R-tree Indexing by Multiple Processors

Edward, Nai-Biu Tam Ada, Wai-Chee Fu
The Department of Computer Science and Engineering
The Chinese University of Hong Kong
Shatin, N.T., Hong Kong
E-mail: {nbtam, adafu}@cs.cuhk.hk

Abstract This work investigates the performance of parallel R-tree where the spatial data are high dimensional objects. Due to the large number of bounding values for each bounding box in the R-tree, one disk page might contain one bounding box only. Parallel Binary R-tree(PBR-tree) is proposed to facilitate the costly node access. An analytical performance evaluation of PBR-tree will be given. Improved Multiplexed R-tree for such data is proposed to achieve high throughput and short response time for concurrent small range queries.

1 Introduction

It is foreseen that the Database Management System(DBMS) of the future shall be able to handle a lot of new data representation. Multimedia data(spatial data) are often represented by vectors in a multidimensional spaces. Typical operation on spatial data involves range queries which retrieve all the data objects intersecting with the user defined hyper rectangle(search rectangle)[4].

Visual data such as image is one of the multimedia data. Features acquisition of an image includes image segmentation, region detection and image histogram[3]. Features such as color, texture, shape[6] and so on are used for indexing. A feature vector is typically extracted for each image. Due to the high complexity of this representation for multimedia data, the DBMS should be able to handle this large index dimensions.

Since the spatial data in multidimensional space cannot be easily mapped into a linear space, several spatial access methods(SAM) have been proposed for handling these data. A recent survey can be found in [9]. R-tree[4] and its variants: packed R-tree[8], R^*-tree[1] and R^+-tree[11] are among the most common cited SAMs.

1.1 R-tree

An R-tree is a B-tree like height-balanced tree structure. Due to space limitations, we omit a detail description of R-tree. Let M and $m(\leq \frac{M}{2})$ be the maximum and minimum number of entries in a node respectively. Let $f(m \leq f \leq M)$ be the average number of entries in a node. If the R-tree contains N index records, then its height will be at most $\lfloor \ln_m N \rfloor$. We assume the size of the bounding box representation is large so that a bounding box fits into one page.

Searching is done by descending the R-tree from the root node in a manner similar to B-tree. The child nodes with its bounding boxes overlapping with the search rectangle will be accessed. Unlike B-tree, there would be branching of search path. The number of pages accessed for a single data object is at least:

$$f|\ln_m N|$$

The performance of searching will be improved by shorter tree. Since more nodes would be accessed in case of branching, the cost of a query increases with search rectangle size.

1.2 Motivation

We need parallelism of R-tree because of the following reasons:

1. Increase the Database capacity.
2. Increase the throughput.
3. Minimize the response time.

A high throughput and short response time DBMS using parallelism of R-tree should satisfy the following requirements[2], [5]:

Scalable No distinguished processor will be a bottleneck. As a corollary, queries with large search rectangle should activate as many processors as possible.

Minimum Load Queries with small search rectangle should activate few processors so that other processors can be used to handle concurrent queries.

Our proposed architecture complies with the above requirements. It is intended to be a server that serves multiple users who submit queries with small search rectangle concurrently. The data objects are high dimensional spatial data and the key(bounding box) size is equal to a page size.

We propose a hardware architecture consisting of several processors connected by a network. Each processor is attached with a disk. The sibling nodes of a R-tree are distributed among different pairs of processors based on proximity index[5]. Each pair of processors build a parallel binary R-tree for accessing the hit subnodes in parallel, so as to minimize the node access time.

The paper is organized as follows. Section 2 surveys previous efforts to parallelize other file structure. Section 3 proposes the Parallel Binary R-tree as a way to facilitate the node access. Section 4 analyzes the behavior of the proposed structure. Section 5 proposes the Multiplexed Binary R-tree as a parallel indexing structure for the multimedia data. Section 6 gives the conclusions and the future direction.

2 Survey

We assume the hardware architecture consists of $r(=2h, h \geq 1)$ processors connected by a network and each processor has a local disk attached. There are different approaches in designing parallel R-tree[2], [5], [10]:

1. Independent R-trees
2. Super Nodes
3. Node Distribution(Multiplexed R-tree)

2.1 Independent R-trees

Each processor stores and manipulates a subset of the data set. Independent R-tree is built in each processor. Searching is executed in each processor separately. There are two types of data distribution[5]:

Data Distribution The data objects are assigned to different processors in a round robin fashion. As the data load, $\frac{N}{r}$, of all processors are equal. The maximum height of each R-tree will be $|\ln_m N - \ln_m r|$ which is reduced by a constant factor. Retrieval of a data object in an independent R-tree requires at least $f|\ln_m N - \ln_m r|$ page accesses.

Besides, all the processors should be activated for all size of search rectangle which violates the minimum load requirement.

Space Partitioning The data objects are inserted into the tree according to the multi-dimensional space assigned to the processors. Since the objects may not be evenly distributed in the space, it is possible that most of the data are inserted into the same tree so that this processor becomes the bottleneck of the system.

2.2 Super Nodes

In this scheme, the R-tree is built by large nodes(supernode). A supernode containing at most $dM(d \geq 1)$ entries is partitioned into r subnodes. Each subnode fits into $l(=\frac{dM}{r})$ pages. Every processor stores a different subnode. A node is accessed by reading the constitute subnodes in parallel for each processor. It can be executed in l page access time. A similar design of B-tree can be found in [7], [10].

Due to the large fanout of the node, the maximum height of super node R-tree will be reduced to $\lceil \frac{\ln_m N}{\ln_m \frac{lr}{M}+1} \rceil$. Hence the performance will be improved when the query is small. However, it violates the minimum load requirement since all processors are activated regardless of the size of query.

2.3 Multiplexed(MX) R-tree

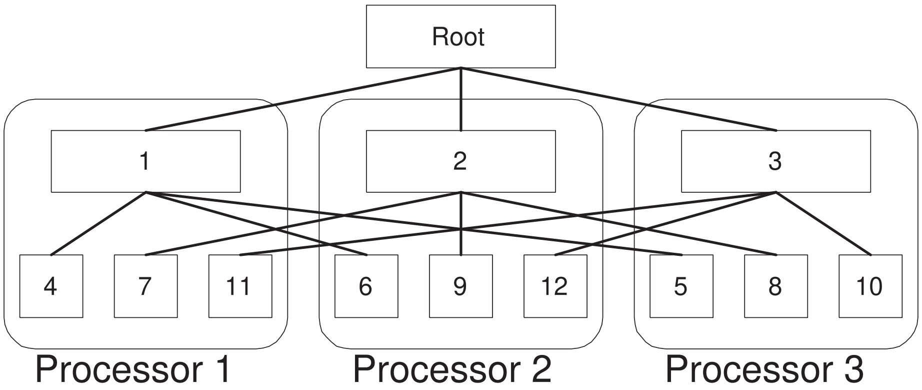


Fig. 1. Structure of Multiplexed R-tree

In this scheme, the child nodes are distributed over the processors, see figure 1, a detail description can be found in [5]. The assignment of the nodes are based on the proximity index[5] of them. Proximity is a measurement of similarity between two rectangles A and B. A node is assigned to a processor so that the nodes stored have the least proximity index.

Each pointer contains the processor ID. At each tree level, the processor of the current node activates the processors storing the child nodes which should be accessed at the next level. For a small query, only one or few processors will be activated at each level. For a large query which overlaps with more than one bounding boxes, the nodes can be accessed in parallel by different processors. Besides, the idle processors can serve the other queries. Hence these fulfill the scalability and minimum load requirements.

As a processor needs f pages access time to access a node, hence at least $f\lceil \ln_m N \rceil$ pages should be accessed for a single data retrieval. Comparing with super node, MX R-tree does not response faster for a single small query(small search rectangle). However, it complies with the requirements.

2.3.1 *Processor Assignment Algorithms*

A survey of the processor assignment heuristics can be found in [5]. Heuristics include:

Round Robin A new node is assigned to a processor in a round robin fashion.

Minimum Area This heuristic tries to balance the area covered by each processor. A new node is assigned to the processor with the smallest area covered.

Minimum Intersection This heuristic tries to minimize the intersection area of the nodes assigned to the same processor.

Proximity Index(PI) This heuristic will assign the node to the processor with the lowest proximity index.

The MXR-tree applies proximity index heuristic as the assignment algorithm. The detail description of the above heuristics is given in [5].

2.3.2 *Proximity Index*

Proximity is a measurement of similarity between two j-dimensional rectangles A and B. It is defined as the probability that a query rectangle will overlap both rectangles[5]. Let q be the number of queries retrieving both A and B and Q be the total number of queries. Thus,

$$Proximity_j(A, B) = \frac{|q|}{|Q|}$$

The proximity index $PI_j(r, R)$ measures the similarity of a rectangle r with a set of rectangles $R = \{R_1, \ldots, R_k\}$. It is defined as[5]:

$$PI_j(r, R) = \max_{R_i \in R} proximity_j(r, R_i)$$

R is the set of sibling nodes, having the same parent node, stored in the same processor[5]. A low PI means that it is improbable that more than one node should be accessed simultaneously in a query after r is inserted.

3 Binary R-tree for parallel processing

As mentioned above, a processor should read in f pages for node access in MXR-tree. However, typically only a few entries overlap with the search rectangle for small query, so that most of the page accesses are wasted. We propose to divide the node into f subnodes and introduce parallel processing of binary R-tree. We will show that this algorithm performs better than sequential reading in f pages.

The proposed architecture consist of two processors connected by a network and a disk is attached to each of them. This approach requires little synchronization.

3.1 *Structure*

The structure of a Parallel Binary R-tree(PBR-tree), figure 2, is similar to that of an ordinary R-tree but the fanout is at most 2. Each node consists of two subnodes. Each subnode contains a tuple $< K, p, s >$ where K is the bounding rectangle. For all non-leaf nodes, p and s are pointers to the child subnode containing the node ID. The leaf subnode contains a pointer p pointing to either another binary R-tree with processor ID or the data object, see figure 8, and s is a null value. The subnodes of the same node are stored in different processors, say processor 1 and processor 2.

The bounding rectangle K denotes the subtree or data object pointed to by p. Hence if K in subnode a overlaps with the search rectangle, the subtree rooted by b and b' should

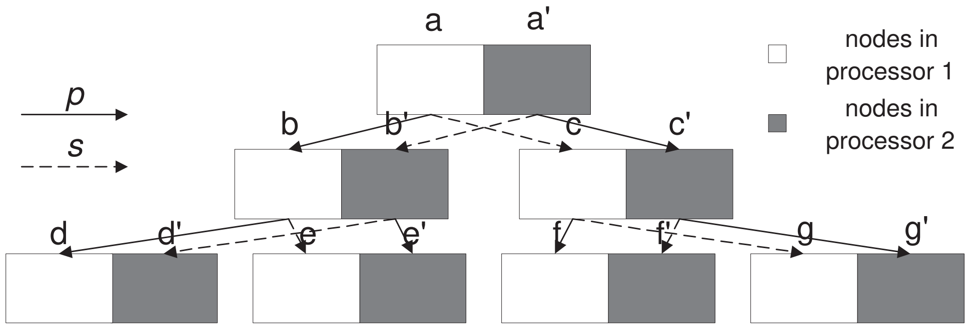


Fig. 2. Structure of a *PBR*-tree

be searched. Otherwise the interested data are not in that subtree. Similarly, if K in a' overlaps with the search rectangle, the subtree rooted by c and c' would be searched.

For each processor, the number of subnode stored is:

$$\sum_{i=0}^{\ln \frac{E}{2}} 2^i \leq E - 1 < E$$

where E is the number of leaf subnodes in the R-tree.

3.2 Searching

A depth first search approach is applied. Each processor starts searching from the root subnode(a for processor 1 and a' for processor 2). If a processor examining subnode with tuple $< K, p, s >$ finds that K overlaps with the query rectangle Q, it proceeds to the subnode pointed to by p. The result of checking $< K, p, s >$ will be a boolean value of $overlap(p, Q)$. This value is sent in a message to the other processor. If the subtree of p is traversed, it proceeds to the subtree pointed to by s. The subnodes in the subtree pointed to by j will not be accessed when message $overlap(j, Q) = False$ is received. The algorithm is shown in figure 4.

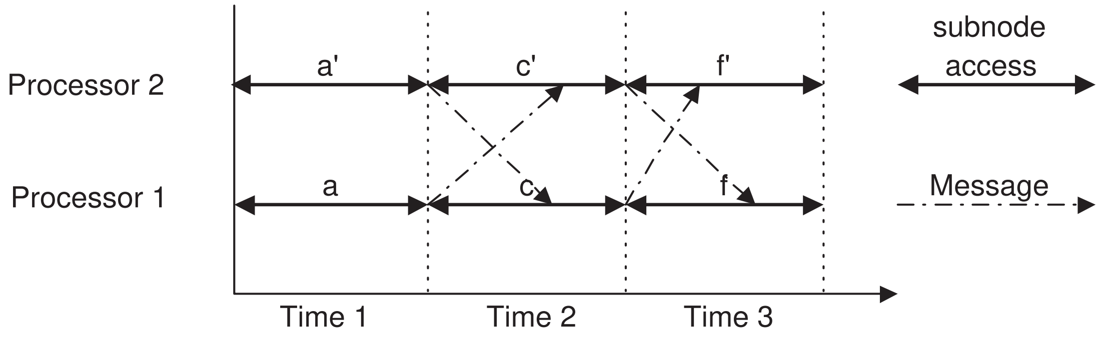


Fig. 3. Timing Diagram for the Example

For example, subnode f and f' overlap with the query. At the beginning, processor 1 and 2 access subnode a and a' respectively, see figure 3. Processor 1 detects that the left branch, see figure 2, does not contain the answer, it sends $overlap(a, Q) = False$ to processor 2 and then access subnode c. At last it accesses subnode f and returns the result. Since processor 2 would send $overlap(g', Q) = False$, the PN of processor 1 becomes empty, searching is terminated. Similarly, processor 2 will retrieve f' and the query is completed.

Searching

For current subnode $< K, p, s >$
If the current subnode is the root subnode
then
PN =: set of all pointers pointing to the nodes stored
$result$ =: $\emptyset$
endIf
If the current subnode is a leaf node
then
If K overlaps with search rectangle Q
then
add the object pointed to by p into $RESULT$
endIf
else
If K overlaps with the search rectangle Q
then
send $overlap(p, Q) = True$
apply Searching on the subnode pointed to by p
else
send $overlap(p, Q) = False$
remove the pointers of the subnodes rooted at p from PN
endIf
If the subnode pointed to by s is in PN
then
apply Searching on the subnode pointed to by s
endIf
endIf

Receiving Message

On receiving a message $overlap(j, Q) = L$
If $L = False$
then
remove the pointers of the subnodes rooted at j from PN
endIf

Fig. 4. Algorithm for Binary R-tree

3.3 Cost

For small query, the *PBR*-tree can retrieve a data in $\ln E$ page access time. For large query where all the data objects should be retrieved, each processor read in E - 1 pages. Hence the time complexity C for a query Q is:

$$C(Q) \in [\ln E, E - 1]$$

Note that most of the time, a processor can determine the next subnode to be accessed without synchronization. It reduces the communication cost.

4 Performance Analysis for *PBR*-tree

The performance($Perf$) of a *PBR*-tree can be formulated into:

$$Perf(n,q,P) = \frac{1}{\#\text{ of possible path}_{n,q}} \sum_{\forall path} \text{expected access time}_p \tag{1}$$

Variable Name	Definition
n	# of leaf subnodes
q	# of leaf subnodes to be retrieved
P	prob. of $Q \cap K_i \neq \emptyset$ given $\forall j, Q \cap K_j = \emptyset$ Q is the search rectangle, K_i is the bounding rectangle of node i, j is the child node of i

The height of the tree is proportional to the number of leaf subnodes n. The value of q is proportional to the query rectangle size. There are C_q^m possible combinations of subnodes retrieved. Since the bounding rectangle of the parent nodes(K_{parent}) contains some sparse area, it is possible that there exists a query with rectangle Q overlapping with K_{parent} but none of that of the objects inserted in the tree. The parameter P is the probability that a processor inspecting node i determines that Q overlaps with K_i given that Q does not overlap with K_j for all child nodes j of node i.

A path($PATH$) is the minimum set of nodes visited from the root node to the leaf nodes for a query. Hence for any n and q,

$$\# \ of \ possible \ path = C_q^n$$

Assuming the cost of message passing and computation are negligible, the access time is proportional to the maximum number of subnodes(pages) accessed among the two processors. For a processor a inspecting subnode i with tuple $< K_i, p_i, s_i >$, the total number of subnode accessed($expNd$) from node i is

$$expNd_P^a(i) = 1 + \begin{cases} 0 & \text{if } PN = \emptyset \\ expNd_P^a(o_i) & \text{if } i \text{ is a leaf node} \\ & \quad \text{or } p_i \notin PN \\ expNd_P^a(p_i) & \text{if } p_i \in PATH \\ P \cdot expNd_P^a(p_i) + (1-P)expNd_P^a(o_i) & \text{if } p_i \notin PATH \end{cases} \tag{2}$$

o_i is the next subnode to be accessed if p_i is not chosen according to the algorithm shown in figure 4. Since the pointer pointing to the subnodes will not be removed from PN if K_i overlaps Q, see algorithm in figure 4, the value of $expNd_P^a(p_i)$ is usually larger than that of $expNd_P^a(o_i)$.

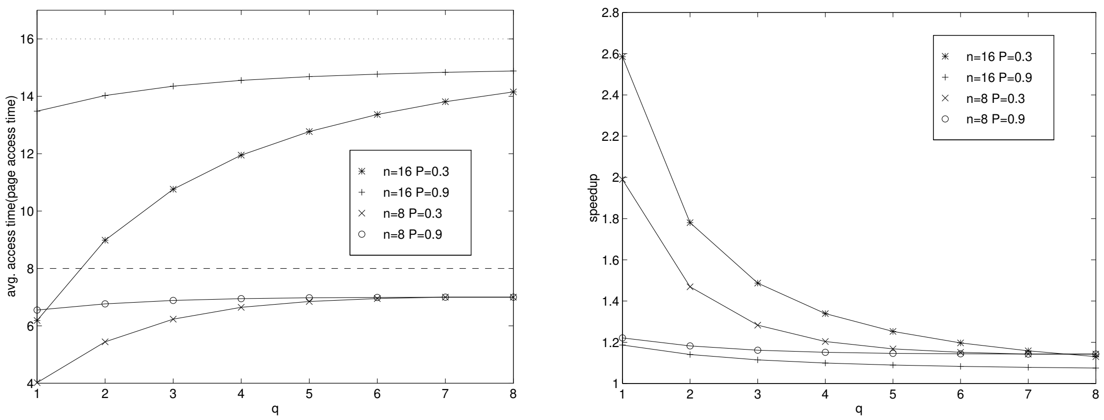
Let r_{0a} and r_{0b} be the root subnodes. Processor a and b are assigned to the *PBR*-tree. Hence equation 1 is re-written as

$$Perf(n,q,P) = \frac{\text{subnode access time}}{\#\text{ of possible path}} \sum_{\forall PATH} \max_{i \in \{a,b\}} expNd_P^i(r_{0i}) \tag{3}$$

4.1 Analysis

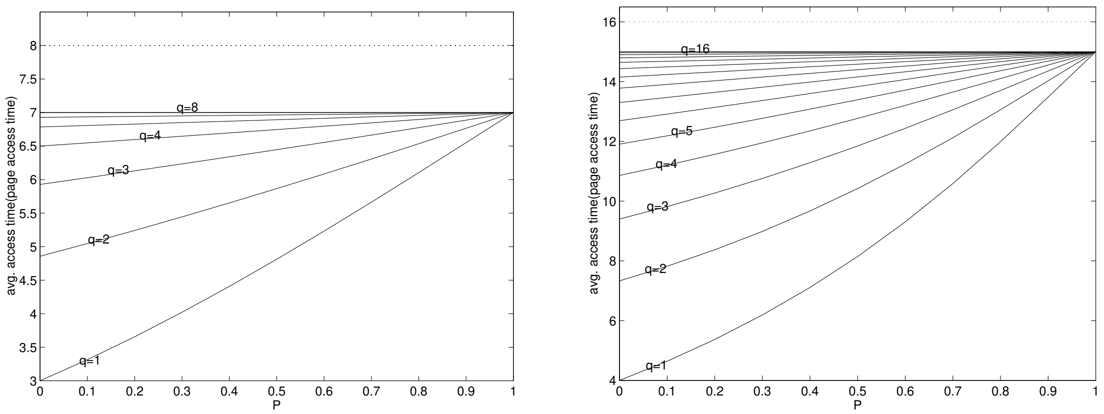
To evaluate the performance of the *PBR*-tree. We have executed a number of simulation experiments on the basis of the mathematical cost model as described in equation 2 and 3. We assume a subnode access costs a page access, the performance($Perf$) of *PBR*-tree is expressed in term of page access time. The parameter values used are:

Parameter	values
n	$\{8, 16\}$
q	$\{1, 2, \ldots, n\}$
P	$\{0, 0.1, 0.2, \ldots, 1\}$



(a) access time (b) speedup

Fig. 5. Performance of PBR-tree



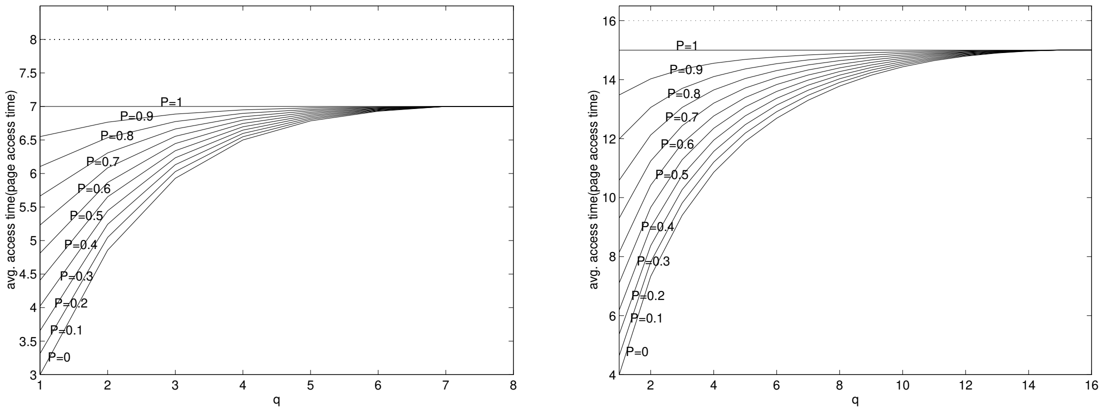
(a) n=8 (b) n=16

Fig. 6. Access time against P

Comparing to sequential reading in n pages, PBR-tree performs better, see figure 5(a) and (b). For the same value of q and P, a smaller n give a shorter response time because of its shorter tree height. According to equation 2, fewer values will be accumulated for shorter tree. However a larger n gives the higher speedup because $\frac{n}{\ln n}$ is a strictly increasing function.

From figure 6, the access time of PBR-tree increases with parameter P. As the nodes is less likely to be removed from PN for higher P, the total number of nodes should be accessed is increased. Hence higher access time will result. When P approach 1, the access time is close to the n - 1 node access time because all nodes should be accessed.

The access time drops dramatically for small q, see figure 7, because the size of $PATH$ is small for small q.



(a) n=8 (b) n=16

Fig. 7. Access time against q

5 Multiplexed Binary R-tree

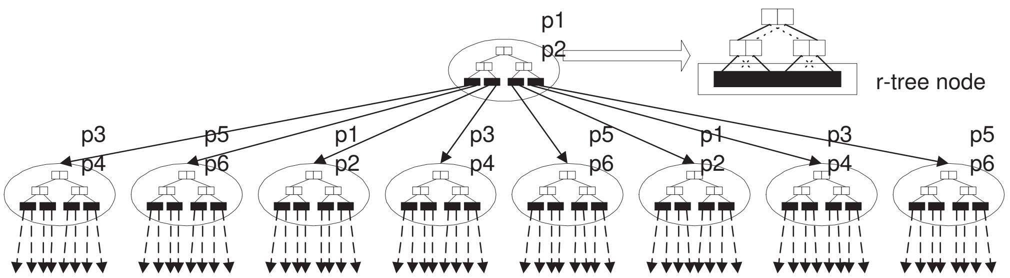


Fig. 8. MXBR-tree

A MXR-tree has better response time than the super node method and it will lead to higher throughput[5]. We adopt the general idea of MXR-tree. However for high dimensions, a node access which involves sequential reading f pages, will be very costly. And most of the subnodes read may not overlap with the query. We propose a Multiplexed Binary(MXB) R-tree which accesses the hit subnodes by searching a binary R-tree, figure 8.

We decompose a R-tree node into subnodes and build a small binary R-tree. A pair of processors are chosen for accessing this tree. The assignments of the processors are based on proximity index[5], hence the binary R-trees built in the same processors for the same level of nodes have the least similarity.

5.1 Searching

The searching algorithm is similar to that of MXR-tree. However, unlike MXR-tree, a node access will activate 2 processors.

For example in figure 8, processor 1 and 2 access the root and determine which child nodes should be accessed by searching its binary R-tree. Then it activates the processors responding for that child nodes, processor 3 and 4 for instance.

5.2 Performance

For small query, the MXBR-tree takes $\ln f$ time to access a node. For large query, it needs $f-1$ pages access time. Hence it would perform better than MXR-tree especially for small queries.

For each node access, 2 processors are activated. The idle processors can serve the other queries. Besides, for large query, all the hit child nodes can be accessed in parallel. Hence MXBR-tree can increase the throughput especially for large queries or multi-users environment.

6 Conclusions

We propose a Multiplexed Binary R-tree for a multi-users environment. The node access is accelerated by using Parallel Binary R-tree. For small query, we can access a node in $\ln f$ pages access time, compared to f pages access time for MXR-tree.

Since only 2 processors are activated for each node access, multiple queries or large queries can be executed in parallel. We can have a higher throughput than a super node method.

The proposed Multiplexed Binary R-tree will be a suitable method for a multiple users environment in which the search rectangle size is typically small.

Future direction includes applying the idea of PBR-tree onto other SAM.

7 Acknowledgment

This research is sponsored partially by XXXXX

References

[1] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The r*-tree: an effecient and robust access method for points and rectangles. In *Proc. ACM SIGMOD*, pages 322–31, September 1990.

[2] David J. DeWitt and Jim Gray. Parallel database systems: The future of database processing of a passing fad? In *Proc. ACM SIGMOD*, volume 19, pages 104–12, December 1990.

[3] Yihong Gong, Hongjiang Zhang, H.C. Chuan, and M. Sakauchi. An image database system with content capturing and fast image indexing abilites. In *Int. Conf. on Multimedia Computing and Systems*, pages 121–30, 1994.

[4] A. Guttman. R-trees: a dynamic index structure for spatial searching. In *Proc. ACM SIGMOD*, pages 47–57, June 1984.

[5] Ibrahim Kamel and Christos Faloutsos. Parallel r-trees. In *Proc. ACM SIGMOD*, pages 195–204, June 1992.

[6] W. Niblack, R. Barber, W. Equitz, M. Flickner, E.Glasman, D. Petkovic, P. Yanker, C. Faloutsos, and G. Taubin. The qbic project: Querying images by content using color, texture, and shape. In *SPIE/IS&T Conference on Storage and Retrieval for Image and Video Databases*, pages 173–87, 1993.

[7] S. Pramanik and M.H. Kim. Parallel proceeding of large node b-trees. In *Trans. on Computers*, pages 1208–12, 1990.

[8] N. Roussopoulos and D. Leifker. Direct spatial search on pictorial database using packed r-trees. In *Proc. ACM SIGMOD*, pages 17–31, May 1985.

[9] H. Samet. *The design and Analysis of Spatial Data Structures.* Addison-Wesley, 1989.

[10] Bernhard Seeger and Per-Ake Larson. Multi-disk b-trees. In *Proc. ACM SIGMOD*, pages 436–45, May 1991.

[11] T. Sellis, N. Roussopoulos, and C. Faloutsos. The r+ tree: a dynamic index for multi-dimensional objects. In *Proc. of the VLDB Conf.*, pages 507–18, England, September 1987.