Fast Matching Of Binary Features

By Marius Muja and David G. Lowe

Presented By: Patricia Oeking & David Zhang

Abstract

- Binary features: fast to compute, compact to store and efficient to compare with each other
- Can still be too slow to use linear search in the case of large datasets
- Introduction to a new algorithm for approximate matching of binary features, based on priority search of multiple hierarchical clustering trees

Related Work

- Salakhutdinov and Hinton: Introduction of the Notion of semantic hashing when they learn a deep graphical model that maps documents to small binary codes
- Torriba et al.: Learn compact binary codes from images with the goal of performing real time image recognition on a large dataset of images using limited memory
- Weiss et al.: Formalize the requirements for good codes and introduce a new technique for efficiently computing binary codes

Binary descriptors

- Binary descriptor composed of
  - A sampling pattern
  - Orientation compensation
  - Sampling pairs
- Recently proposed binary visual descriptors:
  - BRIEF
  - ORB
  - BRISK

Other methods: BRIEF

Randomly sample pair of pixels a and b.
1 if a > b, else 0. Store binary vector.

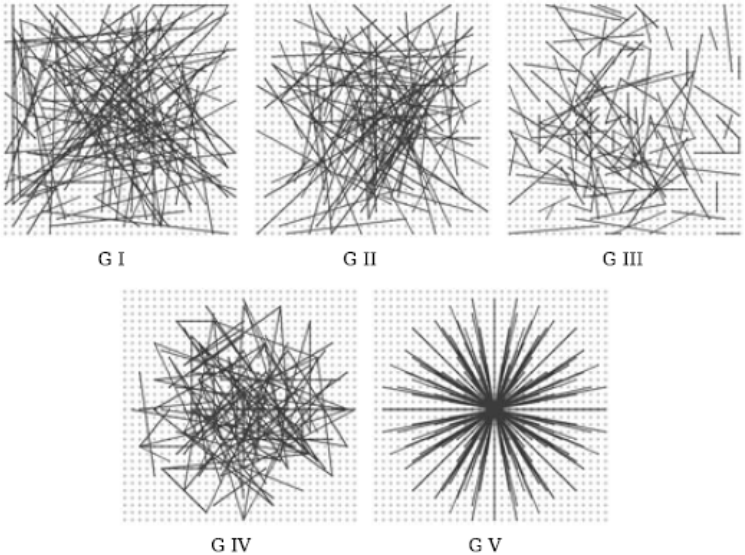


Fig. 2. Different approaches to choosing the test locations. All except the rightmost one are selected by random sampling. Showing 128 tests in every image.

BRIEF: binary robust independent elementary features, Calonder, V Lepetit, C Strecha, ECCV 2010

Slide of Larry Zitnick
(larryz@microsoft.com)

ORB and BRISK descriptors

- ORB:
  - Uses an orientation compensation mechanism, making it rotation invariant
  - learns the optimal sampling pairs, whereas BRIEF uses randomly chosen sampling pairs
- BRISK:
  - having a hand-crafted sampling pattern, composed out of concentric rings

Approximate nearest neighbour search

- Using linear search for matching can be practical only for smaller datasets
- Vector-based Features: SIFT and SURF, use of approximate nearest-neighbor search
  - SURF (Speeded up robust features)
    - relying on integral images for image convolutions
    - building on the strengths of the leading existing detectors and descriptors
    - simplifying these methods to the essential

Approximate matching algorithms

- For matching vector features:
  - kd-tree algorithm
  - hierarchical k-means tree
  - vocabulary tree
- Perform a hierarchical decomposition of the search space
- Not readily suitable for matching binary features
  - Assumes feature space can be continuously averaged

Approximate matching algorithms

- Matching binary features is mostly based on various hashing techniques
  - Locality Sensitive Hashing (LSH)
  - min-hash
  - Geometric Near-Neighbor Access Tree (GNAT)
- here: based on hierarchical decomposition of search space
- Implemented on FLANN

Algorithm Overview

“The algorithm performs a hierarchical decomposition of the search space by successively clustering the input dataset and constructing a tree in which every non-leaf node contains a cluster center and the leaf nodes contain the input points that are to be matched.”

Building the Tree

- Clustering similar to k-medoids in that cluster centers are input data points and not means
- Centers are randomly chosen, we are not trying to minimize the distance between cluster centers and their elements
  - Simpler and more efficient
  - Improves independence when using multiple trees in parallel
- Minimizing squared error and using the greedy approach in choosing cluster centers as in the GNAT tree did not improve performance

Algorithm 1 Building one hierarchical clustering tree

Input: features dataset D
Output: hierarchical clustering tree
Parameters: branching factor K, maximum leaf size S_L

1: **if** *size of* $D < S_L$ **then**
2: *create leaf node with the points in* D
3: **else**
4: $P \leftarrow$ *select* K *points at random from* D
5: $C \leftarrow$ *cluster the points in* D *around nearest centers* P
6: **for** *each cluster* $C_i \in C$ **do**
7: *create non-leaf node with center* P_i
8: *recursively apply the algorithm to the points in* C_i
9: **end for**
10: **end if**

Using Multiple Trees

- Searching multiple randomized trees has been successful for randomized kd-trees but not hierarchical k-means trees
- This algorithm also exhibits improved performance when using multiple trees
  - No further iterations to improve clustering
  - Worst case occurs when the closest neighbor to the query point lies across the boundary of the explored domain, resulting in backtracking
    - Trees are randomized enough so that the closest neighbor is likely to lie in different domains in different trees
    - Increases the likelihood that the closest neighbor is found quickly when trees are searched in parallel

Parallel Tree Search

- Starts with a single traverse of each tree
  - Always picking node closest to the query point and recursively exploring it
    - Adds unexplored nodes to a priority queue PQ.
  - At a leaf node all the points contained within are linearly searched and added to a priority queue R
- Search is continued by dequeuing from PQ the node that is closest to the query and resuming the traversal from there.
- Ends when the number of points examined exceeds a maximum limit, returning the K approximate nearest neighbors from R

Algorithm 2 Searching parallel hierarchical clustering trees

Input: hierarchical clustering trees T_i, query point Q
Output: K nearest approximate neighbors of query point
Parameters: max number of points to examine L_{max}

1: $L \leftarrow 0$ $\{L$ = *number of points searched*$\}$
2: $PQ \leftarrow$ *empty priority queue*
3: $R \leftarrow$ *empty priority queue*
4: **for** each tree T_i **do**
5: *call* TRAVERSETREE(T_i, PQ, R)
6: **end for**
7: **while** PQ *not empty and* $L < L_{max}$ **do**
8: $N \leftarrow$ *top of* PQ
9: *call* TRAVERSETREE(N, PQ, R)
10: **end while**
11: **return** K *top points from* R

procedure TRAVERSETREE(N, PQ, R)

1: **if** *node* N *is a leaf node* **then**
2: *search all the points in* N *and add them to* R
3: $L \leftarrow L + |N|$
4: **else**
5: $C \leftarrow$ *child nodes of* N
6: $C_q \leftarrow$ *closest node of* C *to query* Q
7: $C_p \leftarrow C \backslash C_q$
8: *add all nodes in* C_p *to* PQ
9: *call* TRAVERSETREE(C_q, PQ, R)
10: **end if**

Parallel Tree Search

- L limits the number of points examined
  - Determines degree of approximation
    - Higher L - more exact neighbors, but searching takes longer
  - Relationship between L and the desired search precision is determined empirically for each dataset using cross validation

Algorithm 2 Searching parallel hierarchical clustering trees

Input: hierarchical clustering trees T_i, query point Q
Output: K nearest approximate neighbors of query point
Parameters: max number of points to examine L_{max}

1: $L \leftarrow 0$ $\{L$ = *number of points searched*$\}$
2: $PQ \leftarrow$ *empty priority queue*
3: $R \leftarrow$ *empty priority queue*
4: **for** each tree T_i **do**
5: *call* TRAVERSETREE(T_i, PQ, R)
6: **end for**
7: **while** PQ *not empty and* $L < L_{max}$ **do**
8: $N \leftarrow$ *top of* PQ
9: *call* TRAVERSETREE(N, PQ, R)
10: **end while**
11: **return** K *top points from* R

procedure TRAVERSETREE(N, PQ, R)

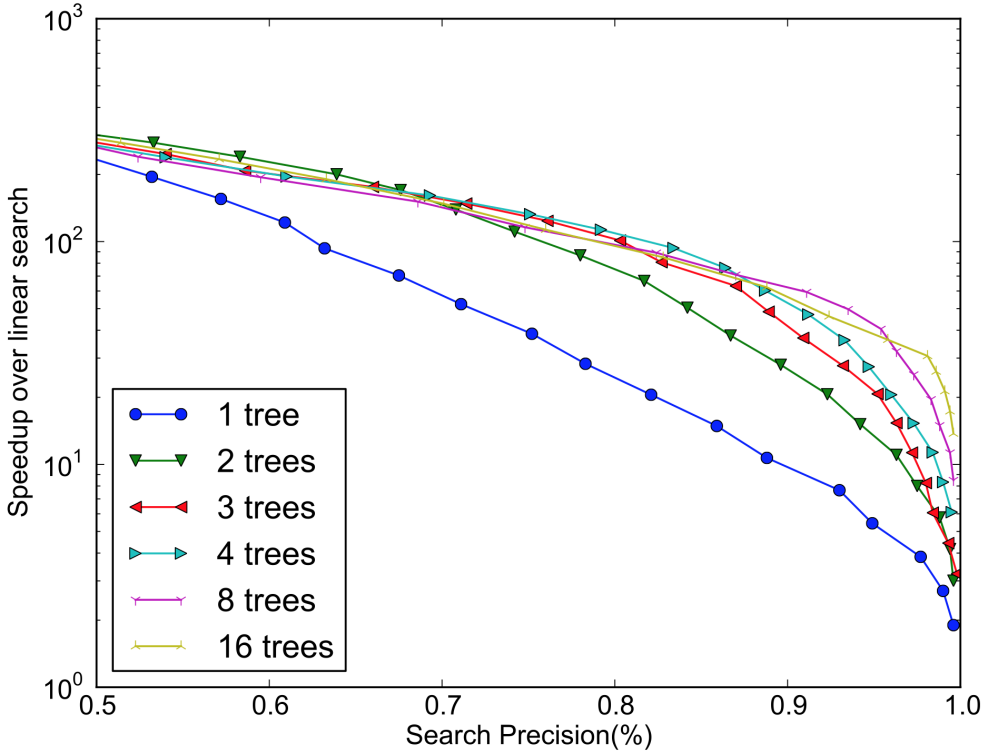
1: **if** *node* N *is a leaf node* **then**
2: *search all the points in* N *and add them to* R
3: $L \leftarrow L + |N|$
4: **else**
5: $C \leftarrow$ *child nodes of* N
6: $C_q \leftarrow$ *closest node of* C *to query* Q
7: $C_p \leftarrow C \backslash C_q$
8: *add all nodes in* C_p *to* PQ
9: *call* TRAVERSETREE(C_q, PQ, R)
10: **end if**

Performance Evaluation

- Speedup over linear search
- Analyze the effect of different parameters
- Compare with other approx-NN algorithms
- Dataset of ~310,000 BRIEF features

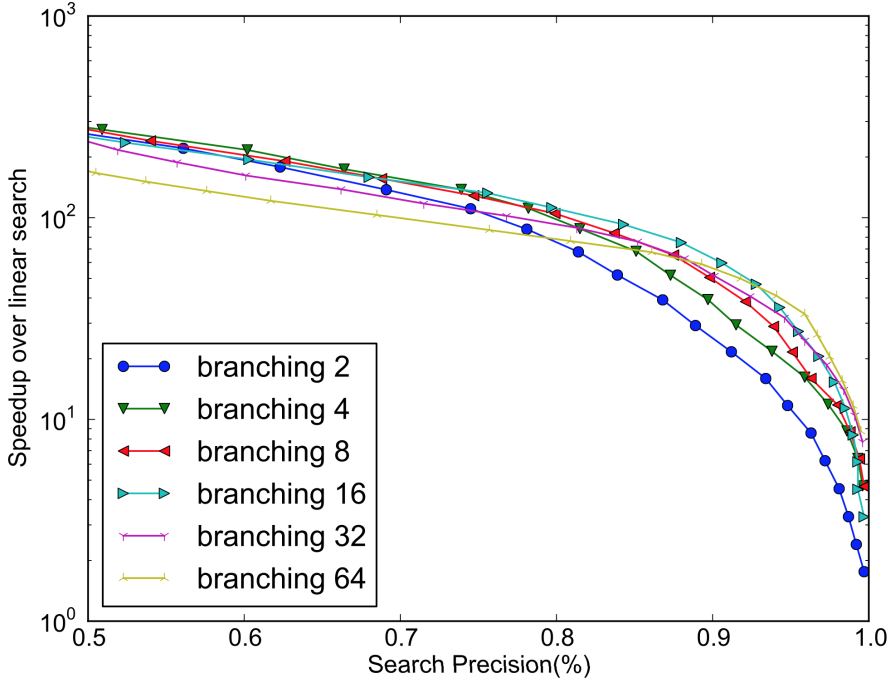
Evaluation: Number of Trees

- The optimum number of trees depends on the desired search precision
- More trees means more memory and longer build time, so the optimum configuration depends on real world constraints



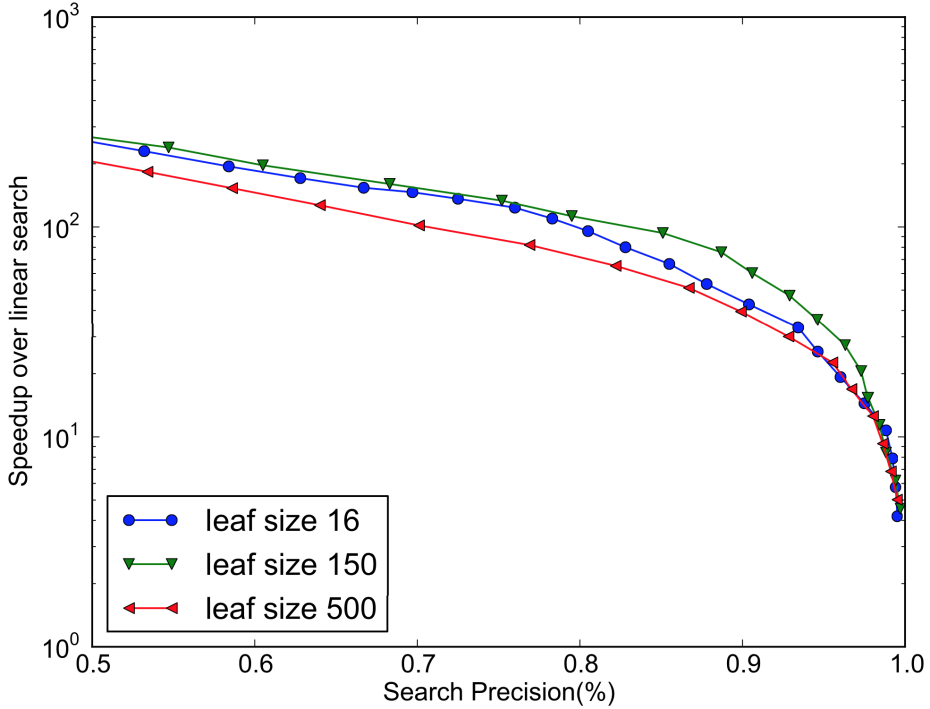
Evaluation: Branching Factor

- Higher branching factors perform better for high precisions (>80%)
    - Little gain for branching factors above 16 or 32
- Very large branching factors perform worse for lower precisions and have a higher tree build time.



Evaluation: Leaf Size

- Maximum leaf size of 150 performs better than a small leaf size (16, which is equal to the branching factor) or a large leaf size (500).
- Computing the distance between binary features is an efficient operation, so for small leaf sizes the overhead of traversing the tree to examine more leaves is greater
- Cost of linearly examining all the features in large leaves ends up being greater than the cost of traversing the tree.



Evaluation: Other Features/Algorithms

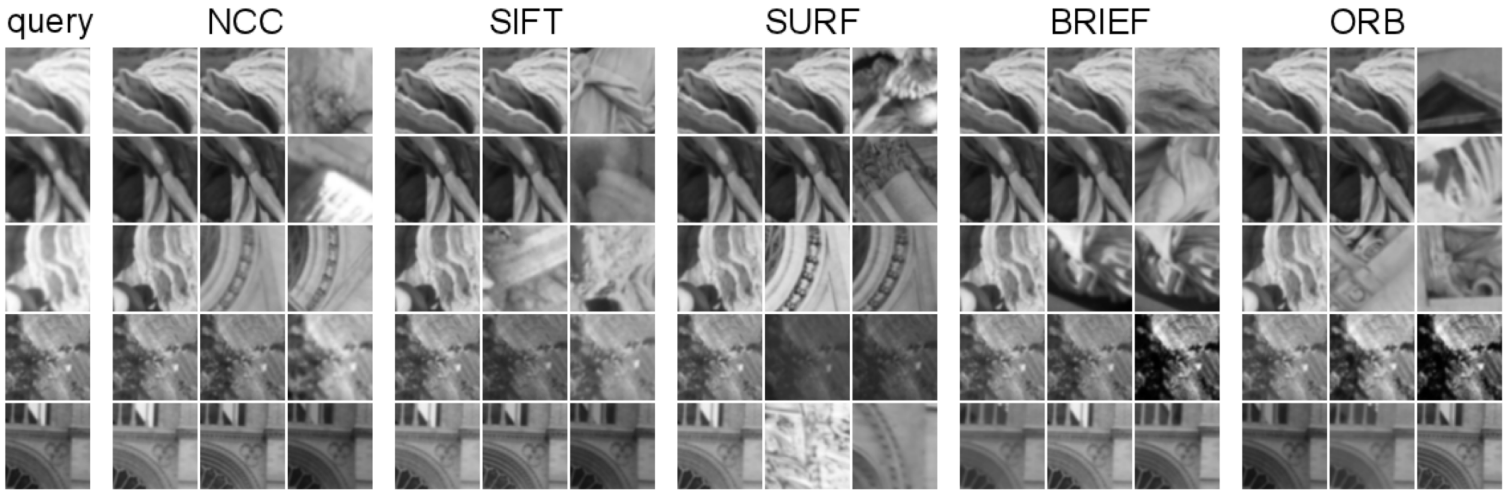
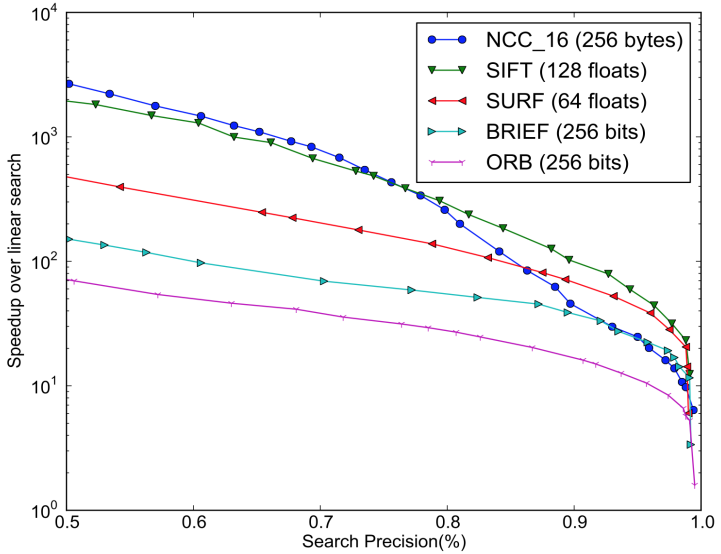
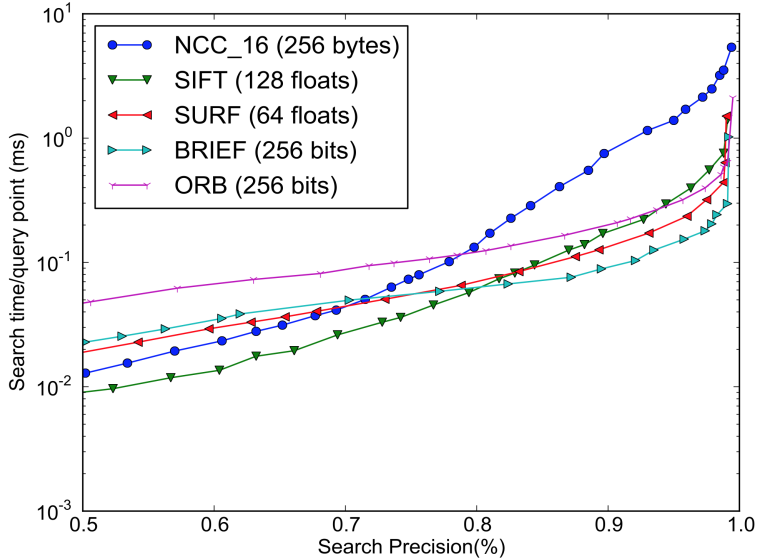


Figure 1. Random sample of query patches and the first three nearest neighbors returned when using different feature types

Evaluation: Other Features/Algorithms





Using best performing algorithm for each particular feature type (randomized kd-trees or hierarchical k-means for SIFT, SURF, NCC and our algorithm for BRIEF and ORB) using the optimum choice of parameters which maximizes the speedup for a given precision.

Evaluation: Other Features/Algorithms

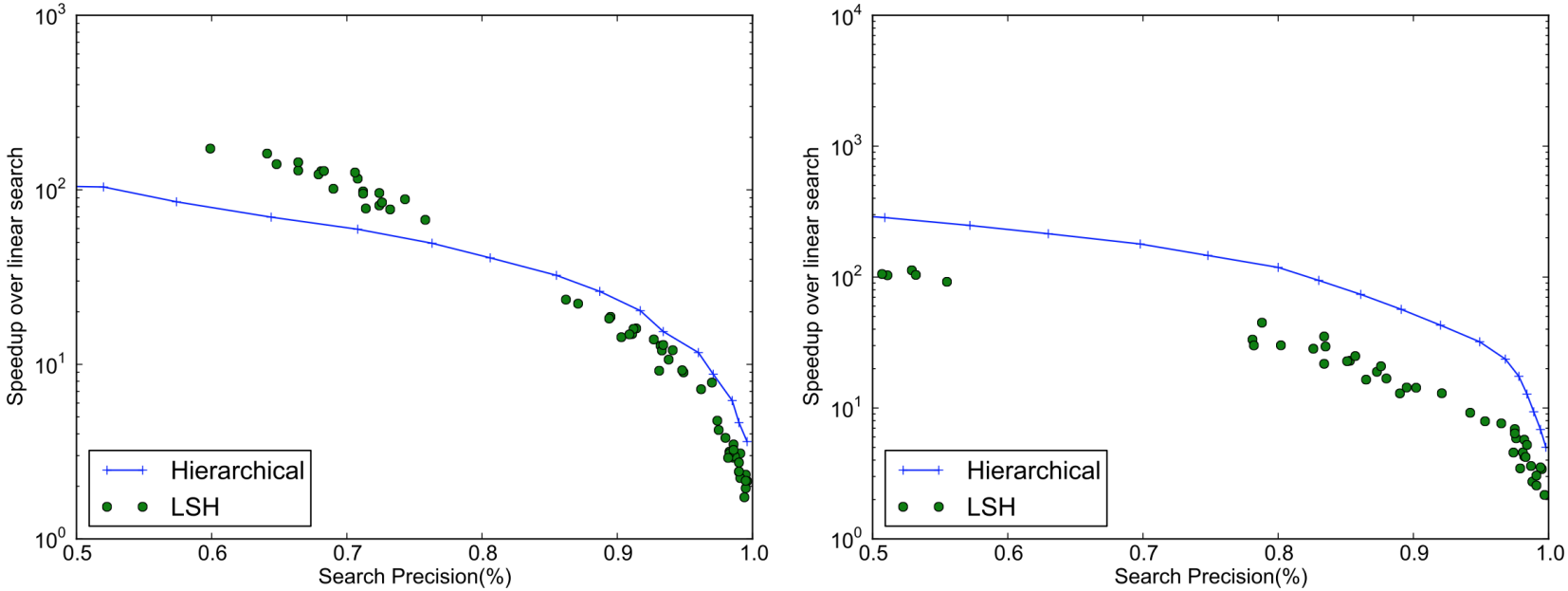
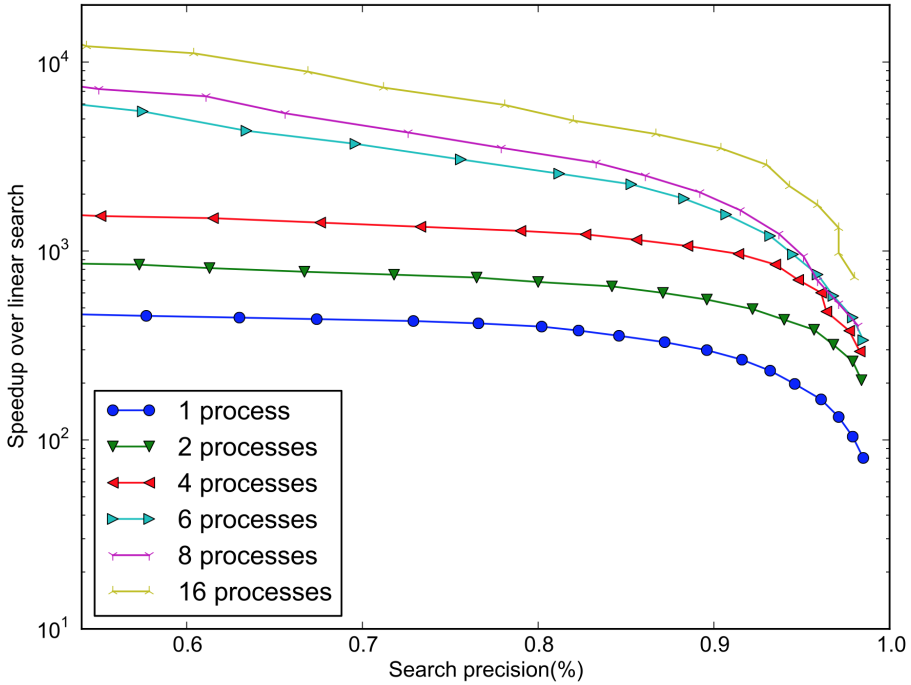


Figure 7. Comparison between the hierarchical clustering index and LSH for the Winder/Brown dataset of about 100,000 features(left) and the Nister/Stewenius recognition benchmark images dataset of about 5 million features(right)

Evaluation: Scalability

- Scales well across multiple computing clusters
- Additional benefit that the size of the dataset is not limited by the memory available on a single machine
    - Binary features are generally more compact, but for very large datasets the memory available on a single machine can become a limiting factor.
- FLANN uses a MapReduce algorithm to run the search on multiple machines and merges the search results at the end.



Conclusion

- New algorithm for fast approximate matching of binary features
- Using parallel searching of randomized hierarchical trees
- Algorithm is implemented on top of the publicly available FLANN open source library
- Simple to implement and efficient to run
- Very effective at finding nearest neighbors of binary features
- Performance of the algorithm is on par or better with that of LSH
- Scales well for large datasets