Metric Suffix Array For Large-Scale Similarity Search

Hisham Mohamed [*]
Université de Genève
Route de Drize 7
Geneva, Switzerland
hisham.mohamed@unige.ch

Stéphane Marchand-Maillet [†]
Université de Genève
Route de Drize 7
Geneva, Switzerland
stephane.marchand-maillet@unige.ch

ABSTRACT

We propose the Metric Suffix Array (MSA), as a novel and efficient data structure for permutation-based indexing. The Metric Suffix Array follows the same principles as the suffix array. The suffix array is mainly used for text indexing. Here, we build the MSA as an alternative for large-scale content based information retrieval. We also show how the MSA is scalable for parallel and distributed architectures. We study the performance and efficiency of our algorithms in a large-scale context. Experimental results show fast response time with high efficiency and effectiveness.

Keywords

Metric Suffix Array, Permutation-Based Indexing, Similarity Search, Large-Scale Multimedia Indexing, Distributed indexing

1. INTRODUCTION

Nowadays, with the tremendous increase in the volume of data, the *exact match* scenario for answering users' queries is not commonly used anymore. Many applications require the most similar results to a certain query and not only the exact match. Examples are text plagiarism to track the similarity between an article against a database of texts, multiple genome comparison to find all the similarities between one or more genes, and multimedia retrieval to find the most similar picture or video to a given example. The similarity search paradigm [11] is thus applicable for these applications.

For a query q and a data collection D, similarity search sorts all the data items in D by their similarity to q according to a given distance function $d : D \times D \longrightarrow \Re$. There are two common search scenarios, namely the K-nearest neighbor search (K-NN) and the range query. The K-NN retrieves the K-top most relevant objects to the query. The range query retrieves all the objects located within a distance range from the query. Several techniques have been developed for improving the performance of similarity search [18], but due to the curse of dimensionality [16], the scalability of similarity search for high-dimensional data is still limited. One of the most promising routes is to perform approximate similarity search [10, 15]. This offers solutions to improve the retrieval of high dimensional data at the price of a reduced precision.

One of the recent techniques for approximate similarity search is the *permutation-based indexing* [8, 1]. In this work, we propose a novel data structure for handling *permutation-based indexes*, inspired by the principle of suffix arrays. Our new data structure requires only half the space for each reference point, when compared to generally available data structures for permutation-based indexing [1, 13, 4, 5]. We also show that it is easy to implement and it is scalable. To validate our claims, we tested MSA for searching within high dimensional large datasets containing millions of objects.

The rest of the paper is organized as follows. In the next section, a review of the related work and a brief background about *permutation-based indexes* and *suffix arrays* is given. In section 3, we detail the main ideas behind our proposed structure and algorithms. In section 4, we detail the parallel and distributed implementations of our algorithms. We finally present our results in section 5 and conclude in section 6.

2. RELATED WORK AND DEFINITIONS

2.1 Prior Work

Several works were proposed to speed up the permutation-based indexing using various techniques. Amato and Savino [1] propose metric inverted files (MIF), which is a data structure to store the permutations based on inverted files. Figuerroa et al. [6] speed up the distance calculation between the objects by indexing the permutation relative to their distance from the reference points. Mohamed and Marchand-Maillet [13] proposed a distributed implementation of the metric inverted files, through three levels of parallelization. They were able to achieve high speed up comparing to the sequential implementation using 40 cores. In [17], authors propose the *brief permutation index* which is a technique to reduce memory usage and speed up the distance calculation. The main idea is to encode the permutation as a binary vector and to compare these vectors using the Hamming distance. In [4, 5], authors proposed the *prefix permutation index*, with a parallel implementation. This PP-Index stores the prefix of the permutations only and measure the similarity between objects based on the length of its shared prefix. Novak et.al. [14] proposed the M-Index algorithm. M-Index maps the objects to a numeric domain. This is done by selecting number of pivots as reference points, which represent the object. The distance values $d(.,.)$ between the objects and the reference points are then normalized by a constant value which is greater than the

[*]Viper group, computer science department

[†]Viper group, computer science department

LSDS-IR '13 El Rome, Italy

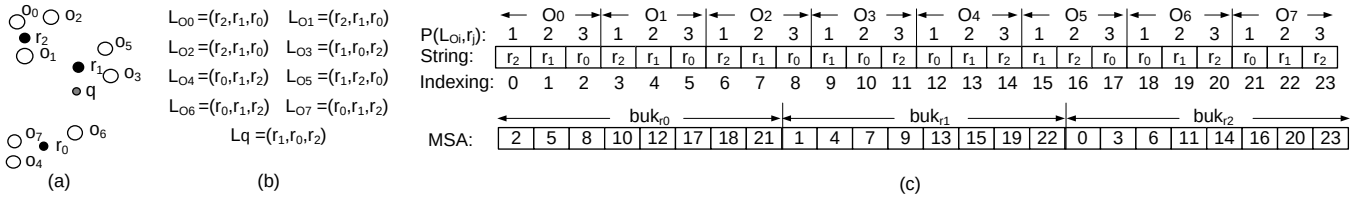

Figure 1: a) White circles are data objects o_i; black circles are reference objects r_j; the gray circle is the query object q b) Ordered lists for all the objects L_{o_i}. c) Example of Metric Suffix Array

maximum distance d between any two objects in the data domain.

2.2 Permutation-based Indexing

The idea of the *permutation-based indexes* is to predict the proximity between elements according to how they order their distances towards a set of reference objects [8, 1]. Given a collection of N objects o_i, which are located in a domain $D = \{o_0 \dots o_N\}$, and a distance function $d : D \times D \to \Re$ between the objects. We assume that the distance function $d(.,.)$ follows the metric space postulates [18] $\forall o_i, o_j, o_k \in D$: *(identity)* $o_i = o_j \iff d(o_i, o_j) = 0$, *(non-negativity)* $d(o_i, o_j) \geq 0$, *(symmetry)* $d(o_i, o_j) = d(o_j, o_i)$ and *(triangle inequality)* $d(o_i, o_k) \leq d(o_i, o_j) + d(o_j, o_k)$.

A set of n reference objects $R = \{r_0, r_1, \dots r_n\} \subset D$ is selected from D. Each object $o_i \in D$ is represented by an ordered list L_{o_i}. The ordered list for each object contains the reference points sorted by their distance d to the object o_i. More formally, L_{o_i} is the ordered permutation of $(r_0, \dots, r_j, \dots, r_n)$ according to the distance function d. $P(L_{o_i}, r_j)$ returns the position of the reference object r_j within the ordered list L_{o_i} of object o_i. For example, $P(L_{o_i}, r_j) = 4$ means that r_j is the 4th nearest reference point to the object o_i. Figures 1a and 1b show a group of objects and their ordered list respectively.

The object ordered lists are saved in the main memory. For a given query q, an ordered list L_q is computed as for the database objects with respect to the same reference points. The similarity between the query q and all the database objects o_i is measured by comparing the ordered lists using *Spearman Footrule Distance(SFD)*[18].

$$SFD(o_i, q) = \sum_{r \in R} |P(L_{o_i}, r) - P(L_q, r)| \qquad (1)$$

2.3 Suffix Array

We recall the principles of indexing with suffix arrays. We will extend these principles for a different usage. Let S be a string of length $m = |S|$ over a finite ordered *alphabet* Σ. Assume that the special symbol \$ is an element of Σ and appears once at the end; $S[m] = \$$. $S[i]$ indicates the character at position i in S, for $0 \leq i < m$. $S[i..j]$ represent the substring S starting with the character at position i and ending with the character at position j. The substring $S[i..m]$ is the i-th suffix of S, and it is denoted by $S(i)$.

The suffix array suff of the string S is an array of integers in the range 0 to m, specifying the lexicographic order of the m suffixes of the string S [12]. That is, $S(\mathsf{suff}[0]), S(\mathsf{suff}[1]), \dots, S(\mathsf{suff}[m])$ is the sequence of suffixes of S in ascending lexicographic order. For clarification, see figure 2. Suffix arrays are used to locate every occurrence of substring B within S in an effective way for many applications such as Bioinformatics [9].

i	0	1	2	3	4	5	6	7
suff	7	0	2	5	1	3	6	4
$S(\mathsf{suff}[i])$	\$	$acactat\$$	$actat\$$	$at\$$	$cactat\$$	$ctat\$$	$t\$$	$tat\$$

Figure 2: The suffix array of the string $S = acactat\$$.

3. MSA: METRIC SUFFIX ARRAY

Let us assume that we concatenate the ordered lists L_{o_i} for all the database objects. We obtain one single string S over a finite set $\Sigma = R$. The length of the string is $m = n \times N$, where n is the number of reference points in R and N is the number of objects. Hence a suffix array can be built for the string S. At each position of the suffix array, an object-id $o_i.id$, a reference-id $r_j.id$ and $P(L_{o_i}, r_j)$ are encoded. This data is used to answer a query q in a fast and effective way without the need to access the string or the ordered lists as discussed in the next sections. The construction of the metric suffix array is performed on three steps as follows:

1. Creating the order lists: For each object o_i an ordered list L_{o_i} is constructed as explained in section 2.2.

2. Concatenation: These ordered lists are concatenated to construct a single unified string $S = [L_{o_0}[0] \dots L_{o_N}[n]]$. String element $S[i]$ indicates a reference point r_j at position i in the string, which we call the global index. The reference point r_j also has a local index value which defines the location of r_j in the ordered list of the object $P(L_{o_i}, r_j)$, where L_{o_i} is a substring within S; $L_{o_i} = S[s_{o_i} \dots e_{o_i}]$, s_{o_i} and e_{o_i} are the start and the end position of the ordered list of object o_i in the string S.

3. MSA construction: For fast construction and usage of the MSA, we divide the MSA into buckets $buk_0 \dots buk_n$, where n is the number of reference points. A bucket buk_i is an interval of the metric suffix array $\mathsf{MSA}[l, r]$ of size N and determined by its left and right ends. Each bucket contains all suffixes $S(\mathsf{MSA}[l]) \dots S(\mathsf{MSA}[r])$ that share a common prefix of length 1, which means all the suffixes with the same starting reference points r_j are located in the same bucket. Hence, the MSA is constructed and partially sorted and there is no need for storing the concatenated string and the ordered lists, see figure 1c. In normal suffix arrays, the suffixes are sorted in lexicographic order. Our MSA is not fully sorted. MSA is only sorted by buckets, but within each bucket the suffixes are ordered randomly. Using the MSA and equation (1), we can answers users' queries in an effective way.

Hence, The main idea is that the MSA provides an organized representation of the reference points r_j in the ordered lists of the objects L_{o_i}, in order to improve the ordered lists comparison.

Comparing to MIF [1, 13], and PP-Index [4, 5] MSA saves half of the memory. In MIF, 8 bytes are needed for each reference point in every ordered list to store the $o_i.id$ and $P(L_{o_i}, r_j)$. In PP-Index, 4 bytes are needed for the reference point and other 4 bytes for the pointer in the prefix tree. In MSA, only 4 bytes are needed to store the global index of the reference point in the concatenated string S, which contains all the needed information. In the next section, we detail the indexing and the searching algorithms using the MSA.

3.1 Indexing

For optimal memory usage, the suffix array is built on the fly without actually constructing the string S. First, we construct an array MSA of size $m = n \times N$. Then, for each object, an ordered list is generated. This ordered list is scanned element by element to fill each position in the MSA. Algorithm 1 shows the indexing process. Lines 1-2 construct the ordered lists for all database objects one by one. Lines 3-4 fill the suffix array, where $r_j.id \times N$ defines the start of the associated bucket, $o_i.id$ defines the position inside the bucket and $(o_i.id \times n) + P(L_{o_i}, r_j)$ defines the global index of r_j in the string S. The processing complexity and the memory usage of the algorithm are $O(m)$.

Algorithm 1 Full permutation indexing

IN: Domain D of N objects,
References list R of n elements,
OUT: Suffix array: MSA;
1. For each $o \in D$
2. L_{o_i} =Create_Ordered_List(o_i, R)
3. For each $r \in L_{o_i}$
4. MSA$[(r_j.id \times N) + o_i.id] = (o_i.id \times n) + P(L_{o_i}, r_j)$

3.2 Searching

To retrieve the K-top results for a query q, L_q has to be compared with every L_{o_i} using equation (1). Hence, we need the object-id $o_i.id$, the reference-id $r_j.id$ and the $P(L_{o_i}, r_j)$. This information is encoded in the MSA. The $r_j.id$ is already defined as the MSA is divided into buckets and each bucket has the global index of all suffixes which start with the same r_j. From each MSA value, we can extract the $o_i.id$ and the $P(L_{o_i}, r_j)$ as given in Algorithm 2. Lines 1-2 creates N accumulators and initialize it by zero. Line 3 creates the ordered list for the query q. Lines 4-5 define the bucket range of each reference point in the query ordered list. Lines 6-7 get the object-id and the position of the reference point r_j in the ordered list of the object from the suffix value. Line 8 updates the accumulator of the object found with the difference between the position of the reference point r_j in the query and its location in the ordered list of the object "RefPos". Lines 9-10 sorts the objects based on the accumulators values and send the results back to the user. Theoretically, the complexity is $O(m)$ for filling the accumulators and $O(N \log N)$ for sorting them as we use quick sort algorithm.

3.3 Using nearest permutations

The information given by the ordering of the farthest reference points is not critical. Hence, in order to reduce the memory used, we can use only the nearest reference points to identify the objects.

Algorithm 2 Full permutation searching

IN: Query: q,
References list R of n elements,
Metric Suffix array: MSA;
OUT: Sorted Objects list: out
1. Create a list of accumulators $A[0 \dots N]$
2. Set accumulators values to 0
3. Create the query ordered list L_q
4. For $r_j \in L_q$
5. For $k \leftarrow (r_j.id \times N)$ *to* $k < (r_j.id \times N) + N$
6. $O_{id} = (\text{MSA}[k] - (\text{MSA}[k] \mod n))/n$
7. RefPos$= (\text{MSA}[k] \mod n) + 1$
8. $Acc[O_{id}] = Acc[O_{id}] + |P(r_j, L_q) - \text{RefPos}|$
9. $sort(Acc)$
10. $out \leftarrow Acc$

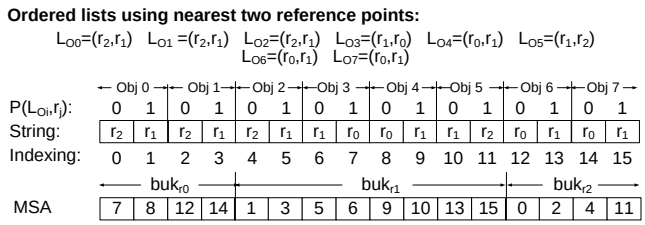

Figure 3: Example of MSA using nearest reference points

This can be done by distributing the reference points in D and every object is indexed with respect to $\bar{R}$, where $\bar{R} \in R$ and represents the $\bar{n}$ reference points closest to the object. The size of the MSA is $\bar{m} = \bar{n} \times N$, where $\bar{n} = |\bar{R}|$ and N is the number of objects. Since $\bar{R}$ is not unique and is based on the location of the object, buckets are not of the same size. Hence, indexing and searching must be adapted as described next. Figure 3 shows an example of indexing using the nearest reference objects.

3.3.1 Indexing

Count sort is used for indexing using the nearest permutations. The basic idea is to determine, for each suffix, the number of suffixes located before it. This helps placing the suffix value directly into its position in the MSA. In Algorithm 3, lines 1-2 create an array buk_count of size n and initialize it with zero to count the number of occurrences of each reference points in all the ordered lists. Lines 3-6 create and scan $L_{o_i} \forall o_i \in D$ element by element. Once a reference point r_j is found, $buk\_count[r_j]$ is updated. Accordingly, the size of each bucket is defined. Lines 7-11 define the start and the end of each bucket. Since buckets are contiguous, the left end l of bucket buk_{r_j} equals to the right end of the previous bucket $buk_{r_{j-1}}.r$. Consequently, the right end r of the bucket equals: $buk_{r_j}.r = buk_{r_j}.l + buk\_count[r_j]$. Lines 12-14 build the MSA. Counter C_{r_i} is created and initialized with zero for each reference point r_j to define the location in its bucket. Ordered lists are scanned element by element and placed directly in their appropriate location in the MSA. Theoretically, the complexity is $O(2\bar{m})$ and the memory usage is $O(\bar{m})$.

3.3.2 Searching

Similar to what is done in Algorithm 2, to rank the database objects for a query q, we need three items: the object-id $o_i.id$, the reference-id $r_j.id$ and the $P(L_{o_i}, r_j)$. After creating L_q, the buck-

Algorithm 3 Nearest permutation indexing

IN: Domain D of N objects,
References list R of n elements,
OUT: Suffix array: suff
1. For each $j \longleftarrow 0$ *to* n
2. $buk\_count[j] = 0$
3. For each $o \in D$
4. L_{o_i} =Create_Ordered_List_With_Nearest_References(o_i, R)
5. For each $r \in L_{o_i}$
6. $buk\_count[r_j] + +$
7. $buk_0.l = 0$
8. $buk_0.r = buk_0.l + buk\_count[0]$
9. For each $j \longleftarrow 1$ *to* n
10. $buk_{r_j}.l = buk_{r_{j-1}}.r$
11. $buk_{r_j}.r = buk_{r_j}.l + buk\_count[r_j]$
12. For each $o \in D$
13. For each $r \in L_{o_i}$
14. MSA$[buk_{r_j}.l + (C_{r_j} + +)] = o_i.id \times \bar{n} + P(L_{o_i}, r_j)$

ets in the MSA are scanned based on the order of the reference points in L_q. In Algorithm 4, lines 1-2 create the accumulators and initialize them by $(\bar{n}+1) \times \bar{n}$, which is the maximum possible distance between an object and the query. Lines 4-8 access the active buckets whose reference points appeared in the query ordered list and update the accumulators, where $(\bar{n}+1)$ shows that a reference point presented in L_q and $|P(L_q, r_j)-$RefPos$|$, gives the difference between the position of r_j in L_q and L_{o_i}. Lines 9-10 sort and gives the output. Theoretically, the complexity is $O(\bar{m}n)$ for filling the accumulators and $O(N \log N)$ for sorting them.

Algorithm 4 Nearest permutation Searching

IN: Query: q,
References list R of n elements,
Suffix array: suff;
OUT: Sorted Objects list: out
1. Create a list of accumulators $Acc[0 \ldots N]$
2. Set accumulators values to $(\bar{n}+1) \times \bar{n}$
3. Create the query ordered list based on the nearest references L_q
4. For $r \in L_q$
5. For $k \longleftarrow$ MSA$[buk_{r_j}.l]$ *to* MSA$[buk_{r_j}.r]$
6. $O_{id} = (\text{MSA}[k] - (\text{MSA}[k] \mod \bar{n}))/\bar{n}$
7. RefPos$= (\text{MSA}[k] \mod \bar{n}) + 1$
8. $Acc[O_{id}] = Acc[O_{id}] - (\bar{n}+1) + |P(L_q, r_j)-$RefPos$|$
9. $sort(Acc)$
10. $out \leftarrow Acc$

4. MULTI-CORE AND DISTRIBUTED MSA

4.1 Multi-Core Implementation

Currently, most computers are multi-core. For indexing, openMP [2] does not support parallel I/O. Hence, we face a bottleneck in accessing the data file. We solve this problem by setting the data read part as a critical region, which means only one thread can access the file at a time. Creating the ordered list, sorting it, and filling the MSA is done in parallel for different objects. As a result, we obtain sub-linear speedup, as shown in the results section.

For searching, we can parallelize the 'for' loops at line 4 or line 5 in algorithms 2 and 4, by distributing the for loop on the available processes. For parallelizing the 'for' loops at line 4 in the two algorithms, we face a race condition problem as some processes access the accumulator of the same object, but in different buckets, which affect the final results. Our solution is to set the accumulator update functions in line 8 in the two algorithms as a critical region. Hence, most of the time is consumed in organizing the access of the processes to the critical region as the rest of work is simple. The best speedup is obtained by parallelizing the 'for' loops at line 5 in algorithms 2 and 4 as we do not need the critical region. For every bucket, all the processes update the accumulators for different objects separately.

4.2 Distributed Implementation

The data domain D of N objects is randomly divided into subdomains of equal sizes $D_0 \ldots D_p$, where p is the number of parallel processes. Using n reference points known by all processes, every process builds its own MSA based on the global reference points and the partial data it has access to. As a result, we obtain multiple partial MSAs, where each process is responsible for a different MSA representing only a sub-domain of the full dataset and not related to the other MSAs (see figure 4). Hence technically, we divide the data file into subfiles and each file is handled on a different machine separately in parallel.

To answer a query q, all the MSAs need to be scanned. A broker process accepts query requests and broadcasts them to all other processes. Upon reception, each process starts to index the query with respect to the list of the global reference points and to apply the search on its local MSA. Once done, every process sends its local accumulators to the broker process. The broker concatenates the accumulators from different processes and sorts the objects based on their accumulator values. More formally, the 'for' loops from line 4 to line 8 in algorithms 2 and 4, are running in parallel on different parts of size N/p, where N is the number of objects and p is the number of processes. There is an extra over head, which is the communication time t_p needed to receive the partial accumulators from different processes. Hence, theoretically the overall complexity is $O(N/p)n + t_p$ for full permutation and $O(N/p)\bar{n} + t_p$ for nearest permutation. MPI [7] is used for our implementation.

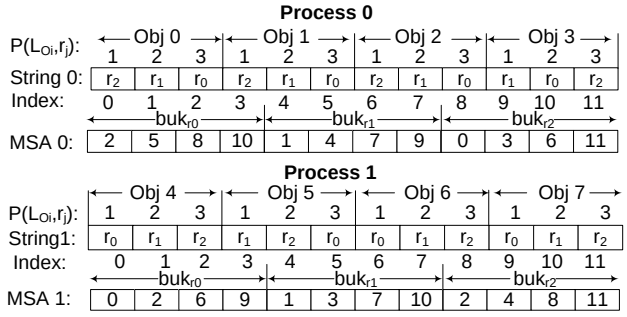

Figure 4: Example of multiple partial MSA for 8 objects and 2 processes.

5. EXPERIMENTAL RESULTS

In [13], we proposed a distributed implementation of the MIF [1]. We were able to achieve high speed up using 40 cores, but we were facing a problem with memory consumption. Here, we compare the performance of our MSA to the performance of MIF as our baseline structure. We have implemented the MSA using C++. The sequential and the multicore experiments were done on

an 8-core machine holding 32Gb of memory and a TeraByte storage capacity. The distributed experiments were done on a computer cluster of 20 machines, each one has 8Gb of memory and 2 cores and a disk of 512GB. Our experiments are based on two different datasets. The first dataset consists of 2,076,399 objects. It was used to compare the MSA and the MIF regarding indexing and searching times for sequential and multicore implementations. Also, we show the recall and the position error (PE) [18] using full and nearest permutations. The second dataset consists of 4,594,734 objects and was used for large scale distributed indexing. The two datasets are visual shape features (21-dimensional), which were extracted from the 12-million ImageNet corpus [3].

5.1 Recall and position error

We measure the average recall and position error based on 10 different queries selected randomly from the datasets. Figure 5 shows the average recall and position error using full and nearest permutations. The selection of the reference points is done by defining a certain threshold, where the minimum distance between any two references is greater than this threshold value. For nearest permutation indexing, we used the closest half reference points. For instance, if we have 1000 reference point, we chose the nearest 500 points only. For full permutations, we can see that increasing the reference points does not really affect the recall after a certain value. In our dataset, after 1000 reference points using full permutation the recall stays constant. For nearest permutations, using the closest 1000 out of 2000 gives higher recall value than using 1000 reference points for full permutation. The reason is that using 1000 reference points out of 2000 reference points makes each object identified by the nearest reference points only, which make a unique identification of each object and leads to an improved recall. Hence, we empirically derive that the best ratio between the number of nearest reference points and the dataset is $n = \sqrt{(N/2)}$ and the closest reference points can be chosen out of $2n$ reference points. For position error, as we can see using half permutation increases the PE, but it decreases when the number of closest reference points increases.

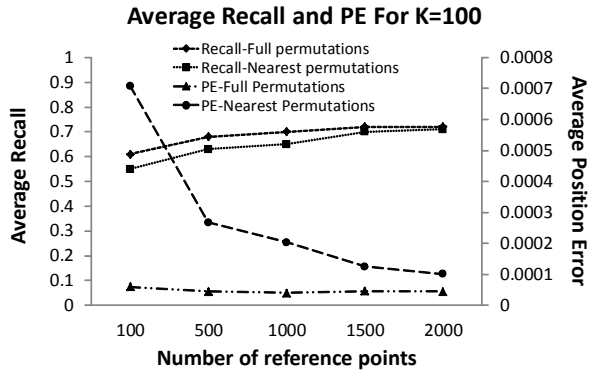

Figure 5: Recall and PE for K=100 using full and nearest half permutations for 100, 500, 1000, 1500 and 2000 reference point.

5.2 Response time

5.2.1 Sequential and Multi-Core Implementation

Figures 6 and 7 show the indexing and the searching time respectively. As we can see from the two figures, using full permutation, MSA outperform MIF. Although theoretically, MSA has the same complexity as MIF. The main reason is that handling the arrays into the main and the cache memories is much better than handling the inverted files. Also, our algorithm consumes half of the memory consumed by MIF. For example, for 2'000 reference points, MIF needs about 30Gb of memory while MSA needs 15GB. The reason is that we encode, in each suffix array value, the position of the reference point and the object-id instead of saving them as two separate values. Also, we can see that the running time decreases using multicore for MIF and MSA, and still MSA outperforms MIF. Using nearest 1000 reference point out of 2000 reference point (which gave the best recall value) the response time is 1.9 second.

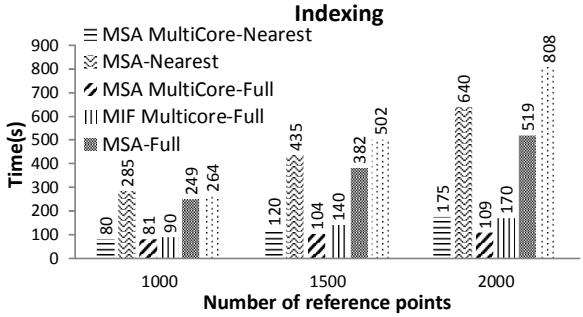

Figure 6: Average Indexing time in seconds. The values above the columns show the running time in seconds

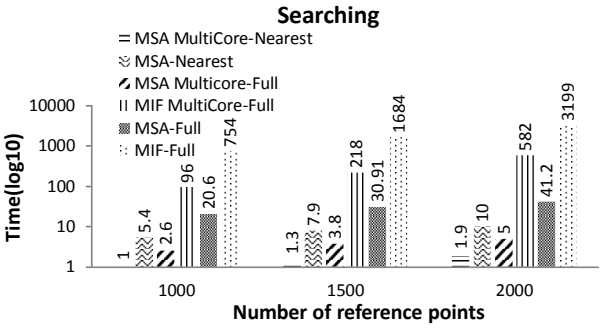

Figure 7: log10 of the average searching time. The values above the columns show the average running time in seconds.

5.3 Distributed Implementation

Figures 8 and 9 show indexing and the searching time respectively for 4,594,734 objects using MSA. As we can see, the time decreases when the number of cores increases with the same recall and PE. Using the closest 1500 points out of 3000 reference points, we are able to retrieve the most similar objects in less than 1 second using 20 cores and with recall value more than 0.7.

6. CONCLUSION

We have presented the Metric Suffix Array data structure, which is a novel data structure for permutation-based indexing of large scale multidimensional data. MSA is adapted to work with full and nearest permutations while saving half of the memory needed for other permutation-based indexing data structures [1, 13, 4, 5].

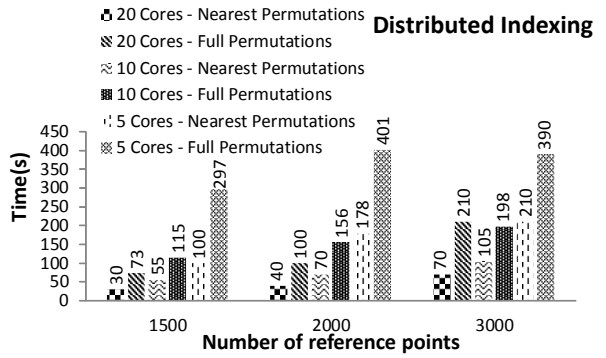

Figure 8: Indexing time using 5, 10, 20 distributed cores. The values above the columns show the running time in seconds.

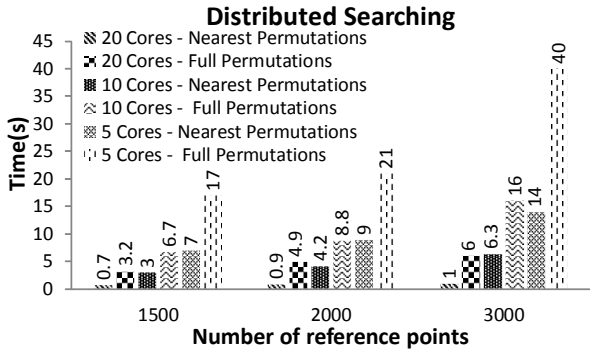

Figure 9: Searching time using 5, 10, 20 distributed cores. The values above the columns show the running time in seconds.

Also, we showed that the MSA outperforms the MIF. We studied the best ratio between the dataset and the reference points and showed how this affects the recall and the position error.

Our current work focuses on proposing a further strategy to speed up the searching using MSA. Also, we will propose some techniques for selecting the reference points to improve the recall. For the distributed implementation, we will provide other strategies for distribution and seek the best possible performance.

7. ACKNOWLEDGMENT

This work is jointly supported by the Swiss National Science Foundation (SNSF) via the Swiss National Center of Competence in Research (NCCR) on Interactive Multimodal Information Management (IM2) and the European COST Action on Multilingual and Multifaceted Interactive Information Access (MUMIA) via the Swiss State Secretariat for Education and Research (SER).

8. REFERENCES

[1] G. Amato and P. Savino. Approximate similarity search in metric spaces using inverted files. In *Proceedings of the 3rd international conference on Scalable information systems*, InfoScale '08, pages 28:1–28:10, ICST, Brussels, Belgium, Belgium, 2008. ICST.

[2] L. Dagum and R. Menon. Openmp: An industry-standard api for shared-memory programming. *IEEE Comput. Sci. Eng.*, 5(1):46–55, Jan. 1998.

[3] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*, 2009.

[4] A. Esuli. Mipai: Using the pp-index to build an efficient and scalable similarity search system. In *Proceedings of the 2009 Second International Workshop on Similarity Search and Applications*, SISAP '09, pages 146–148, Washington, DC, USA, 2009. IEEE Computer Society.

[5] A. Esuli. Pp-index: Using permutation prefixes for efficient and scalable approximate similarity search. *Proceedings of LSDSIR 2009*, i(July):1–48, 2009.

[6] K. Figueroa and K. Fredriksson. Speeding up permutation based indexing with indexing. In *Proceedings of the 2009 Second International Workshop on Similarity Search and Applications*, SISAP '09, pages 107–114, Washington, DC, USA, 2009. IEEE Computer Society.

[7] T. M. Forum. Mpi: A message passing interface, 1993.

[8] E. Gonzalez, K. Figueroa, and G. Navarro. Effective proximity retrieval by ordering permutations. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 30(9):1647 –1658, sept. 2008.

[9] R. Grossi, Jeffrey, and S. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching (extended abstract. In *in Proceedings of the 32nd Annual ACM Symposium on the Theory of Computing*, pages 397–406, 2000.

[10] P. Indyk and R. Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, STOC '98, pages 604–613, New York, NY, USA, 1998. ACM.

[11] H. V. Jagadish, A. O. Mendelzon, and T. Milo. Similarity-based queries. In *Proceedings of the fourteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, PODS '95, pages 36–45, New York, NY, USA, 1995. ACM.

[12] U. Manber and E. W. Myers. Suffix arrays: A new method for on-line string searches. *SIAM J. Comput.*, 22(5):935–948, 1993.

[13] H. Mohamed and S. Marchand-Maillet. Parallel approaches to permutation-based indexing using inverted files. In *5th International Conference on Similarity Search and Applications (SISAP)*, Toronto, CA, August 2012.

[14] D. Novak, M. Batko, and P. Zezula. Metric index: An efficient and scalable solution for precise and approximate similarity search. *Inf. Syst.*, 36(4):721–733, 2011.

[15] M. Patella and P. Ciaccia. Approximate similarity search: A multi-faceted problem. *J. of Discrete Algorithms*, 7(1):36–48, Mar. 2009.

[16] H. Samet. *Foundations of multidimensional and metric data structures*. The Morgan Kaufmann series in computer graphics and geometric modeling. Elsevier/Morgan Kaufmann, 2006.

[17] E. S. Téllez, E. Chávez, and A. Camarena-Ibarrola. A brief index for proximity searching. In *Proceedings of the 14th Iberoamerican Conference on Pattern Recognition: Progress in Pattern Recognition, Image Analysis, Computer Vision, and Applications*, CIARP '09, pages 529–536, Berlin, Heidelberg, 2009. Springer-Verlag.

[18] P. Zezula, G. Amato, V. Dohnal, and M. Batko. *Similarity Search: The Metric Space Approach*, volume 32 of *Advances in Database Systems*. Springer, 2006.