Bulk Message-Passing Model: Metric Space Index a case of study

Verónica Gil-Costa
DCC, UNSL,
San Luis, Argentina
{gvcosta}@unsl.edu.ar

Mauricio Marin
Yahoo! Research,
Santiago de Chile
mmarin@yahoo-inc.com

Marcela Printista
DCC, UNSL,
San Luis, Argentina
{mprinti}@unsl.edu.ar

August 11, 2008

Abstract

Synchronous and asynchronous approaches to model parallel programs are considered as two different schools. Intuition indicates that asynchronous programs should be more efficient since no periodical global time synchronization is required. However, in this paper we parallelize a resent index pivot-based technique for searching in metric spaces designed to improve bulk-message-passing which reduce running times using a synchronous model. This model is designed to balance the load work en each machine between synchronizations and to reduce the number of synchronizations required to finish a batch of queries.

1 Introduction

Most researchers tend to believe that asynchronous methods are more efficient and speed up the response times. But what happens when this is not hold for all cases of study? This is an open debate that has motivated the development of many parallel models, most of them trying to find a suitable cost model with the right level of abstraction [12, 8].

In this paper we present some efficient parallel index strategies based on a bulk-synchronous model for a structure that allows searching for similar multimedia objects in a metric space, the Sparse Spatial Selection - SSS [2] and we compare them with the SSS-Block strategy proposed presented in [6]. One feature of this structure is that it grows rapidly and requires more storage space than the database itself, making it interesting for parallel analyzing. Parallel index construction algorithms were presented and compared in [6].

We selected a metric space index because similarity searches are a suitable way to find any kind of unstructured data, not just text, but video clips, images, finger prints, etc. In a similarity search we do not use the concept of exact matching, we search instead for similar objects. Beside, the computational cost of determining the similarity between two objects makes similarity search an expensive operation. This fact has motivated the development of many research over large collections of data [13, 1, 3, 9].

The parallel algorithms implemented in this work to accept on-line streams of queries under a synchronous model, tend to balance computation and communication among a set of processors, and minimize the number of iterations to reduce the synchronization cost. To balance computation we assign a quantum of load work in a round-robin way. Moreover, algorithms are independent of the architecture and very simple to implement due to its level of abstraction.

2 Theoretical Concepts

A *metric space* $(\mathbb{X}, d)$ is composed of a universe of valid objects $\mathbb{X}$ and a *distance function* $d : \mathbb{X} \times \mathbb{X} \rightarrow \mathbb{R}^+$ defined among them. The distance function determines the similarity between two given objects. The goal is, given a set of objects and a query, to retrieve 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.

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 the objects we are managing.

There are three main queries of interest

- *range search:* that retrieves all the objects $u \in \mathbb{U}$ within a radius r of the query q, that is: $(q, r)_d = \{u \in \mathbb{U}/d(q, u) \leq r\}$;
- *nearest neighbor search:* that retrieves the most similar object to the query q, that is $NN(q) = \{u \in \mathbb{U}/\forall v \in \mathbb{U}, d(q, u) \leq d(q, v)\}$;
- *k-nearest neighbors search:* a generalization of the nearest neighbor search, retrieving the set $kNN(q) \subseteq \mathbb{U}$ such that $|kNN(q)| = k$ and $\forall u \in kNN(q), v \in \mathbb{U} - kNN(q), d(q, u) \leq d(q, v)$.

In this paper we are devoted to range queries. Nearest neighbor queries can be rewritten as range queries in an optimal way [4].

One kind for data structures designed for metric spaces are pivot-based techniques. A pivot technique selects some objects as $pivots$ from the collection and then computes the distance between the pivots and the objects of the database. For a query (q, r) the distances between the query and all pivots are computed. r is the radius used thought the search. The objects x from the collection that hold the condition $|d(p_i, x) - d(p_i, q)| > r$ can be discarded due to the triangle inequality. A candidate list is built with the objects that can not be discarded, and they are compared directly against the query.

Several algorithms, like [13, 1], are almost direct implementations of this idea, and differ basically in their extra structure used to reduce the CPU cost of finding the candidate points, but not in their number of distance evaluations. Some good surveys about metric spaces can be found in [4] and [7].

3 Sequential Sparse Spatial Selection Index

The Sparse Spatial Selection - SSS [2] build algorithm can be divided in two main steps: the pivots selections and the distances computation between the pivots and all database objects. To select the pivots set, 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 αM, being $0 <= \alpha <= 1$ a constant parameter. Therefore, 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 (see Figure 1).

```
PIVOTS ← $\{x_1\}$
for all $x_i \in \mathbb{U}$ do
  if $\forall p \in$ PIVOTS, $d(x_i,p) \geq \alpha M$ then
    PIVOTS = PIVOTS ← $\cup \{x_i\}$
```

Figure 1: Pivots selection algorithm for the sequential SSS algorithm.

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. The steps required to solve the range query (q, r) are shown in Figure 2.

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.

```
foreach pivot $p$ do dq[$p$] ← $d(q,p)$
$n \leftarrow 0$
foreach object $o$ do
    foreach pivot $p$ do
      if ( distance[$o$][$p$] > (dq[$p$]$-r$) and
      distance[$o$][$p$] < (dq[$p$]$+r$) ) then
        $n \leftarrow n+1$
      endif
    endfor
    if ( $n$ = total number of pivots ) then
      add object $o$ to a list of candidate objects $\ell$.
    endif
endfor
foreach object $o \in \ell$ do
    if ( $d(o,q) \leq r$ ) then
      report object $o$ as solution
    endif
endfor
```

Figure 2: Range query searches using the Sparse Spatial Selection index.

4 System Design

4.1 Parallel Environment

The environment selected 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. 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.

Each processor process a sequence of iterations formed by three main operations: receive messages (queries), process messages and send new messages. In the second stage processors may perform three operations according to the messages received: a) broadcasting queries, b) search for similar objects to a query and c) merge results obtained. Program codes are object-oriented programming and we use a skeleton program in a template method design pattern in order to make our codes portable. We avoid sending small packages and favour bulk sending, coping small messages into a single stream.

4.2 Synchronization Methods

In this work we use two message-passing parallel methods to implement the search algorithms using the SSS index. The first one is the well-known asynchronous method where processors work independently, send messages without blocking them self and receive messages from others processors. In this case no barrier synchronization is required. The library selected to implement asynchronous algorithms is PVM [5].

On the opposite side we use the *Bulk Synchronous Parallel- BSP* model of parallel computing [12], where 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 [8].

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 [10]. BSP model 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. In this work, we have used the BSPonMPI library that allows running BSP using the MPI primitives.

We apply the round-robin principle to SSS operations. We refer to the classic round-robin strategy for dealing with a set of jobs competing to receive service from a CPU. In this case each job is given the same quantum of CPU so that jobs requiring large amounts of processing cannot monopolize the use of the CPU. This scheme can be seen as bulk-synchronous in the sense that jobs are allowed to perform a fixed set of operations during their quantum. In our setting we define quanta in computation, disk accesses and communication for the SSS operations which enables a better utilization of resources whilst it improves response times for queries requiring the least use of them.

5 Round-Robin parallel searches

5.1 Index storage

To manage large databases we can see the SSS index as a matrix with a row i for each object and a column j for each pivot. In this matrix each cell (i, j) stores the pre-computed distance $d(x_i, p_j)$, used during the search operation to discard objects without comparing them with the query.

When the index is stored in secondary memory, the processing of the database can be optimized. We divide the matrix in disk pages of size K, and at the beginning of a search operation we can load some disk page with K pairs $\{$id pivot, distance$\}$ for some objects into main memory. Then, we can increase the level of abstraction of this scheme and see this table as a new table $Row\_Page[i, j]$ where each cell is a disk page with the distances between K pivots and K objects. To manage disk pages we use a hierarchical structure to map disk pages identifiers to secondary memory.

5.2 Parallel Searches

Round-robin query processing for a stream of queries takes place as follows. When new queries arrive from the broker, they are broadcasted to the rest of the processors. Then, all processors receiving the query search in their local index for similar objects. For that, in each processor and iteration a different set of pages are brought into main memory and current queries select objects passing condition (1) as their candidate objects. Candidate objects are passed to the next page of pivots with pre-computed distances and so on. This becomes our quantum K of work for the round-robin query processing strategy.

```
distance[o][p] > (distance[q][p] - r)   and
distance[o][p] < (distance[q][p] + r)         (1)
```

Main memory in each processor supports a fixed number of pages. These pages can be loaded in a horizontal or a vertical manner, namely pages covering a wide range of pivots (horizontal fetch) or a wide range of objects (vertical fetch) respectively. In the early stages of processing Q queries a vertical fetch over K columns of the table, i.e., distances from all the objects of the database to a subset of K pivots, will reduce significantly the list of candidate objects for some queries. Then a sequence of horizontal fetches is applied over the candidate lists of objects to accelerate the operation of processing those queries. When horizontal fetches are applied, disk pages are selected according to its priority defined by the number of queries requiring that page.

Thus, a vertical or horizontal fetch is performed in each iteration, and pivots are used to discard as many objects as possible, having a smaller candidate list for the next iteration. When some queries are finished because all disk pages were processed and similar objects were find, a new batch is injected and a vertical fetch is processed. But if some queries end up because no more candidates were find, new queries are injected into a pipelining way following the sequence of horizontal fetches. Therefore, we keep balanced the load work in all iterations. In addition, every disk page retrieved from secondary memory is shared by all queries being processed in the current iteration. Notice that the order in which individual queries "visit" these pages is not relevant for the outcome.

The round-robin principle can be implemented on top of three query processing strategies which we call SSS B-Trees, SSS-Block and Hybrid:

(i) We build one B-tree per pivot and it stores the distance from the pivot to all objects belonging to the disk pages. The candidate objects obtained by each tree are intersected to obtain the candidate objects lists. Notice that every time we load some disk pages, we have to begin the searches from the top of the pivot's trees and then not only intersect the results obtained with the current pages but also with candidates objects obtained in previous iterations. B-trees are useful for pruning useless objects in large data-sets.

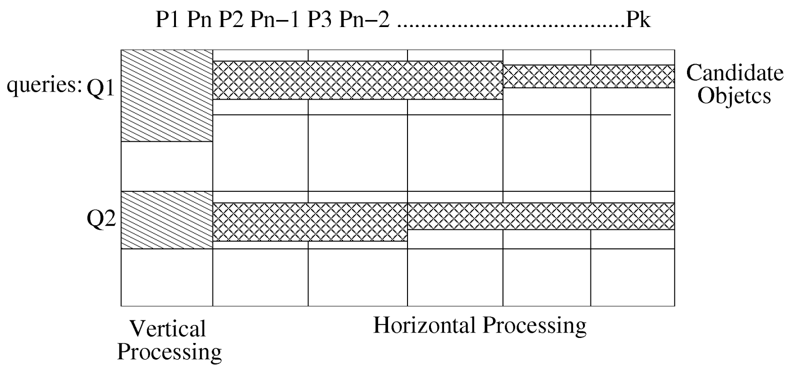

Figure 3: After the first vertical fetch is completed using K pivots, rows one and three are required for more horizontal fetches.

(ii) The SSS-Block [6] (see Figure 3) uses a more primitive search strategy but free of the intersection cost which can be very expensive with large data-sets. It does not use a pruning structure and all page objects' loaded in main memory are analyzed. It has a high cost for vertical fetches, due to it has to compare queries against all objects, but for horizontal fetches it reduce the distances evaluations analyzing only the candidates objects obtained in previous iterations instead of all objects of the disk page. Thus, the load work performed by each processor per query may vary depending on the lists of candidate's objects obtained in previous iterations and the kind of fetched performed (vertical or horizontal).

(iii) The Hybrid algorithm combines the B-tree for the first vertical fetch and the SSS-Block strategy for horizontal fetches in order to reduce the number of distance computations.

6 Experiments

6.1 Data Settings

The experiments were performed with a data set composed of 1,400,000 words in Spanish. It was obtained from crawled Web documents. The distance function determines how similar two words are each other (we use the edition distance function for calculating similarity, the function returns the number of characters we have to delete, modify or insert to make two words equal). Queries were selected at random from a set of 127,000 queries taken from the todocl log.

Experiments were performed in a cluster with dual processors (2.8GHz) that use NFS mounted directories and the $Lustre$ [11] file system. We used up to 32 processors (CPUs) located in different nodes. We call this cluster $C1$. To validate results we also performed the experiments on a second cluster $C2$ with 16 processors connected by a fast switching technology. All results were obtained using 4, 8, 16 and 32 processors as we found this to have a practical relation between the size of the index and the number of processors.

Before any experiment were performed, we selected the α value. We choose a $\alpha = 0.5$ which means that every pivot is farther than a half of the maximum distance from the already selected

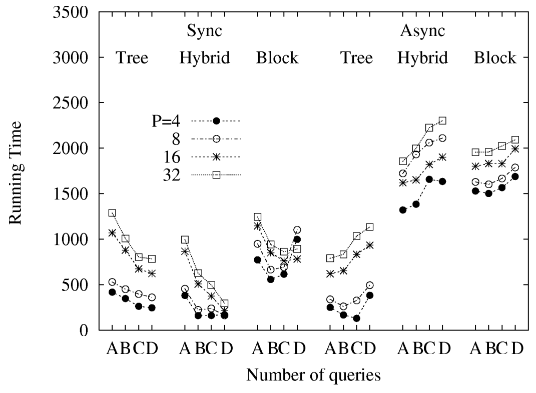

Figure 4: Running time obtained for all three index searching strategies using a synchronous[left] and asynchronous[right] parallel model over $C1$.

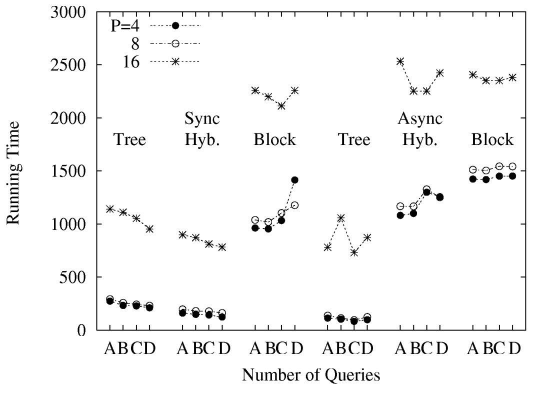

Figure 5: Running time obtained for all three index searching strategies using a synchronous[left] and asynchronous[right] parallel model over $C2$.

pivots. A greater alpha would imply less pivots and a lost of a covered region. This value was selected experimentally as the one reducing the number of distance evaluations. The resulting index has 46410 pivots and a size of 183Gb. The index is larger than the data-sets due to it has to store the distance between the pivots and all objects.

In every run we process 10,000 queries in each processor, therefore is the total number of queries processed in each experiment reported below is $10,000P$. Thus running times are expected to grow with P. Beside, in each iteration we inject batches of queries={1,32,128 and 256}.

6.2 Results

Figures 4 and 5 show the running times obtained in $C1$ and $C2$ for all three parallel strategies varying the number of queries injected (A=1, B=32, C=128 and C=256) in each iteration or superstep if BSP is used. At the left synchronous algorithms implemented under BSP and at the right asynchronous algorithms with PVM. For the synchronous algorithms the Hybrid strategy reduce the running time as batches of queries are increased and also presents a better performance than the others two strategies, balancing the load work and the communication. This is most noticeable in $C1$. B-tree seems to work well in a server with few processors, but when the collection of data in each processor is smaller (using P=16 or P=32) its performance is worse. The SSS-Block strategy obtains larger running times because it has a very high disk access cost compared with the others strategies.

Asynchronous algorithms tend to increase the running time as batches of queries are bigger, contrary to the behaviour of most synchronous algorithms. Beside, the Hybrid algorithm is the one reporting the highest running time opposite to its synchronous version. The SSS-Block strategy is more stable and obtains better results in this asynchronous implementation.

Figures 6 and 7 shows communication efficiency for synchronous and asynchronous implementations respectively. In the second one, results present more fluctuations and are more unstable.

Finally, Figure 8 shows computation efficiency for the synchronous algorithms using only P=32. In this case all strategies requires 35 supersteps to finish the execution of Q queries and the Hybrid strategy is the one presenting a better computation efficiency. On the other side, Figure 9 shows the computation efficiency for the asynchronous algorithms with P=32. This graphic shows how the efficiency decreases with more iterations. Notice that it is difficult to measure the efficiency at a given moment in an asynchronous system due to different processors may be in different iterations, and not

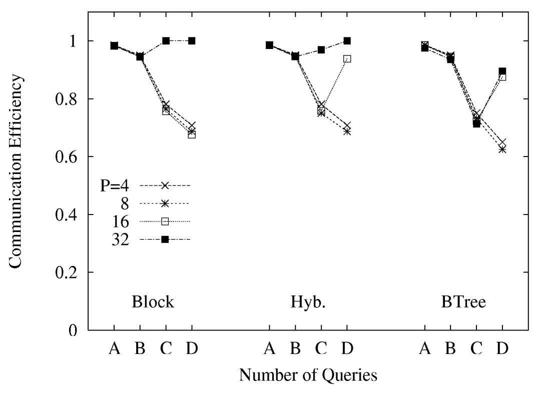

Figure 6: Results obtained under a synchronous method using $C1$.

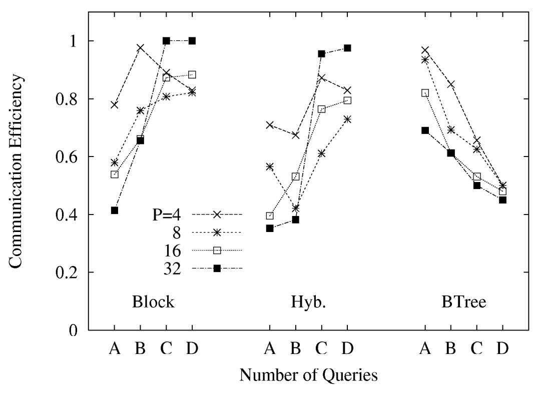

Figure 7: Results obtained under an asynchronous method using $C1$.

all processors require the same number of iterations. Depending on the parallel strategy, the numbers of iterations performed by each processor varies. Using a B-tree the maximum number of iterations is 280, while the Hybrid requires at most 98 iterations and finally SSS-Block only 48. Despite the fact that B-tree requires more iterations (it is more unbalance) it presents better running times due to it can prune more useless objects and reduce the distance computations. Beside, this unbalance is reflected whit higher running times due to the cost of synchronization in Figures 4 and 5.

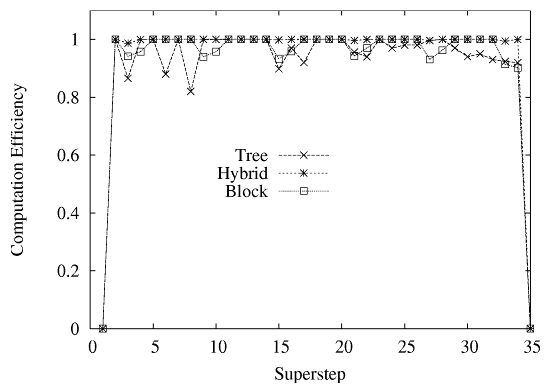

Figure 8: Computation efficiency for all three parallel strategies under the BSP model and using $C1$.

7 Conclusions and Future Work

In this paper we presented a bulk-message-passing server accepting on-line streams of queries for searching for similar objects in a metric space. We used the SSS index due to it selects efficiently the pivots objects and it has been compared successfully against other pivots techniques.

Algorithms were implemented under two different message-passing parallel methods: synchronous and asynchronous. Due to the fact that algorithms favour bulk-sending and most of them presented balance communication and computation, minimizing the number of supersteps and reducing the number of synchronizations, synchronous algorithms tend to be more effective.

Parallel strategies using a B-tree present good performance for large data-sets and a small number of processors. But the Hybrid strategy combines the advantage of both B-tree and SSS-Block for

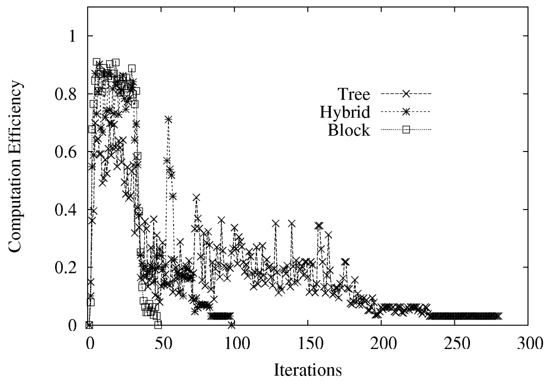

Figure 9: Computation efficiency for all three parallel strategies using PVM and $C1$.

vertical and horizontal fetches, optimizing its performance for synchronous implementations. The asynchronous method seems to work better with unbalance strategies.

As future work we intend to verify the behaviour obtained with others no conventional indexes like the GNAT or Egnat, and then formalize these empirical results using DAG or Petri Nets.

References

[1] R. Baeza-Yates, W. Cunto, U. Manber, and S. Wu. Proximity matching using fixed-queries trees. In *Proc. 5th Combinatorial Pattern Matching (CPM'94)*, LNCS 807, pages 198–212, 1994.

[2] 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.

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

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

[5] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Mancheck, and V. Sunderam. *PVM: Parallel Virtual Machine - A Users Guide and Tutorial for Network Parallel Computing*, 1994. MIT Press.

[6] V. Gil-Costa and M. Marin. Distributed sparse spatial selection indexes. *In 16th Euromicro International Conference on Parallel, Distributed and Network-based Processing (EuroPDP 2008)*, February 13-15, 2008,Toulouse, France.

[7] H. Samet. *Foundations of Multidimensional and Metric Data Structures (The Morgan Kaufmann Series in Computer Graphics and Geometric Modeling)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

[8] 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.

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

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

[11] URL. LUSTE, http://wiki.lustre.org/index.php?title=main_page.

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

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