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

A Parallel Search Algorithm for the SAT

V. Gil Costa, A. M. Printista, N. Reyes [*]
University of San Luis
San Luis, Argentina
{gvcosta,mprinti,nreyes}@unsl.edu.ar

M. Marín
Computing Department
University of Magallanes
Punta Arenas, Chile
mmarin@ona.fi.umag.cl

ABSTRACT

In order to be able to perform multimedia searches (like sounds, videos, images, etc.) we have to use data structures like the Spatial Approximation Tree (SAT). This structure is a nice example of a tree structure in which well-known tricks for tree parallelization simply do not work. It is too sparse, unbalanced and its performance is too dependent on the work-load generated by the queries being solved by means of searching the tree. The complexity measure is given by the number of distances computed to retrieve those objects close enough to the query. In this paper we examine some alternatives to parallelize this structure through the MPI library and the BSPpub library.

Keywords: SAT, metric spaces, searches, MPI, BSP, distances evaluations.

1. INTRODUCTION

Data parallelism is one of the most successful efforts to introduce explicit parallelism to high level programming languages. The approach is taken because many useful computation can be framed in terms of a set of independent sub-computations, each strongly associated with an element of a large data structure. Such computations are inherently parallelizable. 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.

One of these problems is the search in metric spaces by spatial approximation. A metric space is formed by a collection of objects U and a distance function d defined among them, which satisfies the triangle inequality. The goal is given a set of objects and a query, retrieve those objects close enough to the query [3]. The Spatial Approximation Tree (SAT) is a data structure devised to support efficient searching in high-dimensional metric spaces [6, 7]. It has been compared successfully against other data structures [2, 4] and update operations have been included in the design of the SAT[1, 8].

Some applications for the SAT are non-traditional databases (e.g. storing images, fingerprints or audio clips, where the concept of exact search is not used and we search instead for similar objects); text searching (to find words and phrases in a text database allowing a small number of typographical or spelling errors); information retrieval (to look for documents that are similar to a given query or document); etc.

A typical query for this data structure is the $range\ query$ which consists on retrieving all objects within a certain distance from a given query object. The distance in a dimensional space is expensive to compute and is usually the relevant performance metric to optimize, even over the secondary memory operation cost [1][3]. This problem is more significant in very large databases, making it relevant to study efficient ways of parallelization.

In this paper we propose three parallel algorithms for range query operations for the SAT data structure using the MPI library [9] and the BSPpub [12].

2. SEQUENTIAL SAT

The SAT construction starts by selecting at random an element a from the database $S \subseteq U$. 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 neighbours 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 resulting 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 $N(a)$. Some comparations are saved at search time by storing at each node a its covering radius $R(a)$, i.e., the maximum distance between a and any element in the subtree rooted by a.

It is 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

[*] This work has been partially supported CYTED VII.19 RIBIDI Project (all authors)

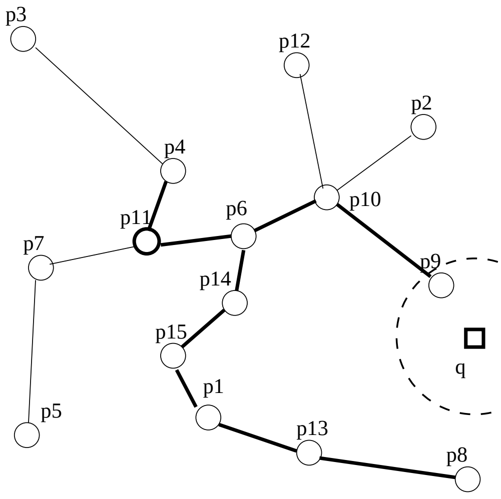

Fig. 1: An example of the search process in a SAT.

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' sought 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.

Besides, we can use another improvement to prune the search. At any node b of the search we keep the track of the minimum distance d_{min} to q seen up to now across this path, including neighbours. We enter only neighbours that are not farther than $d_{min} + 2r$ from q.

Figure 1 illustrates the search process, starting from p_{11} (tree root). Only the element p_9 is in the result, but all the bold edges are traversed.

We depict below the algorithm to search an element $q \in U$ with radius r in a SAT. It is firstly invoked as RangeSearch (a,q,r,$d(a,q)$), where a is the root of the tree. It can be noticed that in the recursive invocations $d(a,q)$ is already computed.

```
procedure RANGESEARCH(Node a, Query q,
    Radius r, Dist. d$_{min}$)
  if (d(a,q) $\leq$ R(a)+r) then
    if (d(a,q)$\leq$ r) then
      report a
    end if
    d$_{min}$ $\leftarrow$ 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$_{min}$)
      end if
    end for
  end if
end procedure
```

3. EXPERIMENTAL ENVIRONMENT

The implementations of the proposed strategies were performed using the MPI and the BSPpub libraries. The database used in our experiments is a 69K-word English dictionary and queries are composed by words selected uniformly at random. The distance is the edit distance, that is, the minimum number of character insertions, deletions, and replacements to make two strings equal. We assume a demanding case in which each query has one word and the search is performed with four different radiuses (1,2,3 and 4).

For the parallelization of the SAT, we assume a server operating upon a set of P machines, each containing its own memory. Clients request service to a $broker$ machine, which in turn distributes those requests 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 queries.

4. PARALLEL SEARCH ON SAT

Now we will be describe and we will analyze three strategies that can be used for the parallelization of the SAT data structure.

A first, but intuitive, approach to parallelization is simply assume that the processors have enough memory to maintain each one a complete copy of the SAT data structure. In this case the queries are distributed evenly onto the processors and their processing is straightforward as we just apply the sequential algorithm locally. No interprocessors communication is required and every query can be solved in just one step.

Fig. 2 shows results for this strategy with a search radius 1,2,3 and 4, $P = 4$ processors and 10 batches of queries. Here the SAT is initialized with the 90% of the dictionary words and the remaining 10% are left as query objects (randomly selected from the whole dictionary). As the radius grows up also does the number of distances computations, because we can compare the queries elements with more objects of the SAT than with a small radius.

Then the Fig. 3 and the Fig. 4 shows the results for a search with radius 1 and 2, the experiments performed with radius 3 and 4 have the same behavior presented in these figures. Here as the percentage of the database is increased, the query search cost is reduced because the number of queries is smaller. But this strategy presents fluctuations where the maximum numbers of distance computations are performed with the 50% of the database. These graphs show how dependent is this strategy on the query distribution. Beside the overconsumption memory problem, this strategy is not convenient since it in fact is not able to achieve a good performance.

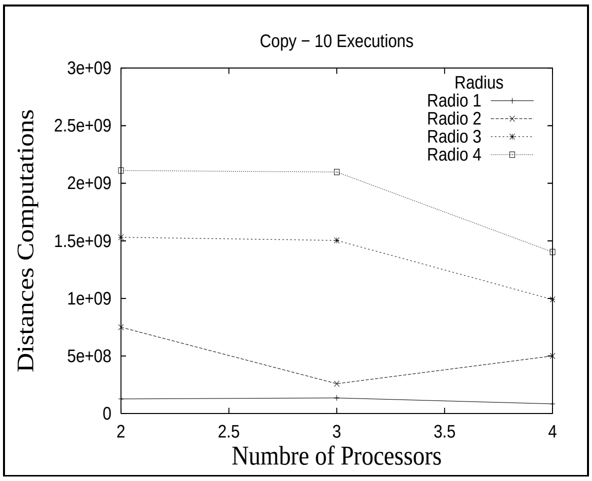

Fig. 2: The copy approach for 10 batches of queries with radius 1,2,3 and 4.

Therefore, this strategy does not present a parallelism overhead, but it neither can scale, because to build the SAT tree it is necessary that each processor has the complete database in its local memory, and as the database grows up, the construction cost also does.

Finally, the workload of the processors will depend on the queries that each one gets, since all they perform the same operations. In the following we propose a different approach to parallelize the SAT data structure.

A second approach to parallelize the SAT, would be to divide the problem in parts and to distribute them among the processors of the server. In this second strategy, a type of well-known partition named domain decomposition is used, where the data associated to the problem are divided and then each parallel task works on a portion of these data. Here the database is divided and distributed among the processors at random. Once each processor gets its portion of the database, it can begin to build the SAT structure using its local data.

All the queries go to the processors, because the objects can be from different data types (sound, text, videos, etc.), and we do not know which processor has information for the query.

Next, the Fig. 5 shows the results obtained for the execution of 10 batches with this strategy using up to 10 processors, with a radius between one and four. With a small radius, as the number of processors is increased the cost is reduced, but with a bigger radius this behavior changes, due the processors has to explore a bigger space to find the answers.

In this strategy, as in the previous one, it is not necessary to send and to receive messages during the search process over the tree, because each processor works using the tree stored in its own memory locally. Communication only exists at the beginning of the search operation, when the queries are sent to the processors. But contrary to the previous case, the communication is bigger, because the queries are sent to the machines (broadcast). The goal of using a random distribution of the database, is to minimize the unbalance presented during the queries search, since the number of distances computations in each pro-

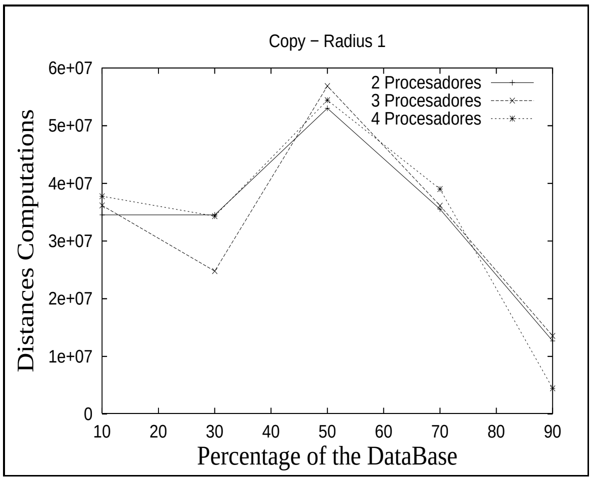

Fig. 3: Results obtained with the copy strategy with radius 1 using different size of a textual database.

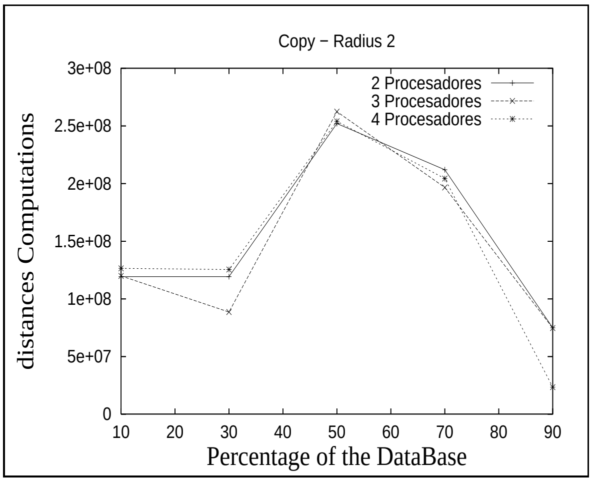

Fig. 4: Results obtained with the copy strategy with radius 2 using different size of a textual database.

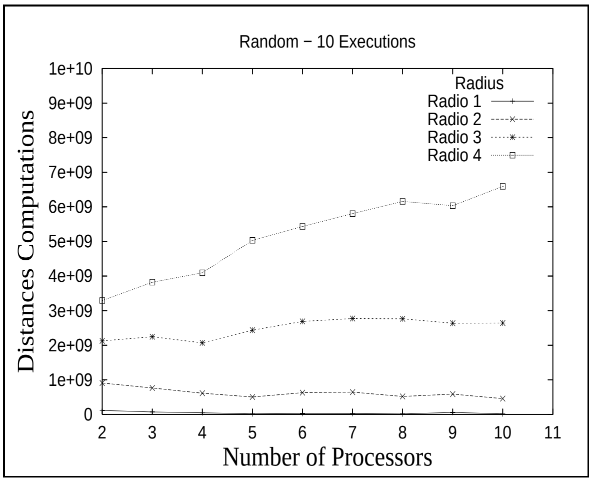

Fig. 5: Results obtained with the random approach using the 90% of the database with radius 1,2,3 and 4.

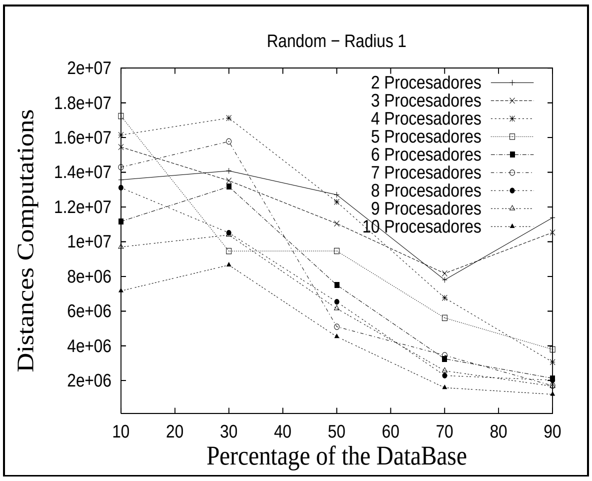

Fig. 6: Distances evaluations computed using the random strategy with radius 1 for different number of processors.

cessor will depend on the data that it has received to build its SAT tree, and on the query that is being processed.

The Fig. 6 and Fig. 7 show the results for this second strategy (Random). With a small radius we can see that as the number of processors is bigger we have less distances evaluations. Now with a bigger radius if we use more than the 50% of the database this holds; but with less than the 50% of the database the results are very dependent on the queries.

Finally we present the last strategy for the SAT data structure, where not only the database and the queries are distributed through a hash function. This kind of strategies may be use to perform searches in a Web dictionary, where the queries search are not exact. That is to say when the system has to find similar words. So this strategy can be used when the database from where the objects of the metric space are obtained, is formed by text (words).

The Fig. 8 shows the results for this strategy with the 90% of the database, and the Fig. 9 and Fig. 10 show the results as the percentage of the database is increased.

In this case we have a more concentrated space, because all the similar words from the dictionary go to the same processor, and that implies a harder search metric space due the triangle inequality permits discarding less elements.

5. COMPARATION OF THE STRATEGIES

The Fig. 11 and Fig. 12 show the distance evaluations obtained by each strategy with $P = 4$ processors as the database size is increased. In all the cases (working with radius 1,2,3 and 4), the second strategy shows a better performance than the others. The reason why the random strategy presented allows to reduce the number of distance evaluations is because the elements of the database are spread over the processors, and each one will have an easy space where the elements are sparse, and in this case the SAT structure performs better than in a hard space [6].

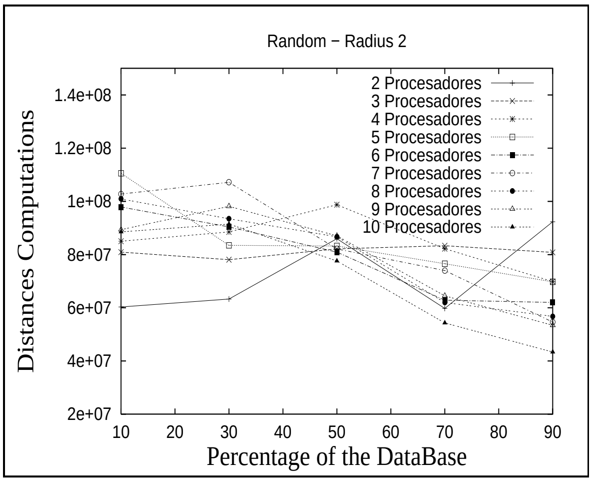

Fig. 7: Distances evaluations computed using the random strategy with radius 2 for different number of processors.

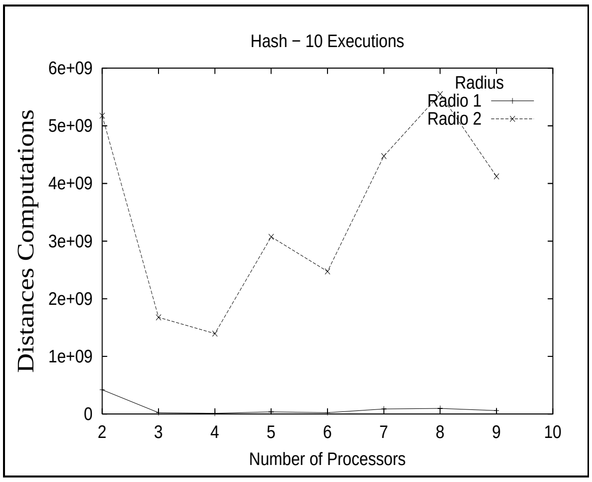

Fig. 8: Results obtained with the hash strategy using the 90% of the database with radius 1 and 2.

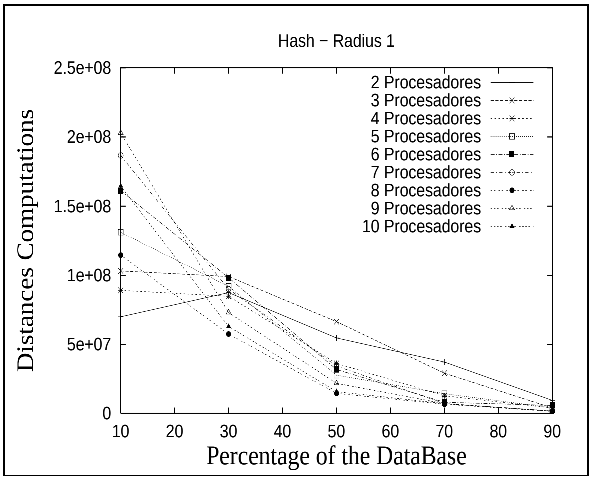

Fig. 9: **Distance evaluations computed with the hash strategy with radius 1.**

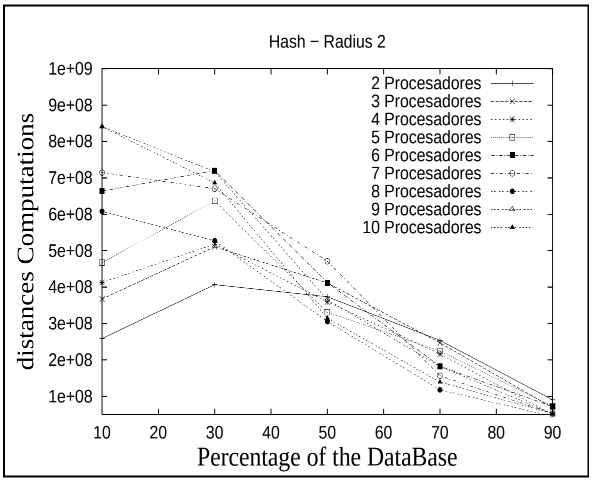

Fig. 10: **Distance evaluations computed with the hash strategy with radius 2.**

Lastly the Fig. 13 shows the results for the copy strategy with four processors, the random with ten processors and the hash with eight processors. Here you can see that the random strategy outperform the other two, while the hash strategy works well with a big database size, where each processor can work with more different elements.

6. FINAL COMMENTS AND FUTURE WORKS

In this paper we examine some alternatives to parallelize the search on the SAT structure using both MPI[9] and BSPpub[12] libraries. The differences presented between both implementations are exclusive of the characteristics that each library has, and how the programming model adapts to the problem.

The BSP model [10] establishes a new style of parallel programming to write programs of general purpose, whose main characteristic are its easiness and writing simplicity, its independence of the underlying architecture (portability). BSP achieves the previous properties elevating the abstraction level with which the programs

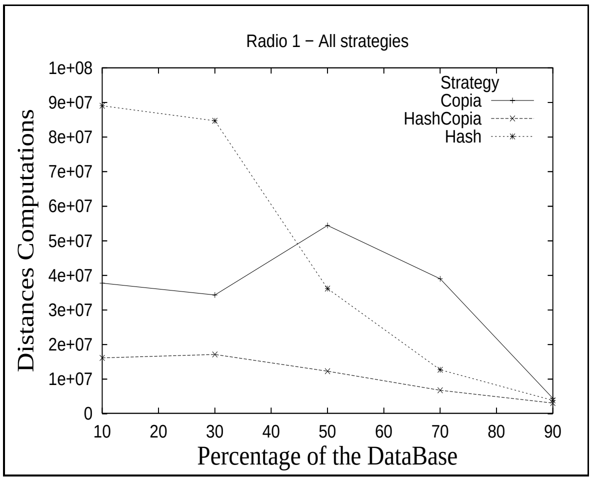

Fig. 11: **Number of distance evaluations obtained by the three strategies with $P = 4$ processors and radius 1, as the database size is increased.**

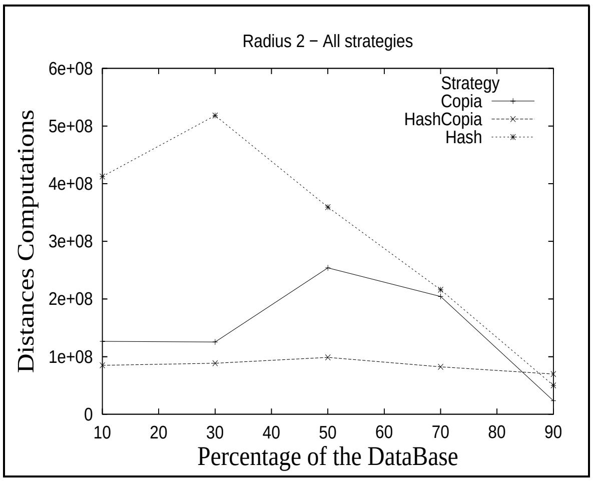

Fig. 12: **Number of distance evaluations obtained by the three strategies with $P = 4$ processors and radius 2, as the database size is increased.**

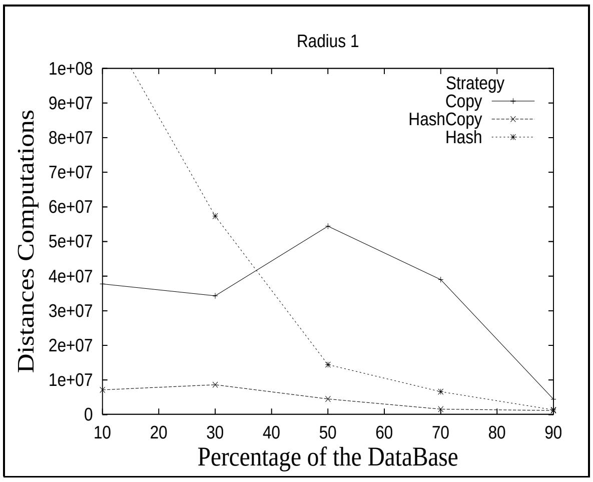

Fig. 13: **Best results obtained by each strategy proposed in this work, as the database size is increased and the number of queries is smaller.**

are written.

With MPI, we do not need to synchronize the processors, so when any machine finishes a batch, then it can continues with the next batch without wasting time. While BSPpub works with supersteps and at the end of each one there is a barrier synchronization, making that every processor has to wait for the others to continue with the next batch of queries. In a real system where the time is important it will be harmful.

All presented strategies do not exhibit data dependence, since each processor has a copy of the SAT, or it builds its own tree using a portion of the database. Consequently none of these strategies require a synchronization barrier at the moment to exchange data. Besides this, we use the number of distances evaluations as the complexity measure, so the results obtained for each strategy are independent of the library (MPI or BSPpub) that we used. However, we use the SAT BSPPub implementation to predict the costs of the algorithms. So, to select the right implementation for our problem we should evaluate the functional requirements and the execution environment of the application to choose the API that has the wanted characteristics.

As future work, we are going to use an approach in which the tree data structure will be distributed across the processors. Probably this strategy can carry out an unbalanced workload of the processors and increase the runtime communication. A point to emphasize is that the SAT structure has nodes with a diverse number of children, and each son can cause a distance comparison. Therefore it is important to be able to balance the number of distances comparison performed in each processor. Then, it is desirable to map the nodes of the tree among the processors considering the distance comparisons that they can be potentially performed in each subtree.

References

[1] D. Arroyuelo, F. Muñoz, G. Navarro, and N. Reyes. Memory-adaptative dynamic spatial approximation trees. In Proceedings of the 10th International Symposium on String Processing and Information Retrieval (SPIRE 2003), LNCS 2857, pages 360-368. Springer, 2003.

[2] C. Bohm, S.Berchtold, and D. Kein. Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases. ACM Computing Surveys, 33(3):322-373,2001.

[3] E. Chávez and G. Navarro and R. Baeza-Yates and J.L. Marroquin. Searching in Metric Spaces. ACM Computing Surveys. 33(3):273-321, 2001.

[4] V. Gaede and O. Gunther. Multidimensional access methods. ACM Computing Surveys, 30(2):170-321,1998.

[5] M. Marín and G. Navarro. Distributed query processing using suffix arrays. In Proceedings of the 10th international Symposium on String Processing and Information Retrieval (SPIRE 2003), LNCS 2857, pages 311-325. Springer, 2003.

[6] G. Navarro. Searching in metric spaces by spatial approximation. The Very Large Databases Journal (VLDBJ), 11(1):28-46, 2002.

[7] G. Navarro and N. Reyes. Fully dynamic spatial approximation trees. In Proceedings of the 9th International Symposium on String Processing and Information Retrieval (SPIRE 2002), LNCS 2476, pages 254-270. Springer, 2002.

[8] G. Navarro and N. Reyes. Improved deletions in dynamic spatial approximation trees. In Proc. of the XXIII International Conference of the Chilean Computer Science Society (SCCC'09), pages 13-22, IEEE CS Press, 2003.

[9] Snir M., Otto S., Huss-Lederman S., Walker D., Dongarra J. MPI: The complete Reference, Cambridge MA: MIT Press, 1996.

[10] L. Valiant. A Bridging Model for Parallel Computation. Communications of the ACM, Vol. 33, Pp 103-111, 1990.

[11] WWW.BSP and Worldwilde Standard, http://www.bsp-worldwide.org

[12] WWW.BSP PUB Library at Paderborn Univertity,