Parallel Processing of Spatial Joins Using R-trees

Thomas Brinkhoff[1]*, Hans-Peter Kriegel*[1]*, Bernhard Seeger*[2]

[1]Institut für Informatik, Universität München, Leopoldstr. 11 B, D-80802 München, Germany

[2]Fachgebiet Informatik, Universität Marburg, Hans-Meerwein-Str, D-35032 Marburg, Germany

e-mail: {brink,kriegel,bseeger}@informatik.uni-muenchen.de

Abstract

In this paper, we show that spatial joins are very suitable to be processed on a parallel hardware platform. The parallel system is equipped with a so-called shared virtual memory which is well-suited for the design and implementation of parallel spatial join algorithms. We start with an algorithm that consists of three phases: task creation, task assignment and parallel task execution. In order to reduce CPU- and I/O-cost, the three phases are processed in a fashion that preserves spatial locality. Dynamic load balancing is achieved by splitting tasks into smaller ones and reassigning some of the smaller tasks to idle processors. In an experimental performance comparison, we identify the advantages and disadvantages of several variants of our algorithm. The most efficient one shows an almost optimal speed-up under the assumption that the number of disks is sufficiently large.

1 Introduction

Spatial database systems (SDBS) must cope with vast amounts of spatial objects such as points, lines, polygons, etc. One of the most important design goals is therefore to equip a SDBS with efficient implementations of the basic spatial operators. Among these operators, the *window query* and the *spatial join* are considered to be the most important ones. The window query is restricted to a scan through a single spatial relation, whereas the spatial join combines two (or more) spatial relations into one. In contrast to ordinary relational joins, the join predicate refers to a spatial predicate, e.g. the test of polygons for intersection. An example of a spatial join is the query "find all forests which are in a city" assuming that there are two spatial relations "forests" and "cities".

In this paper, we address the problem of exploiting CPU- and I/O-parallelism to improve the efficiency of spatial join processing. The reason for investigating parallelism is twofold. First, although the run time of sequential spatial join processing has considerably been improved over the last few years, the response time of the most efficient sequential algorithms is far from meeting the requirements of an interactive user who expects answers within a few seconds. Second, a current hardware trend is the development of inexpensive parallel computer systems from conventional memories, processors and disks. It is obvious that such hardware can only be exploited to a full extent by a SDBS when the system is directed explicitly to parallelism. For natural joins, it has impressively been shown that algorithms can take great advantage from parallel hardware, see [Gra 93] for a survey. Since, however, processing spatial joins is different from processing natural joins, the same approach cannot be used for spatial joins. The general approach of applying parallelism to implementing parallel SDBS has attracted research attention recently, for example in the Paradise project [DeW 94]. Therefore, the question arises whether parallelism is also a cost-effective approach to improving the efficiency of spatial joins.

Since on the one hand shared-everything multiprocessors offer only a limited potential of parallelism and on the other hand shared-nothing multiprocessors are difficult to program (e.g. load balancing), a current trend is to design hybrid multiprocessor systems that avoid those deficiencies of the classical architectures. In the following, we examine spatial join processing primarily for such a hybrid architecture that can be viewed as a *shared-disk architecture*. Our hardware platform consists of 24 processors each of them equipped with 32 MB of main memory. The processors are connected by a network with a throughput of 32 MB/s. Although the number of processors is admittedly rather small, it is still considerably higher than the number of processors of a typical shared-everything multiprocessor system. In contrast to the pure shared-nothing architecture, our hardware platform offers only one *data processor* dedicated to providing the interface to secondary storage. A more important difference to the pure shared-nothing architecture is also the availability of *shared virtual memory (SVM)* that provides a global address space. SVM facilitates the design and the implementation of parallel algorithms that require communication and dynamic load balancing. Query processing on a SVM shared-nothing architecture has received very little attention in the database literature. The work of Shatdal and Naughton [SN 93] is the only one that we are aware of. They showed that drastic performance improvements can be achieved on parallel database systems in the presence of data skew.

Recently, Hoel and Samet ([HS 94b], [HS 94c]) also examined parallel processing of spatial joins. Our approach is, however, completely different for several reasons. First, their approach is designed for a special-purpose platform, whereas our approach is implemented on a hybrid of shared-nothing and shared-memory architecture. Second, the I/O-cost of spatial join processing is considered in our approach, whereas this is not the case in the work of Hoel and Samet. Third, our approach is based on *R-trees* [Gut 84]. The reason for using R-trees in our approach is that the results of our previous work on sequential join processing [BKS 93] demonstrated that R-trees are a very efficient data structure to support spatial joins. Moreover, the R-tree is a well-known multidimensional spatial access method already implemented in several research prototypes (e.g. Paradise [DeW 94]) and commercial products (e.g. Illustra [Mon 93]).

The rest of this paper is organized as follows. Section 2 gives a review on previous approaches to spatial join processing. In particular, we present the most important techniques used in the sequential spatial join algorithms based on R-trees. Section 3 is concerned with the parallel processing of spatial joins. In detail, we present different approaches to organizing buffers and discuss their impact on performance. Moreover, we also investigate how to distribute the work load among the processors and how the load balancing can be achieved in presence of load skew. Section 4 presents results obtained from a set of experiments which were performed on a real machine (KSR1). Finally, section 5 concludes the paper and gives an outlook to future work.

2 Sequential Processing of Spatial Joins

In the following, we first give a brief review of the previous work on spatial join processing. Next, we discuss the sequential strategy of processing spatial joins using R-trees. This strategy is also the starting point of our approach to parallel processing of spatial joins.

2.1 Review of previous work

Recently, spatial join processing has gained much attention in the database literature. The central idea in almost all papers is that join processing consists of at least one filter step and one refinement step. In a *filter step*, the spatial join is not computed on the original relations, but on collections of simple conservative approximations. For each object, there is one approximation which can refer to a set of cells obtained from a equidistant grid [OM 88] or to a single geometric primitive, e.g. rectilinear *minimum bounding rectangle (MBR)*. A filter step produces a set of *candidates* that contains all answers of the spatial join and some others (*false hits*) which do not fulfill the join predicate. In order to eliminate the false hits, a *refinement step* is necessary where the exact geometry of the candidates is tested against the join predicate identifying answers and false hits.

Most of the investigations have focused on the improvement of the first filter step. The approaches can be classified depending on whether a spatial index exists on none, one or both spatial relations. Becker and Güting [BG 90] examined strategies that belong to the first two classes, whereas Lo and Ravishankar [LR 94] proposed a method based on the assumption that an index already exists on one of the relations. Most research attention has however been given to the case when an index exists on each of the relations. Orenstein and Manola [OM 88] proposed to use B-trees combined with z-ordering for processing spatial joins, whereas Brinkhoff et al. [BKS 93] proposed a filter step based on R-trees that organize the MBRs of the spatial objects. We will follow this approach for parallel processing of spatial joins and, therefore, a detailed discussion on this method will be presented in the next subsection. Other approaches are based on grid files [BHF 93] and generalization trees [Gün 93] which can be viewed as a generalization of R-trees.

Relatively little work has been done on the refinement step of the spatial join. In the refinement step, the remaining candidates, which are still not identified to be false hits or answers, have to be checked whether they satisfy the join predicate or not. This requires that the exact geometry has to be read from secondary storage into main memory and that the join predicate is checked by using the exact geometry. Brinkhoff and Kriegel [BK 94] showed that (spatial) clustering of spatial objects considerably reduces the time required for loading the exact geometry. In [BKSS 94], it was found that an appropriate exact representation of the objects can also considerably reduce the CPU-time required for checking the join predicate. Moreover, another filter step can further reduce the total cost of sptail joins. [BKS 94]. Since a second filter step and the refinements step do not influence the parallel design of spatial joins, it is not considered in the following.

2.2 Processing the First Filter Step using R-trees

In the following, we discuss how to perform the filter step using R-trees [Gut 84]. Among the members of the R-tree family, the *R*-tree* [BKSS 90] has frequently been referenced as the most promising approach so far. Therefore, our approach is based on R*-trees although it is directly applicable to the other members of the family.

The basic idea of performing a filter step with R*-trees is to use the property that directory rectangles form the minimum bounding rectangle of the data rectangles in the corresponding subtrees. Thus, if the rectangles of two directory entries, say E_R and E_S, do not have a common intersection, there will be no pair $(rect_R, rect_S)$ of intersecting data rectangles where $rect_R$ is in the subtree of E_R and $rect_S$ is in the subtree of E_S. Otherwise, there might be a pair of intersecting data rectangles in the corresponding subtrees.

The algorithm presented in [BKS 93] starts from the root s of the trees and traverses both of the trees in a depth-first order. For each qualifying (intersecting) pair of directory rectangles, the algorithm follows the corresponding references to the nodes stored on the next lower level of the trees. Results are found when the leaf level is reached. In order to reduce the cost of processing, several tuning techniques are applied to the algorithm. These techniques will be discussed below in more detail.

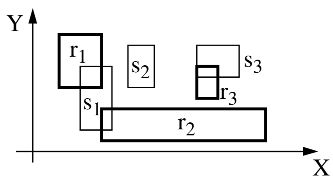
The R*-tree makes use of a so-called *path buffer* accommodating all nodes of the path which was accessed last. In order to be more efficient with respect to I/O, an additional buffer is used for single pages, not complete paths, independently of the path buffer. The buffer, called *LRU-buffer*, follows the least recently used policy. The reason for two different buffers is that the path buffer exclusively belongs to the R*-tree, whereas the LRU-buffer is considered as a buffer of the underlying database or operating system.

Performance Tuning Techniques

In order to reduce CPU-time, we examined two approaches [BKS 93]: (i) for a given pair of nodes, we restrict the search space of the join such that only a small number of entries in the original algorithm has to be considered, (ii) entries are sorted according to their spatial location and thereafter, an algorithm based on the plane-sweep paradigm [PS 85] is used to compute the desired pairs of intersecting entries. Since a pair of pages is associated with a pair of (intersecting) entries, the sequence of entries directly results in a sequence of pages to be read from secondary storage. Therefore, the second method also reduces the I/O-time. Both approaches are also used in our parallel processing strategy. Moreover, the second approach also has a great impact on the design of our parallel processing strategy and, therefore, a detailed discussion follows.

The idea of our approach is to sort the entries in a node of the R*-tree according to the spatial location of the corresponding rectangles. Obviously, this cannot be achieved without any loss of locality. A suitable solution with respect to computing the intersection is the following one. Let us consider a sequence $\mathfrak{R} = \langle r_1,...,r_k \rangle$ of k rectangles. A rectangle r_i is given by its lower left corner $(r_i.xl, r_i.yl)$ and its upper right corner $(r_i.xu, r_i.yu)$. A sequence $\mathfrak{R} = \langle r_1,...,r_k \rangle$ is sorted with respect to the x-axis if $r_i.xl \le r_{i+1}.xl$, $1 \le i < k$.

Plane sweep is a common technique for computing intersections. The basic idea is to move a line, the so-called *sweep-line*, perpendicular to one of the axes, e.g. the x-axis, from left to right. Given two sequences of rectangles $\mathfrak{R} = \langle r_1,\ldots,r_k \rangle$ and $\mathfrak{I} = \langle s_1,\ldots,s_m \rangle$ sorted with respect to the x-axis, we exploit the plane-sweep technique without the overhead of building up any additional dynamic data structure. First, the sweep-line moves to the rectangle, say t, in $\mathfrak{R} \cup \mathfrak{I}$ with the lowest xl-value. If the rectangle t is in $\mathfrak{R}$, we sequentially traverse $\mathfrak{I}$ starting from its first rectangle until a rectangle, say s_h, in $\mathfrak{I}$ is found whose xl-value is greater than $t.xu$. For each rectangle s_j, $1 \le j <$ h, we test whether it intersects rectangle t. Otherwise, if rectangle t is in $\mathfrak{I}$, $\mathfrak{R}$ is traversed analogously. Thereafter, rectangle t is marked to be processed. Then, the sweep-line is moved to the next unmarked rectangle in $\mathfrak{R} \cup \mathfrak{I}$ with the lowest xl-value and the same step as described above is repeated for all unmarked rectangles. When the last entry from $\mathfrak{R} \cup \mathfrak{I}$ was processed, all intersections are computed.



sequence of intersection tests

$t = r_1$: $r_1 \leftrightarrow s_1$
$t = s_1$: $s_1 \leftrightarrow r_2$
$t = r_2$: $r_2 \leftrightarrow s_2$, $r_2 \leftrightarrow s_3$
$t = s_2$: -
$t = r_3$: $r_3 \leftrightarrow s_3$

Figure 1: Example for the Local Plane-Sweep Order

An example how the algorithm proceeds is illustrated in Figure . The sweep-line stops at rectangles r_1, s_1, r_2, s_2 and r_3. For each stop, the pairs of rectangles which are tested for intersection are given on the right hand side of Figure . As mentioned above, the sequence of pairs of intersecting rectangles directly results in a sequence which determines the order how pages are read from secondary storage. This order is called the *local plane-sweep order*. When pages are read according to the local plane-sweep order, spatial locality is also preserved in the LRU-buffer.

3 Parallel Processing of Spatial Joins

In spite of the improvements achieved for sequential join processing, the spatial join is a time-consuming operation where the response time is far beyond the expectations of an interactive user. Therefore, it is necessary to investigate the potential that parallel computer architectures offer for accelerating the spatial join.

Spatial join processing cannot directly exploit the technique of data declustering which is generally used as the basis for processing natural joins in parallel (partitioned parallelism [Gra 93]). Given a declustered data placement of spatial relations R and S into p disjoint subsets $R_1,\ldots,R_p$, and $S_1,\ldots,S_p$, respectively, the union of the response sets obtained from processing spatial joins of R_j and S_j, $1 \le j \le p$, is only a subset of the response set of the spatial join of R and S. This makes the design of parallel spatial join algorithms more complex. Either data replication or communication between processors is required for parallel processing of spatial joins.

Our parallel approach of join processing will follow the idea of partitioned parallelism (with data replication and processor communication). As a consequence, an appropriate distribution of objects to processors is most important in the design of our algorithm. The distribution of objects is based only on the first filter step such that when a processor has determined a candidate in the first step, the same processor will execute further filter steps and, if necessary, also the refinement step. Therefore, we basically restrict our discussion on the filter step using R*-trees.

Let us first discuss the most important cost components which determine the total cost of parallel spatial join processing. Similar to the sequential processing, we particularly have to consider CPU- and I/O-cost. The *CPU-cost* is primarily determined by testing the spatial join predicate, e.g. whether two objects intersect or not. Such a test is considerably more expensive than the simple test of a relational join predicate, e.g. whether two values are equal. The *I/O-cost* consists of reading the pages of the access method from secondary storage into main memory as well as reading the exact geometry of the objects. In a parallel system, additional cost components can occur: *communication cost,* e.g. for transferring data from one processor to another, and *synchronization cost* for accessing or updating data stored in a shared memory.

Our goal is to minimize the CPU- and the I/O-cost as much as possible by using a parallel spatial join algorithm without causing much communication and synchronization cost. First, we will start with an algorithm which needs almost no communication and synchronization. In order to increase the performance of this first approach, we will then introduce additional concepts which require some communication and synchronization.

3.1 A First Approach

The first approach consists of three phases:

1.) Create a set of tasks to be executed in parallel *(task creation)*. For a parallel join processing using R*-trees, e.g. a task refers to performing the sequential algorithm on a pair of subtrees. This phase is sequentially executed on one processor.

2.) Because the number of tasks is generally higher than the number of processors, we need an algorithm for assigning each task to a processor *(task assignment)*. The tasks assigned to one processor form the *work load* of this processor. This step is also performed sequentially.

3.) Execute the tasks assigned to a processor without any communication to the other processors *(task execution)*. This phase is completely performed in parallel.

Obviously, the question arises: What is a suitable task creation and task assignment for spatial joins? We assume that one task corresponds to processing the spatial join on a pair of subtrees of the R*-trees where the affiliated two MBRs intersect. Because we want to avoid any communication between the processors, we should try to define work loads which are as much as possible independent from each other. Otherwise, one object may belong to work loads of different processors. Note that this is a property of the spatial join which cannot occur for natural joins. As a consequence, each of these processors would individually read the object from disk which causes high I/O-cost. In order to reduce the number of objects belonging to different work loads, we use spatial adjacency as the criterion for the task assignment. To put it in concrete terms: a work load consists of a set of spatially adjacent pairs of subtrees. For creating such a work load, we can use the *local plane-sweep order* again (see section 2.2).

In the following, m denotes the number of intersecting MBRs in the roots of the participating R*-trees and n the number of processors. We assume that m is much larger than n. If this condition is not fulfilled, the next lower level of the R*-trees will be considered for the task assignment. Such a pair of intersecting MBRs corresponds to a task. Then, we traverse these tasks according to the local plane-sweep order. The first m modulo n processors receive $\lceil m / n \rceil$ pairs of subtrees according to the order, whereas the others receive $\lfloor m / n \rfloor$ pairs. Each of these n groups corresponds to one work load. Because the tasks are assigned completely before the task execution starts, this type of task assignment is termed *static range assignment*.

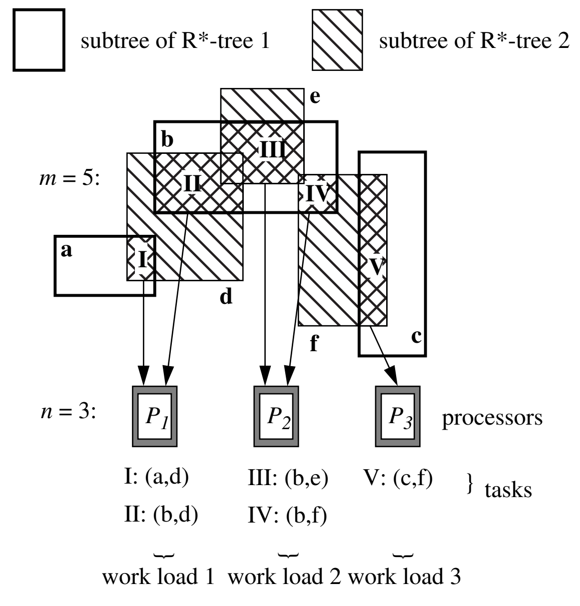


Figure 2: Example for the Static Range Assignment.

In Figure 2, an example illustrates the static range assignment. Each root of the R*-trees consists of 3 entries, m is 5 and n is 3.

After the task assignment, each processor joins the subtrees of its tasks independently from the other processors. The spatial join is finished as soon as all tasks are completely processed. The presented approach has one major advantage: As mentioned before, the task execution avoids communication between the tasks and, in particular, no shared memory is used. This is of great importance when the interconnection network is slow. However, our first approach for parallel spatial join processing has also several disadvantages:

- Due to the independent processing of tasks, the following situation may occur: Two or more processors operate on the same object at the same time. In Figure 2, this situation can occur for objects in subtree *b* which will be processed by the processors P_1 and P_2. The I/O-cost for reading the object from disk is generally higher than the cost for transferring it between processors. In such a case, it is therefore reasonable that only one processor reads the required page(s) from disk into its buffer and that the other processors read the page(s) from the buffer of the first processor. However, in the approach presented so far, the processors do not know about the pages kept in the buffers of the other processors. Therefore, they will independently read the page(s) from disk and thus cause much higher I/O-cost.
- The second observation is related to the work loads: In general, they will not be balanced among the processors. In fact, the number of tasks is approximately the same for all processors but the time for executing different tasks varies resulting in a varying execution time for the work loads.

In order to avoid these problems, we will propose more sophisticated algorithms for parallel spatial join processing in the next sections.

3.2 Buffer Organization

The first problem described in the last section is caused by the missing knowledge about the pages stored in the *local buffers* of the other processors. A page has to be read from disk although it is already in the local buffer of a processor.

Local buffers are used in shared-nothing and shared-disk architectures. When a fast bus is available, we can modify this approach by allowing the processors to access the buffer of other processors. For such an approach, a processor has to know where the requested page is stored. Using a virtual shared memory architecture, a *(virtual) global buffer* is easy to implement: The global buffer consists of the sum of the local buffers. The access to a page in the global buffer is directed by the manager of the virtual shared memory. The only difference between a processor accessing its own buffer and accessing the buffer of another processor concerns the access time: the access to the own buffer is by a factor of about 10 times faster (see table 2 in section 4). For other parallel architectures without (virtual) shared memory, the problem of implementing a global buffer can be solved by using address tables for the local buffers and remote procedure calls.

The advantage of a global buffer is that a page occurs at most once in one of the local buffers. Thus, the number of disk accesses is lower compared to the case when every processor organizes its local buffer independently. However, a global buffer needs the implementation of locking mechanisms for a synchronization between the processes. Moreover, the communication on the bus increases since an access to a page found in the buffer almost always requires a transfer on the communication network whereas an access to the local buffer has no impact on the network. The communication is however reduced by the path buffers of the R*-trees which are stored in the local memory of the processors. Nevertheless, the increased communication on the bus may compensate the benefits of the global buffer.

3.3 Task Assignment

The goal of the static range assignment presented in section 3.1 is to keep those pages in the local buffer which are spatially close to each other. This strategy cannot be maintained anymore for a global buffer since the relevant pages of a processor are distributed among all local buffers: Instead of assigning tasks with spatially adjacent pages to one processor, these tasks should be distributed over different processors in order to process them simultaneously. Since the processors receive spatially adjacent pairs of MBRs, this strategy increases the probability that different processors require the same page at almost the same time. Therefore, such a simultaneous processing of subtrees increases the probability that processors read the required pages from the global buffer instead of reading them from disk. The new strategy proceeds as follows: we sort the *m* intersecting pairs of MBRs in the roots of the participating R*-trees again according to the local plane-sweep order. Instead of assigning adjacent subtrees, we now assign them in a round-robin fashion according to the plane-sweep order to the processors. Correspondingly, this task assignment is called *static round-robin assignment*. It is illustrated by an example in Figure 3..

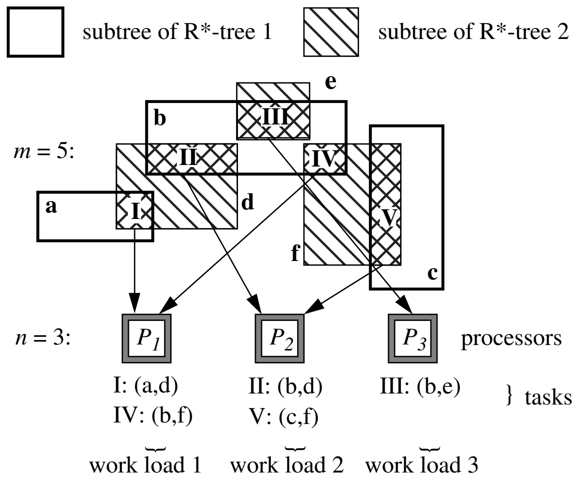


Figure 3: Example for the Static Round-Robin Assignment.

In order to distribute the load more evenly on the processors, we present another approach suitable for global buffers. This is based on giving up the consecutive execution of the two last phases of parallel spatial join processing. Instead, these two phases are alternately performed: First, *n* tasks are assigned to the processors (recall that *n* denotes the number of processors). As soon as one processor has finished its task, the next task is requested. For this so-called *dynamic task assignment*, a small queue describing all remaining tasks is required. This *task queue* must be accessible by all processors. Figure 4 depicts an example for the dynamic task assignment.

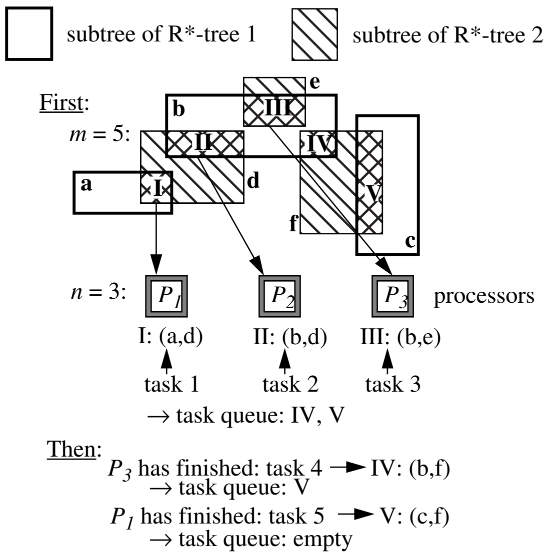


Figure 4: Example for the Dynamic Task Assignment.

3.4 Load Balancing Through Task Reassignment

One major disadvantage of the first approach presented in section 3.1, is the non-uniformly distributed load on the processors. Because the time for processing different pairs of subtrees varies, the time for processing the work loads is not the same. One solution to the problem would be to use a good estimation of the run time for each task and to modify the size of the work loads according to this estimation. However, this is difficult to achieve for spatial joins. Therefore, we follow a different approach which is called *task reassignment*: First, we process the spatial join as described in the sections before. When a processor has finished its tasks and there is no other task in the task queue, the processor offers its help to another operating processor. This operating processor divides its work load into two, where one part remains to be its own work load and where the other part is reassigned to the idle processor. Such a reassigned work load consists of one or more pairs of subtrees on the root level or on any other directory level of the R*-trees. Thereafter, both processors independently execute these new work loads - this is the main difference to the proposal of Shatdal and Naughton in [SN 93] where such processors have to work simultaneously on the same data structure (i.e. on the same hash table). However, due to the branch property of R*-trees, an independent processing can efficiently be supported.

The next time when one of the cooperating processors will be idle, help is given again to its "buddy" processor. This strategy is repeated until both of them are idle. Thereafter, they operate independently of each other and offer help to other processors. For the case of local buffers, this strategy keeps the number of the disk accesses low since the probability is high that pages in the buffer can still be used when a pair of processors exchange work loads more than once.

The first interesting question concerns the minimum size of the work load which is worth to be divided into two. The reassignment causes some algorithmic overhead and additional communications. When the size is too small, the improvements obtained from balancing the load can be compensated by the additional cost. A second question is, which of the processors receive the help of the idle processor? Shatdal and Naughton propose to choose an arbitrary processor. An alternative is to select the processor with the highest work load. For determining that processor, the idle processor needs additional information. Therefore, each processor reports the number *ns* of non-processed pairs of subtrees on the highest level *hl* where such pairs exist. The idle processor reads the current values of *hl* and *ns* for selecting the processor with the highest expected work load.

4 Evaluation

In order to evaluate the performance of the parallel spatial join, we investigate in this section several join algorithms based on the concepts presented in section 3.

4.1 Test Data

The maps used in our experiments are obtained from files of the US Bureau of the Census [Bur 89] describing some Californian counties. *map 1* consists of 131,443 streets whereas *map 2* represents administrative boundaries, rivers and railway tracks. The second map consists of 127,312 objects. The MBRs of the objects from each map were organized by an R*-tree with a page size of 4 KB. For the representation of an entry in a directory page, 40 bytes are used and for an entry in a data page, 156 bytes are reserved (including the MBR and a pointer to the exact object representation). Table 1 gives an overview of the main characteristics of the R*-trees. *m* denotes the number of pairs of intersecting MBRs stored in the root pages.

	tree1	*tree2*
height	3	3
number of data entries	131,443	127,312
number of data pages	6,968	6,778
number of directory pages	95	92
m (number of tasks)	404	404

Table 1: Parameters of the R*-trees

4.2 Test Environment

The experiments were performed on a multiprocessor machine with a virtual shared memory: the KSR1 of the Kendall Square Research Corporation. At most 24 processors were available for out tests. During the experiments, each processor was completely available for computing the spatial join and the bus of the KSR1 was free from other communications. Table 2 shows the most important parameters of the KSR1 concerning the memory.

memory	size of address space	transfer unit (in bytes)	band width (in MB/sec)	latency (in sec^{-6})
cache	256 KB	64	64	0.1
main memory	32 MB	128	40	1.2
main memory of other processors	768 MB	128	32	9

Table 2: Parameters of the KSR1 Concerning the Memory.

Because we were not able to control the distribution of the R*-tree nodes over the disks of the existing disk array, we decided to use a simulated disk array: Each page of an R*-tree was assigned to a disk by using its page number and a modulo function, i.e. spatial aspects have no impact on the selection of the disk where the page is stored. In the following, we assume an average seek time of 9 msec, an average latency time of 6 msec and a transfer time for one page (i.e. for 4 KB) of 1 msec. These parameters are typical values for current disks and result in 16 msec for reading a page. Moreover, the exact geometry is clustered on disk as described in [BK 94]. Consequently, there is a one-to-one relationship between a data page and the cluster where the exact geometry representations of the entries in the data page are stored. Thus, a data page access includes the access to the corresponding cluster. For a cluster of 26 KB (the average size in our experiments), the time for such an access is 37.5 msec.

In order to control the time necessary for testing the exact geometry of the objects for intersection, we replaced this test by waiting periods whose lengths depend on the degree of overlap between the corresponding MBRs. The average time to test one pair of objects is 10 msec; it varies between 2 msec and 18 msec depending on the degree of overlap. Experiments with real data have confirmed this approach assuming a plane-sweep algorithm used for the intersection test [BKSS 94].

On each processor, we provided an LRU-buffer which was implemented according to the description in [GR 93]. In the following, the size of these buffers is expressed by the number of R*-tree pages that can be stored in the buffer. Note that we need proportionally more memory for buffering the exact geometry of the objects. The joins start with cold buffers, i.e. they are empty at the beginning.

4.3 Investigation of the Buffer Organization and the Task Assignment

First, we investigated the use of different types of buffers and the different techniques for the task assignment. For this purpose, three variants of parallel spatial join processing were compared:

1.) local buffers with a static range assignment (***lsr***),
2.) a global buffer with a static round-robin assignment (***gsrr***), and
3.) a global buffer with a dynamic task assignment (***gd***).

These variants were investigated with LRU-buffers of a total size varying between 200 and 3,200 pages. The number n of processors used in these experiments was 8 and 24 with the same number of disks. A task reassignment was performed on the root level of the R*-trees.

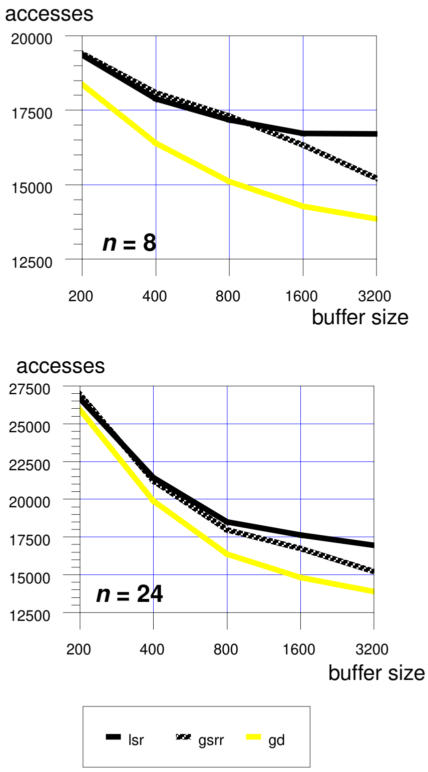


Figure 5: Disk accesses using 8 and 24 processors

Figure 5 depicts the total number of disk accesses as a function of the size of the LRU-buffer. For the total run time of all tasks which includes the CPU-time, the synchronization and the communication cost between the processors, we obtained comparable results. Note that the number of disc accesses is higher when the number of processors increase from 8 to 24. This is because the buffer space of a single processor decreases with an increasing number of processors.

Local buffers combined with a static range assignment (lsr) and the global buffer using a static round-robin assignment (gsrr) do not differ very much in the number of disk accesses. However, the global buffer profits more from using larger buffers than the local buffers. The results demonstrate that a global buffer with dynamically assigned tasks (gd) has a better performance than a global buffer using a static task assignment (gsrr). This is caused by the different run times for processing a pair of subtrees. The example depicted in Figure 6 illustrates this effect. As a consequence, pairs of spatially adjacent subtrees that should be processed at the same time, will be processed at different times using the static round-robin assignment. Thus, the number of disk accesses increases compared to the technique using dynamically assigned tasks where such differences can be balanced.

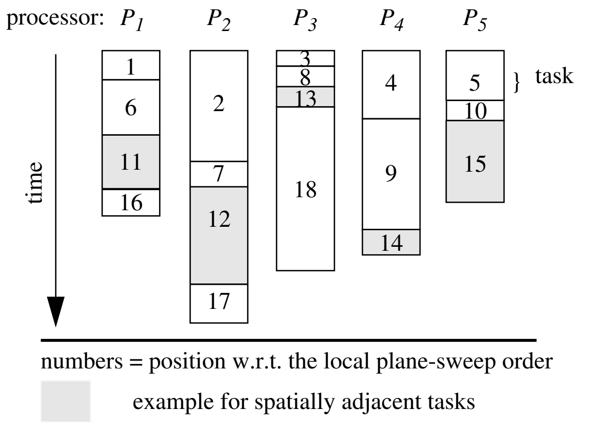


Figure 6: Example for Losing Spatial Adjacency.

4.4 Investigation of the Task Reassignment

Now, let us investigate the effect of the task reassignment to the performance of the parallel spatial join. Three variants were compared for this purpose:

1.) without a reassignment,
2.) with a reassignment on the level of the roots of the R*-trees,
3.) with a reassignment on all levels of the R*-tree directories.

These variants were compared using local buffers with a static range assignment (lsr), a global buffer with a static round-robin assignment (gsrr), and a global buffer with a dynamic task assignment (gd). The total size of the buffer is 800 pages. 8 processors and 8 disks were used in these experiments.

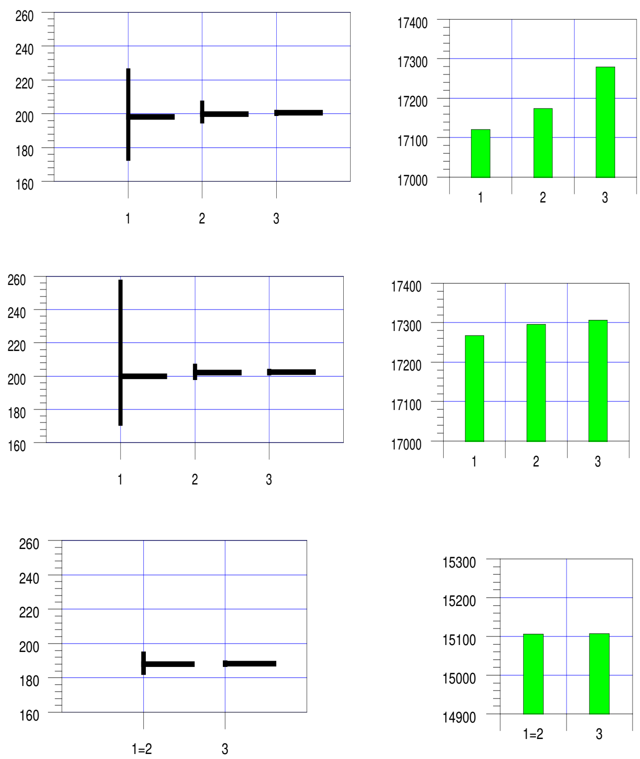


Figure 7: Performance with and without a task reassignment

The left diagrams of Figure 7 show the run time of the processor finishing first (lower end of the vertical line), of the processor finishing last (upper end of the vertical line), and the run time on average (horizontal line). The number of disk accesses is depicted in the right diagrams.

The results demonstrate that the reassignment minimizes the variation between the run times of the processors as well as the run time of the processor that has finished last. Especially, the difference between the variants 1 and 2 is considerable for the test series (lsr) and (gsrr). The total run time of all tasks is only slightly increased by the reassignment. The increase is not caused by an additional algorithmic cost of the reassignment which is at most 100 msec in our set of experiments. The reason is - especially for the test series using local buffers (lsr) - a higher number of disk accesses caused by the fact that a processor which has taken over some of the work load from another processor, often does not find the required pages in its buffer. Additionally, the reassignment is concentrated in the final phase of spatial join processing. As a result, waiting periods may occur.

Using a global buffer with dynamically assigned tasks (gd), a slightly different situation can be observed: By using the dynamic task assignment, a task reassignment on the root level is not necessary because the work load is requested task-by-task in this case. Consequently, the results of the variants 1 and 2 are the same and the decrease of the response time for completing the spatial join is smaller. The fact of a non-increasing number of disk accesses indicates again that this approach maintains spatial locality well.

- In the following experiment we investigate two strategies selecting the processor that receives help from the idle processor. In the test series a, the reassignment algorithm selects the processor with the most extensive work load. This strategy was also used in the experiments before. In test series b, an arbitrary processor is chosen for the task reassignment. This technique follows the proposal of [SN 93]. Our experiments showed that the overhead for determining the processor with the most extensive work load is completely negligible. Therefore, Figure 8 depicts only the number of disk accesses for the different test series. The number of processors is 8. For the test series using a local buffer, we can observe a small increase of the number of disk accesses when an arbitrary processor is chosen. The reason is an increased number of reassignments where the helping processor does not find the required pages in its buffer. When a global buffer is used, there is no difference between the different strategies for determining the processor to be helped.

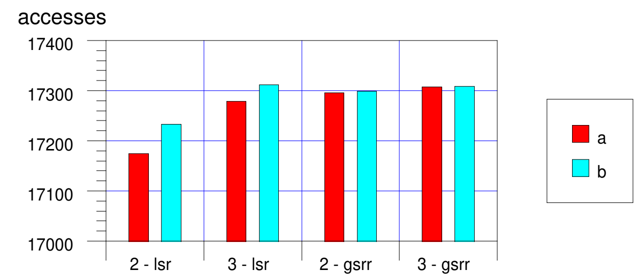


Figure 8: Comparison of different techniques for determining the processor to be helped.

4.5 Investigation of Response Time and Speed up

For the best variant of parallel spatial join processing - i.e. for the variant using a global buffer with a dynamic task assignment and a task reassignment on all levels of the R*-tree directory - we investigate now its response time *t(n)* and its speed up depending on the number of processors *n* used in the experiment. The *response time* is the wall-clock time between starting the spatial join and computing the last pair of intersecting objects; it is determined by the processor finishing last. The *speed up* for using *n* processors is measured by the quotient between the response time *t(1)* using 1 processor and the response time *t(n)*. In the ideal case, we want to decrease the response time *t(n)* by the factor *n* compared to the response time *t(1)*; in other words: the speed up *t(1) / t(n)* should be *n*. However, initialization periods, synchronization periods, and the communication between the processes generally prevent to obtain a linear speed up.

In the case of the spatial join investigated in our experiments, the initialization period is negligible compared to the other cost. Even in the worst case, it was smaller than 0.1% of the response time. The influence of the remaining factors - i.e. the communication (particularly in order to read pages located in the main memory of other processors) and the synchronization (especially at the disks) - will be examined in the following.

In the following experiment, the number of processors varies between 1 and 24. The total size of the buffer increases linearly with the number of processors: for 1 processor 100 pages of the R*-tree can be stored in the buffer and for 24 processors the buffer capacity is 2,400 pages. For the number *d* of disks, we run three test series: 1.) 1 disk ($d = 1$), 2.) 8 disks ($d = 8$), and 3.) the number of processors and of disks are the same ($d = n$). For these test series, Figure 9 shows the response time depending on the number of processors used.

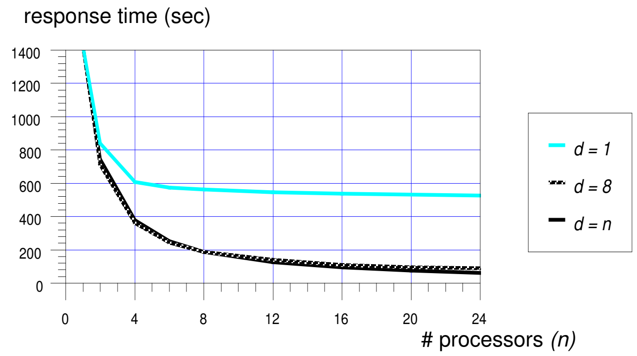


Figure 9: Response Time varying in the number of processors and disks.

In the experiment using only 1 disk, the secondary storage becomes the bottleneck. For 4 or more processors, the response time stays at about 550 sec. Using 8 or *n* disks, the response time decreases when the number of processors increases. However for more than 10 processors, the decrease of the response time is smaller in the case of 8 disks compared to the variant of *n* disks where a response time of 62.8 sec can be obtained using 24 processors. The speed up, depicted in Figure 10, demonstrates this effect more clearly.

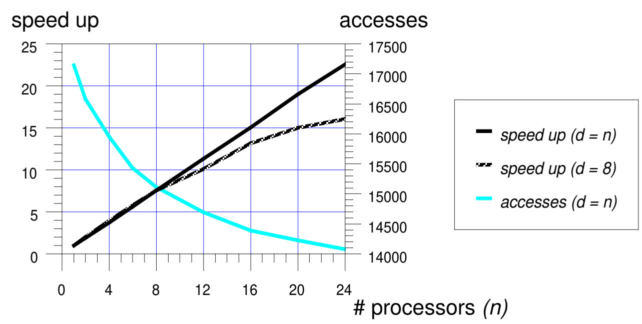


Figure 10: Speed up and disk accesses varying in the number of processors.

For the case of 8 disks, the increase of the speed up drops when more than 10 processors are used. In contrast to this observation, the speed up is linear for the case of n disks. The speed up for $n = d = 24$ is 22.6 which is a very good result. One reason for this high speed up is the good performance of the bus of the KSR1. An additional explanation is given by the number of disk accesses (also depicted in Figure 10): Using a growing global buffer, the number of disk accesses decreases and compensates for some of the additional communication and synchronization cost

The total run time of all tasks was about 7% higher for 4 processors than for 1 processor in our experiments. Using more than 4 processors, this time even falls with an increasing number of processors. Therefore, we expect that there will be only a modest decline of the *throughput* by using the parallel spatial join with a large number of processors.

4.6 Summary

The major results of our experimental investigations are as follows:

- The global buffer combined with a dynamic task assignment is the most efficient assignment technique according to our tests. This technique preserves spatial locality. Consequently, most of the page requests can be satisfied by the LRU-buffer.
- By a task reassignment on all levels of the R*-tree directories, the load is balanced and the response time is additionally shortened. For local buffers, the selection of the processor with the most extensive work load shows the best performance. Otherwise, an arbitrary processor can be chosen for the reassignment.
- Using only one disk, the speed up will not improve for more than 4 processors computing the spatial join.
- We achieve a linear speed up close to n when the data is stored on n disks (e.g. the speed up is 22.6 for 24 processors and disks).
- Using the parallel spatial join with a large number of processors, has almost no influence on the total run time of all tasks and hence, we also expect almost no decline of the throughput.

5 Conclusions

The spatial join is among the most important operations of a spatial database system. Although the run time of sequential spatial join processing has considerably been improved over the last few years, the spatial join is still a very expensive operation. Therefore, the question arises whether parallelism is a cost-effective approach for improving the efficiency of a spatial join.

In this paper, we examined different approaches for a parallel spatial join on a so-called shared-virtual-memory architecture (SVM). The selection of the SVM-architecture was also motivated by the increasing transfer rates of networks. We suppose that shared-nothing architectures available soon will be comparable to a state-of-the-art SVM-architectures respect to their performance.

We started with a first approach where the spatial join was executed in three phases: task creation, task assignment and (parallel) task execution. The most important characteristics of this approach are a task assignment according to the local plane-sweep order and the avoidance of communication between the processors while the join is processed. In order to reduce the response time, we introduced additional techniques concerning the buffer organization, the task assignment and the task reassignment. Using the same number n of processors and of disks, we achieved for the most efficient algorithm a linear speed up close to n (e.g. 22.6 for 24 processors). The total run time for all tasks was only slightly increased.

In our future work, we are particularly interested in a distributed spatial join processing using a shared-nothing architecture. We plan investigations on workstation clusters that are connected through a fast interconnection network, e.g. ATM-switches. In contrast to the SVM-model, in a shared-nothing architecture the assignment of the data to the different disks is of special interest. Furthermore, we want to integrate the spatial join in a larger framework for parallel spatial query processing where also other operations such as neighbor and window queries are efficiently supported.

References

[BG 90]Becker L., Gütimg R. H.: '*Rule-Based Optimization and Query Processing in an Extendsible Geometric Database System*', ACM Trans. on Database Systems, 17, 1992, 247-303.

[BHF 93]Becker L., Hinrichs K., Finke U.: *'A New Algorithm for Computing Joins with Grid Files'*, Proc. 9th Int. Conf. on Data Engineering, Vienna, Austria, 1993, pp. 190-197.

[BK 94]Brinkhoff T., Kriegel H.-P.: *'The Impact of Global Clustering on Spatial Database Systems'*, Proc. 20th Int. Conf. on Very Large Databases, Santiago, Chile, 1994, pp. 168-179.

[BKS 93]Brinkhoff T., Kriegel H.-P., Seeger B.: *'Efficient Processing of Spatial Joins Using R-trees'*, Proc. ACM SIGMOD Int. Conf. on Management of Data, Washington, DC, 1993, pp. 237-246.

[BKSS 90]Beckmann N., Kriegel H.-P., Schneider R., Seeger B.: *'The R*-tree: An Efficient and Robust Access Method for Points and Rectangles',* Proc. ACM SIGMOD Int. Conf. on Management of Data, Atlantic City, NJ, 1990, pp. 322-331.

[BKSS 94]Brinkhoff T., Kriegel H.-P., Schneider R., Seeger B.: *'Multi-Step Processing of Spatial Joins'*, Proc. ACM SIGMOD Int. Conf. on Management of Data, Minneapolis, MN, 1994, pp. 197-208.

[Bur 89]Bureau of the Census: *'TIGER/Line Precensus Files, 1990 Technical Documentation'*, Washington, DC, 1989.

[DeW 94]DeWitt D.J., Kabra N., Lou J., Patel J.M., Yu J.-B.: *'Client-Server Paradise'*, Proc. 20th Int. Conf. on Very Large Databases, Santiago, Chile, 1994, pp. 558-569.

[[GR 93]Gray J., Reuter A.: *'Transaction Processing: Concepts and Techniques'*, Morgan Kaufmann, 1993.

[Gra 93]Graefe G.: *'Query Evaluation Techniques for Large Databases'*, ACM Computing Surveys, Vol. 25, No. 2, 1993, pp. 73-170.

[Gün 93]Günther, O.: *'Efficient Computation of Spatial Joins'*, Proc. 9th Int. Conf. on Data Engineering, Vienna, Austria, 1993, pp. 50-59.

[Gut 84]Guttman A.: *'R-trees: A Dynamic Index Structure for Spatial Searching',* Proc. ACM SIGMOD Int. Conf. on Management of Data, Boston, MA, 1984, pp. 47-57.

[[HS 94b]Hoel E., Samet H.: *'Data-Parallel Spatial Join Algorithms'*, Proc. Int. Conf. on Parallel Processing, St. Charles, IL, 1994.

[HS 94c]Hoel E., Samet H.: *'Performance and Data-Parallel Spatial Operations'*, Proc. 20th Int. Conf. on Very Large Databases, Santiago, Chile, 1994, pp. 156-167.

[LR 94]Lo M.-L., Ravishankar C.V.: *'Spatial Joins Using Seeded Trees'*, Proc. ACM SIGMOD Int. Conf. on Management of Data, Minneapolis, MN, 1994, 209-220.

[Mon 93]Montage Software, Inc.: *'The Montage SPATIAL DataBlade™'*, 1993.

[NHS 84]Nievergelt J., Hinterberger H., Sevcik K.C.: *'The Grid File: An Adaptable, Symmetric Multikey File Structure',* ACM Trans. on Database Systems, Vol. 9, No. 1, 1984, pp. 38-71.

[OM 88]Orenstein J.A., Manola F.A.: *'PROBE Spatial Data Modeling and Query Processing in an Image Database Application'*, IEEE Trans. on Software Engineering, Vol. 14, No. 5, 1988, pp. 611-629.

[PS 85]Preparata F.P., Shamos M.I.: *'Computational Geometry'*, Springer, 1985.

[SN 93]Shatdal A., Naughton J.F.: *'Using Shared Virtual Memory for Parallel Processing'*, Proc. ACM SIGMOD Int. Conf. on Management of Data, Washington, DC, 1993, pp. 119-128.