Large-Scale Distributed Locality-Sensitive Hashing for General Metric Data

Eliezer Silva[2], Thiago Teixeira[1], George Teodoro[1], and Eduardo Valle[2]

[1] Dep. of Computer Science, University of Brasilia thiagotei@gmail.com, teodoro@cic.unb.br
[2] RECOD Lab. / DCA / FEEC / UNICAMP ⋆
{eliezers,dovalle}@dca.fee.unicamp.br

Abstract. Locality-Sensitive Hashing (LSH) is extremely competitive for similarity search, but works under the assumption of uniform access cost to the data, and for just a handful of dissimilarities for which locality-sensitive families are available. In this work we propose Parallel Voronoi LSH, an approach that addresses those two limitations of LSH: it makes LSH efficient for distributed-memory architectures, and it works for very general dissimilarities (in particular, it works for all metric dissimilarities). Each hash table of Voronoi LSH works by selecting a sample of the dataset to be used as seeds of a Voronoi diagram. The Voronoi cells are then used to hash the data. Because Voronoi diagrams depend only on the distance, the technique is very general. Implementing LSH in distributed-memory systems is very challenging because it lacks referential locality in its access to the data: if care is not taken, excessive message-passing ruins the index performance. Therefore, another important contribution of this work is the parallel design needed to allow the scalability of the index, which we evaluate in a dataset of a thousand million multimedia features.

1 Introduction

Content-based Multimedia Information Retrieval (CMIR) is an alternative to keyword-based or metadata-based retrieval. CMIR works by extracting *features* from the multimedia content itself, and using those feature to represent the multimedia objects. The features capture perceptual characteristics (color, texture, motion, etc.) from the documents, helping to bridge the so-called "semantic gap": the disparity between the amorphous low-level multimedia coding (e.g., image pixels or audio samples) and the complex high-level tasks (e.g., classification or retrieval) performed by CMIR engines.

That way, searching for similar multimedia documents becomes the more abstract operation of finding the closest features in the feature space — an operation known as similarity search. CMIR systems are usually complex and may include several phases,

⋆ E. Silva was supported by a CAPES scholarship. All authors thank to CAPES, CNPq (including project 2012/14: Examind) and FAPESP (including FAPESP-JP: 2010/05647-4) for financial support to this research. We also thank the XSEDE for granting us access to the Keeneland cluster, where the large-scale experiments where conducted.

but similarity search will often be a critical step. Similarity search for multimedia descriptors is a complex problem that has been extensively studied for decades [1]. Challenges of this task include (i) the very large and increasing volume of data to be indexed/searched; (ii) the high dimensionality of the multimedia descriptors; (iii) the diversity of dissimilarity functions employed [2].

Among indexing algorithms for efficient searching in high-dimensional datasets, *Locality-Sensitive Hashing* (LSH) [3–5] deserves attention as one of the best performing and most cited algorithms in the literature [3–5]. LSH uses functions that hash together, with higher probability, points in the space that are close to each other. For architectures with uniform access cost to the data, and for a handful of dissimilarities (mainly the Hamming, Euclidean and Manhattan metrics) LSH will often be the technique with best compromise between precision and speed. However, the need to discover a new family of locality-sensitive hashing functions for each dissimilarity function precludes the generalization of LSH; in addition, the poor referential locality of LSH makes its adaptation to distributed-memory systems — essential for scalability — very challenging.

In this work, we address those two shortcomings with *Parallel Voronoi LSH*, which extends LSH for very general dissimilarities (in particular, any metric distances), and for distributed-memory architectures, enabling the index to scale-up to huge datasets. Each hash table of Voronoi LSH works by selecting a sample of the dataset to be used as the seeds of a Voronoi diagram. The Voronoi cells are then used to hash the data. Because Voronoi diagrams depend only on distances, the technique is very general.

Parallel Voronoi LSH builds upon previous works in literature that employ, implicitly or explicitly, the notion of Voronoi cells in order to perform similarity search using the principles of locality-sensitive hashing [6–8]. We aim at tackling a large class of dissimilarity functions (including all metric distances), and at scaling-up the index to very large corpora. As far as we know, our work is the first to tackle at once both problems of general metric spaces and very large scales (thousand of millions) for locality-sensitive hashing. An additional contribution of this work is the evaluation of adaptive locality-sensitive functions for metric data. First proposed for Euclidean data [6], in this work we adapt them for general metric data, and measure the impact of chosing adaptive functions versus random ones.

2 Background

A comprehensive review of similarity search for multimedia would include hundreds of papers and is beyond the scope of this work. For a starting point, the reader is referred to [1, 9]. In this section, we focus on the key papers of Locality-Sensitive Hashing and some recent developments. Since our focus is on practical index design instead of theoretical computational geometry, we review less the recent theoretical advances of LSH, and more the papers with a practical/experimental slant.

LSH relies on the existence of families H of locality-sensitive hashing functions, to map points from the original space into a set of hash keys, such that points that are near in the space are hashed to the same value with higher probability than points that are far apart. The seminal work on LSH [3] proposed locality-sensitive hashing families for Hamming distances in Hamming spaces, and for Jacquard indexes in spaces of sets. Later, extensions for L_1-normed (Manhattan) and L_2-normed (Euclidean) spaces were proposed by embedding those spaces into Hamming spaces [4]. The practical success of LSH, however, came with E2LSH[3] (Exact Euclidean LSH) [5], which proposed a new family of locality-sensitive functions, based upon projections onto random lines, which worked "natively" for Euclidean spaces.

LSH works by boosting the locality sensitiveness of the hash functions. This is done by building from the original $\{h_i \in H\}$ locality-sensitive function family, a family $\{g_j \in H\}$, where each g_j is the concatenation of M randomly sampled h_i, i.e., each g_j has the form $g_j(\mathbf{v}) = (h_1(\mathbf{v}), ..., h_M(\mathbf{v}))$. Then, we sample L such functions g_j, each to hash an independent hash table. As M grows, the probability of a false positive (points that are far away having the same value on a given g_j) drops sharply, but so grows the probability of a false negative (points that are close having different values). But as L grows and we check all hash tables, the probability of false negatives falls, and the probability of false positives grows. LSH theory shows that it is possible to set M and L so to have a small probability of false negatives, with an acceptable number of false positives. That allows the correct points to be found among a small number of candidates, dramatically reducing the number of distance computions needed to answer the queries.

The need to maintain and query L independent hash tables is the main weak point of LSH. In the effort to keep both false positives and false negatives low, there is an "arms race" between M and L, and the technique tends to favor large values for those parameters. The large number of hash tables results in excessive storage overheads. Referential locality also suffers, due to the need to random-access a bucket in each of the large number of tables. More importantly, it becomes unfeasible to replicate the data on so many tables, so each table has to store only pointers to the data. Once the index retrieves a bucket of pointers on one hash table, a cascade of random accesses ensues to retrieve the actual data.

Multiprobe LSH [10] considerably reduces the number of hash tables of LSH, by proposing to visit many buckets on each hash table. It analyses the relative position of the query in the boundaries of the hash functions of E2LSH to estimate the likelihood of each bucket to contain relevant points. *A posteriori LSH* [11] extends that work by turning the likelihoods into probabilities using priors estimated from training data. Those works reduce the storage overhead of LSH at the cost of greatly increasing the number of random accesses to the data.

2.1 Unstructured Quantizers, General Spaces

In *K-means LSH* [6], the authors address LSH for high-dimensional Euclidean spaces, introducing the idea of hash functions adapted to the data, generated by running a K-means or hierarchical K-means on a sample, using the centroids obtained as seeds to a Voronoi diagram. The Voronoi diagram becomes the hash function (each cell induces a hash value over the points it contains). They call those hash functions *unstructured*

[3] *LSH Algorithm and Implementation (E2LSH).* `http://www.mit.edu/~andoni/LSH/`

quantizers, in contrast to the regular hash-functions (intervals on projections over random lines, cells on lattices), which are blind to the data, that they call *structured quantizers*. Their experimental results show that the data-adapted functions perform better than the data-blind ones.

DFLSH (Distribution Free Locality-Sensitive Hashing) [7], works on the same principle of Voronoi-diagram induced hash functions, but instead of applying the K-means, randomly chooses the centroids from the dataset. The advantage of the scheme is generality: while the averaged centroids of K-means LSH imply an Euclidean (or at the very least coordinate) space, DFLSH work for any space in which Voronoi diagrams work.

Another LSH technique for general metric spaces is based on *Brief Proximity Indexing (BPI)* [8]. Similarity is inferred by the perspective on a group of points called *permutants* (if point p sees the permutants in the same order as point q, p and q are likely to be close to each other). This is similar to embedding the data into a space for which LSH is available and then applying a LSH function for that space. Indeed the method consists of those two steps: first it creates a permutation index; and then it hashes the permutation indexes using LSH for Hamming spaces [3].

M-Index [12] is a Metric Access Method for exact and approximate similarity search constructed over a universal mapping from the original metric space to scalars values. The values of the mapping are affected by the permutation order of a set of reference points and the distance to these points. In a follow-up work [13], this indexing scheme is analyzed empirically as a locality-sensitive hashing for general metric spaces.

Works on generalizing LSH for all metric spaces have focused more on proposing practical techniques and show that they work empirically rather than in proving theoretically that the scheme strictly follows the axioms of locality sensitiveness as proposed by Indyk and Motwani [3]. K-means LSH offers no theoretical analysis. In [7] a proof sketch showing that the hashing family is locality-sensitive is presented, but only for Euclidean spaces. Permutation-based index employs Hamming spaces, whose locality-sensitive family was proved to be so in the seminal LSH paper, but they offer no formal proof that the embedding they propose into Hamming spaces preserves the metric of the original space.

In this paper we combine and extend the efforts of those previous works. As in DFLSH we employ Voronoi cells over points chosen from the dataset, allowing us to tackle very general dissimilarity functions[4]. As in K-means LSH, we are interested in evaluating the impact of the adaptation to the data in the performance of the technique. Unlike existing art, we are concerned on how to scale-up the index to *very* large datasets. In the next two sections we will describe the basic technique, and the distributed implementation proposed to tackle those large collections.

[4] A precise characterization of which dissimilarities are compatible with our technique (and with DFLSH, for that matter) seems very complex, but it is sure that it includes all metric distances (e.g. Euclidean and Manhattan distance on coordinate spaces, edit distance on strings, etc.), and all non-decreasing functions of metric distances (e.g., squared Euclidean distance).

3 Voronoi LSH and Parallel Voronoi LSH

Each hash table of Voronoi LSH employs a hash function induced by a Voronoi diagram over the data space. If the space is known to be Euclidean (or at least coordinate) the Voronoi diagram can use as seeds the centroids learned by a Euclidean clustering algorithm, like K-means (in which case Voronoi LSH coincides with K-means LSH). However, if nothing is known about the space, points from the dataset must be used as seeds, in order to make as few assumptions as possible about the structure of the space. In the latter case, randomly sampled points can be used (in which case Voronoi LSH coincides with DFLSH). However, it is also possible to try select the seeds by employing a distance-based clustering algorithm, like K-medoids, and using the medoids as seeds.

Neither K-means nor K-medoids are guaranteed to converge to a global optimum, due to dependencies on the initialization. For Voronoi LSH, however, obtaining *the* best clustering is not a necessity. Moreover, when several hash tables is applied, the difference on the clustering results between runs is an advantage, since it is one of the sources of diversity among the hash functions (Figure 1). We further explore the problem of optimizing clustering initialization below.

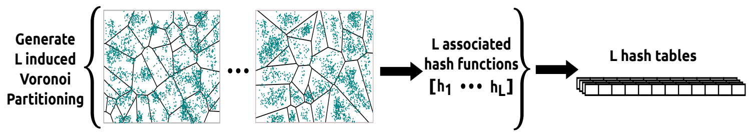


Fig. 1: Each hash table of Voronoi LSH employs a hash function induced by a Voronoi diagram over the data space. Differences between the diagrams due to the data sample used and the initialization seeds employed allow for diversity among the hash functions.

We define more formally the hash function used by Voronoi LSH in Equation 1. In summary, it computes the index of the Voronoi seed closest to a given input object. In addition, we present the indexing and querying phases of Voronoi LSH, respectively, in Algorithms 1 and 2. Indexing consists in creating L lists with k Voronoi seeds each ($C_i = \{c_{i1}, \ldots, c_{ik}\}, \forall i \in \{1, \ldots, L\}$). When using K-medoids, which is expensive, we suggest the Park and Jun fast K-medoids algorithm (apud [6]), performing this clustering over a sample of the dataset, and limiting the number of iterations to 30. Then, for each point in the dataset, the index stores a reference in the hash table T_i ($i \in \{1, \ldots, L\}$), using the hashing function defined in Equation 1 ($h_{C_i}(x)$). When using K-means, we suggest the K-means++ variation [14, 15]. The seeds can also simply be chosen at random (like in DFLSH). The querying phase is conceptually similar: the same set of L hash functions ($h_{C_i}(x)$) is computed for a query point, and all hash tables are queried to retrieve the references to the candidate answer set. The actual points are retrieved from the dataset using the references, forming a candidate set (shortlist), and the best answers are selected from this shortlist.

In this work, we focus on k-nearest neighbor queries. Therefore, this final step consists of computing the dissimilarity function to all points in the shortlist and selecting the k closest ones.

We mention, in passing, that several variations of K-medoids clustering exist, from the traditional PAM (Partitioning Around Medoids) [16] to the recently proposed FAMES [17]. We chose the method of Park and Jun [18] due to its simplicity of implementation and good speed. K-medoid is expensive (even more so than K-means, which is already not cheap), but Park and Jun restrict the search for new medoid candidates to other points already assigned to cluster, making it much faster. Still, our technique can in principle be employed with the other variations, if desired.

Definition 1 *Given a metric space (U,d) (U is the domain set and d is the distance function), the set of Voronoi seeds $C = \{c_1, \ldots, c_k\} \subset U$ and an object $x \in U$:*

$$\begin{gathered} h_C : U \to \mathbb{N} \\ h_C(x) = \operatorname{argmin}_{i=1,\ldots,k}\{d(x,c_1), \ldots, d(x,c_i), \ldots, d(x,c_k)\} \end{gathered} \tag{1}$$

```
input : Set of points $X$, number of hash tables $L$, size of the sample set $S$ and
        number of Voronoi seeds $k$
output: list of $L$ index tables $T_1, \ldots, T_L$ populated with all points from $X$ and list
        of $L$ Voronoi seeds $C_i = \{c_{i1}, \ldots, c_{ik}\}, \forall i \in 1, \ldots, L$
for $i \leftarrow 1$ to $L$ do
    Draw sample set $S_i$ from $X$;
    $C_i \leftarrow$ choose $k$ seeds from sample $S_i$ (random, K-means, K-medoids, etc.);
    for $x \in X$ do
        $T_i[h_{C_i}(x)] \leftarrow T_i[h_{C_i}(x)] \cup \{$pointer to $x\}$;
    end
end
```

Algorithm 1: Voronoi LSH indexing phase: a Voronoi seed list (C_i) is independently selected for each of the L hash tables used. Further, each input data point is stored in the bucket entry ($h_{C_i}(x)$) of each hash table.

```
input : Query point $q$, index tables $T_1, \ldots, T_L$, $L$ lists of $M$ Voronoi seeds each
        $C_i = \{c_{i1}, \ldots, c_{ik}\}$, number of nearest neighbors to be retrieved $N$
output: set of $N$ nearest neighbors $\mathrm{NN}(q,N) = \{n_1, \ldots, n_N\} \subset X$
CandidateSet $\leftarrow \emptyset$ ;
for $i \leftarrow 1$ to $L$ do
    CandidateSet $\leftarrow$ CandidateSet $\cup\ T_i[h_{C_i}(q)]$ ;
end
$\mathrm{NN}(q,N) \leftarrow \{$k closest points to q in CandidateSet $\}$;
```

Algorithm 2: Voronoi LSH querying phase: the input query point is hashed using same L functions as in indexing phase, and points in colliding bucket in each hash table are used as nearest neighbors candidate set. Finally, N closest points to the query are selected from the candidate set.

Initialization. K-means and K-medoids strongly depend on the initial centroid/medoid selection. K-means++ [14, 15] solve the $O(\log k)$-approximate K-means problem by carefully choosing those initial centroids. Moreover, because K-means++ initialization employs only distance information and sampling, it can be transposed to

K-medoids. Park and Jun [18] also propose a special initialization for their fast K-medoids algorithm, based on a distance-weighting scheme. We evaluated both of those special initializations, as well as random initialization.

3.1 Parallelization strategy

The parallelization strategy is based on the dataflow programming paradigm. Dataflow applications are typically represented as a set of computing *stages*, which are connected to each other using directed *streams*.

Our parallelization decomposes Voronoi LSH into five computing stages organized into two conceptual pipelines, which execute the index building and the search phases of the application. All stages may be replicated in the computing environment to create as many copies as necessary. Additionally, the streams connecting the application stages implement a special type of communication policy referred here as *labeled-stream*. Messages sent through a labeled-stream have an associated label or tag, which provides an affordable scheme to map message tags to specific copies of the receiver stage in a stream. We rely on this communication policy to partition the input dataset and to perform parallel reduction of partial results computed during a query execution. The data communication streams and processes management are built on top of Message Passing Interface (MPI).

The index building phase of the application, which includes the Input Reader (IR), Bucket Index (BI), and Data Points (DP) stages, is responsible for reading input data objects and building the distributed LSH indexes that are managed by the BI and the DP stages. In this phase, the input data objects are read in parallel using multiple IR stage copies and are sent (1) to be stored into the DP stage (message i) and (2) to be indexed by the BI stage (message ii). First, each object read is mapped to a specific DP copy, meaning that there is no replication of input data objects. The mapping of objects to DPs is carried out using the data distribution function *obj_map* (labeled-stream mapping function), which calculates the specific copy of the DP stage that should store an object as it is sent through the stream connecting IR and DP. Further, the pair <object identifier, DP copy in which it is stored> is sent to every BI copy holding buckets into which the object was hashed. The distribution of buckets among BI stage copies is carried out using another mapping function: *bucket_map*, which is calculated based on the bucket value/key. Again, there is no replication of buckets among BIs and each bucket value is stored into a single BI copy. The *obj_map* and *bucket_map* functions used in our implementation are modulo operation based on the number of copies of the receiver in a stream. We plan to evaluate other hashing strategies in the future.

The index construction is very compute-intensive, and involves many distance calculations between the input data objects and the Voronoi seeds. For Euclidean data, we implemented a vectorized code using Intel SSE/AVX intrinsics to take advantage of the wide SIMD instructions of current processors. Preliminary measurements have shown that the use of SIMD instructions sped-up the index building 8 times.

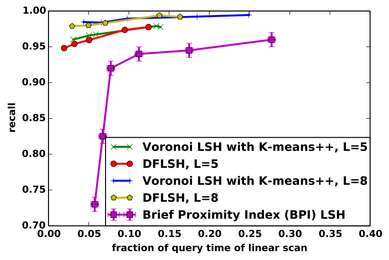
The search phase of the parallel LSH uses four stages, two of them shared with the index building phase: Query Receiver (QR), Bucket Index (BI), Data Points (DP), and Aggregator (AG). The QR stage reads the query objects and calculates the bucket values into which the query is hashed for the L hash tables. Each bucket value computed for a query is mapped to a BI copy using the *bucket_map* function. The query is then sent to those BI stage copies that store at least one bucket of interest (message iii). Each BI copy that receives a query message visits the buckets of interest, retrieves the identifier of the objects stored on those buckets, aggregates all object identifiers to be sent to the same DP copy (list(obj_id)), and sends a single message to each DP stage that stores at least one of the retrieved objects (message iv). For each message received by a DP copy, it calculates the distance from the query to the objects of interest, selects the k-nearest neighbors objects to the query, and sends those local NN objects to the AG stage. Finally, the AG stage receives the message containing the DPs local NN objects from all DPs involved in that query computation and performs a reduction operation to compute the global NN objects. The DP copies (message v) use the query_id as a label to the message, guaranteeing that the same AG copy will process all messages related to a specific query. As a consequence, multiple AG copies may be created to execute different queries in parallel. Although we have presented the index building and the search as sequential phases for sake of simplicity, their executions may overlap.

The parallelization approach we have proposed exploits task, pipeline, replicated and intra-stage parallelism. Task parallelism results from concurrent execution that allows indexing and searching phases to overlap, e.g. during an update of the index. Pipeline parallelism occurs as the search stages, for instance, execute different queries in parallel in a pipeline fashion. Replicated parallelism is available in all stages of the application, which may have an arbitrary number of copies. Finally, intra-stage parallelism results of the application's ability to use multiple cores within a stage copy. This parallelism has the advantages of sharing the same memory space among computing cores in a stage copy, and a reduced number of messages exchanged, since a smaller number of state partitions may be used.

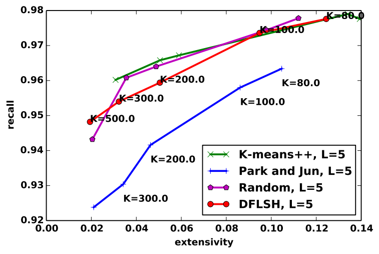
4 Experimental Evaluation

Datasets: two datasets were used: the English dictionary of strings with Levenshtein distance from SISAP Metric Library [19]; and BigANN [20]. The English dataset has 69,069 strings, 500 strings are randomly removed from the set to serve as query dataset. The BigANN contains a thousand million (10^9) 128-dimensional SIFT local feature vectors extracted from images, and 10,000 query feature vectors. Euclidean distance is used for SIFT. We perform k-NN searches, with k=5, 10, and 30 for the Dictionary dataset, and k=10 for the BigAnn dataset.

Metrics: We employ the *recall* as a metric of quality, and the *extensiveness* as metric of cost for the techniques. The recall is defined as usual for information retrieval, as the fraction of relevant answers that was effectively retrieved. The *extensiveness* metric is the fraction of the dataset selected into the shortlist for linear scan. As the indexes work by selecting a (hopefully small) fraction of the dataset as candidates, and then computing the actual distance to the query for those candidates, the size of the shortlist corresponds to the number of distances computed for that query. The related *selectivity* metric ($selectivity = 1 - extensivity$) has distinct (and sometimes conflicting) use in the database and image retrieval research communities: some authors used it as a syn-



(a)



(b)

Fig. 2: (a) Voronoi LSH recall–cost compromises are competitive with those of BPI (error bars are the imprecision of *our interpretation* of BPI original numbers). To make the results commensurable, time is reported as a fraction of brute-force linear scan. (b) Different choices for the seeds of Voronoi LSH: K-medoids with different initializations (K-means++, Park & Jun, random), and using random seeds (DFLSH). Experiments on the English dictionary dataset.

onymous for extensiveness, while others and we use it as a complementary notion (as selectivity grows, extensivity drops). We also employ the *query runtime* as metric of cost. Runtimes are difficult to compare across the literature, due to differences in code optimization, programming language, and execution environment, but in controlled experiments like ours they can offer a perspective on the cost of different techniques.

4.1 Comparison of Voronoi LSH and BPI-LSH

We compared our Voronoi LSH with Brief Permutation Indexing (BPI) LSH [8] in a sequential (non-distributed) environment. We implemented our Voronoi LSH on Java with The Apache Commons Mathematics Library[5]. Also included is Distribution-Free LSH (DFLSH) [7], which we evaluate as a specific configuration of our implementation of Voronoi LSH with the seeds of the Voronoi diagram chosen at random.

In order to allow the comparison with BPI LSH, we followed an experiment protocol as close as possible to the one described in their paper, and used the recall and query runtimes reported there. In order to remove the effects of implementation details and execution environment differences, we normalize both our times and theirs by the runtime of the brute-force linear scan. Because the authors report their numbers graphically, we added error bars to indicate the imprecision of *our* reading of their results.

The experiments used the English Dictionary dataset. Figure 2 summarizes the results. We compare the impact of the Clustering algorithm initialization to the search quality of the K-medoids nearest neighbors results. For sake of this analysis, the initialization strategy proposed in K-means++ and in the work of "Park and Jun" and the naive random initialization are considered. Figure 2b presents the recall and query time for varying number of centroids. As shown, the K-means++ initialization and the

[5] *Commons Math: The Apache Commons Mathematics Library.* `http://commons.apache.org/proper/commons-math/`

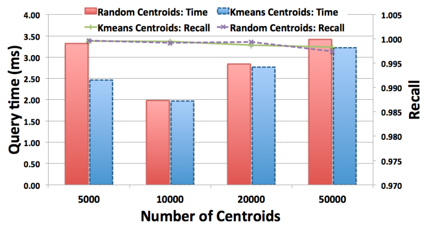


Fig. 3: Random seeds vs. K-means++ centroids on Parallel Voronoi LSH (BigAnn). Results reported are averaged over 10,000 queries. Well-chosen seeds have little effect on recall, but occasionally lower query times.

random initialization performance are not very distinguishable. On the other hand, the initialization of Park and Jun is clearly poorer than the others.

4.2 Large-Scale Distributed Voronoi LSH

The large-scale evaluation used a distributed-memory machine with 70 nodes interconnected through a FDR Infiniband switch. Each computation node was equipped with a dual-socket Intel E5 2.60 GHz Sandy Bridge processor with a total of 16 CPU cores, 32 GB of DDR3 RAM, running Linux OS kernel version 2.6.32. Parallel Voronoi LSH was implemented on C++ and MPI.

The Figure 3 shows the query phase execution time and recall of parallel Voronoi LSH using BigANN dataset with Voronoi seeds chosen at random, and by using K-means++ centroids. Although the difference in recall is neglectable, using K-means++ centroids in general resulted in better query times than using random seeds. The best execution time in both cases was with 10,000 Voronoi seeds. For minimizing the distance computations, there is an expected compromise between too few seeds (cheap hash functions, large shortlists due to more points in buckets) and too many (costly hash functions, smaller shortlists). The theoretical sweet spot is obtained with $\sqrt{n}$ seeds, where n is the dataset size (around 30,000 for BigAnn). However, in our tests, the empirical best value was always much smaller than that, favoring, thus, the retrieval of fewer buckets with many points in them, instead of many buckets with fewer points. That suggests that the cost of accessing the data is overcoming the cost of computing the distances.

Our parallelism efficiency analysis employs a scale-up (weak scaling) experiment in which the reference dataset and the number of computing cores used increase proportionally. A scale-up evaluation was selected because we expect to obtain an abundant volume of data for indexing, which would only fit in a distributed system. For each CPU core allocated to the BI stage 4 CPU cores are assigned to the DP stage, resulting in a ratio of computing cores by BI:DP of 1:4. A single core is used for the AG stage.

The efficiency of the parallel Voronoi LSH is presented in Figure 4. As the number of CPU cores and nodes used increase, the application achieves a very good parallel efficiency of 0.9 when 801 computing cores are used (10 nodes for BI and 40 nodes for DP), indicating very modest parallelism overheads. The high efficiency attained is a result of (i) the application asynchronous design that decouples communication from computation tasks and of (ii) the intra-stage parallelization that allows for a single multi-

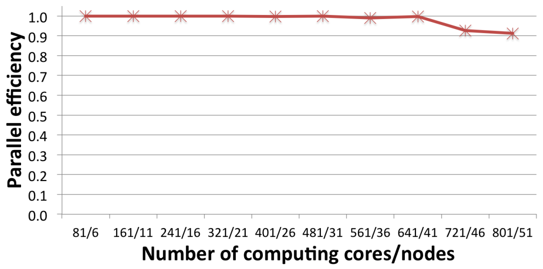


Fig. 4: Efficiency of Parallel Voronoi LSH parallelization as the number of nodes used and the reference dataset size increase proportionally. The results show that the parllelism scheme scales-up well even for a very large dataset, with modest overhead.

threaded copy of the DP stage to be instantiated per computing node. As a consequence, a smaller number of partitions of the reference dataset are created, which reduces the number of messages exchanged by the parallel version (using 51 nodes) in more than $6\times$ as compared to an application version that instantiates a single process per CPU computing core.

5 Conclusion

Efficient large-scale similarity search is a crucial operation for Content-based Multimedia Information Retrieval (CMIR) systems. But because those systems employ high-dimensional feature vectors, or other complex representations in metric spaces, providing fast similarity search for them has been a persistent research challenge. LSH, a very successful family of methods, has been advanced as a solution to the problem, but it is available only for a few distance functions. In this article we propose to address that limitation, by extending LSH to general metric spaces, using a Voronoi diagram as basis for a LSH family of functions. Our experiments show that employing Voronoi diagrams to index the data works well both for metric and for Euclidean data. The experiments do not show any clear advantage in learning the seeds of the Voronoi diagram by clustering: a random choice seems to work just as well. The lack of effect of clustering on recall is somewhat disappointing, and must be confirmed by evaluating a more diverse selection of datasets. However, if confirmed, it will also be an important hint for scalability, since learning the seeds by clustering is expensive. On the other hand, clustering might in some cases affect *query times*, which is surprising. This seems to be due to a more uniform partition of the data, since random seeds tend to create an unbalanced distribution of the dataset on the buckets of the hash tables. The large-scale experiments show that our proposed parallelization has very modest overhead and scales-up well even for a very large collection.

As a future work, we would like to explore very large collections of non-Euclidean metric data. This is currently a challenge because creating an exact ground truth for such corpora is very expensive.

References

1. Chávez, E., Navarro, G., Baeza-Yates, R., Marroquín, J.L.: Searching in metric spaces. **33**(3) (September 2001) 273–321
2. Akune, F., Valle, E., Torres, R.: MONORAIL: A Disk-Friendly Index for Huge Descriptor Databases. In: 20th Int. Conf. on Pattern Recognition, IEEE (August 2010) 4145–4148
3. Indyk, P., Motwani, R.: Approximate nearest neighbors: towards removing the curse of dimensionality. In: Proc. of 13th ann. ACM Symp. on Theory of Comp. (1998) 604–613
4. Gionis, A., Indyk, P., Motwani, R.: Similarity search in high dimensions via hashing. In: Proc. of the 25th Int. Conf. on Very Large Data Bases. (1999) 518–529
5. Datar, M., Immorlica, N., Indyk, P., Mirrokni, V.S.: Locality-sensitive hashing scheme based on p-stable distributions. In: Proc. of the 20th ann. Symp. on Computational Geometry. (2004) 253
6. Paulevé, L., Jégou, H., Amsaleg, L.: Locality sensitive hashing: A comparison of hash function types and querying mechanisms. **31**(11) (August 2010) 1348–1358
7. Kang, B., Jung, K.: Robust and Efficient Locality Sensitive Hashing for Nearest Neighbor Search in Large Data Sets. In: NIPS Workshop on Big Learning (BigLearn), Lake Tahoe, Nevada (2012) 1–8
8. Tellez, E.S., Chavez, E.: On locality sensitive hashing in metric spaces. In: Proc. of the Third Int. Conf. on Similarity Search and Applications. SISAP '10, New York, NY, USA, ACM (2010) 67–74
9. Zezula, P., Amato, G., Dohnal, V., Batko, M.: Similarity Search: The Metric Space Approach. Volume 32 of Advances in Database Systems. Springer (2006)
10. Lv, Q., Josephson, W., Wang, Z., Charikar, M., Li, K.: Multi-probe LSH: efficient indexing for high-dimensional similarity search. In: Proc. of the 33rd Int. Conf. on Very large data bases. VLDB '07, VLDB Endowment (2007) 950–961
11. Joly, A., Buisson, O.: A posteriori multi-probe locality sensitive hashing. In: Proc. of the 16th ACM Int. Conf. on Multimedia. MM '08, New York, NY, USA, ACM (2008) 209–218
12. Novak, D., Batko, M.: Metric Index: An Efficient and Scalable Solution for Similarity Search. In: 2009 Second Int. Workshop on Similarity Search and Applications, IEEE Computer Society (August 2009) 65–73
13. Novak, D., Kyselak, M., Zezula, P.: On locality-sensitive indexing in generic metric spaces. In: Proc. of the Third Int. Conf. on Similarity Search and Applications - SISAP '10, New York, New York, USA, ACM Press (2010) 59–66
14. Ostrovsky, R., Rabani, Y., Schulman, L., Swamy, C.: The Effectiveness of Lloyd-Type Methods for the k-Means Problem. In: focs, IEEE (December 2006) 165–176
15. Arthur, D., Vassilvitskii, S.: K-means++: the advantages of careful seeding. In: Proc. of the 18th annual ACM-SIAM Symp. on Discrete Algorithms. SODA '07, Philadelphia, PA, USA (2007) 1027–1035
16. Kaufman, L., Rousseeuw, P.J.: Finding Groups in Data: An Introduction to Cluster Analysis. 9th edn. Wiley-Interscience, New York, USA (March 1990)
17. Paterlini, A.A., Nascimento, M.A., Junior, C.T.: Using Pivots to Speed-Up k-Medoids Clustering. **2**(2) (June 2011) 221–236
18. Park, H.S., Jun, C.H.: A simple and fast algorithm for K-medoids clustering. **36**(2) (2009) 3336–3341
19. Figueroa, K., Navarro, G., Chávez, E.: Metric spaces library (2007) Available at `http://www.sisap.org/Metric_Space_Library.html` .
20. Jegou, H., Tavenard, R., Douze, M., Amsaleg, L.: Searching in one billion vectors: Re-rank with source coding. In: ICASSP, IEEE (2011) 861–864