Searching and Updating Metric Space Databases Using the Parallel EGNAT

Mauricio Marin[1,2,3], Roberto Uribe[2], and Ricardo Barrientos[2]

[1] Yahoo! Research, Santiago, Chile
[2] DCC, University of Magallanes, Chile
[3] mmarin@yahoo-inc.com

Abstract. The Evolutionary Geometric Near-neighbor Access Tree (EGNAT) is a recently proposed data structure that is suitable for indexing large collections of complex objects. It allows searching for similar objects represented in metric spaces. The sequential EGNAT has been shown to achieve good performance in high-dimensional metric spaces with properties (not found in others of the same kind) of allowing update operations and efficient use of secondary memory. Thus, for example, it is suitable for indexing large multimedia databases. However, comparing two objects during a search can be a very expensive operation in terms of running time. This paper shows that parallel computing upon clusters of PCs can be a practical solution for reducing running time costs. We describe alternative distributions for the EGNAT index and their respective parallel search/update algorithms and concurrency control mechanism [1].

1 Introduction

Searching for all objects which are similar to a given query object is a problem that has been widely studied in recent years. For example, a typical query for these applications is the *range query* which consists on retrieving all objects within a certain distance from a given query object. That is, finding all *similar* objects to a given object. The solutions are based on the use of a data structure that acts as an index to speed up queries. Applications can be found in voice and image recognition, and data mining problems.

Similarity can be modeled as a metric space as stated by the following definitions.

Metric space. A metric space is a set X in which a distance function is defined $d : X^2 \rightarrow R$, such that $\forall\, x, y, z \in X$,

1. $d(x, y) \geq \; and \, d(x, y) = 0$ iff $x = y$.
2. $d(x, y) = d(y, x)$.
3. $d(x, y) + d(y, z) \geq (d(x, z)$ (triangular inequality).

[1] This work has been partially funded by FONDECYT project 1060776.

Y. Shi et al. (Eds.): ICCS 2007, Part I, LNCS 4487, pp. 229–236, 2007.
© Springer-Verlag Berlin Heidelberg 2007

Range query. Given a metric space *(X,d)*, a finite set $Y \subseteq X$, a query $x \in X$, and a range $r \in R$. The results for query x with range r is the set $y \in Y$, such that $d(x, y) \leq r$.

The k nearest neighbors: Given a metric space *(X,d)*, a finite set $Y \subseteq X$, a query $x \in X$ and $k > 0$. The k nearest neighbors of x is the set A in Y where $|A| = k$ and there is no object $y \in A$ such as *d(y,x)*.

The distance between two database objects in a high-dimensional space can be very expensive to compute and in many cases it is certainly the relevant performance metric to optimize; even over the cost secondary memory operations. For large and complex databases it then becomes crucial to reduce the number of distance calculations in order to achieve reasonable running times. This makes a case for the use of parallelism.

Search engines intended to be able to cope with the arrival of multiple query objects per unit time are compelled to using parallel computing techniques in order to reduce query processing running times. In addition, systems containing complex database objects may usually demand the use of metric spaces with high dimension and very large collections of objects may certainly require careful use of secondary memory.

The distance function encapsulates the particular features of the application objects which makes the different data structures for searching general purpose strategies. Well-known data structures for metric spaces are BKTree [3], MetricTree [8], GNAT [2], VpTree [10], FQTree [1], MTree [4], SAT [5], Slim-Tree [6]. Some of them are based on clustering and others on pivots. The EGNAT is based on clustering [7].

Most data structures and algorithms for searching in metric-space databases were not devised to be dynamic ones. However, some of them allow insertion operations in an efficient manner once the whole tree has been constructed from an initial set of objects. Deletion operations, however, are particularly complicated because in this strategies the invariant that supports the data structure can be easily broken with a sufficient number of deletions, which makes it necessary to re-construct from scratch the whole tree from the remaining objects.

When we consider the use of secondary memory we find in the literature just a few strategies which are able to cope efficiently with this requirement. A well-know strategy is the *M-Tree* [4] which has similar performance to the GNAT in terms of number of accesses to disk and overall size of the data structure. In [7] we show that the EGANT has better performance than the M-Tree and GNAT. The EGNAT is able to deliver efficient performance under high dimensional metric spaces and the use of secondary memory with a crucial advantage, namely it is able to handle update operations dynamically.

In this paper we propose the parallelization of the EGANT in the context of search engines for multimedia databases in which streams of read-only queries are constantly arriving from users together with update operations for objects in the database. We evaluate alternatives for distributing the EGANT data structure on a set of processors with local memory and propose algorithms for performing searches and updates with proper control of read-write conflicts.

2 The EGNAT Data Structure and Algorithms

The EGNAT is based on the concepts of Voronoi Diagrams and is an extension of the GNAT proposed in [2], which in turn is a generalization of the *Generalized Hyperplane Tree* (GHT) [8]. Basically the tree is constructed by taking k points selected randomly to divide the space $\{p_1, p_2, \ldots, p_k\}$, where every remaining point is assigned to the closet one among the k points. This is repeated recursively in each sub-tree D_{p_i}.

The EGNAT is a tree that contains two types of nodes, namely a node *bucket* and another *gnat*. All nodes are initially created as buckets maintaining only the distance to their fathers. This allows a significant reduction in space used in disk and allows good performance in terms a significant reduction of the number of distance evaluations. When a bucket becomes full it evolves from a bucket node to a gnat one by re-inserting all its objects into the newly created gnat node.

In the search algorithm described in the following we assume that one is interested in finding all objects at a distance $d \leq r$ to the query object q. During search it is necessary to determine whether it is a bucket node or a gnat node. If it is a bucket node, we can use the triangular inequality over the center associated with the bucket to avoid direct (and expensive) computation of the distances among the query object and the objects stored in the bucket. This is effected as follows,

- Let q be the query object, let p be the center associated with the bucket (i.e., p is a center that has a child that is a bucket node), let s_i be every object stored in the bucket, and let r be the range value for the search, then if holds

  $$Dist(s_i, p) > Dist(q, p) + r$$

  or

  $$Dist(s_i, p) < Dist(q, p) - r\,,$$

  it is certain that the object s_i is not located within the range of the search. In other case it is necessary to compute the distance between q and s_i.

We have observed on different types of databases that this significantly reduces the total amount of distance calculations performed during searches.

For the case in which the node is of type gnat, the search is performed recursively with the standard GNAT method as follows,

1. Assume that we are interested in retrieving all objects with distance $d \leq r$ to the query object q (range query). Let P be the set of centers of the current node in the search tree.
2. Choose randomly a point p in P, calculate the distance $d(q, p)$. If $d(q, p) \leq r$, add p to the output set result.
3. $\forall\, x \in P$, if $[d(q, p) - r, d(q, p) + r] \,\cap\,$ range(p, D_x) is empty, the remove x from P.
4. Repeat steps 2 and 3 until processing all points (objects) in P.
5. For all points $p_i \in P$, repeat recursively the search in D_{p_i}.

3 Efficient Parallelization of the EGNAT

We basically propose two things in this section. Firstly, we look for a proper distribution of the tree nodes and based on that we describe how to perform searches in a situation in which many users submit queries to a parallel server by means of putting queries into a receiving broker machine. This broker routes the queries to the parallel server and receive from it the answers to pass on back the results to the users. This is the typical scheme for search engines. In addition, due to the EGNAT structure we employ to build a dynamic index in each processor, the parallel server is able to cope with update operations taking place concurrently with the search operations submitted by the users.

Secondly, we propose a very fast concurrency control algorithm which allows search and update operations to take place without producing the potential read/write conflicts arising in high traffic workloads for the server. We claim *very fast* based on the fact that the proposed concurrency control mechanism does not incur in the overheads introduced by the classical locks or rollback strategies employed by the typical asynchronous message passing model of parallel computation supported by the MPI or PVM libraries.

Our proposal is very simple indeed. The broker assigns a unique timestamp to each query and every processor maintains its arriving messages queue organized as a priority queue wherein higher priority means lower timestamp. Every processor performs the computations related to each query in strict priority order. The scheme works because during this process it is guaranteed that no messages are in transit and the processors are periodically barrier synchronized to send departing messages and receive new ones. In practical terms, the only overhead is the maintenance of the priority queue, a cost which should not be significant as we can profit from many really efficient designs proposed for this abstract data type so far.

The above described type of computation is the one supported by the bulk-synchronous model of parallel computing [9]. People could argue that the need to globally synchronize the processors could be detrimental and that there could be better ways of exploiting parallelism by means of tricks from asynchronous message passing methods. Not the case for the type of application we are dealing with in this paper. Our results show that even on very inefficient communication platforms such a group of PCs connected by a 100MB router switch, we are able to achieve good performance. This because what is really relevant to optimize is the load balance of distance calculations and balance of accesses to secondary memory in every processor. In all cases we have observed that the cost of barrier synchronizing the processors is below 1%.

Moreover, the particular way of performing parallel computing and communications allows processors further reduction of overheads by packing together into a large message all messages sent to a given processor. Another significant improvement in efficiency, which leads to super-linear speedups, is the relative increase of the size of disk-cache in every processor as a result of keeping a fraction N/P of the database in the respective secondary memory, where N is the total number of objects stored in the database and P the number of processors.

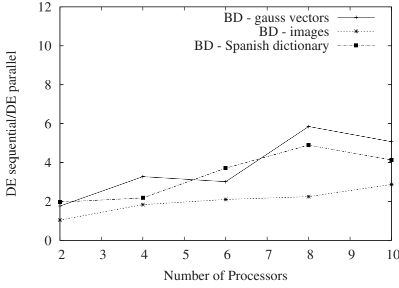
To show the suitability of the EGNAT data structure for supporting query processing in parallel, we evenly distributed the database among the P processors of a 10-processors cluster of PCs. Queries are processed in batches as we assume an environment in which a high traffic of queries is arriving to the broker machine. The broker routes the queries to the processors in a circular manner. We take batches as we use the BSP model of computing for performing the parallel computation and communication.

In the bulk-synchronous parallel (BSP) model of computing [9], any parallel computer (e.g., PC cluster, shared or distributed memory multiprocessors) 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 only perform sequential computations on local data and/or send messages to other processors. The messages are available for processing at their destinations by the next superstep, and each superstep is ended with the barrier synchronization of the processors.

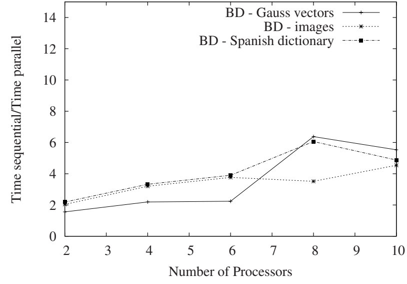
The running time results shown below were obtained with three different metric space databases. (a) A 10-dimensional vector space with 100,000 points generated using a Gaussian distribution with average 1 and variance 0.1. The distance function among objects is the Euclidean distance. (b) Spanish dictionary with 86,061 words where the distance between two words is calculated by counting the minimum number of insertions, deletions and replacements of characters in order to make the two words identical. (c) Image collection represented by 100,000 vectors of dimension 15.

Searches were performed by selecting uniformly at random 10% of the database objects. For all cases the search of these objects is followed by the insertion of the same objects in a random way. After we searched 10 objects we randomly chose one of them and insert it into the database, and for every 10 objects inserted we delete one of them also selecting it at random. Notice that we repeated the same experiments shown below but without inserting/deleting objects and we observed no significant variation in the total running time. This means that the overheads introduced by the priority queue based approach for concurrency control we propose in this paper has no relevant effects in the total running time.

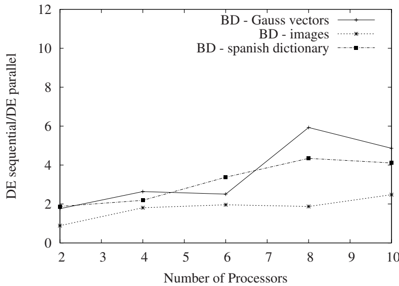
We used two approaches to the parallel processing of batches of queries. In the first case, we assume that a single EGNAT has been constructed considering the whole database. The first levels of the tree are kept duplicated in every processor. The size of this tree is large enough to fit in main memory. Downwards the tree branches or sub-trees are evenly distributed onto the secondary memory of the processors. A query starts at any processor and the sequential algorithm is used for the first levels of the tree. After this copies of the query "travel" to other processors to continue the search in the sub-trees stored in the remote secondary memories. That is queries can be divided in multiple copies to follow the tree paths that may contain valid results. These copies are processed in parallel when are sent to different processors. We call this strategy the *global index* approach.



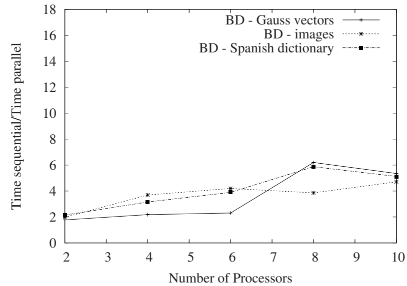
(a) Distance calculations
1 new query per superstep



(b) Running time
1 new query per superstep



(c) Distance calculations
10 new queries per superstep

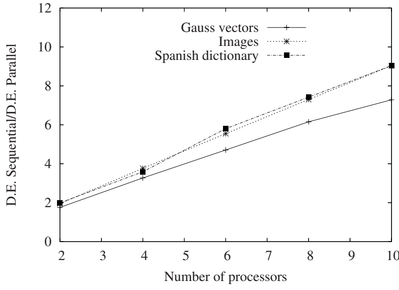


(d) Running time
10 new queries per superstep

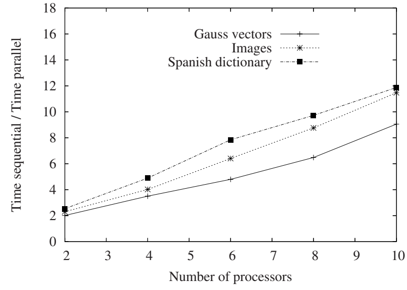
Fig. 1. Results for the global index approach. Figures (a) and (c) show the ratio number of sequential distance evaluations to parallel ones, and figures (b) and (d) show the respective effect in the running times.

Figure 1 shows running time and distance calculation measures for the global index approach against an optimized sequential implementation. The results show reasonable performance for small number of processors but not for large number of processors. This is because performance is mainly affected by the load imbalance observed in the distance calculation process. This cost is significantly more predominant over communication and synchronizations costs. The results for the ratio of distance calculations for the sequential algorithm to the parallel one show that there is a large imbalace in this process. In the following we describe the second case for parallelization which has better load balance. Notice that this case requires more communication because of the need for broadcasting each query to all processors.

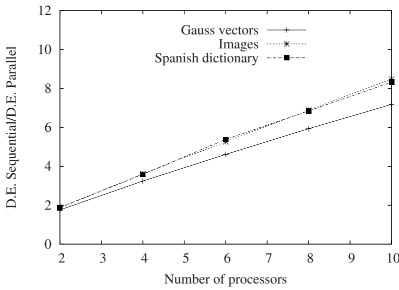
In the second case, an independent EGNAT is constructed in the piece of database stored in each processor. Queries in this case start at any processor at the beginning of each superstep. The first step in processing any query is to send a copy of it to all processors including itself. At the next superstep the searching



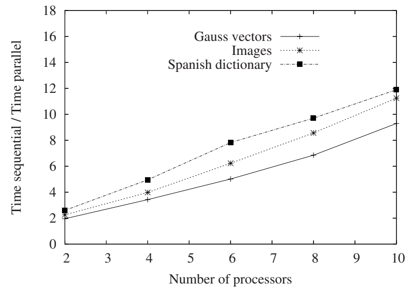
(a) Distance calculations
1 new query per superstep



(b) Running time
1 new query per superstep



(c) Distance calculations
10 new queries per superstep



(d) Running time
10 new queries per superstep

Fig. 2. Results for the local index approach. Figures (a) and (c) show the ratio number of sequential distance evaluations to parallel ones, and figures (b) and (d) show the respective effect in the running times.

algorithms is performed in the respective EGNAT and all solutions found are reported to the processor that originated the query. New objects are distributed circularly onto the processors and insertions are performed locally. We call this strategy the *local index* approach.

In the figures 2 we present results for running time and distance calculations for Q=1 and 10 new queries per superstep respectively. The results show that the local index approach has much better load balance and thereby it is able to achieve better speedups. In same cases, this speedup is superlinear because of the secondary memory effect. Notice that even processing batches of one query per processor is good enough to amortize the cost of communication and synchronization. Interestingly, the running times obtained in figure 2 are very similar to the case in which no write operations are performed in the index [7]. This means that the overhead introduced by the concurrency control method is indeed negligible.

4 Conclusions

We have described the efficient parallelization of the EGNAT data structure. As it allows insertions and deletions, we proposed a very efficient way of dealing with concurrent read/write operations upon an EGNAT evenly distributed on the processors. The local index approach is more suitable for this case as the dominant factor in performance is the proper balance of distance calculations taken place in parallel.

The results using different databases show that the EGNAT allows an efficient parallelization in practice. The results for running time show that it is feasible to significantly reduce the running time by the inclusion of more processors. This is because a number of distance calculations for a given query can take place in parallel during query processing. We emphasize that for the use of parallel computing to be justified we must put ourselves in a situation of a very high traffic of user queries. The results show that in practice just with a few queries per unit time it is possible to achieve good performance. That is, the combined effect of good load balance in both distance evaluations and accesses to secondary memory across the processors, is sufficient to achieve efficient performance.

References

1. R. Baeza-Yates and W. Cunto and U. Manber and S. Wu. Proximity matching using fixedqueries trees. 5th Combinatorial Pattern Matching (CPM'94), 1994.
2. S. Brin. Near neighbor search in large metric spaces. The 21st VLDB Conference, 1995.
3. W. Burkhard and R. Keller. Some approaches to best-match file searching. Communication of ACM, 1973.
4. P. Ciaccia and M. Patella and P. Zezula. M-tree: An efficient access method for similarity search in metric spaces. The 23st International Conference on VLDB, 1997.
5. G. Navarro and N. Reyes. Fully dynamic spatial approximation trees. In the 9th International Symposium on String Processing and Information Retrieval (SPIRE 2002), pages 254–270, Springer 2002.
6. C. Traina and A. Traina and B. Seeger and C. Faloutsos. Slim-trees: High performance metric trees minimizing overlap between nodes. VII International Conference on Extending Database Technology, 2000.
7. R. Uribe, G. Navarro, R. Barrientos, M. Marin, An index data structure for searching in metric space databases. International Conference on Computational Science (ICCS 2006), LNCS 3991 (part I) pp. 611-617, (Springer-Verlag), Reading, UK, May 2006.
8. J. Uhlmann. Satisfying general proximity/similarity queries with Metric Trees. Information Processing Letters, 1991.
9. L.G. Valiant. A bridging model for parallel computation. Comm. ACM, 1990.
10. P. Yianilos. Data structures and algoritms for nearest neighbor search in general metric spaces. 4th ACM-SIAM Symposium on Discrete Algorithms, 1993.