View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by El Servicio de Difusión de la Creación Intelectual

Using Parallel Pivot vs. Clustering-based Techniques for Web Engines

Gil-Costa Veronica and Marcela Printista
DCC, Universidad de San Luis,
San Luis, Argentina
{gvcosta,mprinti}@unsl.edu.ar

July 25, 2007

Abstract

Web Engines are a useful tool for searching information in the Web. But a great part of this information is non-textual and for that case a metric space is used. A metric space is a set where a notion of distance (called a metric) between elements of the set is defined. In this paper we present an efficient parallelization of a pivot-based method devised for this purpose which is called the *Sparse Spatial Selection* (SSS) strategy and we compare it with a clustering-based method, a parallel implementation of the *Spatial Approximation Tree* (SAT). We show that SAT compares favourably against the pivot data structures SSS. The experimental results were obtained on a high-performance cluster and using several metric spaces, that shows load balance parallel strategies for the SAT. The implementations are built upon the BSP parallel computing model, which shows efficient performance for this application domain and allows a precise evaluation of algorithms.

Keywords: Metric Space, BSP, Parallel Search, Distance Computations.

1 Introduction

The World Wide Web has information useful to millions of people. Some simple browse the Web through entry points such as Yahoo!. But many information seekers use a search engine to begin their Web activity. In this last case usually users submit a query of keywords, and receive a list of Web pages containing the keywords that may be relevant. There is no question that the Web is a huge and challenging to deal with. For helping users to find their answers, the search engine module is responsible for receiving and filling search request from users. The engine relies heavily on the indexes and some time on the page repository. Because of the Web's size, and the fact that users typically only enter one or two keywords, result sets are usually very large. Therefore the ranking operation has the task of sorting the results. The query module is of special interest because traditional information retrieval (IR) techniques have run into selectivity problems when applied without modifications to Web searching: most traditional techniques rely on measuring the similarity of query texts with texts in a collection's documents.

With the growth of non-text content of the Web, it is becoming increasingly important to store, index and search over images, audio, and video collections. Metric spaces and similarity search are used for that kind of objects. The computational cost of the algorithms that determine the similarity

between two objects makes similarity search an expensive operation and a case for its efficient parallelization. This fact has motivated the development of many research works aiming to do efficient similarity search over very large collections of data.

Many research studies have been presented so far about multimedial structures. Some of them are Burkhard-Keller-Tree (BKT) [5], Fixed-Queries Tree (FQT) [2], Fixed-Height FQT (FQHT) [1], Fixed-Queries Array (FQA) [6], Vantage Point Tree (VPT) [22], Approximating and Eliminating Search Algorithm (AESA) [21], LAESA (Linear AESA) [12], Bisector Trees (BST) [9], Generalized-Hyperplane Tree (GHT) [17], Geometric Near-neighbor Access Tree (GNAT) [3] and Spatial Approximation Tree (SAT) [13]. These structures are used to perform similarity searches in a *metric space*.

In this paper we present a parallel strategy for the Sparse Spatial Selection (SSS) index which is a pivot based-technique. We present strategies to reduce the number of distances computations and the I/O requirements. We compare the efficiency of the algorithms implemented with the parallel implementation of a clustering-based technique, the Spatial Approximation Tree (SAT). We use the Bulk Synchronous Parallel - BSP [20] model to perform the parallel querying and data distribution.

The rest of this paper is organized as follow. Section 2 introduces the theoretical concepts needed to understand the problem. In Section 3 the sequential SSS index is presented. Section 4 presents the Sequential SAT algorithm. In Section 5 the parallel platform is explained. In Section 6 we present the parallel SSS index strategies and in Section 7 the parallel SAT algorithms. Section 8 preset the results obtained and finally Section 9 presents the conclusions and future works.

2 Theoretical Concepts

A *metric space* $(\mathbb{X}, d)$ is composed of a universe of valid objects $\mathbb{X}$ and a *distance function* $d : \mathbb{X} \times \mathbb{X} \rightarrow \mathbb{X}^+$ defined among them. The distance function determines the similarity or distance between two given objects. The goal is given a set of objects and a query, retrieval all objects close enough to the query. This function holds several properties: strictly positiveness ($d(x, y) > 0$ and if $d(x, y) = 0$ then $x = y$), symmetry ($d(x, y) = d(y, x)$), and the triangle inequality ($d(x, z) = d(x, y) + d(y, z)$). The finite subset $\mathbb{U} \subset \mathbb{X}$ with size $n = |\mathbb{U}|$, is called dictionary or database and represents the collection of objects where searches are performed.

A k-dimensional vector space is a particular case of metric space in which every object is represented by a vector of k real coordinates. The definition of the distance function depends on the type of objects we are managing. In a vector space, d could be a distance function of the family L_s, defined as $L_s(x, y) = (\sum_{1 \leq i \leq k} |x_i - y_i|^s)^{\frac{1}{s}}$ [4]. For example, $s = 2$ yields Euclidean distance, that is the number of insertions, detentions or modifications we have to performe to make two words equal.

The computational cost of the algorithms that determine the similarity between two objects makes similarity search an expensive operation and a case for its efficient parallelization. This fact has motivated the development of many research works aiming to do efficient similarity search over very large collections of data. Data parallel programming is particularly convenient for two reasons. The first is its easiness of programming. The second is that it can scale easily to large problem sizes.

There are two main queries of interest for a collection of objects in a metric space:

- *range search:* that retrieves all the objects $u \in \mathbb{U}$ within a radius r of the query q, that is: $\{u \in \mathbb{U} / d(q, u) = r\}$;
- *nearest neighbor search:* that retrieves the most similar object to the query q, that is $\{u \in \mathbb{U} / \forall v \in \mathbb{U}, d(q, u) = d(q, v)\}$;

In this paper we are devoted to range queries. Nearest neighbor queries can be rewritten as range queries in an optimal way [8], so we can restrict our attention to range queries. The evaluation of the distance function is very expensive, and therefore searches become inefficient if the collection has a high number of elements. Thus, reducing the number of evaluations of the distance function is the main goal of the methods for similarity search in metric spaces. To do that, they first build indexes over the whole collection. Later, using the triangle inequality, those indexes permit to discard some elements without being necessary to compare them against the query.

Search methods can be classified into two types [7]: *pivot-based* and *clustering-based* techniques. Pivot-based search techniques choose a subset of the objects in the collection that are used as pivots. The index is built by computing the distances from each pivot to each object in the database. Given a query (q, r), the distances from the query q to each pivot are computed, and then some objects of the collection can be directly discarded using the triangle inequality and the distances precomputed during the index building phase. Being $x \in \mathbb{U}$ an object in the collection, we can discard x if $|d(p_i, x) - d(p_i, q)| > r$ for any pivot p_i, since by the triangle inequality, if this condition is true, its distance to q will be $d(x, q) > r$. The objects that can not be discarded by this condition make up the candidate list, and they must be compared against the query. The total complexity of the search is the sum of the internal complexity, the comparisons of q with each pivot, and the external complexity, the comparisons of q with each object in the candidate list.

Clustering-based techniques split the metric space into a set of equivalence regions each of them represented by a cluster center. During searches, whole regions can be discarded depending on the distance from their cluster center to the query. Two good surveys can be found in [8] and [23].

A recent pivot-based technique is the *Sparse Spatial Selection* (SSS) [4]. SSS is a dynamic method since the collection can be initially empty and/or grow later. It works with continuous distance functions and it is suitable for secondary memory storage. The main contribution of SSS is the use of a new pivot selection strategy. This strategy generates a number of pivots that depends on the intrinsic dimensionality of the space. Moreover, this pivot selection strategy is dynamic since it adapts the index when new objects are added to the collection.

A recent clustering-based technique is the *Spatial Approximation Tree* (SAT) devised to support efficient searching in high dimensional metric spaces [13]. This structure has been compared successfully against other data structures [15] and update operations have been included in the original design [14]. The SAT is a nice example of tree data structure in which well-known tricks parallelization simply do not work [11, 10]. It is too sparse, unbalanced and its performance is too dependent on the workload generated by the queries being solved by means of searching the tree.

3 Sequential SSS Index

Let $(\mathbb{X}, d)$ be a metric space, $U \subset \mathbb{X}$ an object collection, and M the maximum distance between any pair of objects, $M = \max\{d(x, y)/x, y \in \mathbb{X}\}$. The set of pivots contains initially only the first object of the collection. Then, for each element $x_i \in \mathbb{U}, x_i$ is chosen as a new pivot if its distance to every pivot in the current set of pivots is equal or greater than $\alpha\, M$, being α a constant parameter. That is, an object in the collection becomes a new pivot if it is located at more than a fraction of the maximum distance with respect to all the current pivots. For example, if $\alpha = 0.5$ an object is chosen if it is located farther than a half of the maximum distance from the already selected pivots. The following pseudocode summarizes the pivot selection process:

PIVOTS $\leftarrow \{x_1\}$
for all $x_i \in \mathbb{U}$ **do**

```
if ∀ p ∈ PIVOTS, d(x_i, p) ≥ α M then
    PIVOTS = PIVOTS ← ∪ {x_i}
```

A key observation here is that the calculations performed to obtain the values of the distance function $d(x_i, p)$ during the construction of the SSS index are not discarded, they actually form the index itself. Namely for each pivot, the SSS index maintains the distance between each database object and all of the pivots. Thus solving the range query (q, r) takes the following steps:

```
foreach pivot p do dq[p]← d(q, p)
n ← 0
foreach object o do
    foreach pivot p do
        if  ( distance[o][p] > (dq[p]−r)  and
              distance[o][p] < (dq[p]+r) )  then
            n ← n + 1
        endif
    endfor

    if  ( n = total number of pivots )  then
        add object o to a list of candidate objects ℓ.
    endif
endfor
foreach object o ∈ ℓ do
    if  ( d(o, q) ≤ r ) then
        report object o as solution
    endif
endfor
```

It seems evident that all the selected pivots will not be too close to each other. Forcing the distance between two pivots to be greater or equal than $M\alpha$, ensures that they are well distributed in the whole space. It is important to take into account that the pivots are not very far away from each others neither very far from the rest of objects in the collection (i.e., they are not *outliers*), but they are well distributed covering the whole space. The hypothesis is that, being well distributed in the space, when a search is performed the set of pivots will be able to discard more objects than pivots selected with a different strategy.

Being dynamic and adaptive is another good feature of the pivot selection technique. The set of pivots adapts itself automatically to the growing of the database. When a new element x_i is added to the database, it is compared against the pivots already selected and it becomes a new pivot if needed. In this way the number of pivots does not depend on the collection size but on its intrinsic dimensionality of the metric space. Actually the collection could be initially empty, which is interesting in practical applications.

Although in this method it is not necessary to state in advance the number of pivots to use, it is necessary to set the value of α. This value determines the number of pivots. It is clear that the bigger the value of α, the smaller the number of pivots that can be "placed" into the space. However, α must always take values between 0.35 and 0.40, depending on the intrinsic dimensionality of the space. That is, the optimal results in SSS are always obtained when α is set to those values and in general a higher α works better when the intrinsic dimensionality is higher [4].

4 Sequential SAT

The SAT construction starts by selecting at random an element a from the database $S \subset U$. This element is set to be the root of the tree. Then a suitable set $N(a)$ of neighbours of a is defined to be the children of a. The elements of $N(a)$ are the ones that are closer to a than any other neighbour. The construction of $N(a)$ begins with the initial node a and its bag holding all the rest of S. We first sort the bag by distance to a. Then we start adding nodes to $N(a)$ (which is initially empty). Each time we consider a new node b, we check whether it is closer to some element of $N(a)$ than to a itself. If that is not the case, we add b to $N(a)$. We now must decide in which neighbour's bag we put the rest of the nodes. We put each node not in $a \cup N(a)$, but in the bag of its closest element of $N(a)$. The process continues recursively with all elements in $N(a)$.

The structure is a tree that can be searched for any $q \in S$ by spatial approximation for nearest neighbour queries. The mechanism consists in comparing q against $a \cup N(a)$. If a is closest to q, then a is the answer, otherwise we continue the search by the subtree of the closest element to q in $N(a)$.

It is a little interest to search only for elements $q \in S$. The tree we have described can, however, be used as a device to solve range queries for any $q \in U$ with radius r. The key observation is that, even if $q \notin S$, the answer to the query are elements $q' \in S$. So we use the tree to pretend that we are searching an element $q' \in S$. Range queries q with radius r are processed as follows. We first determine the closest neighbour c of q among $\{a\} \cup N(a)$. We then enter into all neighbours $b \in N(a)$ such that $d(q, b) \leq d(q, c) + 2r$. This is because the virtual element q' can differ from q by at most r at any distance evaluation, so it could have been inserted inside any of those b nodes. In the process we report all the nodes q' we found close enough to q. Finally, the covering radius $R(a)$ is used to further prune the search, by not entering into subtrees such that $d(q, a) > R(a) + r$, since they cannot contain useful elements. The following pseudocode summarizes the range search:

SEARCH(Node a, Query q, Radius r, Dist. dmin)
if (d(a,q) $\leq$ R(a)+r) **then**
    if (d(a,q) $\leq$ r) **then**
        report a
    endif
    $d_{min} = min\{d(c, q), c \in N(a)\} \cup d_{min}$
    for (b $\in$ N(a)) **do**
        if (d(b,q) $\leq$ d_{min}+2r) **then**
            RangeSearch(b,q,r,d_{min})
        endif
    endfor
endif

5 Parallel Platform

In the BSP model of parallel computing [20], any parallel computer is seen as composed of a set of P processor local-memory components which communicate with each other through messages. The computation is organized as a sequence of supersteps. During a superstep, the processors may perform sequential computations on local data and/or send message to others processors. The messages are available for processing at their destination by the next superstep, and each superstep is ended with the barrier synchronization of processors [16].

The practical model of programming is SPMD, which is realized as C and C++ program copies running on P processors, wherein communication and synchronization among copies are performed by ways of libraries such as BSPlib [18] or BSPpub [19].

We choose the BSP model because it is deadlock-free and has a particular way of organizing computations and the resulting performance is in fact not too far from the one obtained with fully asynchronous message passing realizations. Its main advantage stems from the fact that BSP provides a cost model that allows a seemingly precise evaluation of the communication and computation costs of parallel programs. In this particular work, we have used the BSPonMPI library that allows running BSP using the MPI primitives.

The environment selected to process the queries is a cluster of computers connected by fast switching technology. We assume a server operating upon a set of P machines, each containing its own memory. Client request are sent to a broker machine, which in turn distribute those request evenly onto the P machines implementing the server. Requests are queries that must be solved with the data stored on the P machines. We assume that under a situation of heavy traffic the server start the processing of a batch of $Q = q\,P$ queries in every superstep. Basically every processor has to deal with two kinds of messages, those from newly arriving queries coming from the broker, in which case a search is started in the processor, and those from queries located in others processors that decided to continue their search locally in this processor.

We assume that the broker distribute $Q = q\,P$ queries in every superstep so that q new queries arrive at each processor in each superstep. In this case, it is not difficult to see that the cost of the broadcast operation we employ is $O(q\,P + q\,P\,G + L)$ against the $O(q\,P + q\,P^2\,G + L)$ common practice strategy reported in the literature.

6 Parallel Strategies for the SSS Index

The SSS index is a pivot-based technique where distances between the objects in the collection and the pivots are computed before the search operation starts. It can be seen as a table with so many columns as pivots and so many rows as objects in the database. This strategy can be easily parallelizable distributing the rows among the processors. But in this case the pivots must be replicated in all processors because the query objects have to be compared against them. The replication is due to the SSS index performs an intersection between the database objects and the pivots. Therefore all objects with $d(o, piv) \in \{d(q, piv) + r, d(q, piv) - r\}$ for every pivot in the set of pivots, are selected as candidates to be part of the answer. The non-pivot objects are distributed in a multiplexed way.

This method can be seen as a global strategy because all processors share the same pivots but they have only a piece of the whole database. With this approach we can use two different strategies to perform the queries searches. In the first one called *Strategy-B*, the broker machine sends the query to one processor of the system selected in a circular way as in the first strategy. Then the processor receiving the query performs a broadcast so all processors get the same query. After receiving this query, processors search in their local SSS index the most similar objects, namely the objects that satisfies the range query (q, r) and send them to the requesting processor. This processor finally waits for all the results obtained in the previous step, and sends the results to the broker machine (see Figure 1). For the case of queries requiring the nearest k objects within distance r – we denote this operation (q, r, k) – every processor sends its k nearest objects and the requesting processor determines the best k among the $k\,P$ candidates.

The second strategy applied to this global index approach, called *Strategy-C*, is to build the results for a specific query in several steps. So many steps as processors are in the system (see Figure 1). In this scheme when a new query arrives from a broker machine, the receptionist processor performs a

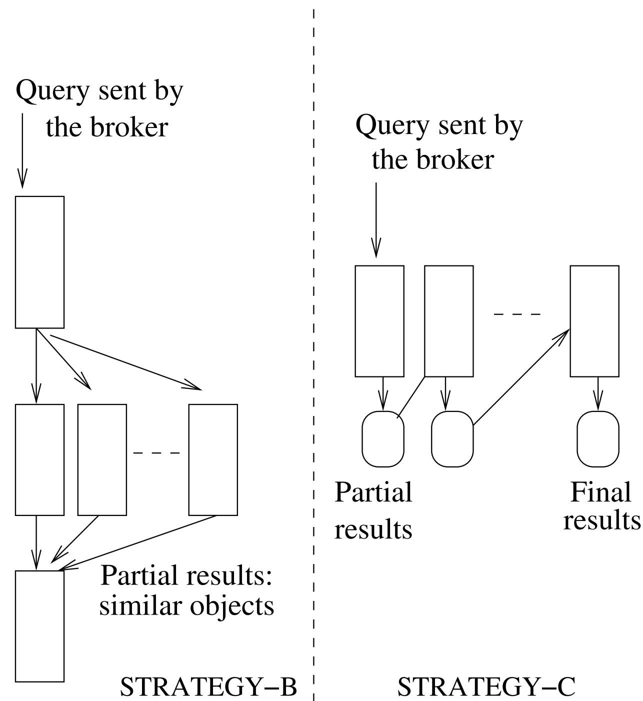

Figure 1: Global index distribution.

local similarity search using its portion of the SSS index. This processor gets a partial result for the query and sends it to the next processor. This is repeated until all processors have searched in their local indexes for similar objects. And the last one sends the final results to the broker machine. For k-nearest queries (q, r, k), together with the message travelling from one processor to the another there is an indication of the maximum distance to q of the set of K candidates contained in the message. This value is used to prone the number of distance calculations in the receiving processor.

This case is very similar to the Strategy-B, but here the communication is performed in smaller chunks. The results are passed from one processor to another until building the final results, so we have P messages with partial results for every query, and all processors are performing the same tasks in every superstep. While in the other strategy, the communication is performed in bigger chuncks and the receptionist machine has to perform the selection of the best results.

Another way to parallelize the SSS index would be to spread the space between the processors and then each one builds its own SSS index with the local data (*Strategy-A*). The problem in this strategy is that each processor will have to select new pivots locally, different from the global ones. And therefore, the number of pivots will be greater than in the other strategies proposed. To avoid the effects of imbalance from objects skewed to particular regions in space, we use multiplexed object distributed strategy, so each object is sent to the processor $p = id_{obj} mod\, P$.

To process a query using this scheme, the broker machine sends the query to one processor from the system selected in a circular way, and this processor sends a copy of the query to all other processors ($O(q\,P + q\,P\,G + L)$ broadcast cost). Then, using the local SSS indexes each processor searches for the most similar objects to the query. The results obtained are sent to the broker machine.

7 Parallel SAT Strategies

A first point to emphasize is that the SAT structure contains nodes of very diverse number of children. Every child node causes a distance comparison, so it is relevant to be able to balance the number of distance comparisons performed in every processor per superstep. In the first parallel strategy for the SAT, we propose to distribute the subtree of the root node among processors at random (R strategy). In this case the root is replicated in all processors and queries are distributed in a circular way. The processor receiving the query determines where it has to be solved.

Another parallel implementation considers the number of nodes that each processor has. So we select the processors with fewer nodes to send a subtree. Queries are distributes in a circular way and the processor receiving one query will determinate where the query must be solved. But, while we were doing some experiments to study the results obtained by this strategy (load processor strategy - LP), we realize the number of comparisons performed depends not only in the number of nodes but also in the query itself. Therefore to reduce the number of comparison distance we present another strategy to map the tree nodes onto the processors by considering the number of distance comparisons that may be potentially performed in every subtree rooted at the children of the SAT's root. That is, the subtrees associated with nodes b in $N(a)$ where a is the root and $N(a)$ is the set neighbour of a. To do that, we replicate the root and each child of the root in every processor and we distribute all the others nodes evenly through the processors, in a multiplexed way. A disadvantage is that every node has to replicate its children locally, to be able to perform the distance comparisons and in this way continue the searching operation.

In the multiplexed strategy (M), we also have to send the query to one processor and then it determinates where the query has to be solved. Therefore, we have more communication and more synchronization during a query processing operation.

These three strategies have a global distribution, because the SAT is sequentially built and then the nodes are distributed in the server. Another way to parallelize this structure (named local strategy L) is to distribute the database among the processors and then each processor builds its own local SAT structure. This case requires broadcasting the queries, because there is no communication between the processors during the query search operation and because they process these queries in a sequential way.

To improve efficiency we set an upper limit V to the number of distance comparisons that are performed per processor in each superstep. During a superstep, every time any processor detects that it has performed more than V distance comparisons, it suspends query processing and waits until the next superstep to continue with this task. Under the BSP model it means that all queries going down in a tree in each processor k has to be sent again to the processor k as a message, exactly as if it found out that the search have to continue in other processor. But no communication cost is involved for these extra messages. Also the processors stop extracting messages from its input queue. Besides, every S supersteps we collect statistics that are used to define the value of V for the next sequence of supersteps. This statistics are independent of the value of V and of the S supersteps used to calculate them. In this way the value of V can adapt itself to the workload changes produced by the flow of queries arriving constantly to the server.

Because of limit V, supersteps can be truncated before processing all the available queries. Therefore real supersteps are not a reliable measure of the real average number of supersteps required to complete a query. To deal with this, we put in every query q a counter of virtual supersteps different from the real ones executed by the BSP computer. Also, we keep a counter for the virtual supersteps in each processor k. Every time a new query is initialized in a processor k we set the virtual supersteps of the query to be equal to the number of batch it belongs to. The broker can do this before sending the

query to the processor. Besides, every time a query has to migrate to another processor we increase the virtual supersteps in one unity, because it takes one virtual superstep to get there. Additionally, we count the total number of distance calculations that has been performed in every processor k. It gives us a precise idea of global load balance (across supersteps).

8 Results

The database collection used in the experiments showed below is a 69Kwords English dictionary and a 51Kwords Spanish dictionary. This system has 32 dual processors (2.8GHz) that use NFS mounted directories. Queries were selected at random from a log of 127,000 queries. In each superstep we introduce $Q = 32$ queries per processor and each of them solves 10,000 queries. So the total number of queries processed in the system is 10,000 P.

We measure the efficiency of each strategy explained in this paper as the ratio A/B where A is the average distance computation performed in each superspsptep by all processors, and B is the maximum number of distance computations performed in that superstep by any processor. This measure gives us an idea about the load balance of the system.

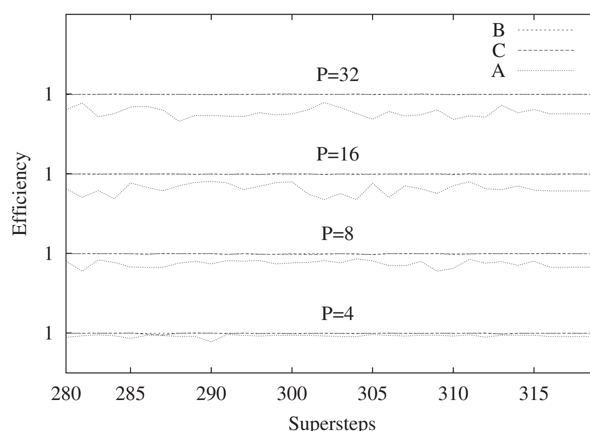

Figure 2: Efficiency obtained by the Strategy-A, Strategy-B and Strategy-C using up to 32 processors with the SSS technique.

Figure 2 shows the efficiency in each superstep. Strategies B and C obtains a good performance (they are overlapped in the graphic) and they show a better load balance than the Strategy-A. This last one presents fluctuations and the efficiency is more unstable. This is because queries requiring more distance computations may fall in the same processor, while in both others strategies the work is distributed among the processors. Figure 3 shows the efficiency obtained by the proposed parallel strategies for the SAT. In this case the L strategy is the one presenting less efficiency and therefore less load balance between supersteps.

Figure 4 shows the running time in seconds divided by the maximum running time required by the parallel implementations. The running time in this graphic increase with a greater number of processor because the amount of queries processed is also larger. With this scheme we can see if the strategies presented allows obtaining scalability. Here, we can see that Strategy-A obtains lower running times than the two others strategies based in a global distribution approach for the SSS index. This strategy only performs a broadcast at the beginning of processing each query, and then the process is performed locally and some processors concentrate a great number of distances computations

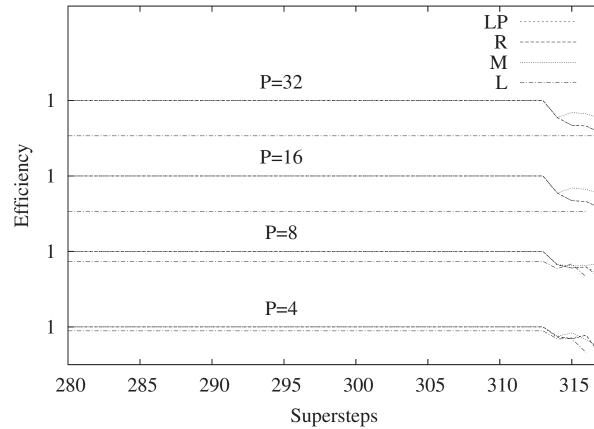

Figure 3: Efficiency obtained with 4,8,16 and 32 processors using the LP, R, M and L parallel strategies proposed for the SAT method.

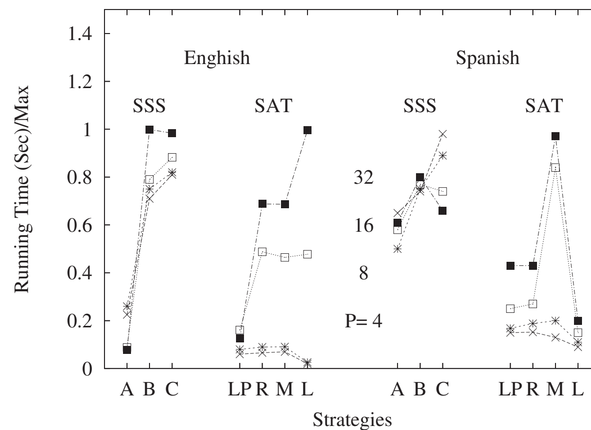

Figure 4: Running time. Each machine process $10,000P$ queries in batches of $Q = 32$. The results are presented with a system with up to 32 processors.

in some processors, while other may remind with less load work.

The parallel SAT strategies proposed used an upper limit V to balance the number of computations performed in each superstep. Figure 5 shows in the $y - axis$ the values obtained by the upper limit per superstep. This is presented for 4-32 processors using the LP, R, M and L strategies for the SAT.

Finally, Figure 6 shows in the x-axis the parallel strategies A, B and C for the SSS index and LP, R, M and L for the SAT. At the left of this graphic we have the average distance computations performed using 4, 8, 16 and 32 processors for the SSS index. In this case the B strategy is the one reporting higher values. On the other hand, the different distance computations presented by the execution of the algorithms with more processors is hard to distinguish. The average efficiency is presented. We can see that the A strategy is the one reporting less distance computations but also has less load balance due to the efficiency reported is very low.

At the right of this figure, we present the average distance computations and efficiency obtained by each strategy of the SAT method with up to 32 processors. We can see that the behaviour stands like the reported before in Figure 3, so the L strategy is the one with less uniform work distribution.

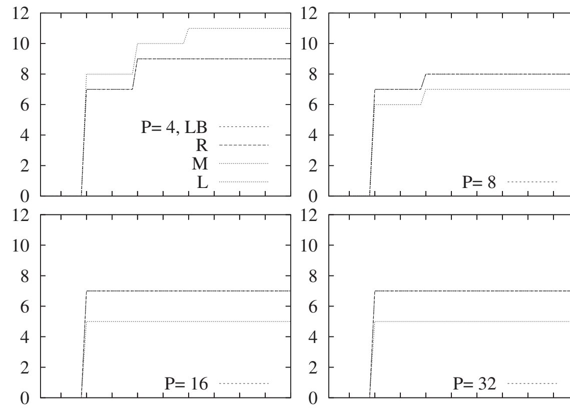

Figure 5: Upper limits adaptation for the parallel strategies of the SAT.

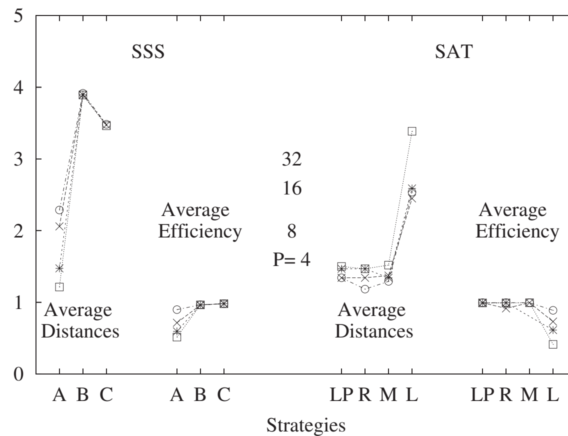

Figure 6: Average measures taken for the three strategies with 4,8,16 and 32 processors, using the English dictionary.

9 Conclusions and Future Works

In this paper we have presented the parallelization of a recent index method used to search similar objects in a metric space and we compared it with some parallel implementations of a clustering-based technique, the SAT. Results show that the SAT is more balance and obtains a better performance decreasing the number of distance computations. The SSS index allows obtaining an optimal number of pivots used to reduce the number of distance computations performed for searching similar objects.

The parallel strategies are based in two main schemes, a local scheme where the database is distributed among the processors and then each processor builds its own index or tree to perform the queries searches. The others strategies are based in a global distribution of the database. In these cases a unique SSS-index is built and then all pivots are replicated in all processors (because the intersection between queries and pivots is required by the SSS method). The database objects are distributed among the processors in a multiplexed way. In the SAT method a unique tree is built and the each node is multiplexed among processors.

As future work we intend to explore the parallel application of this method and compare it with others metric index structures like the EGNAT, and GNAT. The effect of external memory is also an interesting area applied to similarity search problems where parallel computing can allow a significant

reduction of query execution times.

References

[1] R. Baeza-Yates. Searching: an algorithmic tour. *Encyclopedia of Computer Science and Technology*, 37:331–359, 1997.

[2] R. Baeza-Yates, W. Cunto, U. Manber, and S.Wu. Proximity matching using fixed-queries trees. *In Proceedings of the 5th Annual Symposium on Combinatorial Pattern Matching*, pages 198–212, Springer-Verlag, Berlin,1994.

[3] S. Brin. Near neighbor search in large metric spaces. *In 21st conference on Very Large Databases*, 1995.

[4] N. R. Brisaboa, A. Farina, O. Pedreira, and N. Reyes. Similarity search using sparse pivots for efficient multimedia information retrieval. In *ISM '06: Proceedings of the Eighth IEEE International Symposium on Multimedia*, pages 881–888, Washington, DC, USA, 2006. IEEE Computer Society.

[5] W. A. Burkhard and R. M. Keller. Some approaches to best-match file searching. *Communications of the ACM*, 4(16):230–236, 1973.

[6] E. Chavez, J. L. Marroquyn, and G. Navarro. Overcoming the curse of dimensionality. *In European Workshop on Content-based Multimedia Indexing (CBMI99)*, pages 57–64, 1999.

[7] E. Chavez, G. Navarro, R. Baeza-Yates, and J. L. Marroquyn. Searching in metric spaces. *ACM Computing Surveys*, 3(33):273–321, 2001.

[8] E. Chavez, G. Navarro, R. Baeza-Yates, and J. L. Marroquyn. Searching in metric spaces. *ACM Computing Surveys*, 3(33):273–321, 2001.

[9] I. Kalantari and G. McDonald. A data structure and an algorithm for the nearest point problem. *IEEE Transactions on Software Engineering*, pages 631–634, 1983.

[10] M. Marin. Range queries on distributed spatial approximation trees. *International Conference on Databases and Applications (DBA'05)*, Innsbruck, Austria,Feb. 2005.

[11] M. Marin and N. Reyes. Efficient parallelization of spatial approximation trees. *International Conference on Computational Science (ICCS 2005), Lecture Notes in Computer Science 3514 (10031010), (SpringerVerlag)*, Atlanta, May 2005.

[12] L. Mico, J. Oncina, and R. E. Vidal. A new version of the nearest-neighbor approximating and eliminating search (aesa) with linear pre-processing time and memory requirements. *Pattern Recognition Letters*, 15:9–17, 1994.

[13] G. Navarro. Searching in metric spaces by spatial approximation. *n Proceedings of String Processing and Information Retrieval (SPIRE99)*, pages 141–148, IEEE CS Press, 1999.

[14] G. Navarro and N. 3. Fully dynamic spatial approximation trees. In *In Proceedings of the 9th International Symposium on String Processing and Information Retrieval (SPIRE 2002),*, 2002.

[15] S.Berchtold, C. Bohm, and D. Kein. Searching in highdimensional spaces: Index structures for improving the performance of multimedia databases. In *ACM Computing Surveys, (33(3):322373)*, 2001.

[16] D. Skillicorn, J. Hill, and W. McColl. Questions and answers about BSP. Technical Report PRG-TR-15-96, 1996. Also in *Journal of Scientific Programming*, V.6 N.3, 1997.

[17] J. K. Uhlmann. Satisfying general proximity/similarity queries with metric trees. *AInformation Processing Letters*, 40:175–179, 1991.

[18] URL. BSP and Worldwide Standard, http://www.bsp-worldwide.org/.

[19] URL. WWW.BSP PUB Library ar Paderborn Univertity, http://http://www.uni-paderborn.de/bsp/.

[20] L. Valiant. A bridging model for parallel computation. *Comm. ACM*, 33:103–111, Aug. 1990.

[21] E. Vidal. An algorithm for finding nearest neighbors in (aproximately) constant average time. *Pattern Recognition Letters*, 4:145–157, 1986.

[22] P. Yianilos. Data structures and algorithms for nearestneighbor search in general metric spaces. *ACM Press*, pages 311–321, 1993.

[23] P. Zezula, G. Amato, V. Dohnal, and M. Batko. *Similarity search. The metric space approach, volume 32 of Advances in Database Systems*, Springer, 2006.