# Parallel Execution of kNN-Queries on in-memory *K*-D Trees

Tim Hering
Faculty of Computer Science
University Otto-von-Guericke
Universitätsplatz 2
39106 Magdeburg
thering@st.ovgu.de

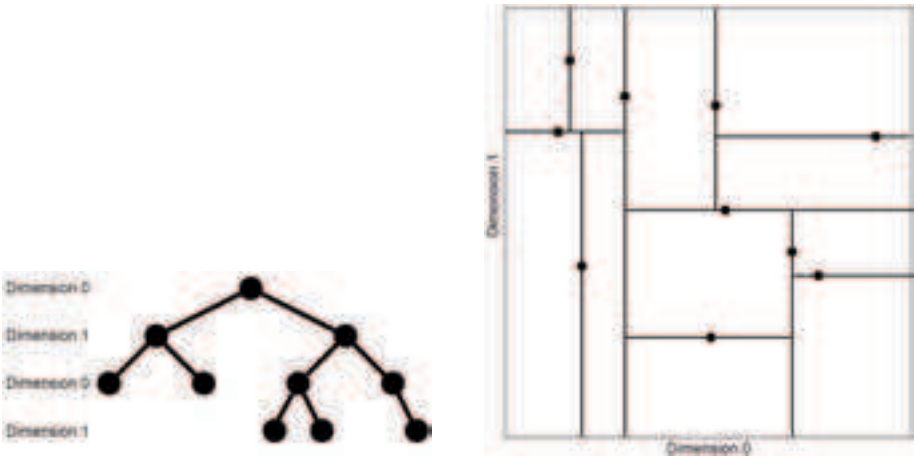
**Abstract:** Parallel algorithms for main memory databases become an increasingly interesting topic as the amount of main memory and the number of CPU cores in computer systems increase. This paper suggests a method for parallelizing the *k*-d tree and its kNN search algorithm as well as suggesting optimizations. In empirical tests, the resulting modified *k*-d tree outperforms both the *k*-d tree and a parallelized sequential search for medium dimensionality data (6-13 dimensions).

## 1 Introduction

Multidimensional datasets are a collections of data points with multiple dimensions. They are produced by a number of applications, one of the most cited of which are multimedia databases [BBK01]. One of the characteristic requirements to storage solutions for multidimensional datasets is the ability to process similarity-based queries fast. One class of those queries are kNN-queries. They retrieve the k points in the dataset that are closest to a given query point. Notable index structures that are designed for multidimensional databases include the R-Tree [Gut84], the Pyramid Technique [BBK98], the VA-File [WB97] and Locality Sensitive Hashing [IM98].

Two developments in computer hardware are starting to change the possibilities of and requirements for index structures. Firstly, the amount of main memory is increasing. As a result, more database tasks can be executed in main memory. Secondly, most computers possess multi-core processors or graphic cards. Both can be used to conduct parallel computations. They reduce the processing time of algorithms, provided they run on main memory data. If disk accesses need to be performed, the speed-up becomes less relevant. The topic of parallelizing multidimensional index structures on main memory databases is largely unexplored. Most attempts at parallelization focus on multiple disks or systems instead of multiple cores (as discussed in section 3.2).

This paper explores the possibility of using *k*-d trees for multidimensional kNN-queries [Ben75]. *K*-d trees are binary trees that are suitable for multidimensional data and were designed for main memory applications. Each level of the tree splits according to the dimension after the one of the previous level. As a consequence, the data space is divided according to a brick wall pattern (Figure 1).




(a) Tree representation (b) Partitioning of the data space

Figure 1: A 2-d tree containing 10 data points

An approach for utilizing multi-core processors to speed up the search on *k*-d trees is subject of this paper. Additionally, several improvements to the resulting index structure are described. They focus on lessening the impact of the disadvantages of parallelization, primarily the need for inter-thread communication. Lastly, suitable parameters for these modifications are determined using empirical tests.

## 2 The *k*-d tree

The *k*-d tree is an index structure for multidimensional data that was introduced in 1975 by Jon Louis Bentley [Ben75]. At its core, it is a binary tree, that can efficiently store data points with more than one dimension by cycling through those dimensions as an algorithm descends down the tree (Figure 1). This means that each level $l$ of the tree has a designated dimension $d_l = l \bmod d_{max}$, where $d_{max}$ is the dimensionality of the data. If an algorithm tries to compare the query point to the discriminator point in a node at level $l$, it uses the values of the points in $d_l$ for comparison. The lower child of that node and all its descendants have a lower value than the discriminator point regarding $d_l$, whereas the higher child and its descendants have higher values. As a consequence, all dimensions receive an approximately equal treatment while the logarithmic running time for insert operations and point queries of binary trees is maintained.

The kNN search algorithm presented in [FBF77] traverses the *k*-d tree recursively. For a given node, the search function executes the following steps: (1) If its point is closer to the query point than any other within a global list if the k best result candidates, update the list with that point. (2) If the node is not a leaf node, the function is recursively called with the child node closer to the query point. (3) The minimum possible distance between the

query point and the other, more distant child node is calculated. If this distance is larger than the maximum distance for a candidate (calculated from the result list), no point in that node or its descendants can be a result candidate. Otherwise, the search function is called with that node.

The algorithm is initiated by calling the search function with the root node. After control returns to the caller, the result list contains the k nearest neighbours to the query point.

# 3 Related work

This section covers four existing categories of index structures that are related to the content of this paper: (1) index structures that are explicitly designed for main memory applications, attempting to optimize the usage of caches, (2) parallel index structures that are not designed for main memory databases, (3) parallel *k*-d trees designed to run on GPUs and (4) parallel index structures for main memory databases that are not designed for kNN-queries.

## 3.1 Cache sensitive index structures for main memory databases

There is a number of index structures for main memory databases that rely on the optimization of cache usage. Cache memories are small, but fast buffers for data blocks in the main memory [Smi82]. Being conscious of these buffers can speed the access times of searches up.

Among the index structures following this idea are Cache-Sensitive Search Trees (CSS-trees) [RR98]. They take advantage of the cache size by possessing leaf nodes that are as large as a block in the cache. Another, similar example is the Cache Sensitive $B^+$-Tree ($CSB^+$-Tree) [RR00]. It is a variant of the $B^+$-Tree that reduces the size of its nodes by having less or no child pointers. Children are instead identified by their address offsets.

## 3.2 Parallel index structures

As parallelization is a commonly used method to speed up computations, various index structures have employed it. In contrast to the parallel *k*-d tree, they tend to focus on parallel disks or parallel systems instead of multiple threads. An example of the former are Parallel R-Trees [KF92]. They are a class of variants of R-Trees, whose nodes are spread over multiple disks to improve query throughput. Additionally, attempts at porting them to an index structure on parallel systems (spreading nodes over multiple computers) have been made [AQSK98].

### 3.3 Parallel *k*-d trees for GPUs

There are various existing papers on parallel *k*-d trees for GPUs. They mostly cover parallel inserts, as fast insert operations are required for 3d rendering [ZHWG08, SSK07]. Examinations on parallel queries are less common, but existing [GDB08]. GPUs have significantly more cores than CPUs, but they also have disadvantages in regards to the execution of queries on *k*-d trees: Firstly, they are less suited for control flow. This means that jumps within the code, like those created by if-statements, execute slower. Secondly, data and queries have to be loaded into the GPU memory. After the execution of the query the results have to be returned. Both processes create an overhead that is avoidable by using the CPU. Additionally, not all computers are equipped with a GPU, especially servers that are remote controlled. This makes GPU solutions less universally applicable than CPU solutions. Nevertheless, they are an interesting alternative, if only for the increase in computational power that can be gained.

### 3.4 Parallel in-memory index structures for other query types

Parallel processing has been utilized by index structures that are designed to answer queries other than kNN-queries. One of them is FAST (Fast Architecture Sensitive Tree), a tree-based index structure built with the characteristics of modern hardware in mind [KCS+10]. The parts of the index are compressed and divided to specifically reduce the limiting influence of memory bandwidth. Another example is the KISS-Tree, which is a prefix tree of 3 levels [KSHL12]. The levels deploy different addressing techniques that are tailored to the requirements of each respective level, allowing the KISS-Tree to reach a performance similar to that of hash-based indexes.

## 4 Modifications to the *k*-d tree

To utilize multi-threading, the kNN search algorithm of the *k*-d tree has to be changed. The basic idea is to introduce a queue that holds all nodes which are yet to be visited. There is a fixed number of threads that work independently and communicate over that queue. In turn, threads retrieve the head of the queue and evaluate it. If one or both of its children are candidates for the query results, they are inserted at the back of the queue (see Figure 2 for the full algorithm). A node is a candidate if the minimum distance of the query point to its bounding rectangle is smaller than the k-th nearest processed neighbour [FBF77].

Communication between threads has the tendency to decrease performance of multi-threaded applications significantly. As one thread uses the communication resource, it is locked and other threads have to wait for it to finish. To avoid introducing the list of intermediate results as a bottleneck (its last entry is needed to determine if a node is a candidate), each thread has its own set of intermediate results. After all threads have finished, their result lists are merged to get the global answer. This measure reduces the speed with which the

```
01  knnQuery(queryVector) {
02    while (not nodeQueues.allEmpty) {
03      node = pollNodeQueues()
04      if (not node.isLeafNode) {
05        if (isCandidate(node.leftChild, queryVector)) {
06          addToNodeQueue(node.leftChild)
07        }
08        if (isCandidate(node.rightChild, queryVector)) {
09          addToNodeQueue(node.rightChild)
10        }
11      } else {
12        addToResultList(node)
13      }
14    }
15    return resultList
16  }

17  node pollNodeQueues() {
18    int rand = random(0, numberOfNodeQueues - 1)
19    if (not nodeQueues.get(rand).isEmpty) {
20      return nodeQueues.get(rand).poll()
21    } else {
22      for (int i from 0 to numberOfNodeQueues - 1) {
23        if (not nodeQueues.get(i).isEmpty) {
24          return nodeQueues.get(i).poll()
25        }
26      }
27    }
28    return lowPriorityQueue.poll()
29  }
```

Figure 2: Pseudo code for the modified kNN search algorithm.

algorithm converges, but experiments showed an overall increase in performance.

The remaining bottleneck is the queue of nodes. It is critical for inter-thread communication and thus cannot be easily removed, but the impact of the delays can be reduced by splitting the queue. Whenever a thread would execute an operation on the queue, it instead selects one of an array of queues at random. The probability for a thread to find a queue in use is almost inversely proportional to the number of queues employed. However, since all queues have to be checked to determine if the query is finished, it is not optimal to maximize the number of queues.

Furthermore, an additional 'low priority queue' is introduced. Whenever a node is a potential, but unlikely candidate for the results, it is inserted into this queue instead of the others. The heuristic for an unlikely candidate is one of two conditions: (1) its sibling node is known to be no candidate, i.e. the node in question is at the border of the region that has to be considered or (2) its sibling node has a lower minimum distance to the query point. When a thread retrieves a node from the queues, it first tries to poll a random queue. If that queue is empty, it searches all queues in order, ending with the low priority queue. This ensures that all normal queues are empty, before the unlikely candidates are considered.

While the search algorithm descends the tree, the minimum distance that a node has to have to be a candidate for the results converges. At the same time, the regions that can be excluded from the search (based on their minimum distance to the query point) get smaller. If only small regions of the data space can be excluded, these comparisons become inefficient and - eventually - an overhead. It is thus advantageous to fall back to a sequential search at a certain depth. The optimal solution is to perform this switch at the break-even-point when the cost to descend a node and its descendants becomes higher than the cost of comparing all points contained in those node sequentially. To support this approach, points are only stored in leaf nodes. Whenever a node is split during the creation of the tree, its points are equally distributed to both children. The split value is the median of the values of all points within the node in the dimension of the split. Side effects are a lower risk of degeneration and a simplified search algorithm (Figure 2).

The resulting index structure is similar to the K-D-B-Tree [Rob81]. The main difference is the size of its nodes - leaf nodes contain up to a user-defined number of points, whereas all other nodes have exactly two children. Additionally, the motivation for having nodes contain more than two points is not the usage of virtual memory, but rather avoiding a search overhead. The K-D-B-Tree was not designed to perform well in a main memory database and is consequently not used as a reference point for evaluation the modified *k*-d tree.

## 5 Empirical parameter optimization

The modifications to the *k*-d tree suggested in this paper require two parameters that were introduced previously: (1) the number of queues and (2) the size of leaf nodes. Their optimum values are dependent on the specific application. This test generates a set of default values that can be used if optimization is not possible or not feasible in consideration of the expected gains.

The experiments are conducted on random, equally distributed data sets with independent dimensions. They are 'neutral' in the sense that they contain neither correlations not clusters. Yet that feature does at the same time separate them from most real data sets.

For the tests, datasets of 100,000 points were generated. Each point had 10 dimensions. Both values were chosen as an average value, being neither particularly high nor low for a multidimensional database. Query points were generated the same way as the data points. They were used as query parameters for 5-NN-queries that were conducted in batches of 1,000 until the size of the confidence interval was smaller than a given threshold. This threshold was chosen to be 0.02 times the size of the mean value, with a confidence interval calculated using $\alpha = 0.05$.

The experiments were executed in 4 threads on a 4-core CPU. The time difference between starting the query and receiving the results was measured, that is to say the optimization target was the mean execution time.

For the number of queues, the optimum value was found to be around 2 to 3, plus the low priority queue (Figure 3). The results are also clearly confirm the advantage of the

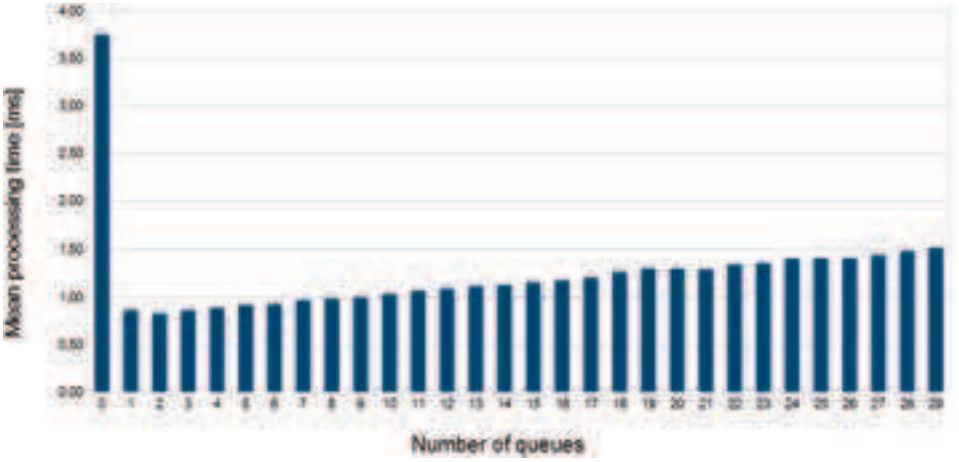



Figure 3: The influence of the number of queues on the performance. Numbers do not include the low priority queue, i.e. the case '0' uses only the low priority queue.

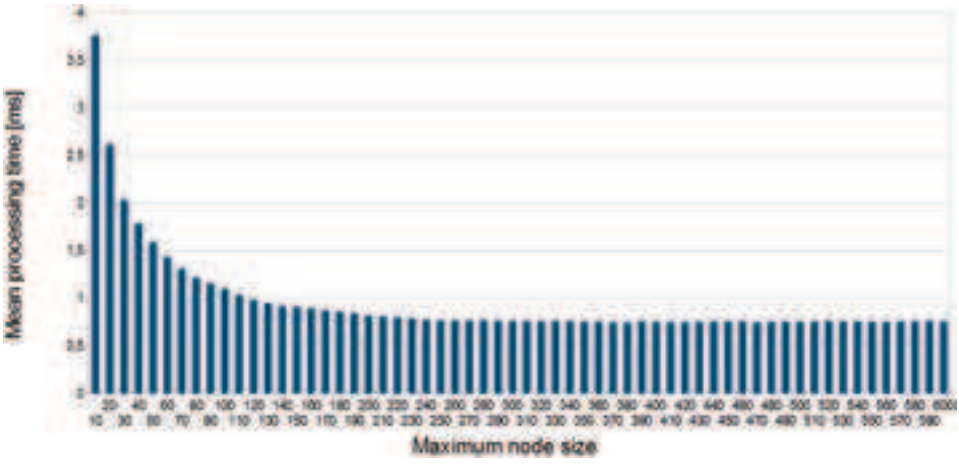



Figure 4: The influence of maximum size of a node on the performance.

separation into high and low priority queues, as the execution time with only one queue total is significantly worse than otherwise.

The optimum node size, which is to say the optimum switching point from descending the tree to a sequential search, was found to be at about 280 points (Figure 4). Consequently, a *k*-d tree for this scenario should have leaf nodes that contain 280 points each.

## 6 Evaluation

For evaluation purposes, the modified *k*-d tree is tested in comparison to the original *k*-d tree as well as a parallel sequential search. The testing process is the same as the one used in section 5. To find out how the index performances behave with different numbers of dimensions, dimensionality was varied from 1 to 20.

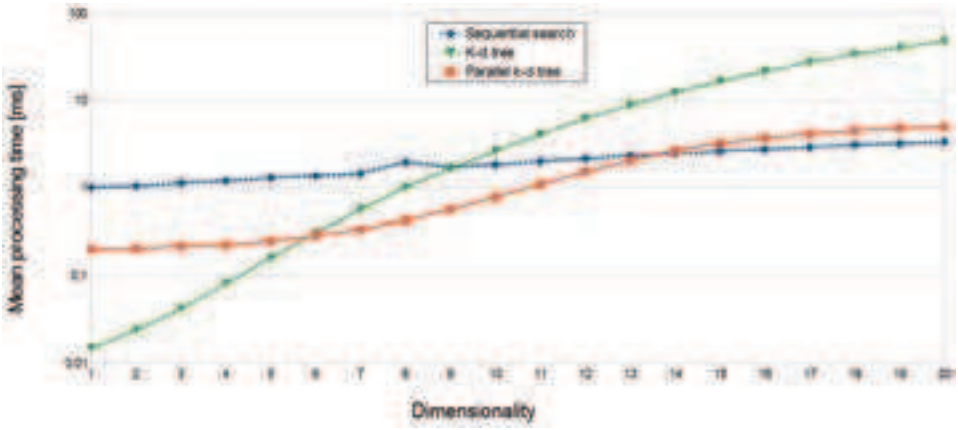



Figure 5: Performance of the performance of sequential search, *k*-d tree and modified *k*-d tree on datasets of varying dimensionality.

The results (Figure 5) show the dependency of the relative performances on the dimensionality. For low dimensionalities, the *k*-d tree is fastest, as it does not have as much of an overhead as the modified version. In the high dimensionality area, the kNN search algorithm for both versions of the *k*-d tree degenerate into a sequential search with an overhead. Nevertheless, the modified *k*-d stays relatively close to the performance of the sequential search. Between both areas, from 6 to 13 dimensions, the modified *k*-d tree is the fastest of the tested algorithms by a factor of up to two. It should also be noted that it is at least the second-best solution across the board, making it the stable choice for an unknown environment.

## 7 Conclusion and future work

A parallelized *k*-d tree implementing the modifications suggested in this paper is a powerful solution to the kNN problem in medium dimensionality data spaces (6-13 dimensions). Additionally, it degenerates slower than the *k*-d tree as the number of dimensions increases. This property makes it useful for unknown environments or 'one size fits all' solutions, where index structures might have to cope with unexpectedly high dimensionalities.

Moving forward, performance tests and parameter optimizations for real datasets are a short term goal. Farther out, the approach for parallelization implemented here can be transferred to other index structures. These could also include suitable disk-based index structures. Lastly, the utilization of GPUs - potentially in parallel to the CPU - is a tempting goal in spite of the difficulties.

## Acknowledgements

The work in this paper has been partially funded by the German Federal Ministry of Education and Science (BMBF) through the Research Program under Contract No. FKZ:13N10817.

## References

[AQSK98] Ning An, Liujian Qian, Anand Sivasubramaniam, and Tom Keefe. Evaluating parallel R-tree implementations on a network of workstations (an extended abstract). In *Proceedings of the 6th ACM international symposium on Advances in geographic information systems*, GIS '98, pages 159–160, New York, NY, USA, 1998. ACM.

[BBK98] Stefan Berchtold, Christian Böhm, and Hans-Peter Kriegal. The pyramid-technique: towards breaking the curse of dimensionality. In *Proceedings of the 1998 ACM SIGMOD international conference on Management of data*, SIGMOD '98, pages 142–153, New York, NY, USA, 1998. ACM.

[BBK01] Christian Böhm, Stefan Berchtold, and Daniel A. Keim. Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases. *ACM Comput. Surv.*, 33(3):322–373, September 2001.

[Ben75] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, September 1975.

[FBF77] 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, September 1977.

[GDB08] Vincent Garcia, Eric Debreuve, and Michel Barlaud. Fast k nearest neighbor search using GPU. In *Computer Vision and Pattern Recognition Workshops, 2008. CVPRW '08. IEEE Computer Society Conference on*, pages 1 –6, June 2008.

[Gut84] Antonin Guttman. R-trees: a dynamic index structure for spatial searching. In *Proceedings of the ACM Special Interest Group on Management of Data International Conference*, pages 47–57. ACM, 1984.

[IM98] Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the 30th annual ACM symposium on Theory of Computing*, pages 604–613. ACM, 1998.

[KCS+10] Changkyu Kim, Jatin Chhugani, Nadathur Satish, Eric Sedlar, Anthony D. Nguyen, Tim Kaldewey, Victor W. Lee, Scott A. Brandt, and Pradeep Dubey. FAST: fast architecture sensitive tree search on modern CPUs and GPUs. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, SIGMOD '10, pages 339–350, New York, NY, USA, 2010. ACM.

[KF92] Ibrahim Kamel and Christos Faloutsos. Parallel R-trees. *SIGMOD Rec.*, 21(2):195–204, June 1992.

[KSHL12] Thomas Kissinger, Benjamin Schlegel, Dirk Habich, and Wolfgang Lehner. KISS-Tree: smart latch-free in-memory indexing on modern architectures. In *Proceedings of the Eighth International Workshop on Data Management on New Hardware*, DaMoN '12, pages 16–23, New York, NY, USA, 2012. ACM.

[Rob81] John T. Robinson. The K-D-B-tree: a search structure for large multidimensional dynamic indexes. In *Proceedings of the 1981 ACM SIGMOD international conference on Management of data*, SIGMOD '81, pages 10–18, New York, NY, USA, 1981. ACM.

[RR98] Jun Rao and Kenneth A. Ross. Cache conscious indexing for decision-support in main memory. 1998.

[RR00] Jun Rao and Kenneth A. Ross. Making B+- trees cache conscious in main memory. *SIGMOD Rec.*, 29(2):475–486, May 2000.

[Smi82] Alan Jay Smith. Cache Memories. *ACM Computing Surveys*, 14(3):473–530, September 1982.

[SSK07] Maxim Shevtsov, Alexei Soupikov, and Alexander Kapustin. Highly Parallel Fast KD-tree Construction for Interactive Ray Tracing of Dynamic Scenes. In *Computer Graphics Forum*, volume 26, pages 395–404. Wiley Online Library, 2007.

[WB97] Roger Weber and Stephen Blott. An approximation based data structure for similarity search. *Report TR1997b, ETH Zentrum, Zurich, Switzerland*, 1997.

[ZHWG08] Kun Zhou, Qiming Hou, Rui Wang, and Baining Guo. Real-time KD-tree construction on graphics hardware. *ACM Transactions on Graphics*, 27(5):126:1–126:11, December 2008.