A Parallel Functional Implementation of Range Queries

Pieter H. Hartel Michiel H.M. Smid* Leen Torenvliet
Willem G. Vree

University of Amsterdam, Departments of Mathematics and Computer Science
Plantage Muidergracht 24, 1018 TV Amsterdam

Abstract

We present two parallel versions of the *range tree* data structure and the corresponding range query algorithm. These structures were implemented for a modest number of points in a parallel functional language. Performance data for the queries were obtained by simulation of a parallel reduction machine. The resulting amount of parallelism is encouraging and gives hope for a substantial gain in efficiency by parallel implementation of query algorithms in large databases.

1 Introduction

The orthogonal range searching problem has received considerable attention, and many data structures have been designed to solve this problem, obtaining various trade-offs between the complexity measures.

Knuth [7] was the first who mentioned the problem:

> Let S be a set of points in d-dimensional space and let $([lo_1 : up_1], [lo_2 : up_2], \ldots, [lo_d : up_d])$ be a hyperrectangle. The *orthogonal range searching problem* asks for all points $p = (p_1, p_2, \ldots, p_d)$ in S such that $lo_1 \leq p_1 \leq up_1, lo_2 \leq p_2 \leq up_2, \ldots, lo_d \leq p_d \leq up_d$.

The problem arises in many applications. As an example, consider a database of major sites in the city of Amsterdam. Map coordinates run from 1 to 100 in east–west direction, and from 1 to 300 in north–south direction. One might seek a list of all those sites in the city center with latitude between 40 and 50 and longitude between 120 and 150.

*This author was supported by the Netherlands Organization for Scientific Research (NWO).

Several data structures have been proposed to solve the problem. For a survey of the state of the art in 1979 concerning the range searching problem, see Bentley and Friedman [3]. More recent data structures, besides, range trees can be found in Edelsbrunner [5], Chazelle [4] and Overmars [9].

The data structure we consider for solving the range searching problem is the *range tree*, due to Bentley [1,2] and Lueker [8]. A d-dimensional range tree for a set of n points has size $O(n(\log n)^{d-1})$, and on a sequential computer a range query can be solved in $O((\log n)^d + t)$ time, where t is the number of reported answers. We shall give simple parallel implementations of such a range tree and queries on the tree. In these implementations, $O(\log n)$ processors are needed, each containing a part of the data structure of size $O(n(\log n)^{d-2})$ in its own local memory.

The results show a new application of the notion of partitioning a data structure into parts, as initiated in [10,11,12,13]. In these papers, a range tree is partitioned in order to maintain it efficiently in secondary memory. In the present paper, we also partition the data structure, and the parts are distributed among the processors.

The paper is organized as follows. In Section 2, we introduce range trees, and we give the algorithm to solve range queries in these trees. In Section 3, we give the two different partition schemes for range trees that we have implemented on a parallel machine. In Section 4 we show how range trees and the sequential query algorithm have been implemented in a functional programming language. Section 5 describes our parallel reduction architecture and the way in which the partitioned range trees are mapped onto a network of processors. Section 7 presents simulated performance results that have been obtained by applying a few queries to a small database. The last section provides a summary of our main results and outlines future research.

2 The range tree data structure

First we consider the one-dimensional case. Let S be a set of n real numbers. We store the elements of S in sorted order in the leaves of a perfectly balanced binary search tree. Let $[lo : up]$ be a query interval. Then we have to report all elements p in S such that $lo \leq p \leq up$. The algorithm is as follows. We search in the binary tree with both lo and up. Assume without of generality that $lo < up$. We have to report the elements in all leaves that lie between the paths to lo and up. (In general, lo will not be stored in the tree. We nevertheless speak about "the search path to lo". This is the path to the smallest element in the tree that is at least equal to lo. Similarly, "the search path to up" refers to the path to the largest element in the tree that is at most equal to up.) Let u be that node in the tree for which the path to lo resp. up proceeds to the left resp. right son of u. Then for each node $v \neq u$ on the path from u to lo, for which the search proceeds to the left son of v, we report the elements in all leaves in the right subtree of v. Similarly, for each node $w \neq u$ on the path from u to up, for which the search proceeds to the right son of w, we report the elements in the leaves of the left subtree of w.

On a sequential processor, this data structure has size $O(n)$, and can be built in $O(n \log n)$ time. A range query can be solved in $O(\log n + t)$ time, where t is the number of reported answers.

Using these one-dimensional binary trees, we build d-dimensional range trees (see [1,2,8]).

Definition 1 Let S be a set of points in d-dimensional space. A *d-dimensional range tree*, representing the set S, consists of the following. For $d = 1$, it is a perfectly balanced binary search tree, as described above. Let $d \geq 2$.

1. There is a perfectly balanced binary tree, called the *main tree*, containing the points of S in its leaves, ordered according to their first coordinate. Internal nodes of this main tree contain information to guide searches.
2. For any internal node v of this main tree, let S_v be the set of points that are in the subtree of v. Then node v contains a (pointer to) an *associated structure*, which is a $(d-1)$-dimensional range tree for the set S_v, taking only the last $d-1$ coordinates into account.

See Figure 1 for a pictorial representation of a 2-dimensional range tree. The main tree T contains the points of S, ordered according to their x-coordinate. Each node v in the main tree contains a pointer to a binary tree that stores the points of S_v in its leaves, ordered according to their y-coordinate.

Range queries are solved as follows. Let $([lo_1 : up_1], \ldots, [lo_d : up_d])$ be a query rectangle, where $d \geq 2$. Then we begin by searching with both lo_1 and up_1 in the main tree. Assume without of generality that $lo_1 < up_1$. Let u be that node in the main tree for which lo_1 lies in the left subtree of u, and up_1 lies in the right subtree of u. Then we have to perform a range query on the last $d-1$ coordinates on all points that lie between lo_1 and up_1 in the main tree. It is sufficient to perform $(d-1)$-dimensional range queries in the associated structure of the right son of each node $v \neq u$ on the path from u to lo_1 for which the search proceeds to the left son of v, and in the associated structure of the left son of each node $w \neq u$ on the path from u to up_1 for which the search proceeds to the right son of w. If these associated structures are one-dimensional structures, we use the query algorithm given above, otherwise we use the same algorithm recursively.

In this query algorithm, each point in the query rectangle is reported exactly once. The number of associated structures in which $(d-1)$-dimensional range queries are performed is at most twice the height of the main tree. In the one-dimensional associated structures, we spend an amount of time that is bounded by

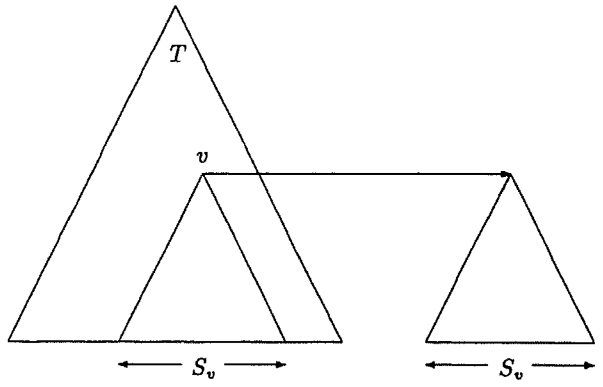


Figure 1: A two-dimensional range tree

$O(\log n + t')$, where t' is the number of answers found in that associated structure. Since the height of the main tree is bounded by $O(\log n)$, it follows by induction on d, that—on a sequential computer—a query in a d-dimensional range tree takes $O((\log n)^d + t)$ time, where t is the total number of reported answers.

Let $S(n,d)$ be the size of a d-dimensional range tree for a set of n points. We prove by induction on the dimension d, that $S(n,d) = O(n(\log n)^{d-1})$. This is clearly true for $d = 1$. So let $d \geq 2$, and suppose the claim is true for $d-1$. Consider a fixed level of the main tree, and let $v_1, \ldots, v_m$ be the nodes on this level. Each node v_i contains an associated structure for a subset S_i of S. By the induction hypothesis, each such associated structure has a size that is bounded by $O(|S_i|(\log|S_i|)^{d-2}) = O(|S_i|(\log n)^{d-2})$. The sets S_i form a partition of S, hence all associated structures on this level together have a size that is bounded by

$$O\left(\sum_{i=1}^{m} |S_i|(\log n)^{d-2}\right) = O\left(n(\log n)^{d-2}\right).$$

Since the main tree contains $O(\log n)$ levels, it follows that the entire range tree has size $O(n(\log n)^{d-1})$, which proves the claim.

Using a method called *presorting*, the range tree can be built in $O(n(\log n)^{d-1})$ sequential time, see [2].

Note 1: In the range tree, several associated structures are never queried. For example, the associated structures of the root of the main tree and its two sons are never used. Also, the associated structures of all nodes in the leftmost path in the main tree (except the leaves) are never queried. Similarly for the rightmost path in the main tree. Therefore, these associated structures can be omitted from the

tree. Then, the size of the data structure remains asymptotically the same. The constant factor will, however, be decreased. (In practice, constant factors are, of course, very important.)

Note 2: In the query algorithm lo_i may be $-\infty$ and/or up_i may be $+\infty$. In the sequential case this would be treated as a degenerate query. However in the parallel case it is advantageous *not* to do so, since more processors will be involved in the query.

3 Solving range queries in parallel

If we want to use a range tree to solve range queries on a parallel computer, we partition the structure into parts, and we distribute these parts among a number of processors. The complexity of a partition will be expressed by

1. the number of processors,
2. the amount of space each processor needs in its own local memory,
3. the time needed to solve a range query,
4. the amount of communication needed to solve a query.

For information about partitions, see [10,11,12,13]. In these papers partitions appear in a different context. There, an efficient partition is one in which each query visits only a small number of parts. In our case, however, this means that only few processors are used to answer the query, or equivalently, most of the processors do nothing.

We shall consider two different partitions. In both cases, we need $O(\log n)$ processors, each containing an amount of $O(n(\log n)^{d-2})$ data. (Note that a range tree has size $O(n(\log n)^{d-1})$.)

3.1 The slice partition

In our first partition we store the main tree T in processor P_m, and for $l = 0, 1, \ldots, \log n$, we store all associated structures of the nodes at level l in processor P_l. This gives us $O(\log n)$ processors, each having an amount of $O(n(\log n)^{d-2})$ data.

The query algorithm is as follows. Let $([lo_1 : up_1], \ldots, [lo_d : up_d])$ be a query rectangle.

1. Processor P_m determines—from top to bottom—all nodes v in T in whose associated structure a $(d-1)$-dimensional range query has to be performed.

2. For each node v found in step 1, say at level $l(v)$, command processor $P_{l(v)}$ to perform a $(d-1)$-dimensional range query in the associated structure of v, with query interval $([lo_2 : up_2], \ldots, [lo_d : up_d])$.

In step 1 of this algorithm, at each level in T at most two nodes are found. Hence each processoı P_l has tc perform at most two $(d-1)$-dimensional range queries. Since all these processors can do their work simultaneously, the time required to solve the entire query is bounded by $O((\log n)^{d-1} + t')$, where t' is the maximal number of points found in any associated structure. Note that in the ideal case almost all processors are involved in the query algorithm. In that case, the value of t' will—in general—be much smaller than the total number of reported answers.

We now give a slight variation of the implementation of the slice partition. For each $l = 0, 1, 2, \ldots, \log n$, we store all nodes of the main tree that are positioned at the levels $0, 1, \ldots, l$, together with the associated structures of the nodes at level l, in processor P_l. Again this gives us $O(\log n)$ processors, each containing an amount of $O(n(\log n)^{d-2})$ data.

Let $([lo_1 : up_1], \ldots, [lo_d : up_d])$ be a query rectangle. Then all processors start working simultaneously in the following way. Processor P_l starts searching in the upper l levels of the main tree for lo_1 and up_1. If the search paths are the same in the highest $l-1$ levels, we are done. Otherwise, let u be the node—at a level $\leq l-2$—for which the search for lo_1 resp. up_1 proceeds to the left resp. right son of u. Then we follow the search for lo_1 in the left subtree of u. If this search proceeds at the node v at level $l-1$ to the left son, this processor performs a $(d-1)$-dimensional range query in the associated structure of the right son of v. Similarly for the search for up_1.

Again the query time is bounded by $O((\log n)^{d-1} + t')$, where t' is the maximal number of points found in any associated structure. In this solution, all processors perform the same algorithm, and no communication among processors is needed (all processors can read the query rectangle from a common input).

3.2 The wedge partition

We first sketch how the range tree is distributed among the processors. We want to give each processor an amount of $O(n(\log n)^{d-2})$ data. The associated structure of the root of the main tree T has size $O(n(\log n)^{d-2})$, and hence we store it in one processor. Now consider the two sons v and w of the root of the main tree. Look at the subtree consisting of v and its two sons. It takes, together with its associated structures, $O(n(\log n)^{d-2})$ storage and, hence, can be stored in one processor. Similarly for w. We have now removed three levels of T; so we are left with 8 sons. For each of these sons u, we make a part consisting of the tree with root u, of depth 8. This subtree, with its associated structures, uses $O(n(\log n)^{d-2})$

space, and is stored in one processor. We have now removed 11 levels. So we are left with 2^{11} sons. For each son, we take a subtree of depth 2^{11}, with associated structures, which takes $O(n(\log n)^{d-2})$ storage, and put it in one processor. Next we are left with $2^{2^{11}+11}$ sons, etc. Note that a node on level i represents $\Theta(n/2^i)$ points.

We describe the above more precisely. We cut the main tree T at level $\lfloor \log\log n \rfloor$. Each node on level $\lfloor \log\log n \rfloor$ is the root of a d-dimensional range tree for $\Theta(n/\log n)$ points. Hence such a range tree has size

$$O\left(\frac{n}{\log n}\left(\log \frac{n}{\log n}\right)^{d-1}\right) = O(n(\log n)^{d-2}).$$

We store each such tree, together with its associated structures in its own processor. This gives us $O(\log n)$ processors, each having an amount of $O(n(\log)^{d-2})$ data. Let T' be the tree, consisting of the highest $\lfloor \log\log n \rfloor$ levels of T. We store T' in one processor. Let $a_0 = 0$, and $a_{k+1} = 2^{a_k} + a_k$ for $k \geq 0$. Let $m = \min\{i \geq 0 | a_i > \log\log n\}$. Now the associated structures of nodes of T' are distributed as follows. For each k, $0 \leq k \leq m-1$, there are 2^{a_k} parts. Each such part is a subtree of T', together with its associated structures, having its root at level a_k, of depth 2^{a_k}. Each part has size $O(n(\log n)^{d-2})$. We store all the associated structures of one part in one processor. The number of parts in which T' is partitioned, and hence the number of processors we need in this way, is

$$\sum_{k=0}^{m-1} 2^{a_k} = O(2^{a_{m-1}}) = O(2^{\log\log n}) = O(\log n).$$

So again we have $O(\log n)$ processors, each containing $O(n(\log n)^{d-2})$ data.

The query algorithm is similar to the first one in Section 3.1. Now the processor containing T' starts working, and it commands the appropriate processors to perform $(d-1)$-dimensional range queries in the appropriate associated structures. It can easily be shown that a range query takes in the worst case $O((\log n)^d + t')$ time, where t' is the maximal number of points reported by any processor. (In a d-dimensional range tree having its root on level $\lfloor \log\log n \rfloor$ in T, it takes $O((\log \frac{n}{\log n})^d + t') = O((\log n)^d + t')$ time to answer a query.) Note that this is the same amount of time as we need in the sequential algorithm of Section 2. In this parallel version, however, the constant factor in the $O((\log n)^d)$ term will be smaller. Also, often the value of t' will be much smaller than the total number of reported answers.

The problem here is how much communication is needed. Let $\log^* n$ be the number of times we have to take the logarithm of n until the result is at most one. It is shown in [11] that to answer a query at most $4\log^* n + O(1)$ processors are needed, where the $O(1)$ factor is a small constant. Also, the multiplicative factor 4 is very pessimistic. Finally, for all practical values of n, we have $\log^* n \leq 5$. (In fact, $\log^* n \leq 5$ for all $n \leq 2^{65536} \approx 10^{19661}$.) Hence, it seems that here very little communication is needed.

4 Functional implementation of range queries

We will describe the parallel implementation of the range query algorithm in two steps. In the next section we present the implementation of the sequential search algorithm. This implementation is used unmodified by the parallel range queries that we will describe in subsequent sections. We use the functional programming language SASL [14]. The essential properties of the language will be explained as we proceed.

4.1 Sequential range query

The implementation of the range tree search algorithm requires two data structures: one to represent the range tree and the other to represent the query. The search process is implemented by a set of three recursive functions and two auxiliary functions. Before discussing these functions we introduce the data structures.

A query is represented by a list of lower and upper bounds $(lo_1 : up_1 : lo_2 : \ldots : ())$ on the intervals of the hyperrectangle. A query on a d-dimensional range tree requires a $2 \times d$ element bounds list. In SASL, lists are constructed from primitive data objects or other lists by the infix operator colon (:). Primitive data objects in SASL include numbers, booleans and the symbol (), which is to be pronounced as "nil". A construction whose last element is nil is usually called a list, one that is not terminated by nil is called a tuple. A d-dimensional range tree for a database with n records consists of a balanced binary tree with n leaves. A node is represented by a 4-tuple (*key* : *left* : *right* : *associated*). The *left* and *right* fields represent the left and right subtrees of the node; the *key* field gives the smallest key value of any node in the right subtree and the *associated* field represents the $(d-1)$-dimensional range tree for the structure that is associated with the current node. If the dimension of the range tree is one, the associated structures represent database record(s). Data base records with the same d-dimensional key are stored as separate nodes.

The functions that operate on a range tree and a query are shown in Figure 2. A SASL program has the form of a set of recursive equations. Function application is denoted by mere juxtaposition and parentheses serve to delineate subexpressions. Function application has higher priority than any other operator. Conditionals have the form *condition* → *then part; else part.*

A single function may have more than one defining equation. The appropriate equation is selected based on matching actual arguments to patterns specified by formal arguments. Patterns within one equation are tried from left to right

```
1.  search  ()             query        =  ()
2.  search  node           ()           =  node
3.  search  (k : l : r : a) query       =  k < lo     →  search  r  query ;
4.                                         up < k     →  search  l  query ;
5.                                         leaf l r   →  search  a  rest ;
6.                                         left l lo rest : right r up rest
..                                         WHERE
..                                         lo : up : rest = query
..
..  left    ()             lo    rest   =  ()
..  left    (k : l : r : a) lo    rest   =  k < lo     →  left    r  lo      rest ;
..                                         leaf l r   →  search  a  rest ;
..                                         left l lo rest : search (assoc r) rest
..
..  right   ()             up    rest   =  ()
..  right   (k : l : r : a) up    rest   =  up < k     →  right   l  up      rest ;
..                                         leaf l r   →  search  a  rest ;
..                                         search (assoc l) rest : right r up rest
..
..  assoc   ()                          =  ()
..  assoc   (k : l : r : a)             =  a
..  leaf    ()             ()           =  TRUE
..  leaf    x              y            =  FALSE
```

Figure 2: Range tree search algorithm

and equations are tried from top to bottom. The elements that may occur in patterns are list constructors, constants and variables. Constants and constructors must occur as written. A variable matches anything and is bound to the data it matches. Any expression may be adorned with a WHERE clause to name commonly occurring subexpressions.

Once the data structures have been established, a range query is performed as follows. The function *search* is initially applied to (the root of) a d-dimensional range tree and a query with $2 \times d$ bounds. Line 1 of the defining equation of *search* specifies that if the root is nil, the result should also be nil. If the query is nil, the current *node* is taken to represent database record(s), because there are no more dimensions to query (line 2). If neither case applies, we arrive at line 3 of the defining equation of *search*. Here the components of the root are isolated by pattern matching (i.e. k is bound to the key of the node, l to the left subtree etc.), such that by comparing the current key (k) to the lower (lo) and upper (up) bound of the current query interval we may decide whether to continue searching to the right (line 3) to the left (line 4) or in both directions. In the first two cases the interval is either entirely to the left or entirely to the right of the key. In the latter case the lower bound is to the left of the current key and the upper bound to the right. Hence the current node satisfies the constraints of the current query interval. The searching process now splits into two separate paths to be handled by the functions *left* and *right* (line 6). These functions search the path to the left respectively the right of the bifurcation point found by *search*. Each requires one bound of the current query interval only.

Leafs have to be handled specially (line 5), because whenever the key of a leaf falls within the current query interval, its associated structure must be searched. Searching associated structures of lower dimension is performed by the function *search* as before, but with two fewer elements in the list *rest* that will serve as the new query. If the current dimension is the last to query (when *rest* is nil), the associated structure of that leaf will be included in the database records to be reported.

The auxiliary function *assoc* extracts the associated structure from a node. The function *leaf* is a predicate that determines whether the current node is a leaf. Only leaves have nil values for both their left and right descendants.

5 Parallel implementation of reduction

Before we can describe the parallel versions of the range query algorithm we need to introduce the concept of our parallel reduction machine. The architecture that we use is a collection of reduction processors, each equipped with its own local memory and interconnected by a (fast) data communication network [6]. A fundamental property of this architecture is that the access time of a processor to its own memory is much shorter than the access time to the other memories. These two access times may differ by one to two orders of magnitude. To account for this property we have decided not to support a global address space. A special mechanism is provided to transport a subexpression from one local memory to another. Our parallel implementation of SASL is based on lazy graph reduction. Lazy evaluation means that arguments to a function are only computed at the time they are needed by the computation. To perform lazy evaluation efficiently, graph reduction shares unevaluated expressions as much as possible.

5.1 Job based parallel reduction

The architectural features discussed above limit the kind of expressions that can be efficiently reduced in parallel. Only certain coarse grain parts of the program that we call "jobs" are allowed to be copied to another processor for parallel evaluation.

A job is an expression with the following properties:

$$\begin{array}{lcl}
G & & <job_1><job_2> \; \dots \; <job_p> \\
\text{WHERE} & & \\
job_1 & = & F_1 \;\; a_{11} \;\; a_{12} \;\; \dots \;\; a_{1m_1} \\
\dots & & \\
job_p & = & F_p \;\; a_{11} \;\; a_{12} \;\; \dots \;\; a_{1m_p}
\end{array}$$

Figure 3: Parallel job annotation

1. it is self-contained (i.e., a subgraph that does not contain references to other points of the graph);
2. its evaluation is needed to compute the final result;
3. the cost to evaluate the expression outweighs the cost involved in transportation.

Job property 3 guarantees that parallel execution of a set of jobs will be faster than their sequential execution. Property 2 makes sure that the result of a job is essentially used in the whole computation and so no actual processing will be wasted. Finally property 1 allows jobs to be evaluated in a separate address space and avoids the need for global garbage collection.

In our implementation of parallel reduction, expressions that classify as a job have to be marked by the programmer, using a special annotation (job brackets) to be presented in the following section. The restriction of parallel reduction to jobs defines a minimum granularity on which the data communication of the architecture can be based. Overhead incurred by transmission protocols can be spread over the cost of transporting a whole subgraph. A disadvantage of job based parallel reduction on a local memory architecture is that sharing of expressions cannot be exploited globally across jobs. Within a job all sharing can be maintained.

5.2 Parallel reduction strategy

To avoid the major disadvantage of copying jobs (duplication of work), a special reduction strategy has been devised on the job level. This strategy guarantees that a job is an application of a function to completely evaluated arguments. Therefore copying jobs can not result in the duplication of work in the arguments.

We have implemented the parallel reduction strategy by a job annotation that has to be given by the programmer. An example is shown in the program fragment of Figure 3. The SASL syntax has been augmented with job annotation through angular brackets. Apart from their meaning as an annotation of parallelism the angular brackets serve the same purpose as normal parentheses.

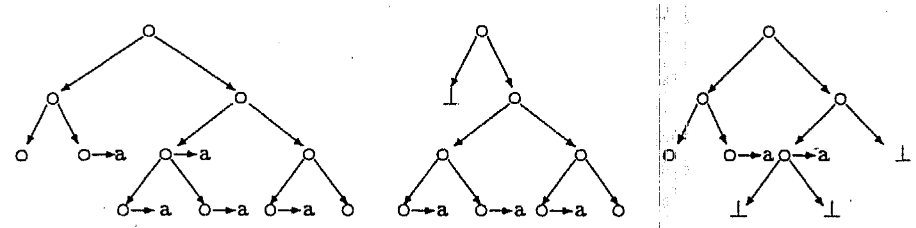
The (angular) job brackets provide the only means in the language to annotate parallel jobs. A program is sequentially evaluated until an expression that contains job brackets is needed. In the case of Figure 3 the application $G < job_1 > \ldots < job_p >$ is then suspended until the parallel evaluations of $job_1, job_2, \ldots, job_p$ have been completed. However, before the jobs are submitted for parallel evaluation, all arguments $a_{i1}\ a_{i2}\ \ldots\ a_{im_i}$ of each job_i are sequentially evaluated. Now copying the evaluated arguments a_{ij}, as part of the jobs, cannot result in extra work. If $a_{i1}\ a_{i2}\ \ldots\ a_{im_i}$ would have been evaluated in parallel, then any function application shared between the a_{ij} would have been copied. This would result in the duplication of work.

5.3 Parallel range queries

On an architecture without a globally accessible store, it is normally difficult to decide how to distribute common data over a network of cooperating processors, because replication of common data can be costly. In the case of range trees there is already a large amount of redundancy in the data structures. Hence we can afford to invest a little more space to solve the distribution problem. In case of the wedge partition the extra space is $O(\log n)$. The slice partition does not require extra space because the subtrees in each part have to be grouped into a single data structure anyway.

For both kinds of range tree partitions we have chosen not only to store the data structures that represent a part itself but also to copy the set of access paths from the root of the main tree to the nodes within the part. All structures beyond a part are omitted. Since the access paths to most parts lead through other parts we must make sure that the same result is reported only once. This is achieved by removing the associated structures from nodes on the access paths. Consider as an example the unpartitioned main tree as shown in Figure 4-a. (We argued at the end of Section 2 that several nodes of the main tree do not have to store an associated structure.) We will show how to construct the slice partition of this tree. The processor appointed to report records in the lowest layer (part 3) requires access to the root and all its right descendants, but not the left subtree of the root. In Figure 4-b the associated structures at level 2 are therefore pruned as well as the left subtree of the root. The main tree for the processor appointed to search layer 2 is shown in Figure 4-c. It has been constructed in a similar fashion.

The organization as sketched above allows all processors to start working at the same time on the query to be answered. In the beginning they will all be doing exactly the same work, since the root of the main tree must first be investigated. When a particular processor enters its own part, other processors will find that the subtree they are trying to enter is either not present and therefore cease to follow such a dead end, or find it devoid from associated structures because they must follow a path to a part further down the main tree. Because the parts are disjoint with respect to the associated structures, the answers reported by the individual processors must be gathered to form the total result of the query.



(a) original tree (b) part 3 (c) part 2

Figure 4: Original tree and two derived slices ("a" is an associated structure)

The construction of parts as described above allows us to apply the sequential search functions of Figure 2 unmodified to each of the parts in parallel. Annotated by job brackets the parallel main program is as shown in Figure 5. The figure also shows that instead of applying the function *search* to a single query, a list of queries is used. The function *map* applies the partial application (*search part$_i$*) to each element of the list of queries. In this way experiments with multiple queries can be performed and a single query can be answered by using a singleton *query_list*.

```
gather <map (search part₁) query_list> ··· <map (search part_p) query_list>
WHERE
gather result₁ ··· result_p = result₁: ··· : result_p
 map  f  ()           =  ()
 map  f  (head:tail)  =  f    head:map  f  tail
```

Figure 5: Parallel main program for a list of database queries

6 Results

We have run experiments on a simulator of our architecture with a 2-dimensional database of 1024 records and three sets of queries. The database describes 1024 major sites (all coordinate pairs are different) in the city of Amsterdam. Map coordinates run from 1 to 100 in east-west direction (first dimension) and from 1 to 300 in north-south direction (second dimension). Most sites are located in the city center.

The experiments assume that a sufficient number of processors is available to absorb all parallelism. With a small database this is realistic but for a real database some parts will have to be grouped onto a single processor. We will not pursue this issue in the current paper. Based on this assumption we may calculate speedup results by comparing the time necessary to perform all queries sequentially to the time that the longest job needs to complete. As an indication of datacommunication cost we count the number of coordinates that are sent to the processor running the longest job and the number of answers this processor returns.

Figure 6 shows the results for the three sets of queries and both partitioning methods. The last column pertains to a single typical query (*center*). Neither of the partitioning methods can achieve a significant speedup for this query. When queries are processed in batches the situation becomes different. The first two columns in Figure 6 (*east-west* and *north-south*) show the speedups that we have measured when processing a list of queries. Instead of constructing a list of random queries we preferred to analyze two extreme situations: one in which queries generate a maximum amount of parallelism (i.e., all processors contribute answers) and the other in which only a few processors are active.

In case of the slice partition all processors will be active when the query interval in the first dimension spans the entire range of points in the data base. On the other hand only a few parts will be active if each query specifies a narrow range in the first dimension. The corresponding results are shown in the columns *east-west* and *north-south* respectively. Both sets of queries specify a list of congruent rectangles, narrow in one and wide in the other dimension. The first set of queries contains 30 rectangles running east–west over the entire map area and the second contains 26 rectangles that run north–south. Both sets of queries cover the entire map area, with some overlap.

It is more difficult to control the amount of parallelism in case of the wedge partition. To enable us to make comparisons we applied the same sets of queries with wedge partition as we did before with slice partition. It is interesting to note that the *north–south* set of queries (narrow in the first dimension) of which each query only visits one part of the partition turns out to yield best results, because the queries in the list activate different parts and together they use most of the available processors. On the other hand the *east-west* set of queries yields lower speedup because each query always visits the same processors, such that using a list of queries does not improve the processor utilization.

The communication involved in the parallel range queries is negligible because only the list of queries has to be sent to all processors and the lists of results have to be returned. Each processor has its private copy of the relevant parts of the data base.

The experiment did not show the inferior behaviour of the wedge partition compared to the slice partition, as could be expected from the worst-case time bounds derived in section 3. (For 1024 data points the performance ratio of both mehtods is $(\log n)^d/(\log n)^{d-1} \approx 10$.) This may be due to special bias of the experimental data or the worst-case time bound may be overly pessimistic. To investigate this we plan to increase the size of the data base and/or to develop a statistical model for expected case analysis.

query	east–west	north–south	center
communicated coordinates	4×30	4×26	4×1
slice partition (8 processors in total)			
number of reporting processors	6	3	3
number of answers busiest processor	704	468	27
speedup	3.6	1.7	1.6
wedge partition (11 processors in total)			
number of reporting processors	6	8	2
number of answers busiest processor	166	421	128
speedup	2.6	4.2	1.3

Figure 6: Performance summary of parallel range queries

Memory usage is not shown in Figure 6 because it is the same ($\Theta(n)$) for all processors in both cases.

7 Conclusions and future research

We have presented a parallel implementation of a well-known query mechanism used in many database applications. We obtained encouraging speedup figures with a small database, which gives hope for a substantial gain in efficiency in real life applications.

The idea of applying functional languages to databases is not a new one. However—to our knowledge—the strategy up till now has been to implement the database's *query language* in a functional style. Here we proposed to implement the index structure in a functional language *independent* of the nature of the query language lying above this structure.

We have considered two different partition schemes, the wedge and the slice partition. If we perform the queries in a batched way, the slice partition gives in general the best results—notwithstanding the results represented by the second column in Figure 6, which are due to the particular type of query. This is not surprising, since in this case the number of processors that are involved in a single query can be logarithmic. In the wedge partition, only $O(\log^* n)$ processors are needed for a query.

If the number of answers in the first coordinate is small, however, the wedge partition gives much better results than the slice partition does. The reason is that in this case, associated structures are needed that are low in the main tree, and many of these associated structures are stored in different processors. Hence, to perform a number of batched queries, processing a query does not have to be postponed until the previous query is completed.

In the slice partition, all associated structures that are low in the main tree, are stored in only a few processors. Hence many queries have to wait until the previous ones are finished.

We expect that for larger n, the slice partition will perform better than the wedge partion, since the number of processors that are needed in the slice partition for a query—which can be proportional to $\log n$—grows faster than the number of processors that are needed in the wedge partition—which is only proportional to $\log^* n$.

In the future, we will extend this work, by experimenting with higher dimensional queries and lager databases. We also plan to extend the parallel implementation such that points can be inserted and deleted in the range tree. Finally, we will investigate whether it is possible to apply the ideas of partitioning to parallel implementations of other data structures.

References

[1] J.L. Bentley. *Decomposable searching problems.* Inf. Proc. Lett. **8** (1979), pp. 244-251.

[2] J.L. Bentley. *Multidimensional divide-and-conquer.* Comm. of the ACM **23** (1980), pp. 214-229.

[3] J.L. Bentley and J.R. Friedman. *Data structures for range searching.* Computing Surveys **11** (1979), pp. 397-409.

[4] B. Chazelle. *A functional approach to data structures and its use in multidimensional searching.* SIAM J. Comput. **17** (1988), pp. 427-462.

[5] H. Edelsbrunner. *A note on dynamic range searching.* Bull. of the EATCS, Number 15 (1981), pp. 34-40.

[6] P.H. Hartel and W.G. Vree. *Parallel graph reduction for divide-and-conquer applications, part II: program performance.* PRM project internal report D-20, Department of Computer Science, University of Amsterdam, 1988.

[7] D.E. Knuth. *The Art of Computer Programming, Vol. 3, Sorting and Searching.* Addison-Wesley, Reading, Mass., 1973.

[8] G.S. Lueker. *A data structure for orthogonal range queries.* Proc. 19-th Annual IEEE Symp. on Foundations of Computer Science, 1978, pp. 28-34.

[9] M.H. Overmars. *Efficient data structures for range searching on a grid.* J. of Algorithms **9** (1988), pp. 254-275.

[10] M.H. Overmars and M.H.M. Smid. *Maintaining range trees in secondary memory.* Proc. 5-th Annual STACS, Springer Lecture Notes in Computer Science, Vol. 294, Springer Verlag, 1988, pp. 38-51.

[11] M.H. Overmars, M.H.M. Smid, M.T. de Berg and M.J. van Kreveld. *Maintaining range trees in secondary memory, part I: partitions.* Report FVI-87-14, Department of Computer Science, University of Amsterdam, 1987. To appear in Acta Informatica.

[12] M.H.M. Smid. *General lower bounds for the partitioning of range trees.* ITLI Prepublication Series CT-88-02, University of Amsterdam, 1988.

[13] M.H.M. Smid and M.H. Overmars. *Maintaining range trees in secondary memory, part II: lower bounds.* Report FVI-87-15, Department of Computer Science, University of Amsterdam, 1987.

[14] D.A. Turner. *A new implementation technique for applicative languages.* Software Practice and Experience **9** (1979), pp. 31-49.

[15] W.G. Vree and P.H. Hartel. *Parallel graph reduction for divide-and-conquer applications, part I: program transformation.* PRM project internal report D-15, Department of Computer Science, University of Amsterdam, 1988.