Parallelizing Multidimensional Index Structures*

K. V. Ravi Kanth Divyakant Agrawal Amr El Abbadi Ambuj Singh
Terence R. Smith
University of California at Santa Barbara
Santa Barbara, CA 93106

Abstract

Indexing multidimensional data is inherently complex leading to slow query processing. This behavior becomes more pronounced with the increase in database size and/or number of dimensions. In this paper, we address this issue by processing an index structure in parallel. First, we study different ways of partitioning an index structure. We then propose efficient algorithms for processing each query in parallel on the index structure. Using these strategies, we parallelized two multidimensional index structures – R and LIB and evaluated the performance gains for the Gazetteer and the Catalog data of the Alexandria Digital Library on the Meiko CS-2.*

1 Introduction

Advances in processing technology and storage management over the last decade have opened up new avenues for handling large and complex map and image data. Efficient management and retrieval of such multidimensional data is of primary concern for several reasons. Many projects such as the Alexandria Digital Library [21], which maintain large collections of spatial and geographic data, have to provide efficient access to such multidimensional data. Most image database systems approximate the images as large multidimensional feature vectors and the retrieval of images in such systems is based on searching in the multidimensional feature space. Similarly, large sequence database systems for handwriting recognition [5] also use multidimensional indices. The attributes in relational database systems can be viewed as forming a multidimensional space. With so many applications being naturally modeled as multidimensional data, there have been many attempts to provide efficient index structures, but due to the inherent complexity of multidimensional data, there appears to be no clear winner among the plethora of indices [2, 4, 7, 12, 14, 1, 13, 16, 11] in the literature. However, some structures like the R^* trees [1] perform reasonably well for rectangular data whereas others like KDB-trees [16] are good for point data. In [10], we have devised a new multidimensional index structure called the LIB-structure that exploits the semantics of the data to gain efficiency. In comparison to the unidimensional index structures such as the B-trees, however, the multidimensional structures still remain slow due to lack of comparable bounds for all types of operations and all types of data. This effect is compounded with the increase in the size of the data in multi-dimensions.

In this paper, we propose an alternative means to gain performance: parallelism. Parallelism can take advantage of the need for searching multiple portions of the index for most of the queries like containment and intersection. Kamel and Faloutsos [6] present the idea of *proximity index* for storing data on multiple disks and use simulations to show performance gains for R-trees with multiple disks on a single processor. In this paper, we focus on parallel performance from multiple processors each with its own disk. We study several ways of parallelizing an index structure in such an environment. These approaches are independent of the index structure and hence can be applied to any index. We also discuss the effect of parallelization on query processing. Since the index is distributed, processors may have to evaluate the query on the remote components of the index. We explore how this can be accomplished without the intervention of the remote processor.

We implemented some of the proposed strategies for parallelization of two index structures – the LIB structure and the R^* tree. We chose the LIB structure because it performs well for certain types of spatial data. We selected the R^* because it is representative of some of the versatile multidimensional indices [4, 1, 20] that cater to arbitrary data. We parallelized these two structures and evaluated their performance on two datasets of the Alexandria Digital Library – the Gazetteer data for the region of California, and the Catalog data for the whole world.

The paper is organized as follows: Section 2 describes the two multidimensional structures. Section 3 presents

*Work supported in part by a research grant from NSF/ARPA/NASA IRI–9411330.

our techniques to parallelize index structures. Section 4 presents the results of parallelizing the two structures on the Meiko CS-2 followed by our conclusions in Section 5.

2 Review of multidimensional index structures

In this section, we describe two multidimensional index structures – the R* trees and the LIB structures, which we chose to parallelize. As we see later, the parallelizing techniques can be applied to arbitrary index structures and are not restricted to these two indices.

The LIB structure [10] segregates the data into different levels of nesting based on the $containment$ relationships of objects. (An object a is *contained* in object b if the region covered by a in the n-dimensional space is a subset of that covered by b). The data at each level of nesting is organized using a separate index structure. The particular index structure we choose for each layer depends on the size of the data and its characteristics. For example, if the data can fit in memory, then we organize it as a tree of trees so that searching in memory is efficient. Otherwise, we organize it as a disk-based index structure like the R^* tree or the KDB-tree – the choice being made based on the data characteristics. If the data consists only point data, then we choose an efficient point data structure like the KDB-tree; if it contains many rectangles then we choose the R^* tree.

When the data is small enough to fit in memory, it is organized in the form of a tree of trees. The tree at each layer indexes the objects using their intervals (projections) on a specific dimension. Figure 1 shows an example LIB structure for 2-dimensional data. Here, the data is organized into k levels of nesting. At each level, the data is organized into a 2-layered tree of trees, one layer per dimension. The first layer indexes the objects using the intervals along X-axis and the second layer along the Y-axis. For indexing intervals along a chosen dimension, we have developed a new unidimensional index structure called the IB-tree [10]. The IB-tree is a B-tree like structure which guarantees $logarithmic$ bounds on all operations when the data is not nested.

Maintaining separate indices for each nesting level has two main advantages. First, the index at each level can be chosen depending upon the characteristics of data at that level. So if a particular level contains only point data, we can take advantage of this by having an efficient point indexing scheme for that level. Second, the indices at different nesting levels are not interdependent and hence can be searched in parallel. However, it is quite possible that most of the data is concentrated at a particular nesting level, limiting the parallelism. In this case, we need to split the index

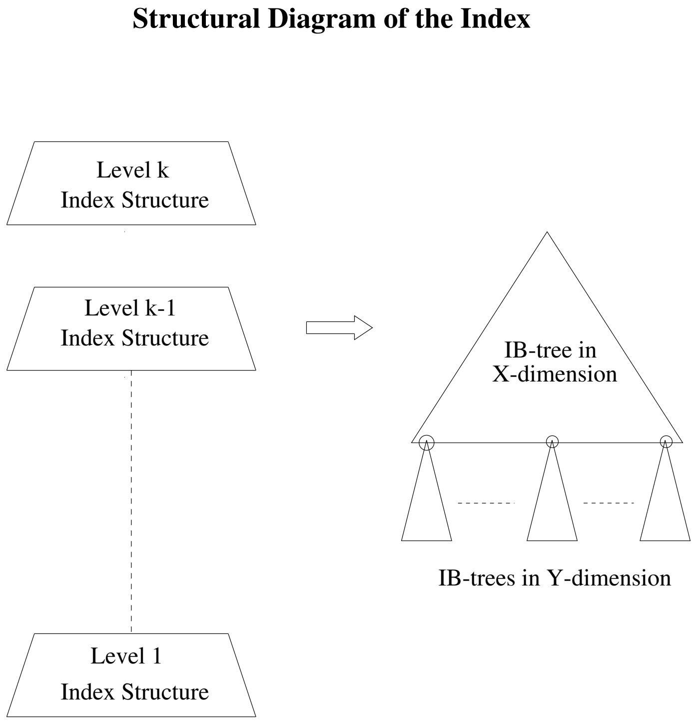

Figure 1. An example LIB structure for 2-dimensional data

at that level into several components so that they may be searched in parallel.

The R^* tree [1], on the other hand, is a single B-tree like structure for multidimensional data. Hence, each node in the tree has between $\lceil m/2 \rceil$ and m children (m being the node capacity), whereas the root has at least 2 children. The leaf nodes store the data objects whereas the non-leaf nodes have entries of the form $< bb, child\_pointer >$, where bb is the $minimal$ bounding box containing all the entries in the $child\_pointer$ node. Unlike B-trees, the index entries in R^* trees can be overlapping regions. Consequently, locating a specific object may involve searching of multiple paths in the tree.

The two most common types of queries on multidimensional index structures are: (1) the *containment* queries, where the index retrieves all data objects that are *contained* in a user-specified query window and, (2) the *intersection* queries, where the index retrieves all data objects that intersect the user's query window. Processing of both these types of queries involves multiple path traversal in the index structure. By parallelizing the searching of such multiple paths, we can improve the performance of the index structures. Note that even in unidimensional indices like the B-trees, containment and intersection queries involve multiple paths. However, the number of such multiple paths is small in comparison to the large number in multidimensional indices which may have overlapping index entries.

3 Parallelizing Index structures

We assume a message-passing network of processors, each having its own local disk. The indices reside either in the processor's memory or on its local disk. If an index structure resides entirely at one processor (either in its memory or on its local disk), then all queries on the index can be processed locally by that processor. However, due to the complexity of multidimensional indices, the query processing is very slow and hence such a sequential solution does not scale up to large databases. Therefore, we propose to speed-up the index performance by processing it in parallel. The parallelization of indices involves two phases: (1) partitioning the index structure into several components and assigning these components to different processors, and (2) parallel searching of the partitioned indices during query processing. We study each of these issues in turn. We model the index structure as a *forest* of *search* trees. Each node of a search tree is associated with a region in the multidimensional space. This region is used to ascertain whether or not the subtree rooted at this node is relevant to a query. The search within a tree proceeds from the root to the leaves. We do not presume any order in processing the search trees of an index. In this framework, the LIB structure of Figure 1 has k search trees whereas an R^* tree always has only one.

The problem at hand is to physically layout the forest of search trees on a given number of processors to achieve small query response times. This problem is similar to the problem of partitioning a task graph [3, 17] on a given number of processors. The difference is that in task graph partitioning, all the nodes (tasks) in the task graph are to be executed while minimizing an objective function (say, parallel time). In contrast, only a few of the search nodes in a forest are evaluated during query, and we have to lay out the forest in such a way that these nodes are processed in parallel.

3.1 Index Partitioning

We propose a two-phase solution to the problem of partitioning a forest of k search trees into p components, where p is the number of processors. In our solution, we measure the load on a processor in terms of the number of search nodes that are assigned to it.

Phase 1 – Assign Search trees

The k search trees are assigned in chunks of $\lfloor k/p \rfloor$ search trees to each processor. The first processor gets the first $\lfloor k/p \rfloor$ search trees, the second processor the next $\lfloor k/p \rfloor$ search trees and so on. The last processors gets all the remaining search trees. This assignment corresponds to *block striping* [9] of the search trees.

Phase 2 – Partition individual search trees

If there is a heavily loaded processor, then the largest search tree (with maximum number of nodes) assigned to that processor is partitioned using the *tree partitioning* scheme described below. This process is repeated until there is an acceptable level of imbalance.

3.1.1 Tree Partitioning scheme

Let G be the search tree that needs to be partitioned. We first estimate the likelihood of two nodes in G being searched in the same query using *proximity-index* [6] for the associated regions. Kamel and Faloutsos define the proximity-index of two regions in multidimensional space as the likelihood of the their being accessed in the same query and devise a way to compute it. We extend this idea to two arbitrary nodes a, b of a search tree G as follows:

- If a, b are connected in G, then $proximity\text{-}index(a, b) = 0$.
- Otherwise, $proximity\text{-}index(a, b)$ is the proximity-index of the regions covered by a and b as defined by Kamel and Faloutsos [6].

If the proximity-index of two nodes is high enough, then we try to assign them to different processors so that they can be searched in parallel. However, for any two nodes that are connected, the proximity-index is set to 0 for the following reasons.

1. Since the edge indicates the relative search order, the two nodes are never searched in parallel in the same query.
2. Assigning them to different processors leads to unnecessary communication overheads for queries that traverse the corresponding tree-edge.

The search tree G is partitioned as follows:

1. Compute the proximity-measure for every pair of nodes in G.
2. Let S_i be the set of all nodes assigned to processor i. This set is initially empty.
3. Assign one node to each processor as follows:
    - For each node, compute the sum of their proximities with respect to every other unassigned node in the tree.

- Find the node with the maximum sum and assign to a new processor, say i. Include it in the set S_i.
- Repeat this assignment until each processor gets a node from the search tree

4. Assign each remaining node n to a processor as follows:
   - for each processor i, compute the sum Sum_i of the proximities of node n with respect to all nodes in the set S_i.
   - Assign node n to the processor i with the minimum sum. Include the node n in the set S_i.

We now describe the issues in query processing for such a partitioned index.

3.2 Query Processing

For every query, all processors do the following:

- Process the query on each search tree independently.
- On traversing a tree edge to a node b assigned to some other processor, the processor sends a remote request to the processor of node b. The query on node b is then evaluated by the remote processor.
- Process any remote requests from other processors (as a consequence of the above step).
- Once the query is evaluated on the local components of all the search trees and all incoming remote requests serviced, the processor participates in a distributed termination protocol.

After termination of query processing, the results are accumulated from all the processors at a *coordinator* processor.

Let us look at the issues in receiving a remote request. Each processor can potentially receive requests from all other processors and hence queuing these requests at a processor and servicing them is akin to the *multiple producers–single consumer* problem. There are many solutions to this fundamental problem [19]. However, to maximize performance of the index structures, we would like to ensure that the destination processor of a remote request can receive it without much overhead. Modern parallel machines like the Meiko CS-2 provide additional support to accomplish queuing at a remote processor without interrupting its computation. Schauser et al. [18] came up with several solutions for emulating remote queues on architectures that support one or more of the following:

- remote *read/writes* ,
- remote *test-and-set*, and
- a separate co-processor to handle communication.

Using such architectural support, they propose the following approaches to emulate a remote queue:

1. Maintain a separate queue on each processor P (consumer) for every other processor (producer). Each queue contains a flag that indicates if it is *empty*. Hence the processor P polls $O(p)$ queue flags (p is the number of processors) to service any incoming remote requests.

2. Maintain one queue on each processor (consumer) to which all remote processors (producers) can write. Writing to this shared queue requires mutually exclusive access, which can be implemented using remote test-and-set operations.

3. Handle requests by co-processor.

In our case, a processor can poll its queue flags when it is idling after processing its own load. Hence polling the queue flags does not interfere with the computation. So the only cost for the first approach is the latency for remote writes. Since the remote writes (DMA) have the least latency of the above three approaches, we choose the first approach to solve the remote-queuing problem. In addition, we may need to queue more than one request from each remote processor. We simply extend the above solution by allocating sufficient storage in each of the queues. Hence, we maintain a storage of $O(N)$ across all the processors, where N is the number of nodes in the precedence graph.

Given the above strategy to solve the remote-queuing problem, the processor P_i on termination of its local computation (i.e., query processing on local components of the index and servicing any incoming remote requests), sends the number of remote requests it has sent and received to the *coordinator* processor which detects and declares termination of the query.

4 Experiments

In this section we discuss our experiments to parallelize the LIB structure and the R^* tree indices for multidimensional data.

We ran our experiments on the Meiko CS-2, which is a 64-node parallel machine. Each node of the Meiko is a 40MHz SPARC viking with 32MB. The interprocessor communication is through the ELAN communication network with a one-way latency of 10us.

The data structures (R* and LIB-structures) are implemented in C under Solaris 2.3 operating system. Using the parallelizing techniques we proposed, we evaluated the performance of the two index structures on a varying number of processors of the Meiko.

4.1 Setup

The first dataset we used in our experiments is a sample of the extents of $features$ in the Alexandria Gazetteer for the state of California. The Alexandria Gazetteer is built by merging the GNIS (Geographic Names Information Systems) Gazetteer, and BGN (Board of Geographic Names) Gazetteer. The GNIS Gazetteer has locations, bounding coordinates (in degrees) for places and other geographical features of interest in United States whereas the BGN has worldwide information. The combined gazetteer has 5 million entries most of which have a point extent. Since the number of nesting levels for the Gazetteer is only 7, the LIB structures are ideally suited for them. We ran our experiments on the California subset of the Gazetteer, which has 72K entries with a total size of 1.5MB. The *spatial* queries to the Gazetteer from the Alexandria Web Page mainly consist of a user specified query box, which is then used to retrieve all the features from the Gazetteer that are *contained* in the query box (*containment* query) or those that *intersect* the query box (*intersection* queries).

The second dataset is the catalog of all holdings in the Alexandria Digital Library. Each holding in the catalog has an associated spatial extent, which is used at query time (in conjunction with other information) for searching the catalog database. To speed-up the retrieval processing, we index these spatial extents. Unlike the Gazetteer, most of these spatial extents are rectangular objects and are spread out over a large number of nesting levels. Specifically, the 400K data of the catalog has 36 levels of nesting. Since each data entry is 20bytes, the total size of the catalog data is 8MB. Note that this high level of nesting is not suited to the LIB structure which maintains a separate index for each level. This will become evident in our experiments.

Since our dataset is 2-dimensional in nature, we experimented with 2-dimensional queries. The queries generated are ensured to span 0.1%, 1% or 10% of the area of the domain (California). Similar assumptions are made in [15] and [8]. Each query window is assumed to be a square and hence the sides of the query window are calculated from the area. These sides are scaled appropriately for the domain we are dealing with. So for a 1% area query, the sides of the query window have to be 10% of the corresponding sides of the domain in each dimension. The location of each query is generated using a random number generator. 100 intersection queries and 100 containment queries for each query span are generated in this manner.

We now discuss our parallelizing strategies for the two index structures on the Gazetteer dataset and analyze the results of parallelization. Although we plot the results for 1% and 10% area queries only, similar results are obtained for 0.1% area query.

Next, we describe the partitioning strategy for the Gazetteer dataset. The catalog is partitioned in a similar fashion.

4.2 Partitioning the LIB structure

As described earlier, the LIB structure maintains a separate index structure for data at each nesting level. Hence, it segregates the 72K data into 7 nesting levels and maintains a separate index for each level. Since 65K of the 72K data are point data, the LIB structure maintains a KDB-tree for this point data and a 2-level index for data at all other levels. While parallelizing the LIB structure, we studied the effect of balancing the load by partitioning individual search trees (Phase 2 of Index partitioning scheme). First, we evaluated the parallel performance gains by just distributing the 7 search trees to different processors. The lowest nesting level (level 0, corresponding to point objects) has 65K data while all the others have only 7K among them. Hence query processing at this level dominates the overall behavior. As a result, we expect very little gain in performance with increase in number of processors for this scheme (LIB1). To improve the situation, we used our second tenet in partitioning: *partition the search trees that cause load imbalance* (as described in Phase 2 of the Index Partitioning). Hence, we partitioned the data at the lowest level using *proximity index* and assigned it to all the p processors. We expect to reap better results with this partitioned index (LIB2).

Figure 2 compares the two schemes for both the containment and intersection queries. The values plotted are the average response times for 100 queries when the indices are kept fully in memory. The dotted lines indicate the curves for intersection queries and the solid lines indicate the curves for containment queries.

Since the size of the result set for an intersection query is higher than that for a containment query, the response times are higher for an intersection in this figure (and in the subsequent ones too). From Figure 2, we note that the LIB1 gains very little in performance as predicted due to load imbalance. However, LIB2 remedies this situation by distributing the lowest level data across all the processors achieving a much better performance gain than LIB1.

Another interesting phenomenon can be observed in Figure 2 for the curves for LIB1. The intersection queries start gaining in performance for just 3 processors whereas the containment queries do so only for 4 processors. This can be explained as follows. The size of the result set for an intersection query is much higher than that for a containment query. Besides, intersections are more expensive than

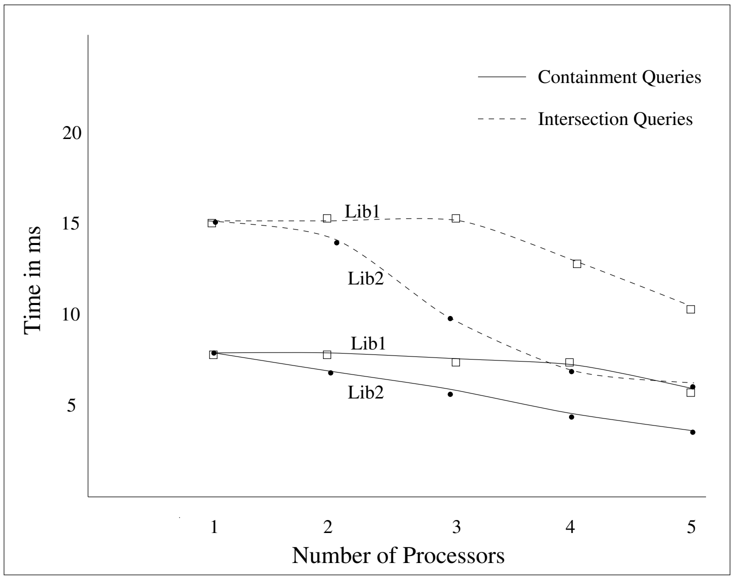

Figure 2. Performance Gains for Parallel schemes for LIB (1% Query)

containment queries in the 2-level index structure of an LIB. Consequently, the excess processing overhead is evenly distributed even for just 3 processors, thus reducing the load imbalance. This explains the difference in the behavior of LIB1 for intersection and containment queries.

As expected, LIB2 has a uniform decrease in response times (for both intersection and containment) with the increase in the number of processors. This indicates the superiority of adaptively partitioning the index depending upon the data.

We now describe the partitioning strategy for R^* trees followed by a comparison of its performance results with those for LIB2 (hereafter referred to simply as LIB).

4.3 Partitioning the R^* tree

For each node in the R^* tree, we first compute the proximity-measure of the node with respect to each of its siblings. Here, we compute the proximity-measure only with respect to its siblings as opposed to computing it with respect to every other node in the search tree. Once the proximity-measures are computed, the children of each node are assigned to different processors using the tree-partitioning technique explained earlier.

Query processing involves traversing the different components of the R^* tree that are distributed among all the processors. After completion of local processing for a query, each processor participates in a query termination protocol to ensure that there are no remote requests that are in transit before processing the next query.

We now describe some of the results of parallelizing the R^* and the LIB structure.

4.4 Performance Evaluations

Figure 3 shows the performance of R^* and the LIB when the indices are in memory. The response times shown are the average for 100 queries. The query spans covered a total of 1% of the domain area. Here we see that the R^* trees gain well as the number of processors is increased. This is due to the parallel processing of multiple nodes at the same level of the tree. For containment queries, LIB does better than R^*. However, for intersection queries, they perform almost the same. Similar results are obtained for a 10% query as shown in Figure 4.

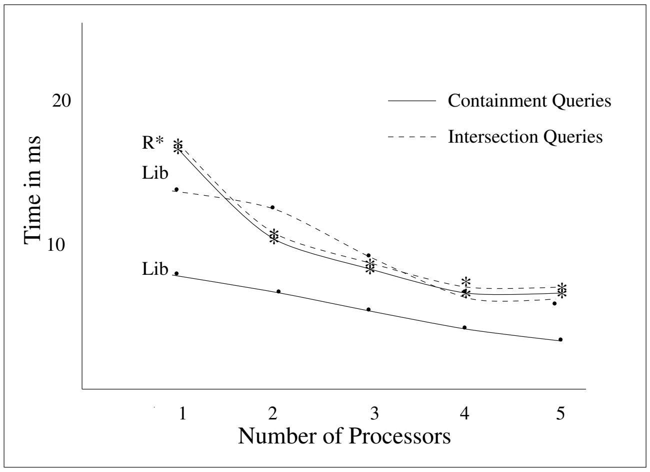

Figure 3. Performance Gains for In-Memory Indices (1% Query)

Since most systems deal with indices that are disk-resident, we placed the indices on disk and evaluated the performance gains due to parallelization. To mimic real-life situations, we allowed a total cache of 40 pages. In order to model large databases, the node capacity in each tree is fixed at 128 (data or index) entries. This corresponds to a page size of 8K, assuming the spatial coordinates are each 8bytes (*double*). Although disk accesses are expensive, the gains in performance for the disk-resident indices are similar to those for in-memory indices. These results are plotted in Figure 5 and Figure 6 for 1% and 10% queries respectively. From the figures, we see that the performance of the data structures improves with the increase in the number of processors. We also note that as the number of processors is increased, the difference in performance of the R^* and the LIB structures reduces. This is expected because the excess

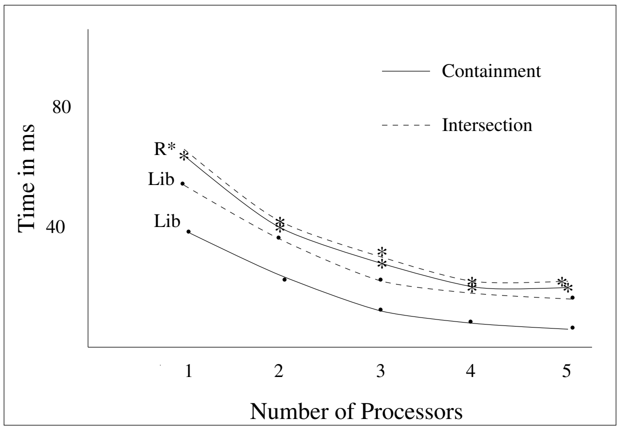

Figure 4. Performance Gains for In-Memory Indices (10% Query)

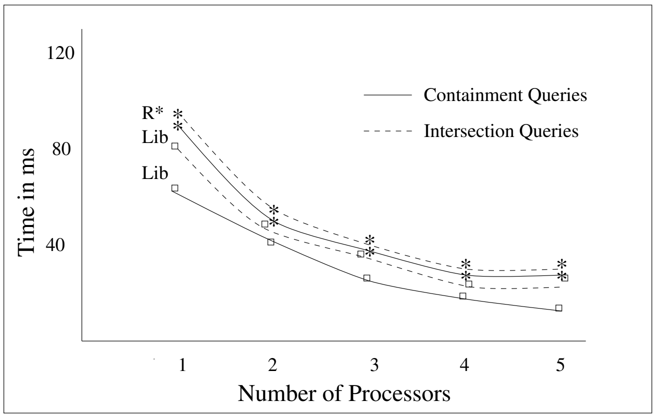

Figure 6. Performance Gains for Disk-Resident Indices (10% Query)

load of searching multiple nodes is nullified by parallelizing such processing. Also, the *flatness* of the curves for 5 processors in the figures implies the possible saturation of performance for the index structures. This indicates that 4-5 processors are sufficient to get reasonable speed-ups in performance for the dataset considered.

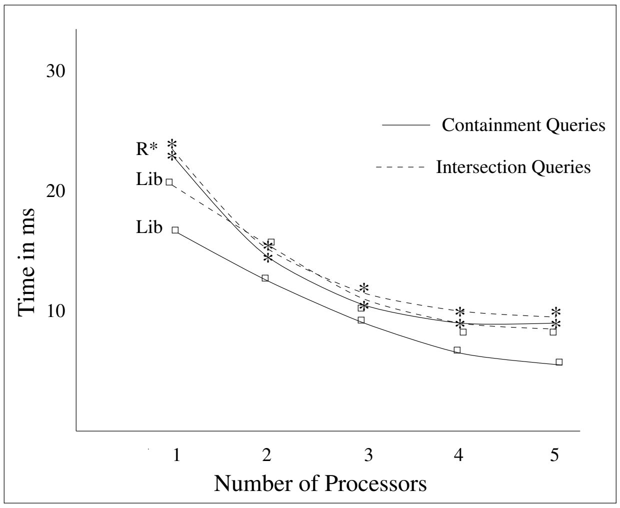

Figure 5. Performance Gains for Disk-Resident Indices (1% Query)

Next, we look at the performance gains by having parallel indices on the catalog data. We experimented with 400K data of the catalog. Since we are dealing with a sufficiently large database, we allowed the nodes to have their full capacity of 200 entries for a disk page size of 8k (corresponds to a 4byte floating point representation for each spatial coordinate). Unlike the gazetteer, most of the objects in the catalog are rectangles and hence we cannot take advantage of efficient point indexing schemes like the K-D-B tree as in the case of gazetteer. In addition, the data is spread over a large number of levels. Since it cannot fit in memory and has non-point objects, the Level-based index (LIB) has to maintain an R^* tree for each nesting level. Since we cannot keep all the 400K data and the associated indices in memory, we only experimented with the indices on disk. To be fair to LIB, which has to deal with different data sizes at each nesting level, we cached the root nodes at each level. This leads to 36 pages of caching for both the structures. Figure 7 plots the average containment query response times of both LIB and R^* for a 1% query area against the number of processors. We see that both R^* and the LIB gain reasonable speed-ups in performance. Unlike for the gazetteer data , R^* performs better than LIB. This is explained as follows. The catalog has 36 nesting levels. Since the data is scattered over these levels, the LIB maintains an independent R^* tree for each such level. Accessing each of these trees may result in the retrieval of one or more disk pages. Consequently, the number of disk pages accessed by LIB increases for each query. Hence the response times degrade as shown in Figure 7. Similar results are obtained for other query ranges.

5 Conclusions

In this paper, we investigated the effects of parallelism for query processing in multidimensional indices. We first described different ways of partitioning an index structure by modeling it as a forest of search trees. Then, we discussed different issues in the query processing of a partitioned index. We noted that remote queues have a significant impact on the performance of an index and therefore

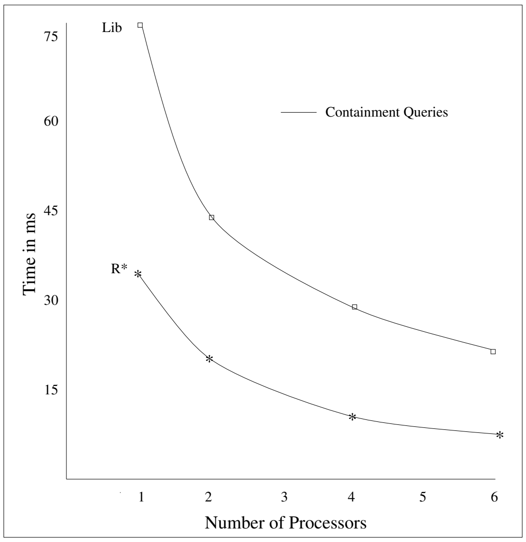

Figure 7. Performance Gains for Disk-Resident Indices on the Catalog (1% Query)

examined different alternatives to emulate such queues and came up with an efficient one for the Meiko CS-2. We then evaluated the parallel query performance of the nodes for each of the data structures LIB and R^*. The speed-up curves obtained are promising and indicate scope for improvement for multidimensional index structures using parallel processing. We observed that the LIB structure outperforms R^* on the gazetteer data. However, R^* does better than LIB for the catalog data which is highly nested. Hence, for datasets that have low nesting, arising either from the data characteristics or by a semantic restriction on the nesting, the LIB performs better than the R^*. Otherwise, the R^* does better. We also showed that performance enhancements through parallelism can be obtained for both memory-resident and disk-resident indices.

As a future direction for this research, we are investigating the effect of parallelism on other operations like inserts and deletes in multidimensional indices. This is of specific importance to commercial database systems because multidimensional index creation involves successive insertions into an initially empty index. Each such insertion involves considerable computation and I/O overhead. Moreover, the complexity of the insertions grows with the size of the database. Hence handling insertions in parallel would be one way to scale up to large datasets.

References

[1] N. Beckmann, H. Kriegel, R. Schneider, and B. Seeger. The R* tree: An efficient and robust access method for points and rectangles. *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, pages 322–331, May 23-25 1990.

[2] M. W. Freeston. The bang file: a new kind of grid file. *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, pages 260–269, May 1987.

[3] A. Gerasoulis and T. Yang. On the granularity and clustering of directed acyclic task graphs. *IEEE Trans on Parallel and Distributed Systems*, 4(6):686–701, June 1993.

[4] A. Guttman. R-trees: A dynamic index structure for spatial searching. *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, pages 47–57, June 1984.

[5] I. Kamel. Fast retrieval of cursive handwriting. *Technical Report MITL-TR-134-94*, 1994.

[6] I. Kamel and C. Faloutsos. Parallel r-trees. *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, pages 195–204, 1992.

[7] C. Kolovson and M. Stonebraker. Segment indexes: Dynamic indexing techniques for multi-dimensional interval data. *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, pages 138–147, May 1991.

[8] H. P. Kriegel, M. Schwietz, R. Schneider, and B. Seeger. Performance comparison of point and spatial access methods. *Design and Implementation of Large Spatial Databases, First Symposium, SSD 89*, pages 88–114, 1993.

[9] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing*. Bejamin-Cumming Publishing Company, Erewhon, NC, 1994.

[10] K.V.Ravi Kanth, D. Agrawal, A. El Abbadi, A. Singh, and T.R.Smith. Indexing hierarchical data. *Technical Report TRCS-95-014*, 1995.

[11] K.Y.Whang and R.Krishnamurthy. Multilevel grid files. *Technical Report, IBM Research Lab.*, 1985.

[12] D. B. Lomet and B. Salzberg. The hb-tree: A multiattribute indexing method with good guaranteed performance. *ACM Trans. on Database Systems*, 15(4):625–658, 1990.

[13] M.A.Ouksel and O.Mayer. A robust and effiecient spatial data structure. *Acta Informatica*, 29:335–373, 1992.

[14] J. Nievergelt, H. Hinterberger, and K. C. Sevcik. The grid file: an adaptable symmetric multikey file structure. *ACM Trans. on Database Systems*, 9(1):38–71, 1984.

[15] B. U. Pagel, H. W. Siz, H. Toben, and P. Widmayer. Towards an analysis of range query performance. *Proc. of 12th ACM PODS Symposium*, pages 214–221, May 1993.

[16] J. Robinson. The k-d-b tree: A search structure for large multidimensional indexes. *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, 10-18, 1981.

[17] V. Sarkar. *Partitioning and Scheduling parallel programs for execution on multiprocessors*. MIT Press, Cambridge, MA, 1989.

[18] K. Schauser and C. Schieman. Experience with active messages on Meiko CS-2. *9th Intl. Parallel Processing Symposium IPPS95*, pages 140–149, 1995.

[19] A. Schilberschatz, J.L.Peterson, and P.B.Galvin. *Operating Systems Concepts*. Addison-Wesley Publishing Co., Erewhon, NC, 1989.

[20] T. Sellis, N. Roussopoulos, and C.Faloutsos. The R+ trees: An dynamic index for multidimensional objects. *Proc. of the 13th Intl. Conf. on Very Large Databases*, pages 507–518, September 1987.

[21] T.R.Smith and J.Frew. Alexandria Digital Library. *Communications of the ACM*, 38(4):61–62, April 1995.