Distributed Kd-Trees for Retrieval from Very Large Image Collections

Mohamed Aly[1]
malaa@vision.caltech.edu
Mario Munich[2]
mario@evolution.com
Pietro Perona[1]
perona@caltech.edu

[1] Computational Vision Group, Caltech
Pasadena, CA 91125 USA
[2] Evolution Robotics
Pasadena, CA 91106 USA

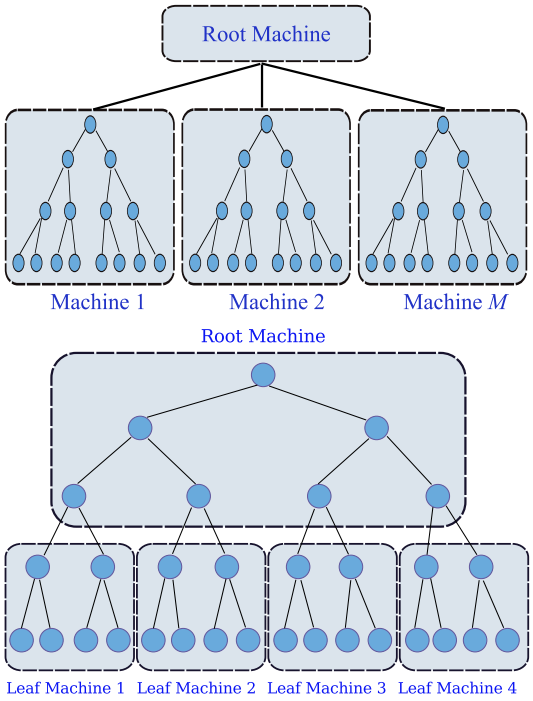

Figure 1: **Kd-Tree Parallelizations**. (*Top*) ***Independent Kd-Tree (IKdt)***. The image database is partitioned into M subsets, each of which resides on one machine that builds an independent Kd-Tree. At query time, all the machines search in parallel for the closest match. (*Bottom*) ***Distributed Kd-Tree (DKdt)***. A single Kd-Tree is built for all images. The *root* machine stores the top of the tree, while the *leaf* machines store the leaves of the tree. At query time, the root machine directs features to a subset of the leaf machines, which leads to higher throughput.

Large scale image retrieval is an important problem with many applications. There are two major approaches for building such databases: Bag of Words (BoW) [4, 5] and Full Representation (FR) [1, 3]. The first has the advantage of using an order of magnitude less storage, however its precision is far worse [1]. Moreover, most past research on large scale image search [1, 4] did not scale past a few million images on one machine because of RAM limits. Therefore, we focus on ways to distribute the image database on an arbitrary number of machines to be able to scale up the recognition problem to hundreds of millions of images using FR with Kd-Trees at its core.

In this paper, we implement and compare two approaches for Kd-Tree parallelizations using the MapReduce paradigm [2]: *Independent Kd-Trees* (IKdt) and *Distributed Kd-Trees* (DKdt) [1], see Fig. 1. We build upon our previous work in [1], and provide practical implementations of the two approaches to parallelize Kd-Trees. We run extensive experiments to assess the different system parameters and compare the two approaches with datasets of up to 100 million images on a computer cluster with over 2000 machines.

Independent Kd-Trees (IKdt): Divide the image database into subsets, where each subset can fit in the memory of one machine. Then each machine builds an independent Kdt for its own subset. A single root machine accepts the query image, and passes the query features to all the machines, which then query their own Kdt. The root machine then collects the results, counts the candidate matches, and outputs the final sorted list of images.

Distributed Kd-Trees (DKdt): Build a single Kdt for the entire database, where the top of the tree resides on a single machine, the root machine. The bottom part of the tree is divided among a number of leaf machines, which also store the features that end up in leaves in these parts. At query time, the root machine directs the search to a subset of the leaf machines depending on where features exit the tree on the root machine. The leaf machines compute the nearest neighbors within their subtree and send them back to the root machine, which performs the counting and outputs the final sorted list of images.

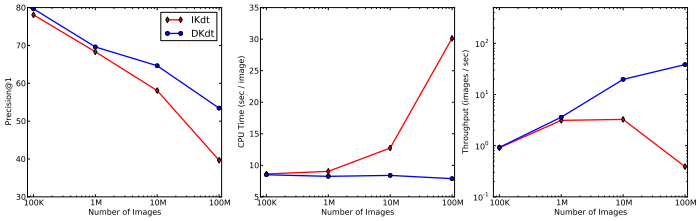

Figure 2: **Effect of Number Images**. The X-axis depicts the number of images in the database. The Y-axis depicts precision@1 (*left*), CPU time (*center*), and Throughput (*right*).

The two main challenges with DKdt are: (a) How to build the Kd-Tree that contains billions of features since it does not fit on one machine? (b) How to perform backtracking in this distributed Kdt? We solve these two problems by noticing the properties of Kd-Trees: (a) We do not build the Kdt on one machine, we rather build a feature "distributor", that represents the top part of the tree, on the root machine. Since can not fit all the features in the database in one machine, we simply subsample the features and use as many as the memory of one machine can take. This does not affect the performance of the resulting Kdt since computing the means in the Kdt construction algorithm subsamples the points anyway. (b) We only perform backtracking in the leaf machines, and not in the root. To decide which leaf machines to go to, we test the distance to the split value, and if it is below some threshold S_t, we include the corresponding leaf machine in the process.

Fig. 2 shows the effect of the number of images indexed in the database. We used 8 machines for 100K images, 32 for 1M images, 256 for 10M images, and 2048 for 100M images. DKdt clearly provides superior precision to IKdt, with lower CPU cost and much higher throughput. For 100M images, DKdt has precision about 32% higher than IKdt (53% vs 40%), with with throughput that's about 30 times that of IKdt (~ 12 images/sec vs. ~ 0.4) i.e. processes images in a fraction of a second. It is clear that by increasing the number of images, the precision goes down. Paradoxically, the throughput goes up with larger databases, and this is because we use more machines, and in the case of DKdt, this allows more interleaving of computation among the leaf machines and thus more images processed per second.

[1] Mohamed Aly, Mario Munich, and Pietro Perona. Indexing in large scale image collections: Scaling properties and benchmark. In *WACV*, 2011.

[2] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, 2004.

[3] David Lowe. Distinctive image features from scale-invariant keypoints. *IJCV*, 2004.

[4] J. Philbin, O. Chum, M. Isard, J. Sivic, and A. Zisserman. Object retrieval with large vocabularies and fast spatial matching. *CVPR*, 2007.

[5] J. Sivic and A. Zisserman. Video google: A text retrieval approach to object matching in videos. In *ICCV*, 2003.