d-Dimensional Range Search on Multicomputers*

Afonso Ferreira$^{(1)\dagger}$ Claire Kenyon$^{(1)}$ Andrew Rau-Chaplin$^{(2)\ddagger}$ Stéphane Ubéda$^{(1)\S}$

(1) LIP ENS-Lyon
46, allée d'Italie
69364 Lyon Cedex 07, France
Firstname.Name@lip.ens-lyon.fr

(2) Technical Univ. of Nova Scotia
P.O. Box 1000, Halifax
Nova Scotia, Canada B3J 2X4
arc@tuns.ca

Abstract

The range tree is a fundamental data structure for multi-dimensional point sets, and as such, is central in a wide range of geometric and database applications. In this paper, we describe the first non-trivial adaptation of range trees to the parallel distributed memory setting (BSP like models). Given a set L of n points in d-dimensional Cartesian space, we show how to construct on a coarse grained multicomputer a distributed range tree T in time $O(\frac{s}{p} + T_c(s,p))$, where $s = n \log^{d-1} n$ is the size of the sequential data structure and $T_c(s,p)$ is the time to perform an h-relations with $h = \Theta(s/p)$. We then show how T can be used to answer a given set Q of $m = O(n)$ range queries in time $O(\frac{s \log n}{p} + T_c(s,p))$ and $O(\frac{s \log n}{p} + T_c(s,p) + \frac{k}{p})$, for the associative-function and report modes respectively, where k is the number of results to be reported. These parallel construction and search algorithms are both highly efficient, in that their running times are the sequential time divided by the number of processors, plus a constant number of parallel communication rounds.

1 Introduction

The range tree is a fundamental data structure for multi-dimensional point sets, and as such, is central in a wide range of geometric and database applications[18]. The design and implementation of efficient parallel versions of this important data structure is one of the primary goals of this year's DIMACS Implementation Challenge[1]. In this paper, we describe the first non-trivial adaptation of range trees in the parallel distributed memory setting.

Our approach is to describe a set of efficient scalable algorithms for the construction and manipulation of a distributed analog to the sequential range tree data structure[4]. We then

---

*Part of this work was completed while the authors were visiting each other in Lyon and in Halifax. Support from the respective Institutions is acknowledged.

†Partially supported by the HCM MAP project of the EC.

‡Partially supported by the Natural Sciences and Engineering Research Council (Canada).

§Partially supported by the EC project NATHAN.

show how to perform $O(n)$ independent range searches on a distributed range tree T, in parallel. Note that the path that an individual search will trace in T is *not* known ahead of time, and must instead be determined "on-line". That is, only when a search query is at a node of T can it determine which node or nodes of T it should visit next. Also note that the paths of the search queries can overlap arbitrarily, such that at any time any node of T may be visited by an arbitrary number of search queries.

The Model

Recently, there has been much interest in "realistic" parallel models (e.g. BSP, LogP, C^3, CGM) that can better predict the performance of parallel algorithms on existing, typically coarse or medium grained, parallel computers[21, 7, 16, 12]. In Valiant's BSP model, each communication round consists of routing a single arbitrary h-relation (i.e. each processor sends and receives $O(h)$ data). Slackness in the number of processors is used to optimally simulate PRAM algorithms on distributed memory multicomputers. However, as Valiant points out, one may want to design "implementations of the BSP model that incorporate features for communications, computation or sychronization that are clearly additional to the ones in the definition"[21].

In this paper, we use the *Coarse Grained Multicomputer* model (CGM(s,p)), also sometimes referred to as the weak-CREW BSP model[15]. This model has been used (explicitly or implicitly) in parallel algorithm design for variety of problems [12, 15, 8, 11, 17, 9, 14] and has led to parallel codes exhibiting good timing results [12, 9, 14]. It consists of a set of p processors P_0 to P_{p-1} with $O(\frac{s}{p})$ local memory each, connected via some arbitrary interconnection network or a shared memory. The term "coarse grained" refers to the fact that the size of each local memory will typically be "considerably larger" than $O(1)$. We will assume $\frac{s}{p} \geq p$ as was assumed in [12], which is clearly true for all existing parallel machines. All algorithms consist of alternating local computation with global communications operations (Supersteps).

In this model, all global communications are performed by a small set of standard communications operations - Segmented broadcast, Segmented gather, All-to-All broadcast, Personalized All-to-All broadcast, Partial sum and Sort, which are typically efficiently realized in hardware. If a parallel machine does not provide these operations they can be implemented in terms of a constant number of sorting operations [12].

Moreover, recently Goodrich [15] has shown that, given $p < n^{1-\frac{1}{c}}$ $(c \geq 1)$, sorting $O(n)$ elements distributed evenly over p processors in the BSP (or LogP) model can be achieved in $O(\log n/\log(h+1))$ communication rounds and $O(n \log n/p)$ local computation time, for $h = \Theta(\frac{n}{p})$, i.e. with optimal local computation and $O(1)$ h-relations, when $\frac{n}{p} \geq p$. Therefore, using this sort, the communication operations of the $CGM(s,p)$ can be realized in the BSP (or LogP) models in a constant number of h-relations, where $h = \Theta(\frac{s}{p})$. Hence, in the remainder, the time required by any of the above global communication operations on the $CGM(s,p)$ will be denoted $T_c(s,p)$.

Finding an optimal algorithm in the CGM model is equivalent to minimizing the number of global communication rounds as well as the local computation time. It has been shown that minimizing the number of supersteps also results in improved portability across different parallel architectures[21, 22].

The Multidimensional Range Search Problem

Consider a collection L of n records, where each record l has a value $key(l)$ and is identified by an ordered d-tuple $(x_1(l), ..., x_d(l)) \in E^d$, the d-dimensional Cartesian space. In the *orthogonal range search* problem, the query specifies a domain q in E^d, and the outcome of the search, depending on the application, may be either the subset $R(q)$ of the points of L contained in q, or the number of such points, or more generally a function $\bigotimes_{l \in R(q)} f(l)$, where $f(l)$ is an element of a commutative semigroup with operation $\otimes$. The former version of this problem is called the *report mode* while the latter version is called the *associative-function mode*. For lower bounds see [6].

There are many sequential data-structures and algorithms for range searching, each offering a different trade-off between storage and time complexity. These structures include k-D trees, multidimensional trees, Super-B trees, range trees, and layered range trees.

Multidimensional binary trees, commonly known as k-D trees are an optimal space solution, requiring $\Theta(dn)$ space, but having a discouraging worst-case search performance of $O(dn^{1-1/d})$ time [18]. Parallel algorithms for the range-search problem based on k-D trees have been studied for the scan computation model [5].

The Range Tree data-structure represents a particularly good balance between storage space and search time. The structure requires $O(n \log^{d-1} n)$ space and construction time, but supports an $O(\log^d n)$ time search algorithm [18]. An improved version of this structure, known as the layered range tree, saves a factor of $\log n$ in the search time. A parallel version of the range tree data structure was introduced for the SIMD hypercube model of computation [19]. It required $O(d \log n)$ search time per query using $O(\log^d n)$ processors. But, the parallelization scheme was based on copying of the data structure onto each processor, therefore requiring $O(pn \log^d n)$ memory space in total which is, in most situations, quite unrealistic. In SODA'95, a derivative of the range tree data structure for secondary memory was described [20]. The one dimensional range search problem is solved in [13].

Our Results

Given a set L of n points in d-dimensional Cartesian space, we show how to construct on a CGM(s, p) a distributed range tree T in time $O(\frac{s}{p} + T_c(s,p))$, where $s = n \log^{d-1} n$ is the size of the sequential data structure. We then show how T can be used to answer a given set Q of $m = O(n)$ range queries in time $\mathrm{O}(\frac{s \log n}{p} + T_c(s,p))$ and $\mathrm{O}(\frac{s \log n}{p} + T_c(s,p) + \frac{k}{p})$, for the associative-function[1] and report modes respectively, where k is the number of results to be reported.

These parallel construction and search algorithms are both highly efficient, in that their running times are the sequential time divided by the number of processors, plus a constant number of parallel communication rounds (i.e., h-relations with $h = \Theta(s/p)$).

Our solution is, in part, based on the Multisearch paradigm first introduced in [10] and later used to solve a variety of problems[12, 10, 3, 2]. It represents a significant advancement over the multisearch method described in [12] in that the lower dimensional substructures pointed to by each node of T is of non-constant size and queries that must visit several

---

[1] In the special case of associative functions with inverses, this problem can be solved using weighted dominant counting [12]

neighbours of a node of T can do so by "splitting" into several subqueries.

In very broad terms, our techniques for solving the range search problem are a judicious combination of the following ideas:

- Partition T into pieces (of differing shapes and dimensions), some of which are processed sequentially, while others are processed in parallel.
- Create multiple copies of those pieces of T for which too many searches need access, and distribute the copies to processors, each of which is responsible for advancing a manageable subset of the "congested" searches. It should be noted that the straightforward strategy of making multiple copies of T, and using one copy for each $\frac{n}{p}$ group of queries, does not work. This is due to the fact that it would not only take too much time to create the p copies, but there is not enough space to store all of these copies of T.
- Implement searches using multiple queries by, when necessary, making copies of those queries visiting a node v of T who require access to more than a single neighbour of v. Some of these queries are advanced sequentially, while others are advanced in parallel.

Of course, the parameters needed to efficiently perform these partitioning, duplication and mapping strategies cannot be precomputed, since the full search paths are computed on-line. Therefore, these parameters must also be determined on-line, as the searches advance through T. The above description is necessarily an over simplification, only a careful look at the details can reveal the exact interplay between the above ideas, as well as the exact nature of each of them.

The organization of the paper is as follows. Section 2 describes the classical range tree and our distribution scheme on p processors. A coarse grained parallel algorithm to build this distributed data structure is described in Section 3. Section 4 gives a coarse grained parallel algorithm to solve n queries in parallel with the distributed range tree.

2 On Range Trees

In this section we first define the segment tree and range tree data structures to be used in the remainder. Then, we define a labeling of the nodes of the range tree, in order to be able to store it efficiently in a distributed memory setting. Finally, we define the "hat" of such a structure, which is fundamental to our partitioning strategy.

2.1 Recall of Basic Definitions

Let a $(1, n)$ segment tree [4] be a complete rooted binary tree with n leaves. Each node is associated with a segment. The segments associated to the leaves are [1..2), [2..3),...point). Each internal node is associated with the segment formed by the union of the two segments associated to its children. Thus, the segment associated with the root is $[1..n]$.

As presented in [18], the range tree structure is a generalization of the segment tree. Let each element of L be a point $l = (x_1(l), x_2(l), \ldots, x_d(l))$, $j = d$, and $L_j = L$. Finally, a segment tree is said to be *in dimension i* if the segments associated to its leaves are obtained by a projection of a subset of L onto dimension i.

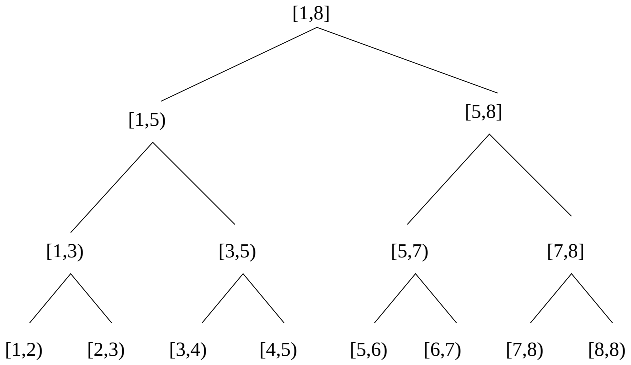

Figure 1: The segment tree structure for (1,8).

Definition 1 *The j-dimensional range tree T for a set L_j is recursively defined as follows.*

- **i** *A primary segment tree T^* in dimension $d-j+1$ corresponding to the set $\{x_{d-j+1}(l) | l \in L_j\}$. For each node v of T^*, let $W(v)$ denote the set of points such that $x_{d-j+1}(l)$ lies in the interval associated with v. Define the $(j-1)$-dimensional set*
  $L_{j-1}(v) = \{(x_{d-j+2}(l), ..., x_d(l)) | l \in W(v)\}$.
- **ii** *Each node v of T^* has a pointer to a range tree for $L_{j-1}(v)$ which is called descendent(v). For each node w in the primary segment tree of descendent(v), we define ancestor(w) $= v$.*

2.2 Labeling

To each node v of the range tree, we associate a unique label denoted path(v) which enables us to refer to nodes and to subtrees of T, which is defined as follows.

Definition 2 *For any node v of a range tree we define the following indices.*

- **i** *Level(v) is the length of the shortest path from v to a leaf (or 0 if v is a leaf).*
- **ii** *Index(v) =*
  - *0 if v is the root of T*
  - *Index(ancestor(v)) if v is a root of any segment tree except T*
  - *2×Index(parent(v)) if v is a left child in a segment tree.*
  - *2×Index(parent(v))+1 if v is a right child in a segment tree.*

  (see Fig. 2)
- **iii** *Path-index(v) = $\langle$index(v),level(v)$\rangle$*
- **iv** *Path(v) =*
  - *path-index(v) if v is a node of T^**
  - *$\langle$path-index(v),path(ancestor(v))$\rangle$, otherwise.*

Lemma 1 *For every segment tree $t \in F$ and all nodes $v \in t$, path(ancestor(v)) uniquely identifies the tree t to which v belongs.*

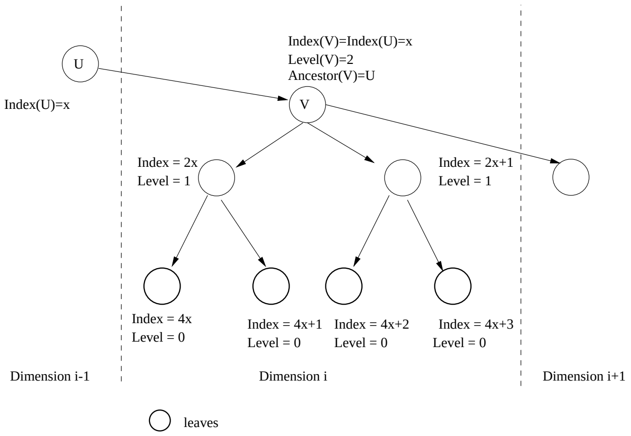

Figure 2: Illustration of *Index* and *Level* of a node of T.

Proof: It is easy to see that for all nodes $v \in T$, path(v) is unique. Furthermore, it follows from Definition 1 that for every segment tree $t \in T$ and each pair of nodes $u, v \in t$, it holds that ancestor(u) = ancestor(v). Hence, path($ancestor(u)$) = path(ancestor(v)) and this can be interpreted as the name of the segment tree t.

2.3 The "hat"

A range tree T for a set of n points is of size $s = O(n \log^{d-1} n)$ [18], which is as large as the total memory available on our CGM(s, p). Therefore, the range tree must be partitioned into substructures where each substructure is of size $O(s/p)$. To support an efficient search strategy, some of these substructures will be stored on a single processor while others will be copied on to all processors such that each processor stores no more than $O(1)$ such structures.

Definition 3 *Given a range tree T,*

- **i** *Let the "hat" H of T denote the subtree of T induced by all nodes v of T in the top* $\log p$ *levels, i.e. with level(v)* $\geq \log n - \log p$.
- **ii** *Let F denote the forest of subtrees of T whose roots are the leaves of the hat H, i.e. the subtrees induced by all nodes v of T with level(v)* $\leq \log n - \log p$. *Note that each element of this forest is a range-tree on n/p points and has dimension $j \in [1..d]$ (see Fig.* ??*)*[2].
- **iii** *For each range tree t in F with root r, let location(t)= i, where i is the index of the leaf of H corresponding to r. Note that such indexes are in the range* $0 \ldots p-1$. *Let* $F_i = \{t \in F | location(t) = i\}$.

Theorem 1 *The following holds for H and F_i as defined above.*

i) *The hat H has size* $O(p \; \log^{d-1} p) = O(s/p)$.

---

[2] In the one-dimensional case, where the range tree is just a segment tree, the hat consists of the top $\log p$ levels of the tree and the forest consists of the p subtrees rooted at level $\log p$.

ii) *For every* i*,* F_i *has size* $O(s/p)$*.*

Proof:

i) Immediate from the fact that the hat is a range-tree with p leaves.

ii) For each $i \in [0..p-1]$, F_i consists of a set of range trees of various dimension (from 1 to d) of n/p points. By definition, the sets F_i are disjoint and have equal size, yielding $|F_i| = O(s/p)$, since the total data size is $O(s)$.

3 Constructing a Distributed Range Tree

Our range search algorithm is based on a distributed representation of a range tree. The size of the range tree data structure for n items is $s = n \ \log^{d-1} n$, therefore we will use a p processor coarse grained multicomputer with $O(\frac{n \log^{d-1} n}{p})$ memory per processor, i.e. CGM(s,p). Without lost of generality, we assume (as in [18]) that all coordinates, in each dimension, are normalized by replacing each of them by their rank in increasing order (i.e. points are in $\{1,\ldots,n\}$) and that $n = 2^k$.

In the following, our distributed range tree will be stored on a CGM(s,p) as follows.

- A copy of the hat H will be stored on every processor and used as an index structure for the forest F.
- Each range tree t in F_i will be stored on processor P_i.

As seen in the previous section, both H and the F_i fit in a single processors memory.

In the following we describe a parallel algorithm for constructing the previously defined distributed range tree. As shown in [18], there exists an optimal sequential algorithm to build a d-dimensional range tree of size $O(n \log^{d-1} n)$, running in time $O(n \log^{d-1} n)$. This algorithm works in a bottom-up fashion in which segment trees are built up from their leaves one dimension after another.

As in sequential, the distributed range tree is constructed in d phases from phase $j = 1$ to phase $j = d$. At the start of phase j, let S^j be a set of records representing leaves of segment trees of T in dimension j. These segment trees must be constructed. More precisely, a record in S^j, corresponding to a point l from the original point set L, consists of two vectors: $l = \{x_1(l),\ldots,x_d(l)\}$ and a label index(l). This record in S^j is to become a leaf of the segment tree in T which is uniquely identified by index(l).

In phase j we first perform the data distribution: S^j will be sorted such that leaves of segment trees $t \in T$ with location$(t) = i$ are routed to processor P_i. Elements of F can then be sequentially constructed. Since the roots of the segment trees in F are the leaves of H, it suffices to perform an all-to-all broadcast of these roots in order to have all information required to complete the construction of the segment trees $t \in H$ in dimension j. Then, the set S^{j+1} is constructed.

The algorithm is as follows.

Algorithm Construct

```
Input: Each processor $P_i$ stores a set $I_i$ of $n/p$ points drawn from $L$, arbitrarily.
Output: Each processor $P_i$ stores
        i) A copy of H;
        ii) The set $F_i$.
  0 Each processor creates for each element $l$ of $I_i$ an initial record $e$ of $S^1$ with $e = (x_1(l), ..., x_d(l))$ and index($e$)= nil. Let $j \leftarrow 1$.
  1 Globally sort $S^j$ by primary key index and secondary key $x_j$.
  2 Each processor receives an ordered set of records from $S^j$ representing leaves of trees $t \in F$ which are in dimension $j$. These trees must now be routed to the correct location. Each processor $P_i$ divides its set into groups of $n/p$ consecutive records, computes the global rank of each group and routes the $k^{\text{th}}$ group to processor $P_{k \bmod p}$, using global sort.
  3 Each processor $P_i$ constructs sequentially the elements of $F_i$. Since the roots of $F$ correspond to the leaves of $H$, all processors perform an all-to-all broadcast of the roots of their $F_i$.
  4 Each processor receives $O(p \log^{d-1} p)$ roots and constructs its own copy of segment trees of $H$ in dimension $j$.
  5 if $j = d$ then exit.
  6 Each record $z \in S^j$ stored in processor $P_i$ belongs to a segment associated with a leaf $y$ of $H$ in dimension $j$ (corresponding to the root of $z$'s tree $t \in F$). In each processor $P_i$, for all $z$, walk from $y$'s parent to the root of $y$'s segment tree and for each node $u$ visited create a new element $s$ of $S^{j+1}$ as follows: $x(s) = x(z)$ and index($s$)=path($u$).
  7 $j \leftarrow j + 1$. Goto step 1.
```

The correctness of Algorithm Construct follows from the sequential construction algorithm in [18], Definitions 1 and 3, Lemma 1, and Theorem 1. Its time complexity is $O(\frac{s}{p} + T_{sort}(s,p))$ where the first term comes from Steps 0, 3, 4 and 6, and the second from Steps 1, 2, and 3.

We thus have the following result.

Theorem 2 *A distributed range tree T can be constructed on a CGM(s,p) in time $O(\frac{s}{p} + T_c(s,p))$.*

This theorem and the weak-CREW BSP sorting algorithm from [15] imply the following.

Corollary 1 *A distributed range tree T can be constructed on a weak-CREW BSP in a constant number of h-relations ($h = \Theta(s/p)$) and $O(s/p)$ local computation time.*

4 Parallel Range Search

As presented in the Introduction, the parallel range search problem consists of answering the set Q of $m = O(n)$ range queries in parallel. In [18], an $O(\log^d n)$ sequential algorithm to solve the single query problem is given. The sequential algorithm for a query q on a range tree T runs as follows. Initially, q *visits* the root of T. When a query visits a node v in dimension j of T, it compares the query in the j^{th} dimension to the interval associated with v. There are four cases.

1. If the two segments are equal and $j < d$ then proceed to the next dimension, and the next node to be visited is the root of descendent(v).

2. If the two segments are equal and $j = d$ then v is the last node on q's search path and the segment tree rooted at v should be *selected* by q (i.e, all of its leaves are in the range of q).
3. If the two segments overlap (but are not equal), then the query q should be split into two queries: q', which is to visit the left child of v, and q'', which is to visit the right child of v.
4. If the two segments do not overlap the query q is deleted.

Note that each query q will visit at most $O(\log n)$ nodes in each dimension of T and $O(\log^d n)$ nodes will be selected in the final dimension d.

4.1 Identifying the results

The parallel algorithm for solving $m = O(n)$ queries takes the same basic approach. Initially, each processor P_i stores a set Q_i of n/p queries drawn from Q arbitrarily, and a distributed range tree T as described in Section 3. Note that a query is ready to report its result only when it visits a segment tree in dimension d of a range tree.

Thus, each processor P_i advances its queries through its copy of the hat H. This set dealt with, some of these queries select segment trees in dimension d of H, while others need to continue in F. The queries that have not completed their search paths and the required elements of F are then evenly balanced such that each processor stores $O(\frac{s}{p})$ queries along with the range trees from F they require. Finally, the queries are sequentially advanced through elements of F until they select segment trees in dimension d.

In the following algorithm, let $\bar{Q}$ denote the queries which have selected a segment tree in dimension d.

Algorithm Search

Input: Each processor P_i stores a set Q_i of n/p queries drawn arbitrarily from Q and a distributed range tree T.

Output: For each query $q \in Q$, a set of selected segment trees in dimension d of T and whose leaves correspond to the points of L in q's domain. Each such selected segment tree is given by an element of $\bar{Q}$.

0 Each processor P_i, advances its queries Q_i through the hat H. The queries which have already selected a segment tree in dimension d of H are put in $\bar{Q}$. Let $\hat{Q}$ denote the remaining queries, which need to visit a node in F.

1 Let $\hat{Q}_{F_j}$ denote those queries wanting to visit a tree $t \in F_j$. Globally, compute $c(j) = |\hat{Q}_{F_j}|/(|\hat{Q}|/p)$.

2 Make $c(j)$ copies of F_j and distribute them evenly.

3 Redistribute $\hat{Q}$ evenly so that every query $q \in \hat{Q}$ is stored on a processor that also stores a copy of the element of F which q is visiting.

4 Each processor P_j thus receives a set of queries and performs the sequential algorithm to select the appropriate segment trees, and puts the corresponding queries in $\bar{Q}$, thus completing $\bar{Q}$.

The load balancing phase, implemented in Steps **1** through **3** evenly distributes queries and forests F_i, such that each processor has $O(1)$ copies of each, as proved in [12].

Therefore, the correctness of Algorithm Search follows from the sequential construction algorithm [18] and from [12]. The time complexity of Algorithm Search is $O(\frac{s \log n}{p} + T_{sort}(s,p))$ where the first term comes from Steps 0, and 4, and the second from Steps 1, 2, and 3.

Theorem 3 *Given a set Q of $m = O(n)$ range queries and a distributed range tree T for a set L of $O(n)$ points in E^d, stored on a CGM(s,p). Each element of Q can identify the subset of points from L in its domain, in time $O(\frac{s \log n}{p} + T_c(s,p))$.*

As in the previous section, combining this result with the weak-CREW BSP sort presented in [15] we get:

Corollary 2 *Given a set Q of $m = O(n)$ range queries and a distributed range tree T for a set L of $O(n)$ points in E^d, stored on a weak-CREW, BSP. Each element of Q can identify the subset of points from L in its domain, in a constant number of h-relations ($h = \Theta(s/p)$) and $O(\frac{s \log n}{p})$ local computation time.*

4.2 Reporting the results

In the range search problem, the query specifies a domain q in E^d, and the outcome of the search depends on the application. It may be either the subset L^q of the points of L contained in q (the *report mode*), or the number of such points, or more generally a function $\bigotimes_{l \in L^q} f(l)$, where $f(l)$ is an element of a commutative semigroup with operation $\otimes$ (the *associative-function mode*).

In this section we describe algorithms for both the associative-function and report modes running in time $O(\frac{s \log n}{p} + T_c(s,p))$ and $O(\frac{s \log n}{p} + T_c(s,p) + \frac{k}{p})$, respectively, where k is the number of results to be reported.

Algorithm Associative-Function

```
Input: A distributed range tree $T$, an associative function $f$, and a set $Q$ of $n$ queries.
Output: $f(q)$ for each query $q \in Q$.
  0 Compute $f(v)$ bottom-up for each node $v$ in dimension $d$ of $T$ as follows:
        - Compute $f(v)$ for each node in trees of $F$ in dimension $d$ sequentially.
        - All-to-all broadcast the values of $f(v)$ for each root of trees of $F$ in dimension $d$.
        - Compute $f(v)$ for each node $v$ of the hat $H$ in dimension $d$.
  1 Perform Algorithm Search.
  2 For each $q' \in \bar{Q}$, we create the pair $(q, f(\text{root of selected segment tree}))$.
  3 Sort the pairs according to their first coordinate $q$.
  4 For each block of pairs sharing a common $q$, compute $f$ over the whole block (using a
    segmented partial sum).
```

Once we have the output of the Algorithm Search, it only remains to report the leaves of each selected segment tree. In order to do this in a balanced manner, we weigh the selected segment trees according to their sizes and redistribute them evenly, using again the load balancing procedure from [12].

Algorithm Report

Input: A distributed range tree T and a set Q of n queries.
Output: For each $q \in Q$ and each $l \in L$ in q's range, the pair (q, l) is on some processor.

0 Perform Algorithm Search to obtain a set of queries $q' \in \bar{Q}_i$ which have, each, selected segment trees in dimension d of T.
1 Compute for all $q \in \bar{Q}$ having selected a segment tree $t \in T$, the weight $w(q) = 2^{\text{level}(\text{root}(t))}$ = number of leaves of t.
2 Sort the elements of $\bar{Q}$ by weight.
3 Compute the partial sum $psw(q)$ for the element q of $\bar{Q}$ with respect to the weight $w(.)$, and let $dest(q) = p\lfloor psw(q)/\sum_{\bar{Q}} w(q)\rfloor$. Perform a segmented broadcast with destination $dest(.)$.
4 Make $w(q)$ copies of each query q and add it to $\bar{Q}$, associating with each copy a path to a leaf of the selected segment tree t. Each such copy corresponds to a pair (query of Q, point of L in q's range).

It is clear that algorithms Associative-Function and Report use only sequential procedures and the load balancing technique from [12]. Therefore,

Theorem 4 *Given a set Q of $m = O(n)$ range queries and a distributed range tree T for a set L of $O(n)$ points in E^d, stored on a CGM(s, p). All queries can be answered in both the associative-function and report modes in times $O(\frac{s \log n}{p} + T_c(s,p))$ and $O(\frac{s \log n}{p} + T_c(s,p) + \frac{k}{p})$, respectively, where k is the number of results to be reported.*

Again, considering the weak-CREW BSP sort presented in [15] we get:

Corollary 3 *Given a set Q of $m = O(n)$ range queries and a distributed range tree T for a set L of $O(n)$ points in E^d, stored on a weak-CREW BSP. Each element of Q can identify the subset of points from L in its domain, in a constant number of h-relations ($h = \Theta(s/p)$) and $O(\frac{s \log n}{p} + \frac{k}{p})$ internal computation time.*

5 Conclusion

In this paper, we have defined a distributed range tree, the first non-trivial adaptation of range trees in the coarse grained multicomputer setting. We use this data structure to perform highly efficient batched range search operations, in associative-function or in report mode. Our search algorithm requires only a constant number of global communication operations in order to answer $O(n)$ queries. From a theoretical perspective it is interesting to note that a $\log p$ factor could be shaved off the sequential component of the search time by using a range tree structure upon which fractional cascading has been applied, although in practice the additional overhead incurred in such an approach would likely make it unattractive.

Our algorithms for constructing and searching the distributed range tree are a combination of standard communication primitives (such as parallel sort, used as a black box) and of standard sequential range tree operations, so that their implementation on any variety of multicomputer should be straightforward.

There are several interesting open problems that might be considered. Firstly, the construction algorithm is not quite optimal, since it uses a constant number of parallel sort operations on sets of size $n \log^{d-1} p$, the number of leaves of the range tree, while ideally we would only wish to sort sets of size at most n, the number of input points. Secondly, the question of using parallelism to speed up just one single query (or a variable number of queries) remains open. However, we must stress that there currently is no viable alternative to the distributed range tree we describe when the database is large enough to require a distributed data structure.

References

[1] The fifth DIMACS implementation challenge:1995-1996, http://www.cs.amherst.edu/ ccm/challenge5.

[2] M.G. Andrews and D.T. Lee. Parallel algorithms for convex bipartite graphs and related problems. In *Proc. of Allerton Conference Communication, Control, and Computing, USA*, pages 195–204, 1994.

[3] M. Atallah, F. Dehne, R. Miller, A. Rau-Chaplin, and J.J. Tsay. Multisearch techniques: parallel data structures on mesh-connected computers. *Journal of Parallel and Distributed Computing*, 20:1–13, 1994.

[4] J.L. Bentley. Decomposable searching problems. *Information Processing Letters*, 8:244–251, 1979.

[5] G.E. Blelloch and J.J. Little. Parallel solutions to geometric problems on the scan model of computation. In *Proc. International Conference on Parallel Processing, St-Charles, USA*, pages 218–222, 1988.

[6] B. Chazelle. Lower bounds for orthogonal range searching, I. The reporting case. *Journal of the ACM*, 37(2):200–212, 1990.

[7] Culler, R. Karp, D. Patterson, A. Sahay, K. Schauser, E. Santos, R. Subrarnonian, and T. von Eicken. LogP: Towards a realistic model of parallel computation. In *Fifth ACM SIGPLAN Symposium on the Principles and Practice of Parallel Programming*, 1993.

[8] F. Dehne, X. Deng., P. Dymond, and A.A. Khokar. A randomized parallel 3D convex hull algorithm for coarse grained parallel multicomputers. In *Proc. of the ACM Sympoisum on Parallel Algorithms and Architectures, july 16-18, Santa Barbara, USA*, 1995.

[9] F. Dehne, A. Fabri, and C. Kenyon. Scalable and architecture independent parallel geometric algorithms with high probability optimal time. In *Proc. of the* 6^{th} *IEEE Symposium on Parallel and Distributed Processing, October, Dallas, USA*, 1994.

[10] F. Dehne and A. Rau-Chaplin. Implementing data structures on a hypercube multiprocessor and applications in parallel computation geometry. *Journal of Parallel and Distributed Computing*, 8(4):367–375, 1989.

[11] X. Deng and N. Gu. Good programming style multiprocessors. In *Proc. of the* 6^{th} *IEEE Symposium on Parallel and Distributed Processing, October, Dallas, USA*, 1994.

[12] A. Fabri F. Dehne and A. Rau-Chaplin. Scalable parallel geometric algorithms for coarse grained multicomputers. In *Proc. of the* 9^{th} *ACM Symposium on Computational Geometry*, pages 298–307, 1993.

[13] A. Fabri and O. Devillers. Scalable algorithms for bichromatic line segment intersection problems on coarse grained multicomputers. In *Proc. of the* 3^{th} *Workshop on Algorithms and Data Structures*, 1993.

[14] A. Ferreira, A. Rau-Chaplin, and S. Ubéda. Scalable 2d convex hull and triangulation for coarse grained multicomputers. In *Proc. of the* 6^{th} *IEEE Symposium on Parallel and Distributed Processing, San Antonio, USA (See M. Diallo, Master Thesis, 1996, LIP ENS-Lyon, France, for implementation results)*, 1996.

[15] T. Goodrich. Communication-efficient parallel sorting. In *Proc. of the* 28^{th} *annual ACM Symposium on Theory of Computing (STOC), May 22-24, Philadephia, USA*, 1996.

[16] S.E. Hambausch and A.A. Khokhar. C^3: An architecture-independent model for coarse-grained parallel machines. In *Proc. of the* 6^{th} *IEEE Symposium on Parallel and Distributed Processing, October, Dallas, USA*, 1994.

[17] H. Li and K.C. Sevick. Parallel sorting by overpartitioning. In *Proc. of the ACM Symposium on Parallel Algorithms and Architectures*, pages 46–56, 1994.

[18] F.P. Preparata and M.I. Shamos. *Range-searching problems*, chapter 3, pages 67–88. Springer-Verlag, 1985.

[19] R. Sridhar, S. Iyegar, and S. Rajanarayanan. Range search in parallel using distributed data structures. *Journal of Parallel And Distributed Computing*, 15:70–74, 1982.

[20] S. Subramanian and R. Ramaswamy. The P-range tree: A new data structure for range searching in secondary memory. In *Proc. of the* 6^{th} *annual Symposium On Discrete Algorithms, San-Francisco, January*, pages 378–387, 1995.

[21] L. G. Valiant. A bridging model for parallel computation. *Communication of ACM*, 38(8):103–111, 1990.

[22] L.G. Valiant. *General purpose parallel architecture.* J. van Leewen, North Holland, 1990.