# Implementation and Analysis of Collision Counting LSH on The Husky Framework

Cheng, Ti-Chung
Chinese University of Hong Kong
Department of Computer Science and Engineering
tcheng@link.cuhk.edu.hk

## Abstract

Locality sensitive hashing (LSH) have been an important topic when it comes to approximation nearest neighbor search problems. Over the past decades, many algorithms has evolved from the original LSH. During this summer research project, we aim to understand the difference between E2LSH and C2LSH. Furthermore, we want to overcome the challenge of LSH on distributing systems and build C2LSH on open source distributed system framework: Husky. The implementation was then verified and analyzed over both algorithms.

## 1 Introduction

Nearest Neighbor Search has been an important question when it comes to image classification, voice recognition, protein structure prediction and item recommendation. However, when given a large set of data with high dimension, it is very difficult to have an efficient algorithm when encountering the curse of dimensionality. Over the last decade, researchers have developed several alternative algorithms to relieve such issue by using approximation methods, generally called c-approximate Nearest Neighbor Search (c-ANN search).

Locality Sensitive Hashing (LSH) [3] is well known as an efficient solution for finding approximate nearest neighbor in high dimensional space. It preprocessed the query and data set by creating a signature hash which allows similar items to have higher probability of being hashed together. Exact Euclidean LSH (E2LSH) package [1] aims to extend the limitation that LSH is originally set only to be used in hamming distance. Collision Counting LSH (C2LSH) [2] was then introduced to improve the limitation of E2LSH by which it would be able to probe dynamically to find nearest neighbors while decreasing the space complexity.

On the other hand, massive data sets cannot be processed immediately on a single machine, distributed systems provide a platform that clusters high computing resources. Husky [6] is an open source, efficient and expressive computing framework that provides the use of flexible APIs to complete the task of implementing distributed applications. Its generality serves the purpose of implementing C2LSH for this research project.

With the significant usage of C2LSH, I aim to implement and analysis this algorithm in a distributed manner within the period of this research project. In the following report, I will demonstrate the implementation concepts as well as the analysis between C2LSH and E2LSH.

## 2 Background

### 2.1 LSH

Locality Sensitive Hashing (LSH) was introduced by Indyk and Motwani [3] as follows:

**Definition 1** *Given a ball $B\ (q,r)$ centered at point $q$ with radius $r$. A LSH family $H = \{h : S \to U\}$ is $(r_1, r_2, p_1, p_2)$-sensitive under similarity* *measure D if for any* $q, p \in S$,

$$1. if\ o \in B(q, r_1),\ then\ P_r[h(o) = h(q)] \geq p_1$$
$$2. if\ o \in B(q, r_2),\ then\ P_r[h(o) = h(q)] \leq p_2$$

To generate a useful LSH family, it requires satisfaction of inequality $r_1 < r_2$ and $p_1 > p_2$. The definition then shows that when two items are similar, they should have a higher probability of being hashed towards the same bucket and vice versa. Many similarity measures include but not limited to Hamming distance, Jaccard distance and Euclidean distance follows such definition.

By intuition, LSH algorithm aims to hash data into buckets so that similar items are grouped upon themselves. That is, to give each data a signature value. To increase the probability for similar items to be hashed together, researchers proposed compounding k-hash functions instead of single hash function. L hash tables were also generated. This allows data with same k-signature vector in the same hash table Li to be hashed together. LSH is proven to achieve a sub-linear querying complexity.

Exact Euclidean LSH (E2LSH) package [1] provides a randomized solution for the high-dimensional NNS in the Euclidean space. The hash function is defined as the following:

$$h_o(v) = \lfloor \frac{\overrightarrow{a} \cdot \overrightarrow{o} + b}{w} \rfloor$$

In this hash function, *a* is a vector drawn randomly over a normal distribution. *a* shares the same dimension with *o*, representing our data. *b* is a real number drawn uniformly from $[0, w)$. $w$ is a user defined constant. To think of this hash function in a more intuitive manner, each data point mapped on a Euclidean Space is to be mapped onto a randomly selected line. This line is then segmented into segments of length w. The data points mapped onto the same segment is seen as mapped into the same bucket.

Based on such definition, E2LSH is able to return a result if a given query is able to find an existing data in the given range; otherwise it would return nothing instead of a nearest neighbor that NNS problem is trying to solve. Therefore, Collision Counting LSH (C2LSH) [2] was then introduced by to resolve the aforementioned issue of E2LSH. C2LSH inherits the hash function from E2LSH but adds an additional probing feature so that the program would not halt until a nearest neighbor is found. The hash function is defind as follows:

$$H^R(o) = \lfloor \frac{h(o)}{R} \rfloor$$

This hash function is defined as $(R, cR, p_1, p_2)$ - sensitive, and R is an integer power of c. Intuitively, the motivation behind this design is to increase the width of each segment so that gradually it would probe the segments near the original target.

### 2.2 Husky

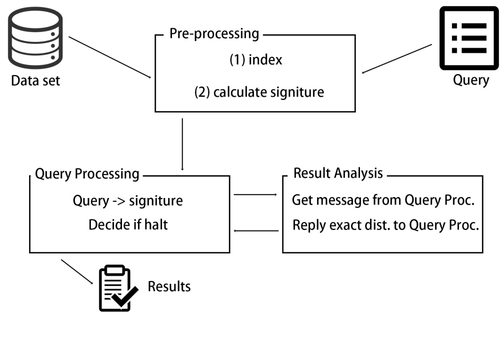
With the explosion of data, distributed systems such as Spark, Flink and Dryad offers frameworks for programmers to implement distributed applications. Husky is an open source project aimed to provide the platform that stands between general-purpose systems and domain-specific systems that leads to the freedom for developers to easily implement complex systems in a much simple way. Husky is able to abstract data into meaningful objects. With concepts such as Map-reduce and Pregel, Husky also emphasised on internal interaction protocol among objects. Objects are designed to execute commands, send and receive messages and other operations.

## 3 Implementation

### 3.1 Implementation Framework

The challenge of implementing C2LSH on a distributed system is to overcome the difficulty of maintaining the hash table for each element while allowing the query to probe the buckets efficiently in order to retrieve the final results. Therefore, we staged the program into three segments: (1) Pre-processing, (2) Query-processing and (3) Result analysis. (Firgure 1) Given that Husky is designed with cooperation between different objects, we can intuitively transform each query and the data within the target data set as respective objects during the Pre-processing phase. By transforming them into objects, we are able to give them an index id. The target data will also be given its signature hash value through the hash function in this stage. During the Query-processing stage, we compute the hash value for each query object so that we know which hash signature to probe. With the `send_message()` API call within the Husky framework, we would be able to send the details of the query data by specifying the exact hash value(s) (buckets we want to look for) we pre-computed for the dataset data. Given the sender, the dataset data would be able to reply the actual results and thus entering the final state: result analysis. Result analysis will decide if the system would halt or would it acquire more buckets to be probe since we are looking for the k-NN of each query. If so, the program would re-enter the Query-process state, otherwise it would output the results.

Figure 1. C2LSH implementation



With such design, no physical hash tables were to be maintained and only necessary communication between objects has to be used. Thus resolves the challenge of implementing a distributed C2LSH.

### 3.2 Implementation Details

Let us first revisit the concept of C2LSH. Since E2LSH is able to approximately preserve distance of the data when we map them to buckets. C2LSH inherits the idea and extends it to probe nearby buckets as C2LSH believes that similar items would also be mapped relatively to their projections.To present this probing feature, we keep track of two pointers that indicate the left most bucket and the right most bucket we have probed in each iteration. During each iteration of Query-processing, we calculate the signature(s) according to these two pointers. The pointers would update after each iteration if there are less then $k$ nearest neighbors obtained. A sorting would be done before the final output of the results to trim the additional nearest neighbors we retrieved from the last iteration.

## 4 Results

### 4.1 Data set and setup

In the experiment we use real data set provided by the TEXMEX [4, 5] research team. This data set is generated especially for A-NNS studies. It consists of 1 million entries of image data each with 128 dimensions. During our testing we run base vectors in the data set over itself, which is 1 million queries over 1 million data. The Husky implementation is setup across one master machine and 20 worker machines. The following experiments would specify the number of machines used respectively. Hadoop file-system is used as data storage system.

### 4.2 Experiment

To verify the implementation of C2LSH, we use the aforementioned set of data and compute using E2LSH. Understanding that E2LSH is the base case of C2LSH in given conditions, we verified that the results from both algorithms after the first iteration is indeed identical, thus C2LSH is implemented correctly. To further test our motivation that to implement C2LSH on a distributed framework is to make use of the growing computation power to supplement the large amount of data we have today. In the traditional setting of C2LSH under a single machine single thread situation. It took over two hours to complete the query the above data set. Th following chart shows the time it took for single machine single thread, single machine multi-thread, 5 machine multi-thread, 10 machine multi-thread and 20 machine multi-thread. We can see that even if we were able to implement a single machine multi-thread C2LSH, a distributed version still ran threee times faster under the same setting. (Chart 1)

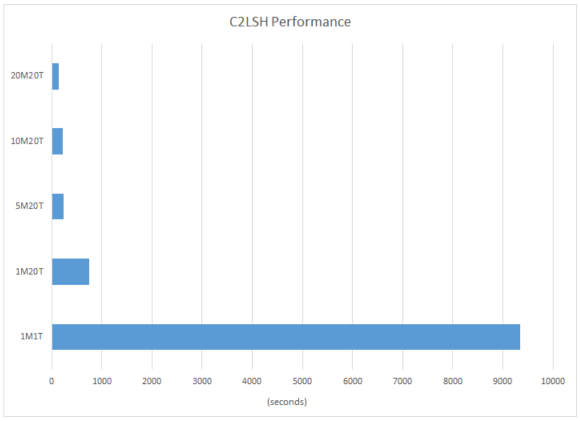


Chart 1. C2LSH performance comparison

On the other hand, to verify the concept of how C2LSH is able to prob more efficiently when it comes to k-NN search, we decide to run an experiment to see the returned result of a E2LSH under a $k = 5$ setting. We use all 20 worker machines running 20 threads each for this experiment. By setting the same width $w$ for C2LSH and E2LSH, we confirm that (1) each query only returns a limited amount of results for E2LSH and (2) it is difficult to obtain a large k-NN search unless fine tuning the width $w$ or other user defined parameters for many times. From the result in Chart 2, we can see that over half of the query finds itself as the only nearest neighbor. Only 1% of the query is able to return the nearest five neighbors for E2LSH indicating that if users want to use E2LSH to complete k-ANN search, many trials would be required for which C2LSH can be easily done.

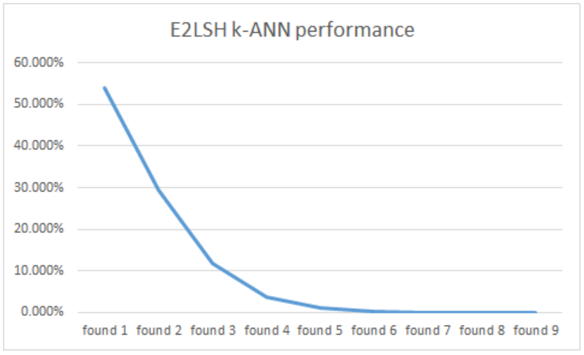


Chart 2. C2LSH and E2LSH under k-ANN search

## 5 Discussion and Conclusion

LSH algorithms has been widely used in industry and research. Leveraging the ability of distributed systems with high computing resources, the combination between both of them allows even more computing ability. From the experiment, we first verified our motivation to combine the ability of LSH and distributed systems. Later we verified that C2LSH on Husky guarantees to return the result and saves users time from fine tuning the parameter for any k-NN search. We not only believe that many topics can be experimented using LSH, but also with more understanding of the algorithm together with other machine learning techniques we would be able to develop better algorithms.

## 6 Acknowledgments

Beside my adviser Prof.James Cheng, I would like to give special thanks to my tutor Mr.Jinfeng Li. Without his patience and guidance it would be nearly impossible for me to complete this research project. Gratitude should also be given to those who finished the implementation of Husky.

## References

[1] A. Andoni and P. Indyk.

[2] J. Gan, J. Feng, Q. Fang, and W. Ng. Locality-sensitive hashing scheme based on dynamic collision counting. In *SIGMOD*, pages 541–552, 2012.

[3] P. Indyk and R. Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *Proceedings of the Thirtieth Annual ACM Symposium on the Theory of Computing, Dallas, Texas, USA, May 23-26, 1998*, pages 604–613, 1998.

[4] H. Jégou, M. Douze, and C. Schmid. Product quantization for nearest neighbor search. *IEEE Trans. Pattern Anal. Mach. Intell.*, 33(1):117–128, 2011.

[5] H. Jegou, R. Tavenard, M. Douze, and L. Amsaleg. Searching in one billion vectors: Re-rank with source coding. In *ICASSP*, pages 861–864, 2011.

[6] F. Yang, J. Li, and J. Cheng. Husky: Towards a more efficient and expressive distributed computing framework. *PVLDB*, 9(5):420–431, 2016.