in core only the affected regions. The point clouds we use are not uniformly distributed in 3-space and partitioning these data sets into regular grids yields a large number of empty regions. Thus, we implement the regular grid as a sparse map and keep track only of the regions which contain interesting information. The time complexity for lookup and insertion of a region, or cell, is in both cases O(log n), since the sparse grid is implemented using red-black trees [Bay72]. A lookup for the nearest-neighbour of a point within a region runs in linear time; we thus use k-d trees to store points within a cell, reducing the lookup complexity to logarithmic time in the number of elements [FBF77].

3.2. Memory Mapped Files

Virtual memory [Den70] is a memory management technique which allows the execution of processes not entirely held in memory by separating the user view of memory from the actual physical memory and provides a mapping function from one to the other. Implementations for virtual memory require hardware support, typically provided by a memory management unit built into the CPU. Paged virtual memory is an implementation of a virtual memory system which divides the logical address space into equal sized memory blocks called pages, permitting the use of memory mapped files (MMF), wherein a file can be manipulated as part of the process address space. This is accomplished by mapping disk blocks to pages in memory using the virtual memory system. Access to memory mapped files uses a demand paging scheme, whereby a block is loaded in memory only if it is needed. The first time a block is accessed, a page fault is generated, and the respective block brought to memory. Subsequent accesses to the specific block occur as memory reads or writes, avoiding the overhead of read and write system calls. Moreover, files which do not fit in memory can still be manipulated with relative ease, as the paged virtual memory system, swaps blocks in and out as required.

4. Concurrent k-NN searches using MMF

Our work addresses the problem of efficiently searching for the k-NN of all points in a point cloud P, when the size of P does not fit entirely in main memory. In order to decrease the memory requirements of the process computing k-NN, we store all point information on disk and iteratively load only those regions in the file which are required in the k-NN computation for a subset of points in P. Points are loaded in memory through the use of MMFs. In general, we would like to exploit all memory available on a machine to achieve the best possible performance, however in order to mitigate

I/O problems which could result from having a process using all system memory, we use a heuristic M to indicate an *approximate upperbound* on the number of points which can simultaneously be present in system main memory. Decreasing the value of M will decrease the memory footprint of the entire process. Whenever we want to use all system memory available, M is set to a value larger than the number of points in P. In order to speed up the time it takes to compute k-NN for each point, all processing elements (PE) available on multi-core computers are utilised.

Algorithm 1 describes the high level structure of our approach. The process starts by first creating and populating a uniform sparse grid G with a count representing the number of points in P which fall within each axis aligned cell in G. This is done by iterating once over all the points in P. Using this information, separate files are created each storing a cell ordered subset of points. Once these clusters of points (stored on disk) are created, they are iteratively loaded in main memory and k-NN is performed for points in these clusters.

Algorithm 1 High-level description of k-NN computation

```
Procedure Search for k-NN of all points p_i ∈ P.
Input Point-cloud P, M, k.
Load Create sparse grid G storing counts for each cell.
Sort Partition P. Persist to disk ordered clusters OC_n.
for each cluster OC_j do
    Memory map points cluster OC_j to main memory
    for each non-ghost grid cell C_k present in OC_j do
        Create local kd-tree
        for each point p_i in C_k do
            Compute k-NN
            Perform operation on p_i using neighbours
        end for
    end for
end for
```

The following sections describe in more detail the stages *load*, *sort* and *compute*. The first stage reads a point cloud binary file and determines spatial locality for all points. This information is then used to sort and divide in clusters of points, depending on M, the inputl point cloud P. This spatially sorted point cloud is then used in the third stage to search for the k-nearest neighbours of each point.

4.1. Loading

The input to this stage are point clouds stored using the Point Cloud Library (PCL) [RC11] binary format, with each point represented as a triple (X,Y and Z coordinates) of type *float*. Since one of our objectives is to decrease the memory footprint of the application used to process a point cloud, whenever the number of points in the cloud is larger than the value of M, which is specified in number of points, a point cloud iterator is used which does not load the entire point cloud in memory. Instead, M points are loaded iteratively from file using MMFs. Since not all points are loaded in memory at any one point in time, each point-cloud is represented as a collection of segments. The maximum size (in number of points) of a segment is M. The index of each point is thus representing using a local offset within the segment and its global index (within the whole point cloud P) is computed from the segment number and local offset. Figure 1 shows the straightforward abstraction adopted.

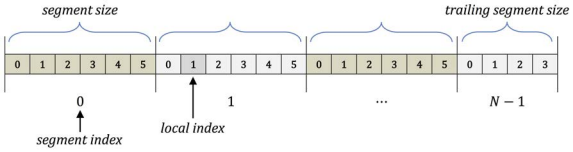


Figure 1: *Input point cloud is loaded in segments.*

The point cloud iterator $GetNext()$ first checks whether the next point to be returned is in the current segment, i.e. whether it's currently addressable in memory. If this is the case then values associated with the next point are returned, otherwise, if the end of segment is reached, the mapped region of the MMF is first deallocated then memory-mapped with the next segment. When the last point in the last segment is reached, $GetNext()$ returns $false$, indicating that all points have been read.

A uniform sparse grid is used to store the number of points contained within each axis-aligned cell in the sparse grid G. This information is used to persist the point cloud to disk ordered by cell index. For each point a key is computed which indicates the cell into which the point should be placed. The key is composed of three values representing cell indices along the X, Y and Z directions. The number of cells along each direction is computed from a user-defined value which specifies the size of each cell. Since points are fitted in a uniform grid, all cells have the same size. As we shall outline later on, this is an important consideration when searching for k-NN concurrently, and also to quickly determine if the correct k-neighbours have been chosen. In the experiments carried out, this value was set to 0.2 for all point clouds. The maximum number of cells in the sparse grid depends on the bounding volume of the input point cloud. For example if the bounding volume is (1,2,3) then the sparse grid would have a maximum of 5 cells along the X direction, 10 cells along the Y direction and 15 along the Z direction. Given that we use a sparse grid, only those cells where points are spatially located are created and stored in system main memory. In the largest point cloud (333M) used to evaluate our approach, the number of cells in the grid is of 97,253.

4.2. Sorting

The output from the previous stage is a sparse grid G holding a count of the number of points contained within each cell. Given this information, together with a value for the approximate number of points in memory M and a specific ordering over grid cells, an optimal set partition of points in P is determined. This set partition groups together clusters of cells, over which k-NN can be computed in-core while adhering as closely as possible to the value of M. The cell ordering employed in our implementation follows in ascending order the X, Y then Z axis as illustrated in figure 2.

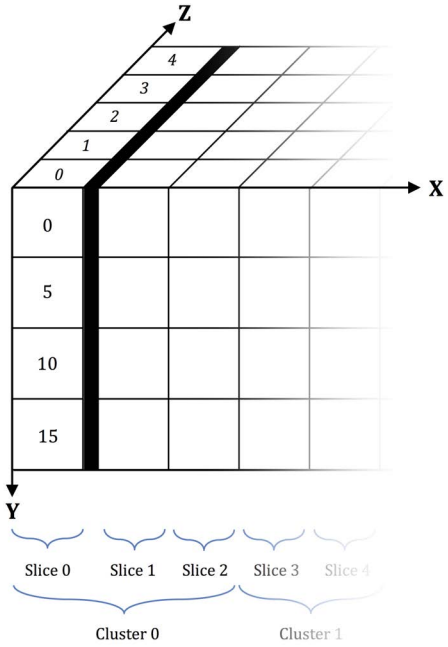


Figure 2: *Sparse grid decomposition and cell ordering*

This ordering implies that the bounding volume of the entire point cloud can be seen as being composed of a number of slices along the x-axis (*X-slices*), where each slice would consist of a number of cells varying along the Y and Z axes. Hence, one valid set partition of P would consisit of cells grouped by X-slice. However, since points are usually not distributed uniformly across the bounding volume of the point cloud, there will be X-slices with many more points than others. Thus, the partitioning process groups together as many X-slices as possible. M is used to determine the size of these clusters of X-slices, with each cluster having approximately M points. For example, if the axis-aligned bounding volume of the point cloud is divided into twelve X-slices, a possible set partition OC could consist of the four clusters {{1,2,3,g},{g,4,g},{g,5,6,g},{g,7,8,9,10,11,12}}. The partitioning process guarantees that the number of points present per cluster over which k-NN can be computed is approximately equal to M. In this case the number of points in the 4th X-slice (alone in the second cluster) is higher than the number of points in the rest of the slices. Hence, it is loaded in main memory on its own. An important aspect that needs to be taken into account when constructing this set partition, is the inclusion of ghost cells/points (represeted using the letter g in the example) within each cluster, i.e. those points for which we do not compute k-NN (within this cluster) but which may actually be one of the k-nearest neighbours for some of the points in the cluster. Figure 3 shows the ghost cells and respective ghost points for point p_i located in the central cell of the 3x3 grid. In the case of a 3D sparse grid, for every cell there can be a maximum of 26 ghost cells. In our implementation, for each cluster OC_i, the last X-slice from OC_{i-1} and the first X-slice from OC_{i+1} are added. Clearly, for OC_0 only the first X-slice from OC_1 is added, whereas for the last cluster OC_n only the last X-slice from OC_{n-1} is added. These additional cells representing the boundary points of the cluster are required to compute k-NN correctly. Since clusters are created over X-slices, the value of M must be reasonably chosen, i.e. it should not be very small. In the results section, the effect of changing this parameter is evaluated with respect to memory usage and performance.

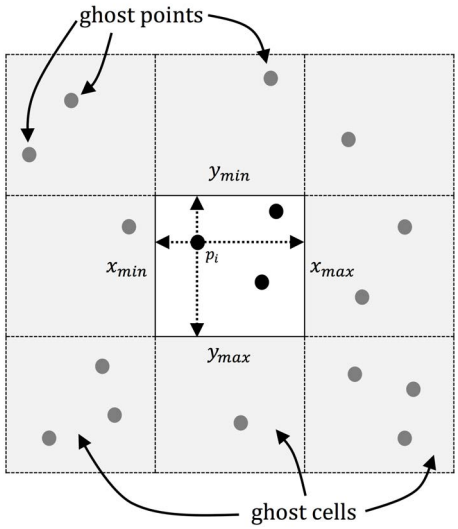


Figure 3: *Ghost cells and points*

The output from this stage is a file for each cluster of X-slices. Each file stores points following the cell ordering described in figure 2. Point ordering within the cell is not important. Taking the example above, this stage would produce four files storing the points from clusters {1,2,3,4},{3,4,5},{4,5,6,7},{6,7,8,9,10,11,12} respectively. In the next stage these files will be efficiently loaded in memory using MMFs.

Algorithm 2 describes the procedure used to sort the input

Algorithm 2 Sorting

Procedure Sort points in P and persist to files
Input P, G with counts for each cell, Clusters OC.
for each cluster OC_i **do**
  Create MMF to store points in OC_i
  Update G with file position offsets of cells in OC_i
  cnt_{total} = number of points in OC_i
  $cnt_{written} = 0$
  for each point $p_j \in P$ **do**
    if p_j falls within this cluster **then**
      Retrieve cell C_k where p_j is located
      Write p_j to file at position offset indicated at C_k
      $cnt_{written} = cnt_{written}+1$
      Increment offset at C_k
    end if
    if $cnt_{written} == cnt_{total}$ **then**
      Flush MMF of OC_i.
      Continue.
    end if
  end for
end for

Algorithm 3 Compute k-NN

Procedure Compute k-NN for all points $p_i \in P$
Input G, Cluster Set OC.
for each cluster OC_i **do**
  Memory map file with points in OC_i
  Update file position offsets of cells in OC_i
  Generate array $CellArr$ storing keys of cells in OC_i
  $cellCount$ = size($CellArr$) - no. of ghost cells in OC_i
  $crtCellIdx$ = index of first non ghost cell
  while $crtCellIdx < cellCount$ **do**
    Atomically assign to PE $crtCellIdx$
    PE generates kd-tree on points in $CellArr_{crtCellIdx}$
    for each point p_j in $CellArr_{crtCellIdx}$ **do**
      Search for k-NN of p_j
      d = shortest dist(p_j,$CellArr_{crtCellIdx}$ planes)
      if dist(p_j, NN_k) > d **then**
        Add p_j to k-NN recomputation list RL
      end if
    end for
    while sizeof(RL) > 0 **do**
      Update kd-tree with points from adjacent cells
      Compute k-NN for p_j
      d += extent of $CellArr_{crtCellIdx}$
      if dist(p_j, NN_k) < d **then**
        Remove from recomputation list RL
      end if
    end while
    Delete kd-tree
    Atomically increment $crtCellIdx$
  end while
end for

point cloud P. For each cluster of X-slices in OC, a file is created. In order to write points at the correct offsets in each file, the information within each cell in G is augmented with file position offsets indicating at which location of the current file the next point contained in that cell should be written. Sorting is currently not very efficient since for each file written, the function $GetNext()$ has to iterate over all points in G. When the size of P is very large (e.g. 333 million points on a 1GB machine) this actually becomes a bottleneck and ends up taking as much time as computing k-NN.

4.3. Concurrent search for k-NN

When computing the k-NN for a given point, our approach ensures that the correct k-nearest neighbours are actually returned. In general, given the two sets of points P_g and P_n, with $P_n \subset P_g$, we need to ensure that the set P_g contains the k-nearest neighbours of all points in P_n. As opposed to the work of [SSV07], i.e. pre-compute the set P_g before searching for the k-NN of points in P_n, we verify that this is the case for each point in P_n once the k-NN are determined. Since each point is located in an axis-aligned cell, the shortest distance d between the position of the point and any one of the boundary planes of the cell can be determined very efficiently. Figure 3 describes how this is done in 2D. After determining k-NN, we check whether the distance between the k^{th} neighbour and the current point is smaller than d. If it is then the currently chosen neighbours are correct and can be returned otherwise the point is flagged for re-computation of k-NN taking in consideration a larger set of adjacent ghost cells. Algorithm 3 describes in detail how the search for k-NN works.

Each processing element (PE) in the system atomically retreives the next available cell in the currently active OC cluster and computes k-NN searches over all points in the cell. k-NN searches are carried out by creating a temporary k-d tree over points in the currently active grid cell. When all searches are done, the k-d tree is deleted from memory. Temporary k-d trees are created and deleted for all cells in G.

5. Results

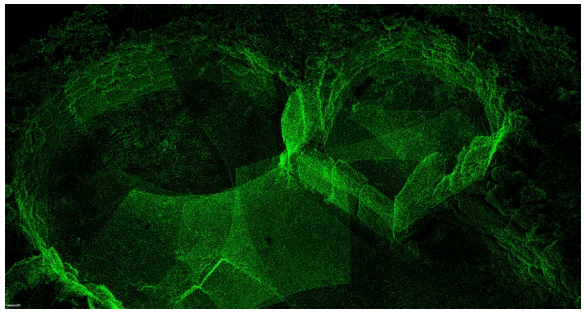
We evaluate our approach on a number of point clouds ranging in size from 53K to 333M points. All experiments are carried out on an Intel Core2Quad machine running Windows7 and SATA2 hard disks. In order to evaluate performance against different memory configurations, the same machine is installed with 1GB, 2GB, 4GB and 8GB of system RAM. Experiments are conducted in order to evaluate the scalability of our approach as the size of the point cloud is increased across these different memory configurations. In addition to an implementation of the concurrent grid based multi k-d tree ($GridXKd$) approach described above, two

further implementations are evaluated for comparison. The first implementation takes the traditional in-core approach where a k-d tree is constructed over all points in the data set. We shall be refering to this implementation as in-core k-d tree (*ICKd*). This implementation should provide the best possible performance whenever enough memory is available to hold the k-d tree. The PCL library [RC11] is used for this implementation which also uses memory mapped binary files to store points. The second implementation works exactly like *GridXKd*, but does not use memory-mapped files and instead loads all points in the sparse grid data structure (rather than just the number of points) before starting to compute k-NN. We shall be referring to this implementation as the in-core concurrent grid based multi-kdtree (*ICGridXKd*). In all cases the FLANN library [ML09] is used to implement kd-tree based kNN searches. The error-bound parameter ε is set in all implementations to zero. Moreover in all implementations all four processing elements available on the computer used are utilised to concurrently compute k-NN.

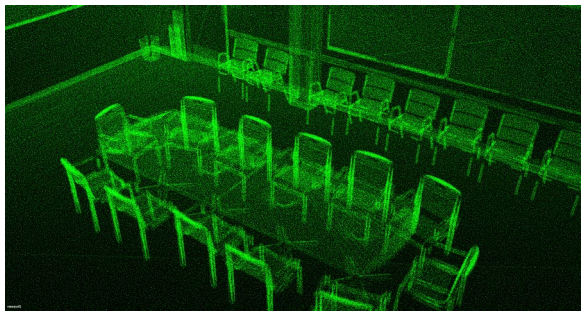
Table 1 lists the point clouds used in the experiments. In all cases (except for Mnajdra and Songo) the data has been generated from polygonal models. In the case of SongoX2, SongoX4 and SongoX8, the original point cloud was up-sampled using a standard up-sampling algorithm in order to increase the number of points. Figure 4 illustrates three of the point clouds used.

Model Name	Size(M)	Cell count in Grid
obelisk	0.053	1097
mnajdra	0.579	6087
conference	2.3	6338
sibenik	6	201,756
songo	41	95,999
songoX2	83	96,940
songoX4	166	96,853
songoX8	333	97,253

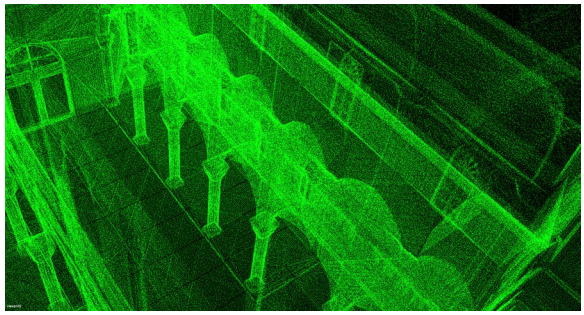
Table 1: *Point clouds, corresponding number of points and number of cells created during loading phase in sparse grid*



(a) Mnajdra 579K points



(b) Conference 2.3M points



(c) Sibenik 6M points

Figure 4: *Views of three point-clouds used in the results*

5.1. Execution Time

We first compare execution time for all three implementations on a machine installed with 8GB of system memory. This is done in order to first establish the best possible results for the three implementations. In the case of the *GridXKd*, parameter *M* is set to a value greater than the number of points in the cloud in order to maximise the use of system memory. *GridXKd* is later evaluated with different values of *M* in order to establish how this constraint effects execution time. Table 2 shows the time it takes for each implementation to calculate k-NN with k set to 16 for the different models.

Note that the readings for *GridXKd*, also include the time taken to populate the sparse grid *G* and persist to file (or files depending on the number of clusters created at the sorting phase) a sorted version of the original point cloud. As the size of the point cloud increases so does the time taken to sort it. This is evident when working with the largest points clouds. As was to be expected *ICKd* performs better in those cases were the accelleration structure can easily fit in main memory. However, as the size of the input data set increases, the performance of our approach (*GridXKd*) is better than that of *ICKD* and *ICGridKd*. Due to the in-core nature of both *ICKD* and *ICGridKd*, both are not able to process the 333 million point data set songoX8. In the case of *GridXKd* the execution time is linearly proportional to the size of the

Model Name	ICKd	ICGridXKd	GridXKd
obelisk	0.127	0.193	0.241
mnajdra	1.164	2.068	1.748
conference	4.864	7.726	5.891
sibenik	12.032	20.039	15.911
songo	101	198	167
songoX2	207	420	353
songoX4	916	-	707
songoX8	-	-	1426

Table 2: *Execution times using 8GB RAM*

input. In the case of the point cloud songoX4, our approach performs better than the in-core *ICKd*.

We now evaluate the execution times of all three implementations whilst decreasing the amount of system memory available. Tables 3, 4 and 5 show execution times for all data sets with 4GB, 2GB and 1GB system memory installed. Table 3 shows the results obtained with 4GB system memory installed. When processing the largest point cloud (songoX8), in order to limit the amount of memory required by *GridXKd*, parameter M is set to 100 million. When M is not set to a value smaller than the size of the dataset, too many points would have been present in system memory resulting in our approach not being able to process songoX8. In order to be able to process this point cloud, we set M to a value smaller than the number of points in the cloud. With M set to 100 million points, *GridXKd* computes all k-NN in 1909 seconds. Given the size of the point cloud a considerable amount of time, 358 seconds, is spent on the sorting phase which partioned the dataset into 5 clusters. Table 6 shows the effect of varying M on both load and sorting times of our approach. The number of segments created at load time and the number of clusters created at the sorting stage are also listed. Once the point cloud is loaded, sorted and persisted to file/s the time taken to compute k-NN is the same across all variations of M with 1GB of system memory installed. These results show that with 1GB of RAM installed, the best results are obtained when setting M to 20 million with the sorting stage partitioning the input point cloud into seven clusters.

Tables 4 and 5 show execution times for all data sets with 2GB and 1GB RAM installed. In all cases *GridXKd* is able to compute all k-NN. When using 1GB, with point clouds of more than 20 million points, M (values shown in table) is used to reduce the number of points which are simultaneously loaded in memory. In all cases the value is set to 30 million or less. As shown in figure 5, as the number of points increases, a considerable amount of time is spent sorting the point cloud. In our current implementation loading and sorting is always performed, however in theory this is not required. Once a sorted point cloud is persisted to file it can be reloaded without incurring the cost of re-sorting. Clearly in this case the sparse grid G would need to be persisted with the rest of the data and reloaded each time. When processing large data-sets this operation is much less expensive than sorting.

Model Name	ICKd	ICGridXKd	GridXKd (M)
obelisk	0.109	0.234	0.172
mnajdra	1.469	2.047	1.921
conference	5.046	8.203	6.031
sibenik	13.875	17.726	16.281
songo	114	205	169
songoX2	443	-	356
songoX4	-	-	786
songoX8	-	-	1909 (100M)

Table 3: *Execution times using 4GB RAM*

Model Name	ICKd	ICGridXKd	GridXKd (M)
obelisk	0.156	0.213	0.157
mnajdra	1.547	2.031	1.673
conference	5.219	7.609	5.957
sibenik	14.641	21.953	16.221
songo	238	-	170
songoX2	-	-	379
songoX4	-	-	1160
songoX8	-	-	1577 (60M)

Table 4: *Execution times using 2GB RAM*

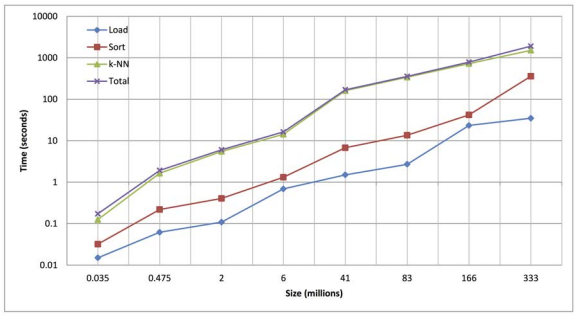
Model Name	ICKd	ICGridXKd	GridXKd (M)
obelisk	0.147	0.243	0.171
mnajdra	1.648	2.323	1.673
conference	5.132	8.102	6.345
sibenik	17.231	24.252	16.454
songo	-	-	300 (20M)
songoX2	-	-	522 (20M)
songoX4	-	-	1541 (30M)
songoX8	-	-	5995 (30M)

Table 5: *Execution times using 1GB RAM*

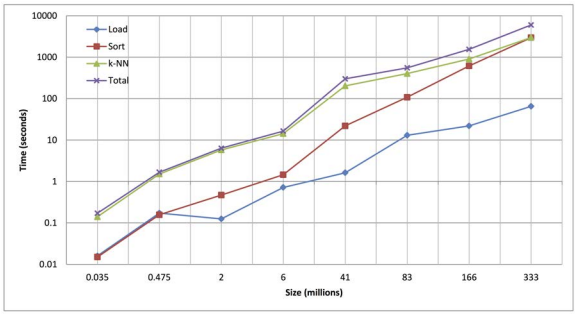
The results of our experiments show that we are able to efficiently compute k-NN searches on very large point clouds. In the case of small point clouds, our results are comparable to the results achieved by an optimal in-core implementation of k-NN search. This demonstrates the scalable nature of our approach. For a neighbourhood of size k=16, using either 8GB, 4GB or 2GB of system memory, we are able to compute approximately 235,000 neighbouhoods/s on an 83 million point data set.

M(million)	Segments	Load(s)	Clusters	Sort(s)
10	9	11.39	20	235
20	5	10.86	7	103
40	3	7.297	3	136
60	2	9.336	2	150
85	1	14.156	1	251

Table 6: *Varying values of M on the songoX2 data set (with 1GB RAM installed)*



(a) Using 4Gb RAM



(b) Using 1Gb RAM

Figure 5: *Execution times for load, sort and compute k-NN*

6. Conclusion and Future Work

The generation of very large 3D point clouds is becoming increasingly common in many areas. Given this huge amount of data, fast k-NN computation methods are required to process this data efficiently. We have presented a procedure which efficiently searches for the k-nearest neighbours of points over very large point clouds. Results have shown that we can easily scale up from a few thousand points to several millions even with limited memory resources. Source code of all implementations presented in this paper is available for download from http://pointcloudsemantics.codeplex.com/

Plans for future development include improvements on the current implementation of the point cloud sorting phase. We are also looking into using different sparse grid cell sizes and analyse the tradeoffs between number of cells in the sparse grid and the average number of points in each cell. Another interesting future direction is that of extending the concept of the error-bound ε used for ANN in k-d trees to include the sparse grid subdivision of space.

Acknowledgements

The point clouds used in this publication originate from a number of sources. For *mnajdra* we would like to thank Heritage Malta, the national agency for museums, conservation practise and cultural heritage in Malta. We also thank Prof Heinz Ruther, Dr Patrick Marias and Dr Christoph Held from the Zamani Project for providing us with the *songo* point cloud. We thank Greg Ward for the Conference scene from the Radiance package and Marko Dabrovic for the Sibenik cathedral model. The *conference* and *sibenik* point clouds were produced from these two models.

References

[AMN*94] ARYA S., MOUNT D. M., NETANYAHU N. S., SILVERMAN R., WU A.: An optimal algorithm for approximate nearest neighbor searching. In *Proceedings of the fifth annual ACM-SIAM symposium on Discrete algorithms* (Philadelphia, PA, USA, 1994), SODA '94, Society for Industrial and Applied Mathematics, pp. 573–582. 2

[Bay72] BAYER R.: Symmetric binary b-trees: Data structure and maintenance algorithms. *Acta Inf. 1* (1972), 290–306. 2

[Cla83] CLARKSON K. L.: Fast algorithms for the all nearest neighbors problem. In *FOCS* (1983), pp. 226–232. 1

[Den70] DENNING P. J.: Virtual memory. *ACM Computing Surveys 2* (1970), 153–189. 2

[EBN11] ELSEBERG J., BORRMANN D., NUCHTER A.: Efficient processing of large 3d point clouds. In *Information, Communication and Automation Technologies (ICAT), 2011 XXIII International Symposium on* (oct. 2011), pp. 1 –7. 1

[FBF77] FRIEDMAN J. H., BENTLEY J. L., FINKEL R. A.: An algorithm for finding best matches in logarithmic expected time. *ACM Trans. Math. Softw. 3*, 3 (Sept. 1977), 209–226. 2

[ML09] MUJA M., LOWE D.: Flann - fast library for approximate nearest neighbors user manual. *Writing* (2009). 2, 6

[RC11] RUSU R. B., COUSINS S.: 3D is here: Point Cloud Library (PCL). In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)* (Shanghai, China, May 9-13 2011). 2, 3, 6

[Rut10] RUTHER H.: Documenting africa's cultural heritage. In *In Proceedings of the 11th International Symposium VAST. Virtual Reality, Archaeology and Cultural Heritage* (2010).

[SSV07] SANKARANARAYANAN J., SAMET H., VARSHNEY A.: A fast all nearest neighbor algorithm for applications involving large point-clouds. *Computers & Graphics 31*, 2 (2007), 157–174. 1, 2, 5

[Vai89] VAIDYA P.: An o(n log n) algorithm for the all-nearest-neighbors problem. *Discrete & Computational Geometry 4* (1989), 101–115. 10.1007/BF02187718. 1