d-Dimensional Range Search on Multicomputers*

Afonso Ferreira[(1)†] Claire Kenyon[(1)‡] Andrew Rau-Chaplin[(2)§] Stéphane Ubéda[(1)]

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

(2) Dalhousie University
P.O. Box 1000, Halifax
Nova Scotia, Canada B3J 2X4
`arc@tuns.ca`

May 30, 1999

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 of n points in d-dimensional Cartesian space, we show how to construct a distributed range tree T on a coarse grained multicomputer in time $O(s/p + T_c(s,p))$, 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-relation 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 $\mathrm{O}((s\log m)/p + T_c(s,p))$ and $\mathrm{O}((s\log m)/p + T_c(s,p) + k/p)$, 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 [21]. The design and implementation of efficient parallel versions of this important data structure was one of the primary goals of the DIMACS Implementation Challenge in 1996 [1]. In this paper, we describe an 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 of the sequential range tree data structure [4]. We then show how to perform $O(n)$ independent range searches on a distributed range tree T, in parallel. Note that the path traced by an individual search traversing 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(s) of T it should visit next. Also note that the paths of the search queries can overlap arbitrarily, so that at any time any node of T may be visited by an arbitrary number of search

*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.

†CNRS. Partially supported by the Région Rhône-Alpes.

‡CNRS.

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

queries.

The Model

Recently, there has been much interest in "more 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 [24, 8, 18, 13]. In Valiant's BSP model, each communication round consists of routing a single arbitrary h-relation (i.e. each processor send 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" [24].

In this paper, we use the *Coarse Grained Multicomputer* model (CGM(s,p)), also sometimes referred to as the weak-CREW BSP model [17]. This model has been used (explicitly or implicitly) in parallel algorithm design for a variety of problems [13, 17, 9, 12, 20, 10, 16] and has led to parallel codes exhibiting good time results [13, 10, 16]. It consists of a set of p processors with $O(s/p)$ 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 $s/p \geq p$ as it was assumed in [13], which is true for all existing parallel machines. All algorithms consist of *supersteps* (see [24]) alternating local computation with global communication operations, in which each processor can receive at most $O(s/p)$ data.

In the CGM(s,p) model, all global communications are performed by a small set of standard 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, each of them can be implemented in terms of a constant number of sorting operations [13].

In addition, it was shown in [17] that, given $p < n^{1-\frac{1}{c}}$ (with $c \geq 1$), 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 $h = \Theta(n/p)$, i.e. with optimal local computation and $O(1)$ h-relations, when $n/p \geq p$. Therefore, using this sort, the communication operations of the CGM(s,p) can be realized in the BSP (or LogP) model in a constant number of h-relations, where $h = \Theta(s/p)$. **Hence, in the remainder, any of the above global communication operations on the CGM(s,p) will be denoted $T_c(s,p)$.**

Finally, designing efficient algorithms in the CGM model is equivalent to minimizing the number of global communication rounds as well as the local computation time. We remark that minimizing the number of supersteps also results in improved portability across different parallel architectures [24, 25].

The Multidimensional Range Search Problem

Consider a collection of n records, where each record has a key-value and is identified by an ordered d-tuple in the d-dimensional Cartesian space. In the *orthogonal range search* problem, the query specifies a rectangular domain in the d-dimensional space, and the outcome of the search, depending on the application, may be either the subset of the points contained in the specified domain (report mode), or the number of such points, or more generally a function computed on a commutative semigroup (associative-function mode) [19].

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 [21].

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 [21]. 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 presents a particularly good tradeoff between storage space and search time. The structure requires $O(n \log^{d-1} n)$ space and construction time, and supports an $O(\log^d n)$ time search algorithm. An improved version of this structure, known as the layered range tree, saves a factor of $\log n$ in the search time[21]. A parallel version of the range tree data structure was introduced for the SIMD hypercube model of computation [22]. It requires $O(d \log n)$ search time per query using $O(\log^d n)$ processors. But the parallelization scheme is based on copying 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 [23], a derivative of the range tree data structure for secondary memory was described while the one-dimensional range search problem (segment search problem) was solved in [6, 14].

Our Results

Given a set of n points in the d-dimensional Cartesian space, where d is considered constant, we show how to construct a distributed range tree T on a CGM(s, p) in time $O(s/p + T_c(s))$, 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 $\mathrm{O}((s \log n)/p + T_c(s, p))$ and $\mathrm{O}((s \log n)/p + T_c(s, p) + k/p)$, 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 optimal, in the sense that their running times are the sequential time divided by the number of coarse-grained multicomputer 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 [11] and later used to solve a variety of problems [2, 3, 11, 13]. It represents a significant advancement over the multisearch method described in [13] in that the lower dimensional substructures pointed to by each node of T are of non-constant size and queries that must visit several neighbors of a node of T can split 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 a *hat* composed of its top few levels and a forest of sub-trees which are hooked to the hat (see Fig. 3). These sub-trees have a range tree structure.
- Make p copies of the hat and distribute them, one to each processor.
- Concurrently in all processors and for the entire set of queries, perform a sequential range search on the hat. The queries will end up in its leaves, each of which corresponds to a specific pendant.
- Create multiple copies of those pendants in T for which too many queries need access, and distribute the copies to processors, along with the queries that need access to that pendant. Again, each processor is responsible for advancing its subset of the "congested" queries by performing a sequential range search. It should be noted that the straightforward strategy

[1] In the special case of associative functions with inverses, the problem can be solved using weighted dominant counting [13].

of making multiple copies of T, and using one copy for each group of n/p 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.

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 queries advance through T. The above description is necessarily an over simplification and 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 contribution of this paper is therefore twofold. First, we explicitly give the exact tuning of the parameters needed for efficiently constructing a distributed range tree and implementing a parallel range search. Second, our algorithms use only standard communication operations and sequential range search, implying that existing sequential code for this problem can be re-used.

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 then described in Section 3. Section 4 gives a coarse grained parallel algorithm to solve n queries in parallel with the distributed range tree.

2 The Distributed Range Tree Data Structure

In this section we first define the range search problem and we present the classsical segment tree and range tree data structures. 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 distributed range tree data structure.

2.1 The 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 rectangular 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 (report mode), 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 $\otimes$ (associative-function mode) [19].

2.2 The Classical Range Tree Data Structure

Let a $(1, n)$ *segment tree* [4] be a complete rooted binary tree with n leaves. To each node is associated a segment. The segments associated to the leaves are $[x_1, x_2)$, $[x_2, x_3)$,...,$[x_{n-1}, x_n)$ and $[x_n, x_n]$ (the last segment is reduced to a point). To each internal node is associated the segment formed by the union of the two segments associated to its children. Thus, the segment associated to the root is $[x_1, x_n]$ (see Fig. 1). 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. Notice that, as presented in [21], the range tree structure is a d-dimensional generalization of the segment tree.

Definition 1 *The d-dimensional range tree T for a set L_d of points of E^d is recursively defined as follows.*

i) *A primary segment tree T^* in dimension d corresponding to the set $\{x_1(l) | l \in L_d\}$.*

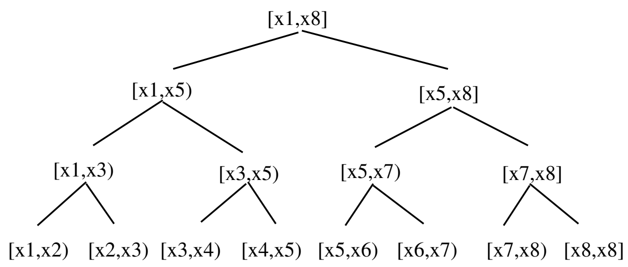

Figure 1: The segment tree structure for (1,8).

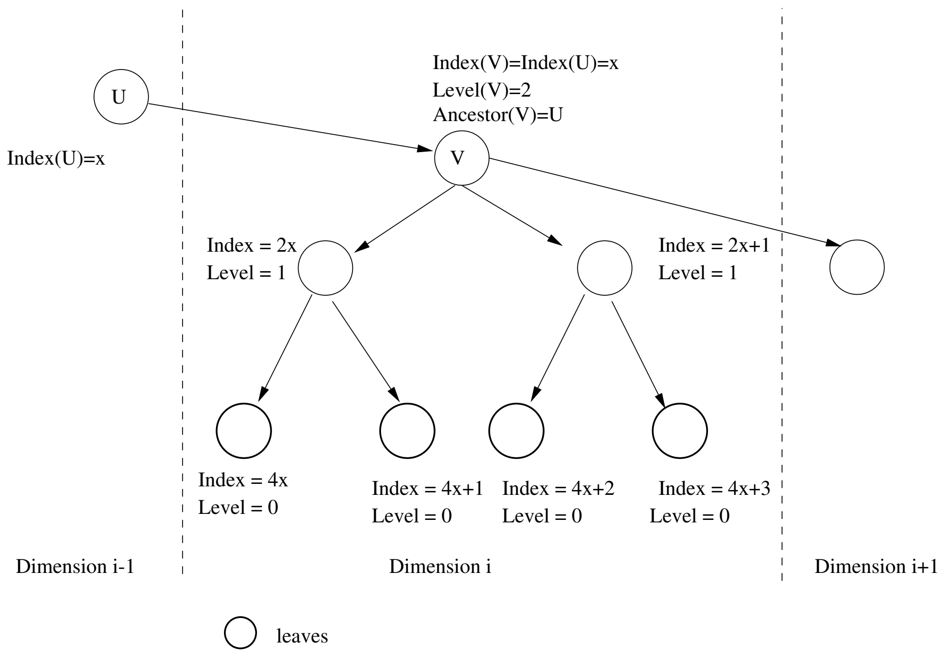

Figure 2: Illustration of *Index* and *Level* of a node of T.

ii) *For each node v of T^*, we define the $(d-1)$-dimensional set*
$L_{d-1}(v) = \{(x_2(l), ..., x_d(l)) | x_1(l)$ *lies in the interval associated to* $v\}$.
Each node v has a pointer to a range tree for $L_{d-1}(v)$ which is called descendent(v). For each node w in the primary segment tree of descendent(v), we define ancestor$(w) = v$.

2.3 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 quantities (see Fig. 2).*

i) *Level(v) is the length of the shortest path from v to a leaf in the segment tree containing v (or 0 if v is a leaf).*

ii)

$$Index(v) = \begin{cases} 0 & \textit{if } v \textit{ is the root of } T^* \\ Index(ancestor(v)) & \textit{if } v \textit{ is a root of any segment tree except } T^* \\ 2 \times Index(parent(v)) & \textit{if } v \textit{ is a left child in a segment tree} \\ 2 \times Index(parent(v)) + 1 & \textit{if } v \textit{ is a right child in a segment tree.} \end{cases}$$

iii) *Path-index*$(v) = \langle level(v), index(v) \rangle$

iv)

$$Path(v) = \begin{cases} path\text{-}index(v) & \textit{if } v \textit{ is a node of } T^* \\ \langle path(ancestor(v)), path\text{-}index(v) \rangle & \textit{otherwise.} \end{cases}$$

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

Proof: It is easy to see that for all nodes $v \in T$, path(v) is unique. Furthermore, by Definition 1, 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. Moreover, it is easy to see that all nodes on the same level of t have distinct indices, and so $\langle level(v), index(v) \rangle$ uniquely determines v within t. □

2.4 The Distributed Range Tree

Our range search algorithm is based on a distributed representation of a range tree. A range tree T for a set of n points is of size s, which is as large as the total memory available on our CGM(s,p), i.e. a p processor coarse grained multicomputer with $O((n \log^{d-1} n)/p)$ memory per processor. 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 in such a way that each processor stores no more than $O(1)$ such structures. This requires preliminary definitions.

Definition 3 *Given a range tree T,*

i) *Let the "hat" H denote the subtree of T induced by all nodes v of T which are either in the top $\log p$ levels of T, i.e. with level(v) $\geq \log n - \log p + 1$, or such that level(v)$= \log n - \log p$ and the parent of v lies in the same dimension as v.*

ii) *Let F denote the forest of subtrees of T induced by all nodes v of T with level(v) $\leq \log n - \log p$.*

iii) *For each range tree t in F with root r, let location(t)= index(r).*

Note that each element of this forest F is a range-tree on n/p points and has dimension $j \in [1, d]$ (see Fig. 3)[2]. The roots of the trees of F are the nodes v of the hat H with $level(v) = \log n - \log p$.

Note that the locations are in the range $[0, p-1]$. 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 $\log p + 1$ levels of the tree and the forest consists of the p subtrees rooted at level $\log n - \log p$ (See [14]).

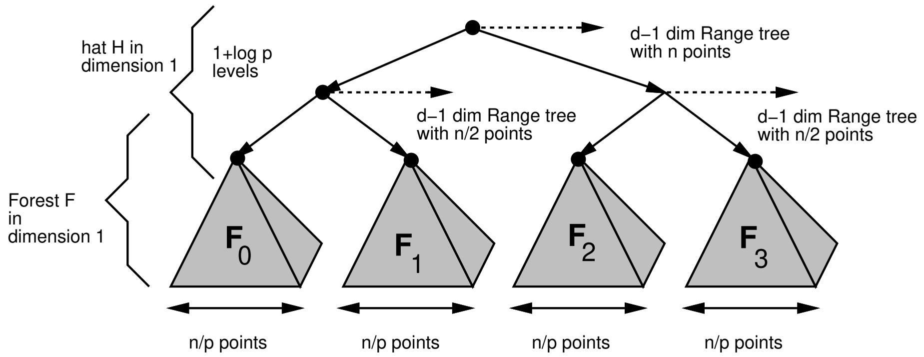

Figure 3: The hat of T in dimension 1, along with the associated part of F, for $p = 4$.

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

Proof:

i) Immediate from the fact that the hat can be seen as a range-tree on p items with a few missing pointers on the bottom level.

ii) For each $i \in [0, p-1]$, F_i consists of a set of range trees of various dimensions (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 $O(s)$. Notice that the path associated to the nodes do not asymptotically increase the size of the structure since each path is just the concatenation of at most d addresses.

□

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 Theorem 1, both H and the F_i fit in a single processor's memory.

3 Constructing a Distributed Range Tree

In this section we describe a parallel algorithm for constructing the distributed range tree defined in subsection 2.4.

In [21], a sequential algorithm is presented to build a d-dimensional range tree of size $O(n \log^{d-1} n)$ in optimal time $O(n \log^{d-1} n)$. The algorithm works in d phases, in a bottom-up fashion in which segment trees are built up from their leaves one dimension after another.

The distributed range tree is also constructed in d phases $j = 1, 2, \ldots, d$. At the start of phase j, we have a set S^j of records representing the leaves of the j-dimensional range trees of F. These range trees must now be constructed. More precisely, a record in S^j contains two informations: the coordinates of a point $l = (x_1(l), \ldots, x_d(l))$ from the original point set L, and a label path(l). Each label "x=path(l)" corresponds to a unique j-dimensional range tree of F, and conversely each such range tree will be constructed from the subset of points of S^j which have label x.

In phase j we first distribute the data: S^j is sorted according to (path,x_j-coordinates) so that records which need to go to the same j-dimensional tree of F are consecutive and stored on the same processor. Thus a tree of F corresponds to a block of n/p consecutive values of S^j, and its location can be computed by a global rank of the blocks. The blocks are then routed to their location.

The j-dimensional range trees of F are then locally constructed. Each tree t of F knows the minimum $\min(t)$ x_j-coordinate of its records.

Since the roots of the range trees of F are also the leaves of H, it suffices to perform an all-to-all broadcast of these minima in order to complete the construction of the hat in dimension j.

Then, the set S^{j+1} is constructed by appropriate duplication of elements of S_j. (The location of the j-dimensional trees of F ensures that the workload is balanced).

The algorithm is detailed below.

Algorithm Construct

Input: Each processor P_i stores a set of n/p points of L.
Output: Each processor P_i stores
 i) A copy of H
 ii) The set F_i.

0 Creation of S^1. Each processor creates for each point l a record of S^1 with two fields: $(x_1(l), ..., x_d(l))$ and path = nil. Let $j \leftarrow 1$.

1 Global sort. Globally sort S^j by primary key path and secondary key x_j.

2 Routing to locations. Each processor P_i divides its set into blocks of n/p consecutive records, computes the global rank of each block and routes the k^{th} block to processor $P_{k \bmod p}$.

3 Local range tree construction. Each processor P_i sequentially constructs the j-dimensional range trees of F_i.

4 Broadcast tree intervals All processors perform an all-to-all broadcast of $\min(t)$ of each tree t built in step 3.

5 Local hat completion. Each processor receives $O(p \log^{d-1} p)$ min values and locally completes its own copy of H in dimension j.

6 If $j = d$ then exit.

7 Local construction of S^{j+1}. Each record $z \in S^j$ stored in processor P_i belongs to a j dimensional range tree hooked to a leaf y of H. For all z, walk in H 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: $x_1(s), \ldots, x_d(s) = x_1(z), \ldots, x_d(z)$ and path(s)=path(u).

8 End of loop. $j \leftarrow j+1$. Goto step **1**.

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

Proof: The correctness of Algorithm Construct follows from the sequential construction algorithm in [21], Definitions 1 and 3, Lemma 1, and Theorem 1. Step 0 takes parallel time $O(n/p)$. Steps 1 through 8 are executed d times. In each phase, steps 1,2 and 4 involve global communications and take time $O(T_c(s,p))$. Steps 3,5,6,7,8 involve local computations only. Step 3 takes time $O(s/p)$ from [21]. Step 5 takes time $O(p)$. Step 7 takes time $O(s/p)$. Steps 6 and 8 take time $O(1)$. The overall time complexity is $O(s/p + T_c(s,p))$. $\square$

This theorem and the weak-CREW BSP sorting algorithm from [17] 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 [21], 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 2. 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 $c(F_j) = \lceil |\hat{Q}_{F_j}|/(|\hat{Q}|/p) \rceil$.

2 Make $c(F_j)$ copies of F_j and distribute them evenly such that each processor stores at most two forests.

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 $\bar{Q}$.

Note that this algorithm depends on a load balancing phase, implemented in Steps **1** through **3**, which evenly distributes queries and forests F_i, such that each processor has $O(1)$ copies of each. This approach to load balancing, which is used here and in Algorithm Report given below, is based on a CGM technique described and analyzed in [13].

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 $O(\frac{s \log n}{p} + T_{sort}(s,p))$.*

Proof: The correctness of Algorithm Search follows from the sequential construction algorithm [21] and the following three observations. First, all forests $F_0, F_1, \ldots, F_{p-1}$ have size $O(s/p)$ (Theorem 1). Then, the total number $\sum_{j=0}^{p-1} c(F_j)$ of all forest copies created in Step 1 is $O(p)$. Finally, in Step 3, the number of queries moved to each processor is $O(s/p)$. The space requirement is $O(\frac{s}{p} + p) = O(p)$. In each step, the local computation time is at most $O(\frac{s}{p} \log n)$. The global communication in each step reduces to a constant number of global sorts and communication operations (see [13]). Hence, the time complexity of Algorithm Search is $O(\frac{s \log n}{p} + T_{sort}(s,p))$.
$\square$

As in the previous section, combining this result with the weak-CREW BSP sort presented in [17] 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 $O(\frac{s \log n}{p} + T_{sort}(s,p))$ and $O(\frac{s \log n}{p} + T_{sort}(s,p) + \frac{k}{p})$, 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 [13].

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 [13]. 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 $O(\frac{s \log n}{p} + T_{sort}(s,p))$ and $O(\frac{s \log n}{p} + T_{sort}(s,p) + \frac{k}{p})$, respectively, where k is the number of results to be reported.*

Again, considering the weak-CREW BSP sort presented in [17] 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 defined a distributed range tree, an adaptation of range trees in the parallel distributed memory setting. This data structure allows optimal batched range search operations, in associative-function or in report mode. 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. Not only should the implementation on any variety of multicomputer should be relatively easy for a range tree expert, but also existing optimized sequential code can be re-used.

With respect to further research in this area, many avenues may be explored. For instance, the construction algorithm uses 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. In addition, answering queries in batches of size n may be unsatisfactory in some applications, where n is very large. The question of using parallelism to speed up just one single query (or a few queries) is open, even in the much simpler case of segment trees, and would be worth studying.

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] A. Chan, F. Dehne, A. Rau-Chaplin. Coarse grained parallel next element search. *Proc. of the* 11^{th} *IEEE International Parallel Processing Symposium, April, Geneva, CH*, pages 320–325, 1997.

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

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] A. Ferreira, C. Kenyon, A. Rau-Chaplin, and S. Ubéda. d-dimensional range search on multicomputers. *Proc. of the* 11^{th} *IEEE International Parallel Processing Symposium, April, Geneva, CH*, pages 616–620, 1197.

[16] 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.

[17] M.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.

[18] S.E. Hambrusch 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.

[19] D.E. Knuth. Range search problem. In *The Art of Computer Programming Volume 3: sorting and searching*, Addison Wesley, Reading Mass, pages 550, 1973.

[20] 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.

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

[22] 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.

[23] 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.

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

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