Similarity Search Implementations for Multi-core and Many-core Processors

Roberto Uribe-Paredes
Computer Engineering Department,
University of Magallanes,
Punta Arenas, Chile.
roberto.uribeparedes@gmail.com

Pedro Valero-Lara
Centro de Investigaciones Energéticas,
Medioambientales y Tecnológicas
Madrid, España.
pedro.valero@ciemat.es

Enrique Arias, José L. Sánchez, Diego Cazorla
Computing Systems Dept,
University of Castilla-La Mancha,
Albacete, España.
{*enrique.arias,jose.sgarcia, diego.cazorla*}*@uclm.es*

ABSTRACT

Similarity search in a large collection of stored objects in a metric database has become a most interesting problem. The Spaghettis *is an efficient metric data structure to index metric spaces. However, for real applications, when processing large volumes of data, query response time can be high enough. In this case, it is necessary to apply mechanisms in order to significantly reduce the average query response time. In this sense, the parallelization of the metric structures processing is an interesting field of research. Modern multi-core and many-core systems offer a very impressive cost/performance ratio. In this paper two new parallel implementations for range queries on Spaghettis data structures have been carried out: one of them on a many-core processor and the other one on a multi-core processor. Both implementations have been compared in terms of execution time and speedup.*

KEYWORDS: Similarity search, metric spaces, parallel processing, multi-core and many-core.

1. INTRODUCTION

In the last decade, the search of similar objects in a large collection of stored objects in a metric database has become a most interesting problem [1–7]. This kind of search can be found in different applications such as voice and image recognition, data mining, plagiarism and many others. A typical query for these applications is the *range search* which consists in obtaining all the objects that are at some given distance from the consulted object.

The increased size of databases and the emergence of new data types create the need to process a large volume of data. The use of parallelism may allow us not only to deal with these data, but also to reduce query response time.

The recent appearance of GPUs for general purpose computing platforms offers powerful parallel processing capabilities at a low price. On the other hand, current multicore processors are becoming another interesting parallel platform also at a low price without the constraints of GPU architecture. Both types of processors offer a very interesting performance/cost ratio, and so they are increasingly used in parallel computing. In this paper two new parallel implementations for range queries on Spaghettis data structures have been carried out, one of them on a GPU and the other one on a multicore processor. Both implementations have been compared in terms of execution time and speedup.

This paper is structured as follows. Section 2 introduces the problem of similarity search in metric spaces. In Section 3 the state of the art of parallelism applied to similarity search in metric spaces is presented. After that, in Section 4 the two new parallel implementations are briefly explained, and in Section 5 the experimental platform is described and the results are discussed. Finally, in Section 6 the conclusions and future work are outlined.

2. SIMILARITY SEARCH IN METRIC SPACES

Similarity is modeled in many interesting cases through metric spaces, and the search of similar objects through

978-1-61284-383-4/11/$26.00 ©2011 IEEE

range search or nearest neighbors. A metric space $(\mathbb{X}, d)$ is a set $\mathbb{X}$ and a distance function $d : \mathbb{X}^2 \rightarrow \mathbb{R}$, so that $\forall x, y, z \in \mathbb{X}$ fulfills the properties of positiveness $[d(x, y) \geq 0$, and $d(x, y) = 0$ iff $x = y]$, symmetry $[d(x, y) = d(y, x)]$ and triangle inequality $[d(x, y) + d(y, z) \geq (d(x, z)]$.

In a given metric space ($\mathbb{X}$,*d*) and a finite data set $\mathbb{Y} \subseteq \mathbb{X}$, a series of queries can be made. The basic query is the *range query* (*x*,*r*), a query being $x \in \mathbb{X}$ and a range $r \in \mathbb{R}$. The range query around *x* with range *r* is the set of objects $y \in \mathbb{Y}$ such that $d(x, y) \leq r$. A second type of query that can be built using the range query is *k nearest neighbors*, the query being $x \in \mathbb{X}$ and object k. *k* nearest neighbors to *x* are a subset $\mathbb{A}$ of objects $\mathbb{Y}$, such that if $|\mathbb{A}| = k$ and an object $y \in \mathbb{A}$, there is no object $z \notin A$ such that $d(z, x) \leq d(y, x)$.

Metric access methods, *metric space indexes* or *metric data structures* are different names for data structures built over a set of objects. The objective of these methods is to minimize the amount of distance evaluations made to solve the query. Searching methods for metric spaces are mainly based on dividing the space using the distance to one or more selected objects. As they do not use particular characteristics of the application, these methods work with any type of objects [1].

Metric space data structures can be grouped into two classes [1], *clustering*-based and *pivots*-based methods.

The *clustering*-based structures divide the space into areas, where each area has a so-called center. Some data is stored in each area, which allows easy discarding the whole area by just comparing the query with its center. Algorithms based on clustering are better suited for high-dimensional metric spaces, which is the most difficult problem in practice. Some clustering-based indexes are *BST*, *GHT*, *M-Tree*, *GNAT*, and many others [1].

There exist two criteria to define the areas in clustering-based structures: *hyperplanes* and *covering radius*. The former divides the space into *Voronoi* partitions and determines the hyper plane the query belongs to according to the corresponding center. The covering radius criterion divides the space into spheres that can be intersected and one query can belong to one or more spheres.

In the *pivots*-based methods, a set of pivots are selected and the distances between the pivots and database elements are precalculated. When a query is made, the query distance to each pivot is calculated and the triangle inequality is used to discard the candidates. Its objective is to filter objects during a request through the use of a triangular inequality, without really measuring the distance between the object under request and the discarded object. Some pivots-based indexes are *LAESA* [4], *FQT* and its variants [5], *Spaghettis* and its variants [6], *FQA* [7] and others.

Many indexes are trees, and, the children of each node define areas of space. Range queries traverse the tree, entering into all the children whose areas cannot be proved to be disjoint with the query region. Other metric structures are arrays; in this case, the array usually contains all the objects of the database and maintains the distances to the pivots.

In this paper, the Spaghettis data structure is considered for range queries. *Spaghettis* [6] is a variant of data structure *LAESA* [4] based on pivots. The method tries to reduce the CPU time needed to carry out a query by using a data structure where the distance to the pivots is sorted independently. As a result, the Spaghettis data structure S is obtained, where each column S_j is associated to each pivot allowing a binary search in a given range.

2.1. Construction

During the construction of the Spaghettis structure, a random set of pivots $p_1, ..., p_k$ is selected. These pivots could belong or not to the database to be indexed. Each position on column S_j represents an object of the database which has a link to its position on the next pivot column. The last column links the object to its position on the database. Figure 1 shows an example considering 17 elements.

2.2. Searching

During the searching process, given a query q and a range r, a range search on an *Spaghettis* follows the following steps:

1. The distance between q and all pivots $p_1, \ldots, p_k$ is calculated in order to obtain k intervals in the form $[a_1, b_1], ..., [a_k, b_k]$, where $a_i = d(p_i, q)$ - r and $b_i = d(p_i, q) + r$.
2. The objects in the intersection of all intervals are considered as candidates to the query q.
3. For each candidate object y, the distance $d(q, y)$ is calculated, and if $d(q, y) \leq r$, then the object y is a solution to the query.

Implementation details are shown in Algorithm 1. In this algorithm, S_{ij} represents the distance between the object y_i and the pivot p_j.

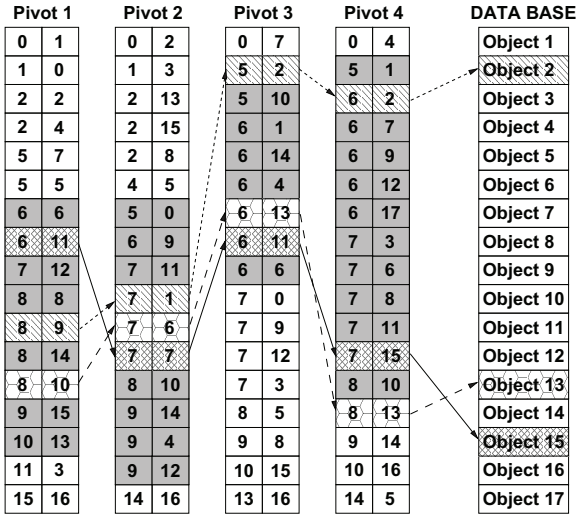


Figure 1. *Spaghettis*: Construction and search. Example for query q with ranges $\{(6,10),(5,9),(2,6),(4,8)\}$ to pivots.

Algorithm 1 *Spaghettis*: Search Algorithm.

rangesearch(query q, range r)

1: {Let $\mathbb{Y} \subseteq \mathbb{X}$ be the database}
2: {Let P be set of pivots $p_1, \ldots, p_k \in \mathbb{X}$}
3: {Let D be the table of distances associated to q}
4: {Let S be Spaghettis}
5: **for all** $p_i \in P$ **do**
6: $D_i \leftarrow d(q, p_i)$
7: **end for**
8: **for all** $y_i \in \mathbb{Y}$ **do**
9: $discarded \leftarrow false$
10: **for all** $p_j \in P$ **do**
11: **if** $D_j - r > S_{ij} \,||\, D_j + r < S_{ij}$ **then**
12: $discarded \leftarrow true$
13: break;
14: **end if**
15: **end for**
16: **if** !$discarded$ **then**
17: **if** $d(y_i, q) \leq r$ **then**
18: add to result
19: **end if**
20: **end if**
21: **end for**

Figure 1 represents the data structure *Spaghettis* in its original form. This structure is built using 4 pivots to index a database of 17 objects. The searching process is as follows. Assuming a query q, the distance to the pivots $\{8, 7, 4, 6\}$, and a searching range $r = 2$, Figure 1 shows in dark gray the intervals $\{(6,10), (5,9), (2,6), (4,8)\}$ over which the searching is going to be carried out. Also, in this figure it is possible to see all the objects that belong to the intersection of all the intervals, which are considered as candidates. Finally, the distance between the candidates and the query has to be calculated in order to determine a solution from the candidates. The solution is given if the distance is lower than a searching range.

3. RELATED WORK

Currently, there are many parallel platforms for the implementation of metric structures. In this context, basic research has focused on technologies for distributed memory applications, using high level libraries for message passing such as MPI [8] or PVM [9], and shared memory, using the language or directives of OpenMP [10].

Some studies [11–13] have focused on different structures parallelized on distributed memory platforms using MPI or BSP. In these cases, the aim was not only the parallelization of the algorithms, but also the balanced distribution of data.

In terms of shared memory, [14] proposes a strategy to organize metric-space query processing in multi-core search nodes as understood in the context of search engines running on clusters of computers. The strategy is applied in each search node to process all active queries visiting the node as part of their solution which, in general, for each query is computed from the contribution of each search node. Besides, this work proposes mechanisms to address different levels of query traffic on a search engine.

Most of the previous and current works developed in this area are carried out considering classical distributed or shared memory platforms. However, new computing platforms are gaining in significance and popularity within the scientific computing community. Hybrid platforms based on *Graphics Processing Units* (GPU) is an example.

However, as far as we know, the solutions considered till now developed on GPUs are based on *kNN* queries without using data structures. This means that GPUs are basically applied to exploit its parallelism only for exhaustive search (brute force) [15–17].

In [15] both elements (A) and queries (B) matrices are divided into fixed size submatrices. In this way, the resultant submatrix C is computed by a block of threads. Once the whole submatrix has been processed, *CUDA-based Radix Sort* is applied over the complete matrix in order to sort it and obtain the first k elements as a final result.

In [16] a brute force algorithm is implemented where each thread computes the distance between an element of a database and a query. Afterwards, it is necessary to sort the resultant array by means of a variant of the *insertion sort* algorithm.

As a conclusion, in these works the parallelization is applied in two stages. The first one consists in building the distance matrix, and the second one consists in sorting this distance matrix in order to obtain the final result.

A particular variant of the above proposed algorithms is presented in [18] where the search is structured into three steps. In the first step each block solves a query. Each thread keeps a heap where stores the kNN nearest elements processed by this thread. Secondly, a reduction operation is applied to obtain a final heap. Finally, the first k elements of this final heap are taken as a result of the query.

The purpose of this paper consists in carrying out parallel implementations for range queries considering the *Spaghettis* data structure. These implementations are based on GPU and multicore processors, representing a new contribution on this research area.

4. PARALLEL IMPLEMENTATION

The range query process intrinsically has a high data-level parallelism with high computing requirements. For that reason, the authors propose versions on different parallel platforms in order to obtain a better performance. We present two parallel implementations of the range queries method, one based on a many-core platform and the other based on a multi-core platform. In each one, the *exhaustive search* and *Spaghettis* implementations are presented.

In order to obtain better performance on the target platform, some changes on the original *Spaghettis* structure have been made. In this implementation, each row is associated with an object of the dataset and each column to a pivot. Therefore, each cell contains the distance between the object and the pivot. Moreover, unlike the original version, the array is sorted only by the first pivot. Thus, the cells on the same row refer to the same object.

4.1. GPU-Based Implementation

Given a database and a query, sequential *Exhaustive Search* is an iterative process where in each iteration the distance between the query and an element of the database is calculated, thus determining if this element is or not a valid solution. The parallel implementation of this process is obvious, and due to the special characteristics of GPUs, it consists in launching as many threads as elements in the database.

On the other hand, due to the GPU limitations (number of threads and memory capacity), it is not possible to simultaneously calculate all distances for all queries using only one kernel. Consequently, we consider as many kernels as queries, where each kernel solves one query.

The *Spaghettis* GPU-based implementation has been splitted into three parts, which are the most computationally expensive parts of this algorithm. These parts correspond to the three steps presented in Subsection 2.2.

The first part consists in computing the distances between the set of queries, Q, and the set of pivots, P. In order to exploit the advantages of using a GPU platform, a data structure which stores all distances is needed. Therefore, this structure is implemented as a $Q \times P$ matrix which allows us to compute all distances at the same time in a single call to the kernel. This kernel consists of as many threads as the number of queries. In fact, each thread calculates independently the distance from a query to all pivots.

The second part of the parallel implementation consists in determining whether each element of the database is or not a candidate for every query. This part has been implemented as an iterative process. In each iteration the candidates for a particular query are computed in one kernel. As we have described above, it is not possible to calculate all candidates for every query in only one kernel due to the GPU limitations. In this kernel as many threads as the number of elements of the database are launched. Each thread of this kernel determines, for a given data (y_i) of the dataset, if this data is candidate or not. Thus, this kernel returns a list of candidates for a given query. Finally, when this process finishes we obtain one list of candidates for each query. This task is carried out by a kernel called *KCandidates* (see Algorithm 2).

The kernel which implements the third part determines if each candidate is really a solution. In this kernel, the number of threads corresponds to the number of candidates for each query. Each thread calculates the distance between one candidate and one query, and determines if this candidate is a valid solution. Finally, as result we obtain one list of solutions for each query.

In the three kernels, threads belonging to the same thread block operate over contiguous components of the arrays. Therefore, more efficient memory accesses are allowed. This is due to the above mentioned changes in the Spaghettis structure.

Algorithm 2 CUDA Search Algorithm.

__global__ KCandidates(range r, Spaghettis S, distances D, pivots P, candidates C)

1: {Let P be set of pivots $p_1, \ldots, p_2 \in \mathbb{X}$}
2: {Let D be the table of distances associated q}
3: {Let C be list of candidates for q}
4: {Let i be thread Id }
5: $discarded \leftarrow false$
6: **for all** $p_j \in P$ **do**
7: **if** $D_j - r > S_{ij} \,||\, D_j + r < S_{ij}$ **then**
8: $discarded \leftarrow true$
9: break;
10: **end if**
11: **end for**
12: **if** $!discarded$ **then**
13: add to C (candidates)
14: **end if**

The threads in the same block can use the shared memory which is much faster than global memory. However, the use of this memory is only useful when many threads of the same block have to access the same positions of memory. This fact occurs in all three kernels, and therefore, we can use the shared memory in the three parts of the algorithms. Next, we explain how we use the shared memory in the different kernels.

In the first kernel, the threads have to compute the distance between all queries with all pivots. Therefore, all threads need to access the pivots, that is, they need to access the same positions of memory. For that reason, we store the pivots in shared memory.

In the second kernel, the threads calculate the candidates of a particular query by comparing the distances between the query and the pivots (calculated in the first kernel) with the distances between all data set with the pivots. Hence, all threads have to access to the same positions of memory. Due to this, we store these positions in shared memory.

In the third kernel, the threads have to compute if the candidates are valid solutions by calculating the distances between the query and the candidates. Due to this implementation, all threads have to access to the same positions of memory which store the query. Therefore, we store the query in shared memory.

4.2. Multi-Core Implementation

In order to use the capability of our CPU with 4 cores, carry out a performance evaluation on this platform and compare the results between platforms (GPU and multi-core CPU), we have implemented a multicore-based version of the same algorithm.

We have implemented the three parts described in Section 2.2 which are the most expensive computationally as we have explained above. Mainly, this implementation consists in distributing the different iterations of the loop of each part among the different cores using OpenMP pragmas. In the first part, the calculation of the distance between the queries and the pivots are distributed among the cores. In the second part, we distribute the candidates searching among the cores. Finally, in the third part, each core determines in a subset of candidates the solutions to a given query. Due to that, we obtain a more efficient and faster implementation.

5. EXPERIMENTAL EVALUATION

In this section we show the experimental results obtained from the parallel implementations of the range search based on Spaghettis structure. Previously, the testbed used in this study is presented. We introduce the datasets and hardware plataforms, and describe the experiments.

5.1. Evaluation Conditions

We considered two datasets: a subset of the Spanish dictionary and a color histograms database, obtained from the Metric Spaces Library[1]. The Spanish dictionary we used is composed of 86,061 words. The edit distance was used. Given two words, this distance is defined as the minimum number of insertions, deletions or substitutions of characters needed to make one of the words equal to the other. The second space is a color histogram. It is a set of 112,682 color histograms (112-dimensional vectors) from an image database. Any quadratic form can be used as a distance, so we chose Euclidean distance as the simplest meaningful alternative.

We create the *S*paghettis data structure with the 90% of the data set randomly chosen, and reserve the rest for queries.

The hardware platform used was a PC with the following main components:

- CPU: Intel Core 2 Quad at 2.66GHz and 4GB of main memory.
- GPU: GTX 285 with 240 cores and 1 GB of global memory.

[1] www.sisap.org

The results presented in this section belong to a set of experiments with the following features:

- Pivots were randomly selected.
- The *Spaghettis* structure was built considering 4, 8, 16, and 32 pivots.
- For words space, each experiment has 8,606 queries over a *Spaghettis* with 77,455 objects. For vectors space, we have used a dataset of 101,414 objects and 11,268 queries.
- For each query, a range search between 1 and 4 was considered for the first space, and for vectors space we have chosen ranges which allow to retrieve 0.01, 0.1 and 1% from the dataset.
- The execution time shown in this paper is the total time of all versions. Therefore, in the case of GPU-based version, the execution time also includes the data transfer time between the main memory (CPU) and global memory (GPU).

We have chosen this experimental environment because is the most usual environment used to evaluate this kind of algorithms.

5.2. Experimental Results

In this section, we show and comment the results obtained. Figure 2 shows comparative results of search costs considering the space of words and color histograms for *Spaghettis* metric structure on sequential CPU, multi-core CPU and GPU platforms.

Figures 2(a) and 2(b) show the execution time for all implementations considered: exhaustive search and *Spaghettis* structure. As expected, in all cases, the use of *Spaghettis* structure allows to decrease the number of distance evaluations, and therefore we obtain a smaller execution time. As the number of pivots increases, the execution time decreases. It is due to the fact that with more pivots *Spaghettis* structure is more efficient because less distance evaluations are needed. On the other hand, the execution time increases when range increases because there are more data recovered (candidates) and so more distance evaluations are required. As we can also observe, the GPU-based implementations offer the best performance.

Figures 2(c), 2(d) and 2(e), 2(f) focus on the multi-core and GPU results, respectively. Notice that in some cases, a higher number of pivots does not imply a better performance. This can be observed in Figure 2(c) for ranges 3 and 4. This is due to the fact that the selection of pivots is not always the most appropriate. In this work random selection has been used. The pivot selection is out of the scope of this work. On the other hand, in Figure 2(f) we can observe that the exhaustive search is faster than the *Spaghettis* with 4 pivots for 1% of data retrieved from the database. Notice that when range is increased the behavior of the searching using the *Spaghettis* structure is similar to the exhaustive search. Besides, the cost of the tranferences between memories (CPU and GPU) and memory accesses have to be considered.

Finally, Figures 2(g) and 2(h) show the speedup of the parallel implemententions with respect to the corresponding sequential implementations. Several datails can be observed. First, as expected, the speedup obtained for the GPU platform is higher than that obtained for the multi-core platform. Secondly, regardless of the hardware platform, the speedup for the word space is higher than that obtained for the color histograms. This is due to the computational cost of the distance function, which is much higher in the first case. Third, notice that for low ranges the speedup is higher when a small number of pivots is used because in these cases the performance of the sequential implementation is very poor.

It is also important to remark the speedup obtained when comparing the best results for both platforms with the best sequential result, without considering the brute force implementation. In the case of words space with range 1-4 we obtain a speedup between 1.87 and 3.17 for multicore and between 2.79 and 9.84 for GPU. In the case of color histogram with range 0.01%-1% we obtain a speedup between 1.58 and 1.77 for multi-core, and between 2.18 and 5.55 for GPU. The speedup for color histogram is not so high. This is because the distance function is not so expensive. As a consequence, the sequential part of the algorithm represents a greater percentage of total time.

6. CONCLUSIONS AND FUTURE WORK

In this work, we have presented different implementations of the similarity search using *Spaghettis* structure for parallel platforms based on multi-core and many-core processors. In order to evaluate these implementations we have used two different databases, Spanish dictionary and color histograms.

When considering all the implementations, the brute force implementation included, the behavior of the structure in

both metric spaces is similar, obtaining results of speedup between 1.87 and 3.94 for multi-core implementation, and between 2.08 and 14.04 for the GPU-based platform. These speedup values have been obtained considering the same conditions (exhaustive or *Spaghettis* with the same pivots and range). Considering the best implementations we have obtained a maximum speedup of 3.17 for multi-core and 9.84 for GPU, both cases obtained for the Spanish dictionary, which uses a more expensive computational cost distance function.

Our future work is concerned with using different metric spaces and parallel platforms in order to compare the behavior of the structure over big size databases. Also, a scalability study of the implementations will be carried out.

ACKNOWLEDGMENTS

This work has been partially supported by the Spanish project SATSIM (Ref: CGL2010-20787-C02-02).

REFERENCES

[1] E. Chávez, G. Navarro, R. Baeza-Yates, and J. L. Marroquín, "Searching in metric spaces," in *ACM Computing Surveys*, September 2001, pp. 33(3):273–321.

[2] P. Ciaccia, M. Patella, and P. Zezula, "M-tree : An efficient access method for similarity search in metric spaces." in *the 23st International Conference on VLDB*, 1997, pp. 426–435.

[3] S. Brin, "Near neighbor search in large metric spaces." in *the 21st VLDB Conference*. Morgan Kaufmann Publishers, 1995, pp. 574–584.

[4] L. Micó, J. Oncina, and E. Vidal, "A new version of the nearest-neighbor approximating and eliminating search (AESA) with linear preprocessing-time and memory requirements," *Pattern Recognition Letters*, vol. 15, pp. 9–17, 1994.

[5] R. Baeza-Yates, W. Cunto, U. Manber, and S. Wu, "Proximity matching using fixedqueries trees." in *5th Combinatorial Pattern Matching (CPM'94)*, ser. LNCS 807, 1994, pp. 198–212.

[6] E. Chávez, J. Marroquín, and R. Baeza-Yates, "Spaghettis: An array based algorithm for similarity queries in metric spaces," in *6th International Symposium on String Processing and Information Retrieval (SPIRE'99)*. IEEE CS Press, 1999, pp. 38–46.

[7] E. Chávez, J. Marroquín, and G. Navarro, "Fixed queries array: A fast and economical data structure for proximity searching," *Multimedia Tools and Applications*, vol. 14, no. 2, pp. 113–135, 2001.

[8] W. Gropp, E. Lusk, and A. Skelljum, *Using MPI:Portable Parallel Programming with the Message Passing Interface*, ser. Scientific and Engineering computation Series. Cambridge, MA: MIT Press, 1994.

[9] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, B. Manchek, and V. Sunderam, *PVM: Parallel Virtual Machine – A User's Guide and Tutorial for Network Parallel Computing*. MIT Press, 1994.

[10] L. Dagum and R. Menon, "OpenMP: An industry-standard API for shared-memory programming," *IEEE Computational Science and Engineering*, vol. 5, no. 1, pp. 46–55, 1998.

[11] P. Zezula, P. Savino, F. Rabitti, G. Amato, and P. Ciaccia, "Processing M-trees with parallel resources," in *RIDE '98: Proceedings of the Workshop on Research Issues in Database Engineering*. Washington, DC, USA: IEEE Computer Society, 1998, p. 147.

[12] A. Alpkocak, T. Danisman, and U. Tuba, "A parallel similarity search in high dimensional metric space using M-tree," in *Advanced Environments, Tools, and Applications for Cluster Computing*, ser. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2002, vol. 2326, pp. 247–252.

[13] V. Gil-Costa, M. Marin, and N. Reyes, "Parallel query processing on distributed clustering indexes," *Journal of Discrete Algorithms*, vol. 7, no. 1, pp. 3–17, 2009.

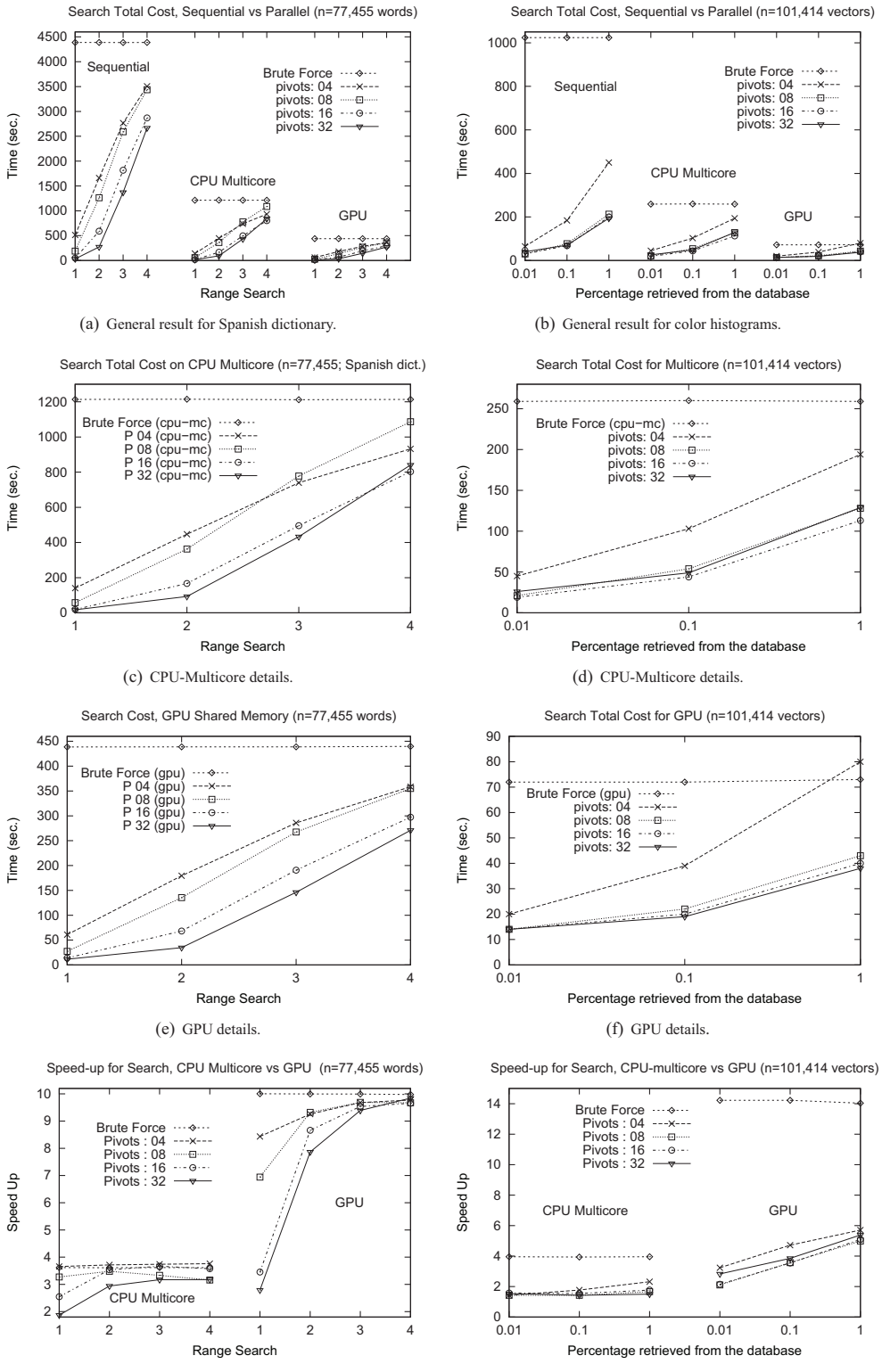
[14] V. Gil-Costa, R. Barrientos, M. Marin, and C. Bonacic, "Scheduling metric-space queries processing on multi-core processors," *Parallel, Distributed, and Network-Based Processing, Euromicro Conference on*, vol. 0, pp. 187–194, 2010.

[15] Q. Kuang and L. Zhao, "A practical GPU based kNN algorithm," *International Symposium on Computer Science and Computational Technology (ISCSCT)*, pp. 151–155, 2009.

[16] V. Garcia, E. Debreuve, and M. Barlaud, "Fast k nearest neighbor search using GPU," *Computer Vision and Pattern Recognition Workshop*, vol. 0, pp. 1–6, 2008.

[17] B. Bustos, O. Deussen, S. Hiller, and D. Keim, "A graphics hardware accelerated algorithm for nearest neighbor search," in *Computational Science (ICCS)*, vol. 3994. Springer, 2006, pp. 196–199.

[18] R. J. Barrientos, J. I. Gómez, C. Tenllado, and M. Prieto, "Heap based k-nearest neighbor search on GPUs," in *Congreso Español de Informática (CEDI))*, Valencia, Sept. 2010, pp. 559–566.



(a) General result for Spanish dictionary.

(b) General result for color histograms.

(c) CPU-Multicore details.

(d) CPU-Multicore details.

(e) GPU details.

(f) GPU details.

(g) Speed Up for Spanish dicctionary.

(h) Speed Up for color histograms.

Figure 2. Comparative results of search costs for the space of words (left) and color histograms (right) for *Spaghettis* metric structure on sequential CPU, Multicore CPU and GPU platforms. Number of pivots: 4, 8, 16 and 32.