Organization of geometric point data information for parallel processing

MARIUS-DORIAN ZAHARIA [*]

Keywords: multidimensional data, spatial data, hierarchical data structure, parallel algorithm

Articolul prezintã structuri de date ierarhice sau bazate pe cataloage tabulare care permit descompunerea bazatã pe considerente spa_iale a datelor de naturã geometricã (punctiforme). Aceste structuri sunt adecvate folosirii în aplica_ii paralele din domeniul vederii asistate de calculator sau al bazelor de date spa_iale.

The paper describes some hierarchical or tabular based spatial decomposition techniques for geometric data representation (multidimensional point data sets). These structures are suitable to be used in parallel applications in the field of computer vision or data bases for graphics processing.

1. Introduction

There are various categories of parallel and distributed software applications, some of them are solving problems where numerical computations are dominant, other problems could be solved using symbolic calculus or artificial intelligence techniques. Other classes of parallel applications consist in analysis, simulation and optimization of systems functionality, as well as acquisition and control of information in geographically distributed systems. Computer graphics (e.g. realistic 2D image synthesis) and image processing applications could be considered as numerically intensive applications.

The development of parallel applications in the above mentioned fields requires the geometric (spatial) decomposition of the data space. Every process of the application will manipulate variables associated with subregions of the two, three or multidimensional data space. The data structures used to most suitably represent the spatial decomposition are hierarchical structures (a tree node corresponds to a spatial region and different paths in the tree could be parallely processed) or grid based structures (in this case every spatial region has an associated array element).

A spatial decomposition technique should divide the data space in infinitely repetitive patterns. There are unlimited spatial decompositions (if every partition element can be splitted in finer (similar) subelements, which will represent the data at higher resolutions) as well as limited decompositions (the atomic pieces can be farther decomposed only by changing the shape of those elements).

In case of a bidimensional data space the decompositions could be polygonal, nonpolygonal, regular (if the atomic pieces are regular polygons) hierarchical (if more adjacent elements of the decomposition may be grouped in more complex (molecular)

[*] Sef lucrãri dr. ing., Department of Computers, Faculty Automation and Computers, University "Politehnica" of Bucharest

elements which generally have different shapes as the atomic pieces). An important class of hierarchical decompositions is selfsimilar decompositions where all elements of the hierarchy have the same shape as the element placed at the base level of the hierarchy.

There are a lot of different modalities to represent multidimensional point data. The implementer may choose the suitable one according to the most heavily used operations, which are expected to be performed on the data. The issues distinguishing those representations are:

- the data space organized by the representation (object space, image space or both - in case of hybrid methods, some attributes are represented in object space others in image space)

- the context in which data will be used (static - i.e. the number of points is apriori known - or dynamic - i.e. the number of data points can grow (vary) during the application run).

2. Hierarchical methods for representation of multidimensional data points

A largely used method of organizing a data point collection is the point quadtree. Every tree node includes only one point, (his coordinates) the associated information, as well as 4 pointers to the node's sons. In order to obtain good processing times the quadtree must be balanced. One algorithm to build a balanced point quadtree starts with a lexicographic sort of data points. The median point (P) is placed in the root of the tree and the building process continues recursively by processing the two halves of the data file containing data points placed below respectively above (P). In the worst case (where the N data points are collinear) the height of the resulting tree will be of order $O(\log_2 N)$.

The pseudo-quadtrees have the same structure as the point quadtrees but the data points are stored only in the leaves. The internal nodes will include so called splitting points, which generally are not in the data point collection. In case of a k-dimensional data space the maximal height of a pseudo-quadtree is $O(\log_{k+1} N)$.

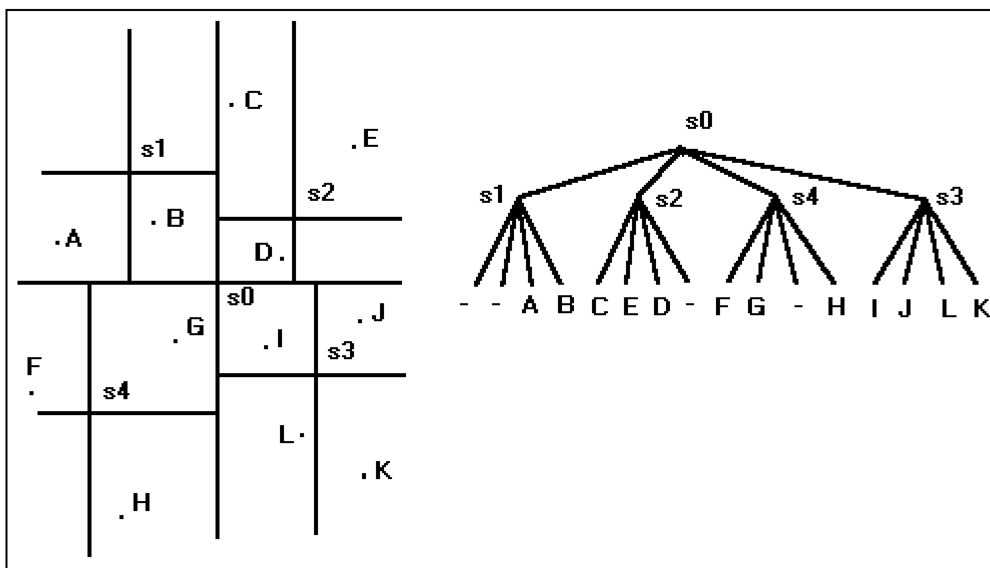

Figure 1

For demonstration let V be a data point collection in a bidimensional data space and card(V)=N. There is a splitting point $H \in V$ such that everyone of the four quadrants determined by H will include at most [N/3] points from V. Let hx, hy be the coordinates of H.

hx is choosed as: $card(\{p \in V | px<hx\}) = N/3$

hy is choused as: $card(\{p \in V | (px>hx)\wedge(py<hy)\}) = N/3$

So H will split the data space in four quadrants and every quadrant will include at most N/3 data points. The tree will have $[\log_3 N]$ levels.

The k-d tree is a data structure similar to the binary search tree, the difference is that the search key varies with the levels of the tree nodes. Every tree node has an associated data point. In the following it will be considered: a k-dimensional data space, a point collection modeled by a k-d tree and a data point search request. If the search process reaches a node (N) having the depth a (the root of the tree is considered placed on level 0) the data space will be splitted accordingly to the value of the j=a mod k coordinate of the point associated with (N).

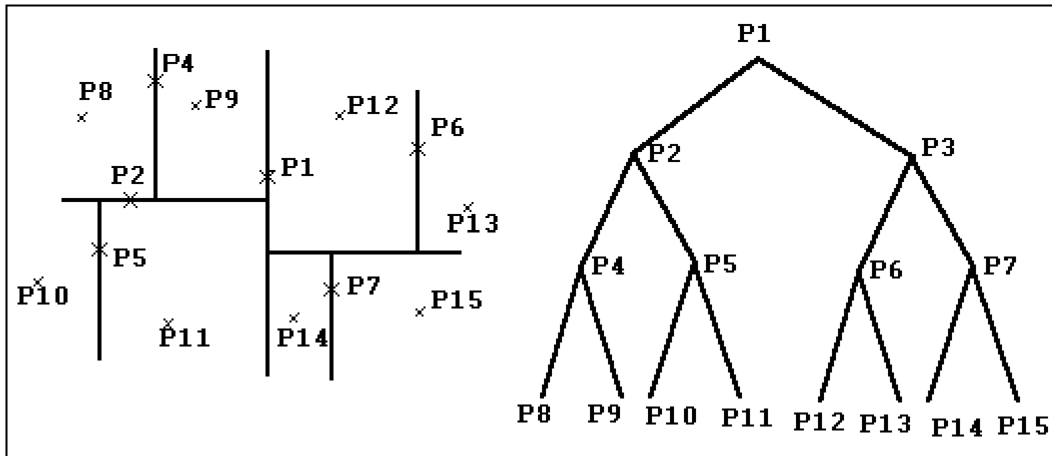

Figure 2

The left subtree of (N) includes the points whose coordinates x_j are less than p_j (j-th coordinate of the point corresponding to (N)).

The k-d tree subdivision hyperplanes are orthogonal to reference axes of the data space. There are examples of binary trees where the hyperplanes which realize the division of the data space are not orthogonal. The binary space partitioning trees (BSP [1], [8]) have subdivision hyperplanes associated to each tree node. In a three-dimensional data space the BSP trees are used for hidden surface removal algorithms in 3D scenes modeled through polygonal meshes. If (N) is an arbitrary node of a BSP tree the processing of the polygons from the left and right subtree of (N) may be parallely done.

A spatial region could be represented by its interior or its boundary. A hierarchical representation of the interior of a region could be done by dividing the data space in blocks. The spatial decomposition methods from the above mentioned category could be classified by the following facts:

- the blocks of the decomposition are (or not) disjoint
- the blocks are placed in standard positions (resulted from successively halving the data space across each coordinate axis)
- the blocks have standard dimensions.

If the regions resulted after the division have equal sizes, the corresponding data structure is a region quadtree. This type of quadtree can be adapted to represent point data. Every square region of the spatial division has only one associated data point (P). That is why (P) could be approximated by one of the corners (or the center) of the square region. This technique is similar to the "fixed grid technique". To represent a $2^n x 2^n$ digital image the splitting process could continue maximum n times. The advantage of the hierarchic representation is that four empty square neighbour regions can be merged in only one cell, associated with one tree node. The shape of the region quadtree is not influenced by the insertion order of the data points in the tree.

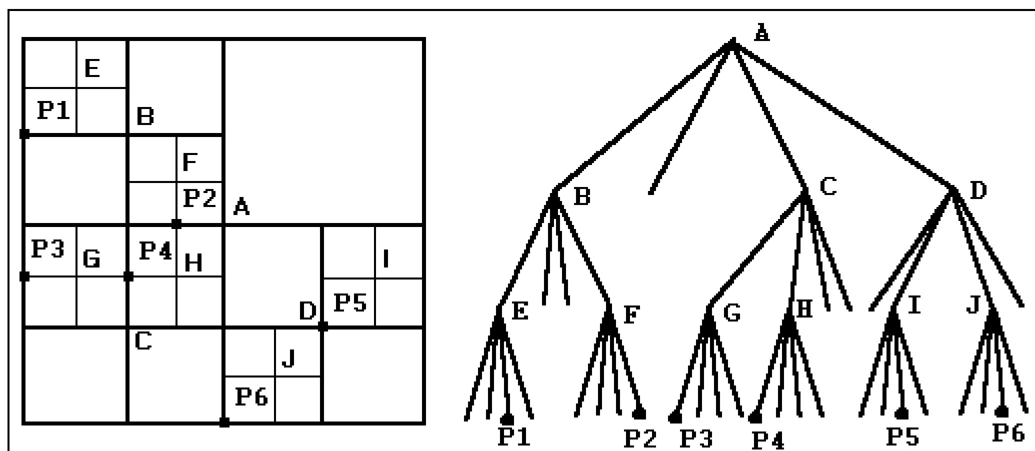

Figure 3

The MX-tree is used to model discrete data spaces as sparse arrays. It allows efficient algorithms for point retrieval. A separate process could search every quadrant of the data space. The data points have distinct coordinates; otherwise every leaf node (L) of the tree will refer a list containing informations associated with every data point having the same coordinates as the point associated to (L).

For example the search of the points from a collection represented by a MX-tree who are included in a rectangular domain (a so called range search query) could be solved as follows.

The MX-tree is described by:

```
typedef struct MXnode {
        enum {BLACK, WHITE, GRAY } tipnod;
        tip_info info;
        struct MXnode *fii[4];
        } *MXtree;
```

A hyperrectangular data space represented by two data points whose coordinates are the center of the rectangle and the sides lengths respectively will be considered. The four sons of an internal node (N) of the MX tree correspond to the following spatial decomposition 0=NV (subquadrant), 1=NE, 2=SV, 3=SE as specified in Figure 3. If the spatial region corresponding to node (N) is the hyperrectangle (c, l) then, in case of a bidimensional data space the four sons of (N) will have the associated regions (c_i, l_i) where:

$$l_i.x = l.x/2,\ l_i.y = l.y/2,\ \ \forall\ 0 \le i < 3.$$

and

$$c_i.x = c.x + dx_i * l.x,\ c_i.y = c.y + dy_i * l.y$$

where dx and dy are two arrays of weight factors:

$$dx=\{-0.5, 0.5, -0.5, 0.5\},\ dy=\{0.5, +0.5, -0.5, -0.5\}$$

The range search of the points from the MX-tree data collection placed in a rectangular domain (r, q) is described by the following routine. (The points satisfying the query are added to a global list LP).

```
procedure rqs(point nc, point nl, point r, point q, MXtree a)

    switch(a->tipnod)
        BLACK: //leaf node containing a data point
            if (interior(nc, r, q)) *add point nc to result point list LP
            break;
        WHITE: //empty leaf node
            break;
        GRAY: //internal node
            for *every son (i) of a do
                c_i.x = nc.x+dx_i*nl.x;  c_i.y = nc.y+dy_i*nl.y;
                l_i.x = nl.x/2;  l_i.y = nl.y/2;
                if (* reclangles (r,q) and (c_i,l_i) intersect)
                    rqs(c_i, l_i, r, q, a->fii[i])
                □
            □
    □

end
```

Obviously the computations associated with every son node could be parallely done.

Another structure suitable to point data representation is the point region tree (PR-tree). This structure allows specifying data points when the minimal distance between two points is not apriori known (i.e. the data points can not be associated to nodes of a fixed grid). It's a tree similar to the region quadtree, the data points being stored in buckets associated with some of the tree's leaves.

Following that, are given a pseudocode description and performance evaluation of two of the fundamental operations over data point collections modeled through PR quadtrees. The operations realize the insertion and the deletion of one data point from a PR quadtree. The PR quadtree could be relatively unbalanced if the points are nonuniformly distributed in a hyperrectangular data space. In case of an insertion operation of a point (P), first is searched the geometric quadrant (Q) which includes (P). If the data page associated to the leaf node corresponding to (Q) is full, the quadrant will be decomposed and the points from the original data page will be redistributed between four pages associated to the four subquadrants of (Q). The splitting process will continue until the addition of (P) will not cause the overflow of the data page.

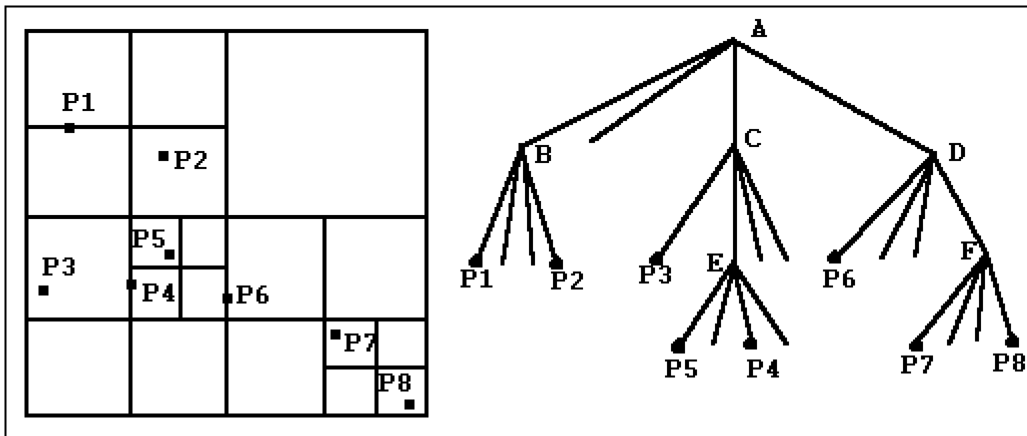

Figure 4

A PR-tree with paged data points is described by the following data_structure:

```
typedef struct aPRp {
        int tipnod; //0=internal node, 2=leaf node
        point_data *pag; //associated data point array
        int N; //the effective number of data points in a page
        struct aPRp *fii[4]; //pointers to the four sons
        } *PRtree;
```

```
function PRinsert(PRtree a, point d, point c, point l): PRtree
/* inserts the data point d, in tree a; points c and l represent the hyperrectangle associated with tree
node    a . The function returns a   pointer to the root of the resulted tree*/
    if(* a is empty)
        a ← *create a leaf node
        *add d to node's "a" data page
        return a
    □
    if(* a is a leaf)
        if(*d is in node's "a" data page)
            return a;
        else
            u ← a
            a ← *create an internal node
```

```
                    *rearrange data points from u->pag in data pages
                    associated to a->fii[i], 0≤i≤3
          □
     □
     temp ← a
     q ←       *find the subquadrant of the rectangle (c, l) where data point d is placed
     while temp->fii[q]≠NULL and *temp->fii[q] is internal node do
          temp ← temp->fii[q]
          *update (c, l) to correspond to the rectangle associated with temp
          q ←       *find the subquadrant of the rectangle (c,l) where data point d is placed
     □
     if(temp->fii[q]=NULL)
          temp->fii[q] ← *create a leaf node
          *add d to node's temp->fii[q] data page
          return a
     else
          u ← temp->fii[q]
          repeat
               temp->fii[q] ← *create an internal node
               temp ← temp->fii[q]
               *update (c, l) to correspond to the rectangle  associated  with temp
               *rearrange data points from u->pag in data pages
               associated to a->fii[i], 0≤i≤3
               free_node(u)
               fdet ← *find the subquadrant of the rectangle (c,l)
                    where data point d is placed
               if(*data page of son "fdet" is not empty)
                    *insert d in data page associated to
                    temp->fii[fdet]
                    return a
               □
               temp ← temp->fii[fdet]
               q ← fdet
          until false
     □
end
```

The running times of the insert/delete algorithms of a point into/from a PR page tree are specified in Table 1. (the case of uniformly distributed data points) and Table 2 (the case of increasingly sorted data points). The evaluations where done on a INTEL 486SX/33MHz processor.

Table 1

PR tree -- Insert/Delete Operations Running Times -- uniformly distributed data points

No. points	Insert time	Delete time	No. leaves	No. internal nodes	Height
100	5	6	16	5	3
1000	99	127	85	28	5
5000	659	769	535	178	6

Table 2

PR tree -- Insert/Delete Operations Running Times -- sorted data points

No. points	Insert time	Delete time	No. leaves	No. internal nodes	Height
100	5	11	13	4	3
1000	99	126	94	31	5
5000	643	769	556	185	6

3. Nonhierarchical methods for representation of multidimensional data points

The fixed grid method ([6]) divides the data space in equally sized square cells. This structure is similar to a k-dimensional array (disk stored); each array element corresponds to one grid cell and is implemented as a simple list containing the interior points of the cell.

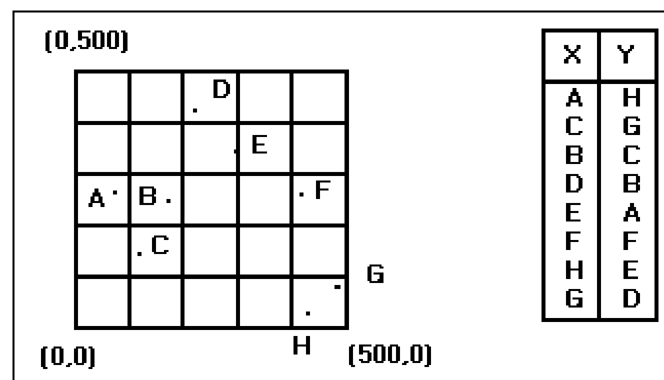

Figure 5

This kind of grid is particularly efficient in case of uniformly distributed data points and of requests which are using a fixed search grid. From the search complexity point of view this technique is similar with hash file indexing technique.

The most often referred method from the above mentioned category is the grid file ([2]). It is a method similar to the fixed grid but the hyperplanes dividing the data space are no more equally spaced. The grid blocks (hyperrectangular cells of the data space) are represented by elements of a multidimensional array. Each such array element is pointing a

data page that stores all the points located in the corresponding grid block. It is possible that more neighbor data blocks (which are forming together a hyperrectangular block) point to the same data page.

The mapping between the domain of a data point coordinate and the corresponding index of the grid directory is made through a collection of unidimensional arrays (one array for each dimension of the data space). These linear scales accomplish partitioning of the domain of each attribute (coordinate) of a data point. Using this method any point could be found with two disk accesses: the first to the grid directory and the second to the data file (the linear scales are usually stored in internal memory).

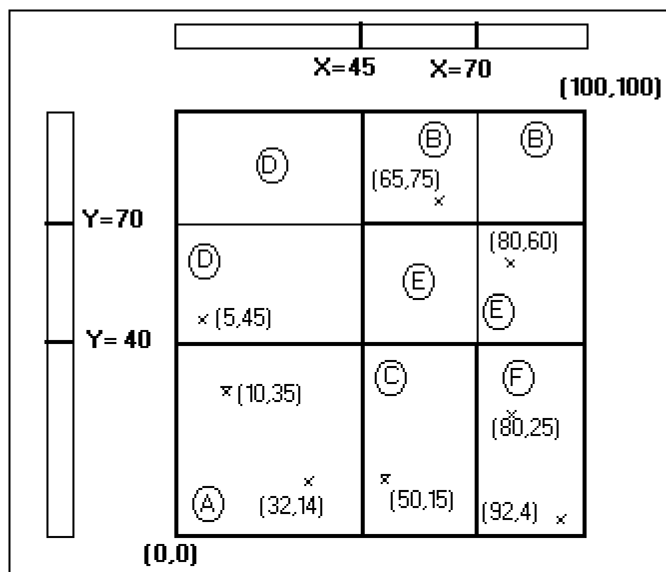

Figure 6

In my implementation a data page has a capacity of 20 points, the linear scales were implemented by simple linked lists.

```
typedef struct scl {
        int info;  /* value of a splitting coordinate */
        struct scl *urm;
} *scala;
typedef struct lscl {
        int N;  /* number of splitting sections */
        scala s;
        struct lscl *urm;
}
```

The mapping between the rectangular blocks of the data space and the data pages is accomplished by the file GRID.DAT who can be considered a column linearisation of a k-dimensional array). The data pages were stored in file PAGGRID.DAT. The file SCALE.DAT is useful for saving the linear scales at the end of a work session.

The procedure divide_grila is called during the insertion process when the capacity of a data page (P) - whose points are placed in the same grid block - overflows. This block

will be splitted through a hyperplane parallel with one of the coordinates hyperplanes. In the current implementation the division coordinate (dim_split) is cyclically choosed and the sectioning of the data block is made across the median value of the dim_split coordinate of the points from the data page. The blocks, resulted after a splitting process, which are pointing the same data page must form together a convex (rectangular) region in the data space.

```
procedure divide_grila(data_page pg, data_point p, ind_address kp, int nb)
        *find the median value (ds) of the dim_split coordinate of the points from pg and p
        *fetch from file "GRILA.DAT" all the page pointers  (corresponding to each block
          of the data space) and place them in tab_grid array
        *update the linear scale corresponding to the dim_split  dimension (inserting ds)
        *allocate a new data page (pag1)
        *rearrange the points from pg and point p between pg and  pag1
        *save pg and pag1 in PAGGRID.DAT file
        *increment total number of pages
        *increment the number of elements of dimension dim_split of  tab_gr
        *modify the pointers from tab_gr
        *save tab_gr in "GRILA.DAT" file
end
```

Due to the fact that in my implementation I used RAM disks to store data files GRILA.DAT, DIMMAX.DAT, PAGGRID.DAT and SCALE.DAT the resulted running times were quite short.

Table 3

Grid File -- Insert/Delete Operations Running Times

No. points	Insert time	Delete time	No. of data pages	Grid dimension	Data page fill factor
100	72	72	7	3*3	0.71
1000	418	302	77	11*11	0.65
5000	4125	1961	373	27*27	0.67

4. Conclusions

The multidimensional point data collections could be used in broad categories of applications in the field of: spatial databases (geographic information systems), relational databases (where tuples could be assimilated to points in a multidimensional data space), computer vision, computer graphics and image processing.

The paper describes two categories of methods used to represent such n-dimensional point sets: the hierarchical methods and the grid based methods. These data modelling techniques are particularly suitable to parallel processing. For example information in

subtrees of an arbitrary internnal node of a hierarchy based spatial decomposition data structure could be parallely processed. The same argument is valid in case of grid based spatial decomposition techniques; the information in every decomposition cell of the data space could be parallely processed.

C-style pseudocode was used to describe some representative multidimensional point sets data structures and associated operations. Performance evaluation was also considered.

REFERENCES

1. *D. Gordon, S. Chen,* Front to Back Display of BSP Trees, IEEE Computer Graphics and Applications, pp. 79_85, september 1991.
2. *J. Nievergelt, H. Hinterberger, K. C. Sevcik,* The grid file: an adaptable, symmetric, multikey file structure, ACM Transactions on DataBase Systems, pp. 38-71, march 1984.
3. *M. H. Overmars*, Geometric Data Structures for Computer Graphics: An Overview, Thoretical Foundations of Computer Graphics and CAD, Springer Verlag Berlin Heidelberg, pp. 21-49, 1988.
4. *H. Samet,* An Overview of Quadtrees, Octrees and Related Hierarchical Data Structures, Theoretical Foundations of Computer Graphics and CAD, Springer Verlag Berlin Heidelberg, pp.51-68, 1988.
5. *B. Seeger, H. Kriegel,* The Buddy-Tree: An Efficient and Robust Access Method for Spatial Data Base Systems, Proceedings of the 16th VLDB Conference, Brisbane Australia, pp. 990-1013, 1990.

6. *D. E. Knuth*, Tratat de Programare a Calculatoarelor, Editura Tehnicã, Bucuresti, pp.565-582, 1976.
7. *K. M. Chandi, J. Misra,* Parallel Program Design, Addison-Wesley Publishing Company, pp. 21-38, 81-150, 1988.
8. *M. Zaharia*, Structuri de date folosite în prelucrarea graficã a informatiei, Litografia U.P.B., pp. 76-93, 96_101, 1994