Computing Intersections and Arrangements for Red-Blue Curve Segments in Parallel

Christine Rüb

MPI-I-92-108 February 1992
Computing Intersections and Arrangements for Red-Blue Curve Segments in Parallel

Christine Rüb

MPI-I-92-108 February 1992
"Das diesem Bericht zugrunde liegende Vorhaben wurde mit Mitteln des Bundesministers für Forschung und Technologie (Betreuungskennzeichen ITR 9102) gefördert. Die Verantwortung für den Inhalt dieser Veröffentlichung liegt beim Autor."
Computing Intersections and Arrangements for Red-Blue Curve Segments in Parallel

Christine Rüb
Max-Planck Institut für Informatik, Im Stadtwald, D-6600 Saarbrücken

Abstract
Let $A$ and $B$ be two sets of "well-behaved" (i.e., continuous and $x$-monotone) curve segments in the plane, where no two segments in $A$ (similarly, $B$) intersect. In this paper we show how to report all points of intersection between segments in $A$ and segments in $B$, and how to construct the arrangement defined by the segments in $A \cup B$ in parallel using the concurrent-read-exclusive-write (CREW-) PRAM model. The algorithms perform a work of $O(n \log n + k)$ using $p \leq n + k/\log n$ ($p \leq n/\log n + k/\log^2 n$, resp.,) processors if we assume that the handling of segments is "cheap", e.g., if two segments intersect at most a constant number of times, where $n$ is the total number of segments and $k$ is the number of points of intersection. If we only assume that a single processor can compute an arbitrary point of intersection between two fixed segments in constant time, the performed work increases to $O(n \log n + m(k + p))$, where $m$ is the maximal number of points of intersection between two segments. We also show how to count the number of points of intersection between segments in $A$ and segments in $B$ in time $O(\log n)$ using $n$ processors on a CREW-PRAM if two curve segments intersect at most twice.

Key Words
Parallel algorithms, Computational geometry, Curve segments, Red-blue intersection reporting, Red-blue intersection counting, Arrangement, Segment tree, PRAM.

1. Introduction

We consider the following problems: Given a set $A$ of "red" and a set $B$ of "blue" non-intersecting well-behaved curve segments in the plane, 1. report all points of intersection between segments in $A$ and segments in $B$, and 2. construct the arrangement defined by the segments in $A \cup B$, using the concurrent-read-exclusive-write (CREW) PRAM model. These problems have applications, e.g., in computer graphics and computer aided design.

The work performed by our algorithms depends on the complexity of handling segments, e.g., how much time it takes a single processor to compute the number of points of intersection between two fixed segments. This determines how efficiently the points of intersection can be distributed among the processors. If the handling of segments is "cheap", e.g., if two segments intersect at most a constant number of times (for details see Section 2), all points

---

1 This work was supported by the DFG, SFB 124, TP B2, VLSI Entwurfsmethoden und Parallelität.
of intersection can be reported using $O(n \log n + k)$ work and $p \leq n + k/\log n$ processors, and the arrangement can be constructed using $O(n \log n + k)$ work and $p \leq n/\log n + k/\log^2 n$ processors, where $n$ is the number of segments and $k$ is the number of points of intersection. This is optimal since $\Omega(n \log n)$ is a lower bound for the element uniqueness problem (cf. [DL79]). If the handling of segments is "expensive", all points of intersection can be computed using $O(n \log n + m(k + p))$ work and $p \leq n + k/\log n$ processors, and the arrangement can be constructed using the same amount of work and $p \leq n/\log n + k/\log^2 n$ processors, where $m$ is the maximal number of points of intersection between two segments. This compares to $O(n \log n + k)$ for the running time of the best known sequential algorithm for this problem, using the same assumptions (cf. [MS88]).

We are not aware of any other parallel algorithm for these problems. However, there are several algorithms for variants of them. Chow (cf. [Ch81]) developed a parallel algorithm that reports all points of intersection between a set $A$ of vertical and a set $B$ of horizontal straight line segments and runs in time $O(\log^2 n + k_{\text{max}})$ using $n$ processors on a CREW-PRAM, where $k_{\text{max}}$ is the maximal number of points of intersection per segment, and Goodrich (cf. [G89]) showed how to construct the arrangement for this input in time $O(\log n)$ using $n + k/\log n$ processors on a CREW-PRAM. Rüb then showed (cf. [R92]) how to solve the red-blue intersection reporting problem for straight line segments with arbitrary slopes within the latter time and processor bounds.

The best known parallel algorithms for reporting all points of intersection between $n$ arbitrary straight line segments and constructing the arrangement defined by them, run in time $O(\log^2 n)$ using $n + k/\log n$ processors (cf. [G89]) and in time $O(\log n \log \log n)$ using $n + k$ processors (cf. [R92]), both on the CREW-PRAM model. The best known sequential algorithm for this problem runs in time $O(n \log n + k)$ (cf. [CE88]). The problem of constructing the arrangement defined by $n$ straight lines was considered in [ABB90], [G91], and [HJW90]. Hagerup et al gave a randomized algorithm that runs in $O(\log n)$ expected time using $n^2/\log n$ processors, and Goodrich gave a deterministic algorithm with the same time and processor bounds.

Our algorithms do not need to know the number $k$ of points of intersection and the maximal number $m$ of points of intersection between two segments in advance. Rather, $k$ and $m$ are determined during the execution of the algorithms and additional processors are requested if necessary. This is done only a constant number of times and thus no spawning (cf. [G89]) of processors is necessary.

Additionally we show how to count the number of points of intersection between segments in $A$ and segments in $B$ in time $O(\log n)$ using $n$ processors if two segments intersect at most twice. Since $\Omega(n \log n)$ is a lower bound for the element uniqueness problem (cf. [DL79]), this is optimal.

This paper is organized as follows: in Section 2 we define curve segments, in Section 3 we define the data structure we use in our algorithms and give some basic lemmas, and Section 4
contains the algorithms.

2. Basic Definitions

The segments that we consider in this paper are "well-behaved" curve segments. A well-behaved curve segment $s$ is the graph of a function $f_s(z)$ that is defined on some closed interval $\text{dom}(f_s)$ of the x-axis and continuous on $\text{dom}(f_s)$. This means that any vertical line intersects $s$ at most once. In the remainder of this paper "segment" means "well-behaved curve segment". A "point" of intersection is a tuple $(p,q,X)$ where $p$ and $q$ are segments, and $X$ is a maximal interval of the x-axis such that $X \subseteq \text{dom}(f_p) \cap \text{dom}(f_q)$ and $f_p(z) = f_q(z)$ for all $z \in X$. In the remainder of this paper we assume, for ease of explanation, that no two segments overlap.

The running times of our algorithms depend on the complexity of handling segments, e.g., how much time it takes a single processor to compute the number of points of intersection between two fixed segments. We consider three different models. In all three of them we assume that the following functions can be evaluated in constant time by a single processor: $f_p(z)$, where $p$ is a segment and $z \in \text{dom}(f_p)$, and $\text{intersect}(p,q)$ that is true iff segments $p$ and $q$ intersect.

When using model 1 we additionally assume that a single processor is able to compute the number of points of intersection between two segments as well as the $i$-th point of intersection from the left between two segments, if it exists, in time $O(1)$. This model allows us to distribute the points of intersection most efficiently among the processors. Note that under this assumption it is possible to compute the $d$ points of intersection between two segments sorted in time $O(1)$ using $d$ processors.

When using model 2 we assume that a single processor is able to compute the number of points of intersection between two segments in time $O(1)$ and the $d$ points of intersection between two segments in time $O(d)$. Note that under this assumption the $d$ points of intersection can be computed sorted in time $O(d)$ by a single processor: after computing a point of intersection $s$, simply clip the parts of the segments lying to the right of $s$ and count the number of points of intersection between these two segments. In the same way, given a point of intersection $s$ between two segments, a single processor can compute the rank of $s$ in the sorted list of all points of intersection between the two segments in time $O(1)$.

When using model 3 we only assume that a single processor can compute an arbitrary point of intersection between two segments in time $O(1)$. Note that under this assumption a single processor is able to compute the $d$ points of intersection between two segments sorted in time $O(d)$.

The arrangement of a set $L$ of segments in the plane is the planar map whose vertices are the endpoints of the segments and the points of intersection between them, whose edges are
maximal connected subsegments of the segments not containing any vertex, and whose faces are the connected components of the complement of the union of the segments.

3. The Plane-Sweep Tree and Some Lemmas

In this section we introduce the plane-sweep tree, i.e., the data structure that allows us to find segment intersections efficiently in parallel. The plane-sweep tree is an extension of the segment tree (cf., e.g., [M84]). It was used by Chazelle et al (cf. [CEGS89]) to count sequentially the number of points of intersection between two sets of non-intersecting straight line segments, by Aggarwal et al (cf. [ACGOY88]) and by Atallah et al (cf. [ACG89]) for line segment intersection detection in parallel, and by Goodrich (cf. [G89]) and Rüb (cf. [R92]) for line segment intersection reporting in parallel. This section also contains some lemmas that we will use in Section 4.

![Plane-Sweep Tree Diagram](image)

**Definition** plane-sweep tree (cf. Fig. 1)

Let \( S = \{l_1, ..., l_n\} \) be a set of well-behaved non-intersecting curve segments and let \( U = \{z_1 < z_2 < ... < z_r\} \subseteq \mathbb{R} \), called the universe, contain the x-coordinates of the endpoints of the segments in \( S \).

A plane-sweep tree \( \text{PST} \) for \( S \) with universe \( U \) consists of a balanced binary tree with \( 2r + 1 \) leaves. Each node \( v \) has associated with it an interval \( I_v \): the leaves from left to right are associated with the intervals \((-\infty, z_1),[z_1, z_1), (z_1, z_2), ..., (z_r, +\infty)\), and every internal node is associated with the union of the intervals of its children. Let \( \Pi_v = I_v \times (-\infty, +\infty) \) be the vertical strip thereby assigned to node \( v \). In addition, every node \( v \) has associated with it a sequence \( H(v) \) and a set \( W(v) \) of segments from \( S \), defined as follows:

\[
W(v) = \{l \in S | l \text{ has an endpoint in } \Pi_v \text{ and does not span } \Pi_v\},
\]
\(H(v) = \{l \in S | l \text{ spans } \Pi_v \text{ but not } \Pi_{\text{parent}(v)}\}\).

The segments in \(H(v)\) are sorted according to the natural ordering among them.

Often we are only interested in the portion of a segment inside the vertical strip of some node. Thus we introduce the following notation: Let \(N(v)\) be some set of segments assigned to a node \(v\) of a plane-sweep tree. Then \(\tilde{N}(v) = \{l \cap \Pi_v | l \in N(v)\}\). We call the elements of \(\tilde{N}(v)\) fragments of the segments in \(N(v)\).

The following lemma gives a bound for the size of a plane-sweep tree and demonstrates how we can use this tree to compute segment intersections.

Lemma 1

a) Let \(PST\) be a plane-sweep tree for a set \(S\) of \(n\) segments with a universe \(U\) of size \(r\). Then the depth of \(PST\) is \(O(\log r)\) and \(\sum_{v \in PST} (|H(v)| + |W(v)|) = O(n \log r)\).

b) Let \(A\) and \(B\) be two sets of segments in the plane and let \(U\) contain all x-coordinates of endpoints of elements in \(A \cup B\). Let \(PST\) be a plane-sweep tree for \(A \cup B\) with universe \(U\) and denote the subsets of the \(H\)- and \(W\)-sets in \(PST\) that consist of segments in \(A\) (\(B\), resp.,) by \(H^A\) and \(W^A\) (\(H^B\) and \(W^B\), resp.). Suppose that a segment \(p \in A\) intersects a segment \(q \in B\) at a point \(d\). Then there exists exactly one node \(v\) in \(PST^A\) such that \(d \in \Pi_v\) and either

(i) \(p \in H^A(v), \ q \in H^B(v)\) or
(ii) \(p \in H^A(v), \ q \in W^B(v)\) or
(iii) \(p \in W^A(v), \ q \in H^B(v)\).

Proof: See, e.g., [R92].

From Lemma 1 we can conclude that we can use plane-sweep trees to report or count all points of intersection and that we find every point of intersection exactly once when doing this since the three cases above are mutually exclusive.

In [R92] it was shown that a plane-sweep tree for a set of non-intersecting straight line segments can be constructed efficiently in parallel. The same proof can be used for well-behaved curve segments:

Lemma 2 Building a Plane-Sweep tree

Let \(S\) be a set of non-intersecting segments and let \(U \subseteq \mathbb{R}\) contain all x-coordinates of endpoints of segments in \(S\). Let \(n = |S| + |U|\).

A plane-sweep tree \(PST\) for \(S\) with universe \(U\) can be built in time \(O(\log n)\) by \(n\) processors on a CREW-PRAM.

Proof: See proof of Theorem D in [R92].

The following lemmas will be used in Section 4.
Lemma 3  Merging
Given two sorted sequences $A$ and $B$ (each stored in an array) with a total of $n$ elements, the sorted sequence $C = A \cup B$ and the rank in $B$ of each element in $A$ and vice versa can be computed in time $O(\log \log n)$ with $n/\log \log n$ processors on a CREW-PRAM.

Proof: See [K83].

Lemma 4  Parallel Prefix Computation
Let $z_1, \ldots, z_n$ be $n$ elements of a semi-group with the associative operation $+$. Let $z_1, \ldots, z_n$ be stored in this order in an array.
Then the $n$ sums $\sum_{j=1}^{i} z_j, 1 \leq i \leq n$, can be computed in time $O(\log n)$ with $n/\log n$ processors on an EREW-PRAM.

Proof: See, e.g., [A89].

Lemma 5  List Ranking
Given a linked list of length $n$, for each element its distance to the head of the list as well as this head can be computed in time $O(\log n \log^* n)$ using $n/(\log n \log^* n)$ processors on a EREW-PRAM.

Proof: See [CV86].

Lemma 6  Fractional Cascading
Let $G = (V, E)$ be a directed acyclic graph of bounded degree where every node $v \in V$ contains a sorted list $C(v) \subseteq U(v)$. The universe of $v$, denoted $U(v)$, is totally ordered and $U(v) \supseteq U(w)$ if $(v, w) \in E$. Let $n = |V| + |E| + \sum_{v \in V} |C(v)|$.
Then it is possible to construct in time $O(\log n)$ with $n/\log n$ processors on a CREW-PRAM a data structure $\hat{G}$ of size $O(n)$ with the following property: Each node $v$ in $\hat{G}$ has assigned a list $M(v)$ such that, given the position of an element $z$ in $M(v)$, a single processor can compute the position of $z$ in $C(v)$ and in $M(w)$ in constant time, where $(v, w) \in E$.

Proof: See [ACG89].

4. The Algorithms
We will prove the following theorem.

Theorem
Let $A$ (resp., $B$) be a set of non-intersecting well-behaved curve segments, let $n = |A| + |B|$, let $k$ be the number of points of intersection between segments in $A$ and segments in $B$, and let $m$ be the maximal number of points of intersection between a segment in $A$ and a segment in $B$.

a) All points of intersection between segments in $A$ and segments in $B$ can be computed by $p \leq n+k/\log n$ processors on a CREW-PRAM using $O(n \log n + k)$ space and $O(n \log n + k)$ time.
work under model 1, $O(n \log n + k + mp)$ work under model 2, and $O(n \log n + m(k+p))$ work under model 3.

b) The arrangement of $A \cup B$ can be computed by $p \leq n/\log n + k/\log^2 n$ processors on a CREW-PRAM using $O(n \log n + k)$ space and $O(n \log n + k)$ work under model 1, $O(n \log n + k + mp)$ work under model 2, and $O(n \log n + m(k+p))$ work under model 3.

c) If every segment in $A$ intersects every segment in $B$ at most twice, the number of points of intersection between segments in $A$ and segments in $B$ can be computed in time $O(\log n)$ by $n$ processors on a CREW-PRAM, using $O(n \log n)$ space.

Our algorithms are all based on lemma 1. This means that we proceed in three or four steps as follows.

Step 1: First we compute the set $U$ containing the $x$-coordinates of all endpoints of segments in $A \cup B$ and build up a plane-sweep tree $PST$ for $A \cup B$ with universe $U$. While doing this we only compute the subsets of the $H$- and $W$- sets in $PST$ that consist of segments in $A$ ($B$, resp.). We call these sets $H^A$ and $W^A$ ($H^B$ and $W^B$, resp.). According to lemma 2, this can be done in time $O(\log n)$ by $n$ processors.

Step 2: In this step we compute, for all nodes $v$ in $PST$ and for all fragments $l \in \tilde{H}^B(v) \cup \tilde{W}^B(v)$, the neighbours of $l$ (the neighbours of the endpoints of $l$, resp.) in $\tilde{H}^A(v)$, i.e., the fragments in $\tilde{H}^A(v)$ that lie directly above and below $l$ (the endpoints of $l$, resp.). For the fragments of segments in $A$ we compute corresponding neighbours. In Section 4.1 we show how this can be done in time $O(\log n)$ by $n$ processors.

Step 3: In this step we compute or count the points of intersections. Essentially we distribute the work equally among the processors using the information gathered in step 2 (cf. Section 4.2).

Step 4: Finally we construct the arrangement of $A \cup B$, using the information gathered in step 3 (cf. Section 4.3).

4.1 Computing Neighbours

We show here how to compute neighbours for fragments of segments in $B$. For each node $v$ in $PST$ and each fragment $l \in \tilde{H}^B(v) \cup \tilde{W}^B(v)$ with left endpoint $p_l(l)$ and right endpoint $p_r(l)$, let $low(l)$ ($low(p_l(l))$, $low(p_r(l))$, resp.) be the rank of the highest fragment in $\tilde{H}^A(v)$ that lies entirely below $l$ ($p_l(l)$, $p_r(l)$, resp.), and let $high(l)$ be the rank of the lowest fragment in $\tilde{H}^A(v)$ that lies entirely above $l$ (cf. Fig. 2). We prove the following two claims.

Claim 1
For all nodes $v$ in $PST$ and all fragments $l \in \tilde{H}^B(v)$ with endpoints $p_l(l)$ and $p_r(l)$, $low(l)$, $high(l)$, $low(p_l(l))$, and $low(p_r(l))$ can be computed in time $O(\log n)$ by $n$ processors.

Proof: We show here how to compute $low(l)$ for each fragment $l \in \tilde{H}^B(v)$ for all nodes $v$. 7
To do this we assign \( \lceil (|H^A(v)| + |H^B(v)|)/\log n \rceil \) processors to each node \( v \) in \( PST \). The processors assigned to a node \( v \) then merge \( H^A(v) \) and \( H^B(v) \), comparing a fragment \( l_a \in H^A(v) \) and a fragment \( l_b \in H^B(v) \) as follows: \( l_a < l_b \) iff \( l_a \) lies entirely below \( l_b \) (appropriate orderings are used to compute the other low- and high-values). At the same time the processors assigned to \( v \) compute \( low(l) \) for each \( l \in H^B(v) \). Since \( \sum_{v \in PST} (|H^A(v)| + |H^B(v)|) = O(n \log n) \) and \( PST \) has \( O(n) \) nodes, this can be done in time \( O(\log n) \) by \( n \) processors.

Fig. 2

Claim 2
For all nodes \( v \in PST \) and all fragments \( l \in H^B(v) \) with endpoints \( p_l(l) \) and \( q_r(l) \), \( low(l) \), \( high(l) \), \( low(p_l(l)) \), and \( low(p_r(l)) \) can be computed in time \( O(\log n) \) by \( n \) processors.

Proof: We show here how to compute \( low(l) \) for all nodes \( v \) and all fragments \( l \in H^B(v) \). We use fractional cascading and proceed in 3 steps as follows.

Step 1: First we turn \( PST \) together with the \( H^A \)-sequences into a fractional cascading data structure. To do this we direct all edges in \( PST \) towards the root and define \( U(v) = \{ s \in A | s \) spans \( \Pi_v \} \) for each node \( v \). Then we apply lemma 6 (fractional cascading) to \( PST \) and the \( H^A \)-sequences. Since \( \sum_v |H^A(v)| = O(n \log n) \), this can be done in time \( O(\log n) \) by \( n \) processors. Now each node \( v \) in \( PST \) has a list \( M(v) \) assigned to it, cf. lemma 6.

Step 2: Now we assign \( \lceil (|M(v)| + |H^B(v)|)/\log n \rceil \) processors to each node \( v \). For each fragment \( l \in H^B(v) \) the processors assigned to a node \( v \) compute, with the help of merging as in Claim 1, the nearest neighbour from below of \( l \) in \( M(v) \), i.e., the highest fragment in \( M(v) \) that lies entirely below \( l \). Since \( \sum_v (|M(v)| + |H^B(v)|) = O(n \log n) \), this can be done in time \( O(\log n) \) by \( n \) processors. Since for each segment \( s \in B \) there exists at most one left (right) node \( v \) on each level in \( PST \) where \( s \in H^B(v) \), we can store this information for each segment in an array of length \( 2 \)\( \text{depth}(PST) \). We will need this information in step 3.

Step 3: Now we assign one processor to each segment \( l \in B \) that computes \( low(l) \) for all fragments \( l_b \) of \( l \) where \( l_{b} \in H^B(v) \) for a node \( v \). Let \( l \) be fixed. The nodes in \( PST \)
where \( l \) is contained in their \( W \)-sets lie on at most two paths in \( PST \). Let \( z_1 \) and \( z_2 \) be the lowest nodes on these paths. The processor assigned to \( l \) first computes \( z_1 \) and \( z_2 \), and then the nearest neighbour from below of \( l_{z_1} \) (\( l_{z_2} \), resp.,) in \( M(z_1) \) (\( M(z_2) \), resp.,) \( \text{low}(l_{z_1}) \), and \( \text{low}(l_{z_2}) \) with the help of binary search. Then it follows simultaneously the paths from \( z_1 \) and \( z_2 \) to the root of \( PST \) (treating nodes with greater depth first) and computes the \( \text{low} \)-values for the nodes on these paths. When moving from a node \( v \) to its parent \( w \), we distinguish two cases.

Case 1: \( l_v = l_w \) (cf. Fig. 3a)
We know the nearest neighbour from below of \( l_v \) in \( M(v) \). Following lemma 6, the processor assigned to \( l \) now computes the nearest neighbour from below of \( l_w (= l_v) \) in \( M(w) \) and \( \text{low}(l_w) \) in time \( O(1) \).

Case 2: \( l_v \neq l_w \) (cf. Fig. 3b)
Let \( u \) be the sibling of \( v \). Then \( l \cap \Pi_u \neq \emptyset \) and thus, since \( l \) does not span \( \Pi_w \), either \( l \in W^B(u) \) or \( l \in H^B(u) \) (cf. Fig. 3b). In the first case the processor assigned to \( l \) has already computed the neighbour from below of \( l_u \) in \( M(u) \), and in the second case this neighbour was computed in step 2. Thus the processor assigned to \( l \) can compute the neighbours from below of \( l_v \) and of \( l_u \) in \( M(w) \) (cf. lemma 6) in time \( O(1) \). Since \( l_w = l_u \cup l_v \), the nearest neighbour from below of \( l_w \) in \( M(w) \) is the lower of these two. Now the processor assigned to \( l \) can compute \( \text{low}(l_w) \) in time \( O(1) \).

Since \( \text{height}(PST) = O(\log n) \), step 3 can be executed in time \( O(\log n) \) by \( n \) processors.

4.2 Computing and Counting Intersections
We show how to count and how to compute all points of intersection in the following representation (cf. Fig. 4).
W.l.o.g. let $l \in B$. For each node $v$ in $PST$ where $l \in H^B(v) \cup W^B(v)$ we compute a list $\text{IntSeg}(l,v)$ that contains all segments $h \in H^A(v)$ that intersect $l$ in $\Pi_v$, sorted according to their rank in $H^A(v)$. Each entry in this list, e.g. for a segment $h \in A$, points to a list $\text{IntPoint}(l,v,h)$ containing the ordered list of all points of intersections between $l$ and $h$ in $\Pi_v$. At each list we additionally store its length and backward pointers, and all $\text{IntPoint}$-lists are contained continuously in an array of size $k$.

Let $l \in \tilde{H}^B(v) \cup \tilde{W}^B(v)$ for a node $v$. Then $l$ intersects exactly those segments in $\tilde{H}^A(v)$ with rank greater than $\text{low}(l)$ and smaller than $\text{high}(l)$ (cf. Fig. 2). Thus we can proceed as follows.

We assign one processor to each fragment $l \in \tilde{H}^B(v) \cup \tilde{W}^B(v)$ for all nodes $v$. The processor assigned to a fragment $l$ decides whether $l$ intersects a fragment in $\tilde{H}^A(v)$, using $\text{low}(l)$ and $\text{high}(l)$. If this is the case, it produces a tuple $(v, \text{low}(l), \text{high}(l), l)$. This step produces $O(n \log n)$ tuples altogether, where each tuple $(v, \text{low}(l), \text{high}(l), l)$ represents $\text{high}(l) - \text{low}(l) - 1$ pairs of intersecting fragments. Assume that the tuples are given in an array in the order $(v_1, \text{low}_1, \text{high}_1, l_1), (v_2, \text{low}_2, \text{high}_2, l_2), (v_3, \text{low}_3, \text{high}_3, l_3), ..., (v_c, \text{low}_c, \text{high}_c, l_c)$.

To compute all points of intersection, we distribute the pairs of intersecting fragments equally among the processors with the help of a parallel prefix computation and binary search. This can be done in time $O(\log n)$ by $n + k/\log n$ processors. The rest of this step depends on the model we use.

If it is possible for a single processor to compute the number of points of intersection between two curve segments in $O(1)$ time (models 1 and 2), the processor assigned to a pair of intersecting fragments $l_1$ and $l_2$ then computes the number $d$ of points of intersection between $l_1$ and $l_2$ and produces a tuple $(l_1, l_2, d)$. This step produces $O(k)$ tuples altogether. To compute all points of intersection we then distribute them equally among the processors. This leads to a performed work of $O(n \log n + k)$ under model 1, and to $O(n \log n + k + mp)$ under model 2, each with $p \leq n + k/\log n$ processors.

If it is not possible to compute the number of points of intersection between two curve segments fast, each processor assigned to a pair of intersecting fragments $l_1$ and $l_2$ then

---

**Fig. 4 Representation of the intersections**
computes all $d$ points of intersection between them in $O(d)$ time. This leads to a performed work of $O(n \log n + m(k + p))$ with $p \leq n + k/\log n$ processors.

To count the number of points of intersection we proceed as follows. Since the segments intersect at most twice, each tuple $(v, \text{low}(l), \text{high}(l), l)$ can represent up to $2(\text{high}(l) - \text{low}(l) - 1)$ points of intersection. With the help of $\text{low}(p_l(l))$ and $\text{low}(p_r(l))$, a single processor can compute in time $O(1)$ how many points of intersection $(v, \text{low}(l), \text{high}(l), l)$ represents, as follows.

![Graph](image)

a) $\text{low}(p_l) = \text{low}(p_r)$

b) $\text{low}(p_l) < \text{low}(p_r)$

Fig. 5

W.l.o.g. let $\text{low}(p_l) \leq \text{low}(p_r)$. Since $l$ intersects each fragment at most twice, each fragment in $H^4(v)$ with rank $\leq \text{low}(p_r)$ and $> \text{low}(p_l)$ is intersected exactly once by $l$ (cf. Fig. 5). The remaining intersected fragments are intersected exactly twice by $l$ except, perhaps, the intersected fragments with minimal and maximal rank. Thus a single processor can determine in time $O(1)$ how many points of intersection the tuple $(v, \text{low}(l), \text{high}(l), l)$ represents, and we can compute the number of points of intersection by adding $O(n \log n)$ numbers. This can be done in time $O(\log n)$ using $n$ processors.

**Comment:** If two segments may intersect more than twice, the method used above cannot be applied. However, we can compute a lower and an upper bound for the number of points of intersection that depends on the number $k'$ of pairs of intersecting fragments: $k' \leq k \leq mk'$ and $k'$ can be computed in time $O(\log n)$ by $n$ processors on a CREW–PRAM.

4.3 Constructing the Arrangement

We show here how to construct the arrangement after all points of intersection have been computed and stored as described at the beginning of Section 4.2. What we need to compute are pointers from each point of intersection $s$ to its at most 4 neighbours on the segments defining $s$.

For each segment $l$ let $L(l)$ be the sorted list of all points of intersection on $l$, and let $t(m) = O(1)$ under models 1 and 2, and $t(m) = \log m$ under model 3. In the remainder of this section we will not explicitly refer to the used model.
According to the three clauses in lemma 1, there are three cases for a segment \( l \) and a point of intersection \( s \) on \( l \): \( s \) is defined by two \( H \)-fragments, \( s \) is defined by a \( W \)-fragment of \( l \) and an \( H \)-fragment \( h \), or \( s \) is defined by an \( H \)-fragment of \( l \) and a \( W \)-fragment \( h \). Following clause 2, we define, for each segment \( l \), a sublist \( L^{WH}(l) \) of \( L(l) \) that contains all points of intersection \( s \) between \( l \) and a segment \( q \) where \( s \in \Pi_v \), \( l \in W(v) \), and \( q \in H(v) \) for a node \( v \). In Section 4.3.1 we show how to compute \( L^{WH}(l) \) for all segments \( l \) using \( O(n \log n + \tau(m)(k + p)) \) work and \( p \leq (n + k/\log n)/\log^* n \) processors. These lists will then guide the computation of all neighbours using \( O(n \log n + \tau(m)(k + p)) \) work and \( p \leq n/\log n + k/\log^2 n \) processors, as shown in Section 4.3.2. In the remainder of this section we will always deal with pointers to the already computed points of intersection.

4.3.1 Computing \( L^{WH}(l) \)

In this section we show how to compute \( L^{WH}(l) \) for all segments \( l \in A \). We do this by first computing, for all segments \( l \in A \), the neighbours of each point of intersection in \( L^{WH}(l) \) and then using list ranking to obtain all \( L^{WH} \)-lists. The latter step can be executed in time \( O(\log n \log^* n) \) by \( (n + k/\log n)/\log^* n \) processors, so let us concentrate on the first one.

![Diagram](https://via.placeholder.com/150)

To compute the neighbours we proceed in two steps as follows.

**Step 1:** This is a preprocessing step. For each segment \( q \in B \) and each node \( v \) where \( q \in H^B(v) \cup W^B(v) \) let \( q \)'s nearest neighbours above \( v \) be the at most two fragments that are contained in \( H^B(v) \) or in the \( H^B \)-set of an ancestor of \( v \) and are nearest (from above or from below) to \( q \) among these (cf. Fig. 6). Call these neighbours \( Nbelow(q, v) \) and \( Nabove(q, v) \). In this step we compute all these neighbours in time \( O(\log n) \) with \( n \) processors. To do this we use the following observation: Let \( q \in H^B(v) \cup W^B(v) \) for a node \( v \) in \( PST \). Then
a nearest neighbour of \( q \) above \( v \) is a nearest neighbour of \( q \) above \( \text{parent}(v) \), or it is a neighbour of \( q \) in \( H^B(v) \). Thus we proceed as follows.

First we compute, for each endpoint \( p \) of a segment in \( B \), the neighbours of \( p \) in all lists \( H^B(v) \) where \( p \in \Pi_v \). This can be done in time \( O(\log n) \) by \( n \) processors with the help of fractional cascading. For each segment \( q \) and each node \( v \) where \( q \in W^B(v) \) this gives us the neighbours of \( q \) in \( H^B(v) \).

Then we assign one processor to each segment in \( B \). The processor assigned to a segment \( q \) computes the nearest neighbours of \( q \), starting at the root of \( PST \), according to the above observation. Thus all nearest neighbours above some node can be computed in time \( O(\log n) \) by \( n \) processors.

---

**Fig. 7**

Step 2: In this step we compute all \( L^{WH} \)-lists.

Let \( l \in A \), let \( s_1 \) and \( s_2 \) be neighbours in \( L^{WH}(l) \), let \( s_1 \) lie on \( h \in \widehat{H}^B(v) \), and let \( s_2 \) lie on \( q \in \widehat{H}^B(w) \). W.l.o.g., let \( \text{depth}(v) \geq \text{depth}(w) \). We distinguish 3 cases. In case 1 \( w = v \) or \( w \) is an ancestor of \( v \) and \( s_2 \in \Pi_v \), in case 2 \( w \) is an ancestor of \( v \) but \( s_2 \notin \Pi_v \) (cf. Fig. 7a), and in case 3 \( w \) is not an ancestor of \( v \) (cf. Fig. 7b). We deal with the 3 cases in reversed order.

**Case 3:** Note that this case can occur at most once for each segment \( l \in A \). This can be seen as follows. Let \( l \in A \), let \( u \) be the lowest node in \( PST \) where both endpoints of \( l \) are contained in \( \Pi_u \), and let \( g(l) \) be the vertical line that separates the strips of \( l \)'s children. Then \( s_1 \) and \( s_2 \) are the points of intersection in \( L^{WH}(l) \) that are nearest to \( g(l) \) (cf. Fig. 7b).

Thus, for all segments \( l \in A \), we can compute the two points of intersection that comply with case 3, if they exist, as follows. First we compute the vertical line \( g(l) \) for each segment \( l \in A \). This can be done in time \( O(\log n) \) by \( n \) processors altogether. Then we compute, for each segment \( l \in A \), the two points of intersection \( s_1(l) \) and \( s_2(l) \) that are nearest to \( g(l) \)
from the left (right, resp.). This can be done in time $O(\log n)$ by altogether $n + k/\log n$ processors.

What remains is to decide for each segment $l \in A$ whether $s_1(l)$ and $s_r(l)$ comply with case 3, i.e., whether $\Pi_v \cap \Pi_w = \emptyset$ where $s_1(l)$ ($s_r(l)$, resp.) is found at node $v$ ($w$, resp.). This can be done in time $O(1)$ by $n$ processors.

Case 2: Here we compute, for each segment $l \in A$, all neighbours $s_1$ and $s_2$ in $L^{WH}(l)$ where $s_1$ lies on $h \in \widehat{H}^B(v)$, $s_2$ lies on $q \in \widehat{H}^B(w)$, $w$ is an ancestor of $v$ and $s_2 \not\in \Pi_v$ (cf. Fig. 7a). W.l.o.g. we assume that $s_2$ lies to the right of $s_1$. Then $s_1$ is the rightmost point of intersection between $l$ and fragments in $\widehat{H}^B(v)$, and only $l$'s left endpoint is contained in $\Pi_v$. Thus we proceed in two steps as follows.

In step 1 we assign one processor to each segment $l \in A$ and call this processor $P_l$. Let $l$ be fixed, and let $v_1, \ldots, v_r$ be the nodes in $PST$ where $l \in W(v_i)$ and only $l$'s left endpoint lies in $\Pi_{v_i}$, $1 \leq i \leq r$, sorted according to their depth. For each $i$, $1 \leq i \leq r$, let $s_i$ be the rightmost point of intersection between $l$ and the fragments in $\widehat{H}^B(v_i)$ and let $q_i$ be the segment in $H^B(v_i)$ that contains $s_i$. In this step we compute at most two fragments for each $s_i$ such that one of these fragments contains $s_i$'s right neighbour in $L^{WH}(l)$ iff this neighbour lies to the right of $\Pi_{v_i}$.

![Fig. 8](image)

W.l.o.g. we assume that $s_i$, $1 \leq i \leq r$, exists. $P_l$ computes for $s_r, \ldots, s_1$, if necessary and in this order, the at most two fragments. While doing this we maintain the following invariant: for each $i$, $1 \leq i \leq r$, let $I_{right}$ be the rightmost point of intersection in $\{s_r, \ldots, s_{i+1}\}$. Before we treat $v_i$, $I_{right}$ is the only element in $\{s_r, \ldots, s_{i+1}\}$ for which the fragments containing its neighbour possibly still have to be computed, and if this is the case the neighbour lies to the right of $\Pi_{v_i+1}$ (cf. Fig. 8).

Assume that $P_l$ has advanced up to node $v_i$. If the fragments containing the neighbour of $I_{right}$ still have to be computed, $P_l$ tests whether $l$ intersects one of the nearest neighbours above $v_i$ of $h_i$ to the right of $\Pi_{v_i+1}$, where $h_i \in B$ contains $I_{right}$. If this is the case, the right neighbour of $I_{right}$ lies on one of these neighbours.

Next $P_l$ examines $s_i$. If $s_i$ is one of the points of intersection that complies with case 3, $P_l$ stops. Otherwise it tests whether $s_i$ lies to the left of $I_{right}$. In this case the right neighbour
of $s_i$ in $L^{WH}(l)$ lies in $\Pi_{w_i}$ and thus $s_i$ and its right neighbour do not comply with case 2 (cf. Fig. 8). If $s_i$ lies to the right of right$_i$, $P_l$ tests whether $s_i$ intersects one of the nearest neighbours of $q_i$ above $v_i$ to the right of $\Pi_{w_i}$. If this is the case, $s_i$ and the nearest point of intersection between $l$ and these comply with case 2. Thus this step can be executed in time $O(\log n)$ by a single processor per segment in $A$.

In step 2 we assign one processor to each point of intersection complying with case 2 that actually computes the point's neighbour. Using the appropriate IntPoint-lists, this can be done using $O(n \log n + t(m)(k+p))$ work and $p \leq n+k/\log n$ processors.

Case 1: Here we compute, for each segment $l \in A$, all neighbours $s_1$ and $s_2$ in $L^{WH}(l)$ where $s_1$ lies on $h \in \overline{H}(v)$, $s_2$ lies on $q \in \overline{H}(w)$, $w = v$, or $w$ is an ancestor of $v$ and $s_2 \in \Pi_v$. We will use the following observation: Under the above conditions, $q$ is a nearest neighbour of $h$ above $v$. To compute all neighbours we will employ a technique that will also be useful in Section 4.3.2. Thus we do not solve this problem directly but prove the following lemma.

Lemma 7

For each segment $l \in A$, let $\text{Nodes}(l)$ be a set of nodes in $\text{PST}$, let $S(l)$ be a set of points of intersection on $l$, and let $\text{Pre}(l) \subseteq S(l) \times S(l)$ be symmetrical, with the following properties:

(i) $S(l)$ is the set of all points of intersection between $l$ and the fragments in $\overline{H}(v)$ where $v \in \text{Nodes}(l)$.

(ii) For each ancestor $w$ of $v$ where $v \in \text{Nodes}(l)$ and $l$ intersects a fragment in $\overline{H}(w)$, $w \in \text{Nodes}(l)$.

(iii) For each node $v \in \text{Nodes}(l)$ and each fragment $q \in \overline{H}(v)$ that is intersected by $l$, we know the list $L(l,q)$ of all points of intersection between $l$ and $q$, sorted according to their $x$-coordinates and stored at $v$.

(iv) Let $\overline{S}(l)$ be the list of all points of intersection in $S(l)$, sorted according to their $x$-coordinates. Let $s_1$ and $s_2$ be neighbours in $\overline{S}(l)$, let $s_1$ lie on $h \in \overline{H}(v)$ and let $s_2$ lie on $q \in \overline{H}(w)$. Then $(s_1,s_2) \in \text{Pre}(l)$, or $s_1, s_2 \in \Pi_v \cap \Pi_w$. (The latter case means that either $h = q$ and $s_1$ and $s_2$ are neighbours in $L(l,q)$, or $q$ is a nearest neighbour of $h$ above $v$, or $h$ is a nearest neighbour of $q$ above $w$.)

(v) For no pair $(s_1,s_2) \in \text{Pre}(l)$ exists another pair $(s_3,s_2) \in \text{Pre}(l)$ where $s_3$ lies on the same side of $s_2$ as $s_1$, and given $s_2$'s entry in $L(l,q)$, where $s_2$ lies on $q$, a single processor can access $(s_1,s_2)$ in $O(1)$ time.

Then the sorted lists $\overline{S}(l)$, for all $l \in A$, can be computed using $O(n \log n + t(m)(k+p))$ work and $p \leq (n+k/\log n)/\log^* n$ processors.

Proof: Note that we do not necessarily know the sets $\text{Nodes}(l)$ and $S(l)$ for each segment $l \in A$. First we compute, for all segments $l \in A$, the neighbours of each element in $\overline{S}(l)$, as shown below. Then we use list ranking to compute all $\overline{S}$-lists in additional time $O(\log n \log^* n)$ with $(n+k/\log n)/\log^* n$ processors.
We compute the neighbours in 3 steps. In step 1 we compute, for all \( l \in A \) and all \( s \in S(l) \), "candidates" \( c_{\text{left}}(s, l) \) and \( c_{\text{right}}(s, l) \) for \( s' \) left (right, resp.,) neighbour in \( S(l) \). In step 2 we then take the Pre-sets into consideration, and in step 3 we finally compute the neighbours. We first describe the three steps and then show that their execution computes all neighbours.

**Step 1: Computing Candidates**

For each segment \( l \in A \), we assign one processor to each point of intersection in \( S(l) \). Let \( s \in S(l) \) lie on \( h \in \tilde{H}^B(v) \). The processor assigned to \( s \) examines \( h \)'s nearest neighbours \( h_1 \) and \( h_2 \) above \( v \) and computes the at most 6 points of intersection in \( L(l, h) \), \( L(l, h_1) \) and \( L(l, h_2) \) that are nearest to \( s \). This can be done in time \( O(t(m)) \). Then the point of intersection among these that is nearest to \( s \) from the left (right, resp.,) is assigned to \( c_{\text{left}}(s, l) \) (\( c_{\text{right}}(s, l) \), resp.,).

**Step 2: Pre-sets**

In this step we take the Pre-sets into consideration. We assign one processor to each pair \((s_1, s_2) \in \text{Pre}(l)\) for all \( l \in A \). Let \((s_1, s_2) \in \text{Pre}(l)\) and let, w.l.o.g., \( s_1 \) lie to the left of \( s_2 \). The processor assigned to \((s_1, s_2)\) replaces \( c_{\text{right}}(s_1, l) \) with \( s_2 \) if \( s_2 \) lies between \( s_1 \) and \( c_{\text{right}}(s_2) \) (\( c_{\text{left}}(s_2) \) is treated correspondingly). Because of condition (v) there are no write conflicts. Thus step 2 can be executed in time \( O(\log n) \) by \( n + k/\log n \) processors.

![Fig. 9](image)

\( c_{\text{right}}(s_1, l) = s_3 \) at the beginning.

When the neighbour of \( s_2 \) from the left has been computed, \( c_{\text{right}}(s_1, l) \) is replaced by \( s_2 \).

**Step 3: Computing the neighbours**

In this step we finally compute all neighbours. We do this in \( \text{depth}(PST) - 1 \) steps, one for each level of \( PST \) except the root, moving from the leaves upwards. For each \( i, 1 \leq i \leq \text{depth}(PST) \), let \( k_i \) be the sum of the lengths of all lists \( L(l, q) \) where \( l \in A \) and \( q \in \tilde{H}^B(v) \) for a node \( v \) at depth \( i \).

Step 3.i, \( 1 \leq i \leq \text{depth}(PST) - 1 \):
Let \( \text{depth}(i) = \text{depth}(PST) + 1 - i \). We assign one processor to each point of intersection \( s \in L(l, q) \) where \( l \in A \) and \( q \in \mathcal{H}^B(v) \) for a node \( v \) at depth \( \text{depth}(i) \). The processor assigned to \( s \) examines \( \text{cand}_i(s, l) \) and tests whether \( s \) lies between \( \text{cand}_i(s, l) \) and \( \text{cand}_r(\text{cand}_i(s, l)) \). If this is the case, it sets \( \text{cand}_r(\text{cand}_i(s, l)) \) to \( s \) (cf. Fig. 9). A similar rule is applied to \( \text{cand}_r(s, l) \) and \( \text{und}_r(\text{und}_r(s, l)) \).

Thus step 3 can be executed in time \( O(\sum_{i=1}^{\text{depth}(PST)}[k_i \log n/k]) \) = \( O(\log n) \) by \( n + k/\log n \) processors, where \( k = \sum k_i \).

We still have to prove that steps 1 to 3 compute all neighbours, and do this by induction on \( i \). The induction hypothesis is that after the execution of step 3.i, \( 0 \leq i \leq \text{depth}(PST) - 1 \), the neighbours are computed correctly for all points of intersection in \( L \)-lists of nodes at depth \( \geq \text{depth}(i + 1) \). This is obvious for step 3.0 (= step 2). So let \( 1 \leq i \leq \text{depth}(PST)-1 \). Assume that after the execution of step i there exists a point of intersection \( s \) at depth \( \text{depth}(i + 1) \) where one of the candidates for its neighbours, say \( \text{cand}_i(s, l) \), does not point to \( s \)'s neighbour from the left in \( \mathcal{S}(l) \). Then this neighbour is stored at a node below level \( \text{depth}(i + 1) \) (cf. condition (iv) and steps 1 and 2). But the neighbours of this point of intersection have been computed correctly before the execution of step 3.i, and taken into consideration at the latest in step 3.i. This is a contradiction. This argument also shows that there are no write conflicts in step 3.i, \( 1 \leq i \leq \text{depth}(PST) \). ■

We still have to show how to apply lemma 7 to compute all \( L^{WH} \)-lists. For each segment \( l \in A \), let \( \mathcal{S}(l) = L^{WH}(l) \), let \( \text{Nodes}(l) \) be the set of all nodes \( v \) where \( l \in W^A(v) \), let \( L(l, q) = \text{IntPoint}(l, v, q') \) where \( q \in \mathcal{H}^B(v) \), \( q \) is a fragment of \( q' \), and \( v \in \text{Nodes}(l) \), and let \( \text{Pre}(l) \) be the set of all neighbours in \( L^{WH}(l) \) that comply with cases 2 or 3. Then conditions (i) to (v) of lemma 7 are fulfilled and thus all \( L^{WH} \)-lists can be computed within the bounds claimed.

In Section 4.3.2 we will not only need the \( L^{WH} \)-lists, but also the sorted lists \( W_r(l) \) and \( W_l(l) \) for each segment \( l \in A \), defined as follows. For each segment \( l \in A \), \( W_r(l) \) (\( W_l(l) \), resp.,) is the list of all segments \( h \in B \) where \( l \in W^A(v) \) and \( h \in H^B(v) \) for a node \( v \), \( l \) and \( h \) intersect in \( \Pi_v \), and \( l \)'s right (left, resp.,) endpoint lies in \( \Pi_v \), sorted according to their y-coordinates. Additionally \( W_r(l) \) (\( W_l(l) \), resp.,) contains the lowest and the highest segment in \( B \) that lies above (below, resp.,) \( l \)'s right (left, resp.,) endpoint of \( l \) and is not intersected by \( l \). Next we show, w.l.o.g., how \( W_r(l) \), for all \( l \in A \), can be computed in time \( O(\log n) \) by \( (n + k)/\log n \) processors.

Let \( l \in A \). The lowest and highest elements in \( W_r(l) \) can be computed as follows. In Section 4.1 we computed the lowest (highest, resp.,) segment in \( H^B(v) \) that lies entirely above (below, resp.,) \( l \) for each node \( v \) where \( l \)'s right endpoint lies in \( \Pi_v \). The lowest (highest, resp.,) of these is the highest (lowest, resp.,) element of \( W_r(l) \). Thus the extremal segments in \( W_r(l) \) can be computed by comparing \( O(\log n) \) segments.
We compute the remaining segments in $W_r(l)$ with the help of the $L^{WH}$-lists as follows. First we delete all points of intersection $s$ from $L^{WH}(l)$ where $s$ lies on $h \in H^B(v)$ and only $l$'s left endpoint lies in $\Pi_v$ or $s$ is not the rightmost point of intersection between $l$ and $h$. This can be done in time $O(\log n)$ by $1 + |L^{WH}(l)|/\log n$ processors. The remaining list $L^{WH}_r(l)$ contains exactly one entry for each segment in $W_r(l)$, except the extremal ones. Let $h$ be the segment in $W_r(l)$ that is intersected furthest to the right by $l$. Next we compute two sublists $W^1_r(l)$ and $W^2_r(l)$ of $L^{WH}_r(l)$, where $W^1_r(l)$ ($W^2_r(l)$, resp.) contains the segments in $W_r(l)$ that lie below (above, resp.) $h$ with the help of a parallel prefix operation. Since the segments in $W^1_r(l)$ ($W^2_r(l)$, resp.) are sorted according to descending (ascending, resp.) y-coordinates (cf. Fig. 10), we can afterwards compute $W_r(l)$ with the help of merging. Thus $W_r(l)$ can be computed in time $O(\log n)$ by $n + k/\log n$ processors for all $l \in A$.

Before $h_3$ is intersected, $h_2$ has to be intersected

Fig. 10

4.3.2 Computing all neighbours

In this section we show how to compute the neighbours of all vertices in the arrangement of $A \cup B$ using $O(n \log n + t(m)(k + p))$ work and $p \leq n/\log n + k/\log^2 n$ processors, using the $L^{WH}$-lists as computed in the previous section. W.l.o.g., we restrict our attention to the segments in $A$.

We will use the following observation: Let $l \in H^A(v)$ and $h \in W^B(v)$ for a node $v$ in $PST$, and let $s$ be a point of intersection between $l$ and $h$ such that $s \in \Pi_v$. Then there exists exactly one descendant $w$ of $v$ where $h \in H^B(w)$ and $s \in \Pi_w$.

This leads to the following three main steps. For each node $v$ in $PST$ let $Copy(v)$ be the list of all segments $l \in A$ where $l \in H^A(w)$ for an ancestor $w$ of $v$, including $v$, and $l$ intersects a fragment in $\tilde{H}^B(v)$, sorted according to their y-coordinates. In step 1 we compute the list $Copy(v)$ for each node $v$ in $PST$. For each node $v$, each segment $l \in Copy(v)$ and each fragment $q \in \tilde{H}^B(v)$ that is intersected by $l$, we additionally compute the sorted list $L^C(l,q)$ of points of intersection between $l$ and $q$. 

18
What do we achieve by this step? Consider a segment \( l \in A \) and a point of intersection \( s \) in \( L(l) \). Then there exists exactly one node \( v \) in \( PST \) where \( l \in W^B(v) \cup \text{Copy}(v) \), and \( s \) lies on a fragment in \( \tilde{B}^B(v) \), i.e., we have divided the fragments in \( B \) that contribute to \( L(l) \) into subsets where the elements in a subset are totally ordered. We have done so by increasing the number of segments stored in \( PST \) to \( O(n \log n + k) \). Next we want to apply lemma 7 to compute all neighbours. For each segment \( l \in A \), \( L(l) = L(l) \), and \( \text{Nodes}(l) = \{ v \in PST \mid l \in W^A(v) \cup \text{Copy}(v) \} \). Additionally, \( L(l, q) = \text{IntPoint}(l, v, q) \) if \( l \in W^A(v) \), and \( L(l, q) = L^C(l, q) \) if \( l \in \text{Copy}(v) \) where \( q \in \tilde{B}^B(v) \) and \( q \) is a fragment of \( \tilde{q} \).

What remains to be done is to compute the \( \text{Pre} \)-sets such that conditions (iv) and (v) from lemma 7 are fulfilled. In step 2 this will be done using \( O(n \log n + t(m)(k + p)) \) work and \( p \leq n/\log n + k/\log^2 n \) processors. In step 3 we then apply lemma 7 to compute all neighbours.

Step 1: In this step we compute the list \( \text{Copy}(v) \) for each node \( v \) in \( PST \) and the \( L^C \)-lists for each segment in \( \text{Copy}(v) \). We show here how to compute the list \( \overline{\text{Copy}}(v) = \text{Copy}(v)/H^A(v) \) for each node \( v \). \( \text{Copy}(v) \) can then be computed with the help of merging.

For each segment \( h \in B \) and each node \( v \) where \( h \in H^B(v) \), let \( \text{Sub}(h, v) \) be the list of all points of intersection \( s \in L^{WH}(h) \) where \( s \in \Pi_v \). First we compute the \( \text{Sub} \)-lists for all segments \( h \) in \( B \). Since we know the \( L^{WH} \)-lists already, this can be done in time \( O(\log n) \) by \( n + k/\log n \) processors with the help of merging. Next we compute the list \( \overline{\text{Copy}}(v) \) for each node \( v \) in \( PST \), using the \( \text{Sub} \)-lists. Let \( l \in \overline{\text{Copy}}(v) \) for a node \( v \). Then there exists a list \( \text{Sub}(h, v) \) that contains at least one point of intersection between \( l \) and \( h \). Thus we first compress each list \( \text{Sub}(h, v) \) to a list \( \text{Copy}(h, v) \) that contains at most one point of intersection for each segment in \( A \) and is sorted according to ascending or descending \( y \)-coordinates of the corresponding segments, as follows.

For each list \( \text{Sub}(h, v) \) we compute the lowest and the highest segment that contributes a point of intersection to \( \text{Sub}(h, v) \), and the leftmost points of intersection \( s_1 \) and \( s_2 \) with these in \( \text{Sub}(h, v) \). Then we delete all points of intersection from \( \text{Sub}(h, v) \) that do not lie between \( s_1 \) and \( s_2 \). Note that each segment in \( \text{Copy}(h, v) \) is still represented in this list. Next we delete all points of intersection \( s \in \text{Sub}(h, v) \) between \( h \) and a segment \( q \) where \( s \) is not the leftmost point of intersection between \( h \) and \( q \) that lies between \( s_1 \) and \( s_2 \) (cf. Fig. 11). The remaining list contains one point of intersection for each segment in \( \overline{\text{Copy}}(v) \) that intersects \( h \), sorted according to ascending or descending \( y \)-coordinates of the segments.

Next we reverse the \( \text{Copy} \)-lists, if necessary, and concatenate for all nodes \( v \) all lists \( \text{Copy}(h, v) \) according to the rank of \( h \) in \( H^B(v) \). Then we delete all copies of segments represented in this list except the lowest one to obtain \( \overline{\text{Copy}}(v) \) for all nodes \( v \). Thus the \( \text{Copy} \)-lists can be computed in time \( O(\log n) \) by \( n + k/\log n \) processors.

All that remains is to compute the \( L^C \)-lists. With the help of the \( \text{Sub} \)-lists and the \( \text{IntPoint} \)-lists this can be done in time \( O(\log n) \) by \( n + k/\log n \) processors.

19
Step 2: In this step we compute the $Pre$-lists.

Let $l \in A$, let $s_1$ and $s_2$ be neighbours in $L(l)$, let $s_1$ lie on $h \in H^B(v)$ and $s_2$ on $q \in H^B(w)$, and let $s_1 \in \Pi_v$ and $s_2 \in \Pi_w$. Only the cases where $s_2 \not\subseteq \Pi_v$ or $s_1 \not\subseteq \Pi_w$ need our attention.

We distinguish 2 cases. In case 1 $l \in W^A(v)$ and $l \in W^A(w)$, and in case 2 $l \in Copy(v)$ or $l \in Copy(w)$. Below we show how to compute all neighbours that comply with one of these cases. While doing this we possibly compute pairs of points of intersection that do not comply with one of them, or compute "neighbours" for a point of intersection in more than one case. The first does not matter since for such a pair the second clause of (iv) in lemma 7 is fulfilled, and the second does not matter since we treat the cases one after another and can thus compute the nearest of these "neighbours" in constant time per point of intersection.

Case 1: $l \in W^A(v)$ and $l \in W^B(w)$. Then $s_1$ and $s_2$ are neighbours in $L^{WH}(l)$. Thus, for each segment $l$ in $A$, all pairs of neighbours in $L^{WH}(l)$ are inserted into $Pre(l)$.

Case 2: $l \in Copy(v)$ or $l \in Copy(w)$. This case is quite involved, and the next few pages will be concerned with it. To compute these neighbours we use the fact that in this case $s_1$ is extremal among the points of intersection of $l$ with fragments in $\tilde{H}^B(v)$, or $s_2$ among those with fragments in $\tilde{H}^B(w)$.

Thus we define for each node $v$ two lists $Search_r(v)$ and $Search_l(v)$ of $H^A$-segments that "search" for neighbours for a point of intersection on them that lies in $\Pi_v$. $Search_r(v)$ ($Search_l(v)$, resp.) contains an entry for each segment $l \in A$ where $l \in H^A(u)$ for an ancestor $u$ of $v$ and $l \in Copy(w)$ for a descendant $w$ of $v$, both times including $v$. This entry represents the rightmost (leftmost, resp.) point of intersection between $l$ and a fragment in $\tilde{W}^B(v) \cup H^B(v)$, and is associated with the segment in $W^B(v) \cup H^B(v)$ that contains this point of intersection. The segments in $Search_r(v)$ ($Search_l(v)$, resp.) are sorted according to their $y$-coordinates. For each list of segments $M$ where each segment $l \in M$ is associated with some segment $q$, we call this segment $Seg(l, M)$. 
Now let us investigate how the Search-lists can help us to find all neighbours that comply with case 2. Remember that \( l \in A \), \( s_1 \) and \( s_2 \) are neighbours in \( L(I) \), \( s_1 \) lies on \( h \in B(v) \) and \( s_1 \in \Pi_v \), and \( s_2 \) lies on \( q \in B(w) \) and \( s_2 \in \Pi_w \). W.l.o.g. we assume that \( l \in \text{Copy}(v) \) and that \( s_2 \) lies to the right of \( \Pi_v \). Then \( s_1 \) is the rightmost point of intersection between \( l \) and fragments in \( \tilde{B}(v) \cup \tilde{W}(v) \) and thus \( l \in \text{Search}(v) \) and \( l \)'s entry in \( \text{Search}(v) \) represents \( s_1 \). Let \( u \) be the ancestor of \( v \) where \( l \in A(v) \). We distinguish two cases.

Case 2.1: \( s_1 \) is the rightmost point of intersection on \( l \) in \( \Pi_u \). Then \( l \in \text{Search}(u) \) and \( l \)'s entry in this list represents \( s_1 \), and \( l \not\in \text{Search} \text{(parent}(u)) \). Moreover, there exist only \( O(\log n) \) such points of intersection for each segment \( l \). To compute all neighbours that comply with case 2.1 we thus first compute, for each node \( z \) where \( l \in A(z) \), the points of intersection on \( l \) that are represented by \( l \)'s entries in \( \text{Search}(z) \) and \( \text{Search}(z) \), if they exist. Call the sorted list of all such points of intersection \( L_{\text{C}}(l) \). If \( w \) is an ancestor of \( v \), then \( s_2 \in L_{WH}(l) \), and if \( w \) is not an ancestor of \( v \), then \( s_2 \) is the leftmost point of intersection between \( l \) and fragments in \( \tilde{B}(w) \), and \( s_2 \in L_{WH}(l) \) or \( s_2 \in L_{\text{C}}(l) \). Thus, next we merge \( L_{\text{C}}(l) \) and \( L_{WH}(l) \). Except for the computation of the Search-lists, this can be done in time \( O(\log n) \) by \( n + k/\log n \) processors for all segments \( l \in A \), and yields all pairs of neighbours that comply with this case.

Case 2.2: \( s_1 \) is not the rightmost point of intersection on \( l \) in \( \Pi_u \). We distinguish 2 cases.

Case 2.2.1: \( l \)'s entry in \( \text{Search}(u) \) still represents \( s_1 \). Since \( s_1 \) is not the rightmost point of intersection on \( l \) in \( \Pi_u \), \( w \) is an ancestor of \( u \). Thus \( q \) spans \( \Pi_u \) and \( q \cap \Pi_w \) is a nearest neighbour of \( h \) above \( u \).

To compute all neighbours that comply with this case we thus proceed as follows. For each node \( z \) where \( l \in A(z) \cap \text{Search}(z) \) we compute the nearest neighbours of \( Seg(l, \text{Search}(z)) \) above \( z \) and the point of intersection between \( l \) and these that is nearest to \( s \) from the right, where \( s \) is the point of intersection represented by \( l \)'s entry in \( \text{Search}(z) \). Besides the computation of \( L_{\text{C}}(l) \), this can be done using \( O(n \log n + t(m)(k + p)) \) work and altogether \( p \leq n + k/\log n \) processors with the help of the appropriate IntPoint-lists.

![Fig. 12](image)

Case 2.2.2: \( l \)'s entry in \( \text{Search}(u) \) does not represent \( s_1 \). Let \( z \) be the highest ancestor of \( v \)
where l’s entry in $\text{Search}_r(z)$ represents $s_1$. Then $z$ is a left child and $l \in \text{Search}_1(\text{Sibling}(z))$, or $l \in \text{Copy}(\text{parent}(z))$. W.l.o.g. we assume that both is true, and let $s_3$ be the point of intersection represented by l’s entry in $\text{Search}_1(\text{Sibling}(z))$, and $s_4$ the point of intersection between $l$ and the fragments in $\tilde{H}^B(\text{parent}(z))$ that is nearest to $s_1$ from the right. Then $s_2 = s_3$, or $s_2 = s_4$, or $q$ is a nearest neighbour of $h$ above parent($z$) (cf. Fig. 12).

To compute all neighbours that comply with this case we thus proceed as follows. For each left node $z$ and all segments $l \in \text{Search}_r(z)$ where $l \in \text{Search}_1(\text{Sibling}(z))$, or $l \in \text{Copy}(\text{parent}(z))$ and $l$ intersects a fragment in $\tilde{H}^B(\text{parent}(z))$ further to the right than $\text{Seg}(l, \text{Search}_r(z))$, we compute the point of intersection $s$ represented by l’s entry in $\text{Search}_r(z)$ and the one represented by l’s entry in $\text{Search}_1(\text{Sibling}(z))$, if it exists, as well as the point of intersection between $l$ and the fragments in $\tilde{H}^B(\text{parent}(z))$ that is nearest to $s$ from the right. The last information can be computed in time $O(1)$ by altogether $k$ processors. Using the observation above, we can then compute all neighbours that comply with case 2.2.2 using additional work $O(n \log n + t(m)(k + p))$ and $p \leq n + k / \log n$ processors. Thus we now only need to show how to compute the information given by the Search-lists.

Note that all Search-lists together may contain up to $O(k \log n)$ elements. Thus we store only compressed versions of the Search-lists, defined as follows. Let $v$ be a node in PST and let $\text{Search}_r(v)$ ($\text{Search}_1(v)$, resp.,) be divided into maximal sublists where the elements in a sublist are associated with the same segment in $W^B(v) \cup H^B(v)$. In the compressed version of $\text{Search}_r(v)$ ($\text{Search}_1(v)$, resp.,), called $C\text{Search}_r(v)$ ($C\text{Search}_1(v)$, resp.,), we store only the lowest and highest element in each sublist. The following lemma shows that the size of all CSearch-lists in PST is bounded by $O(n \log n)$.

Lemma 8 Size of the CSearch-lists
Let $v$ be a node in PST. Then $|C\text{Search}_r(v)| + |C\text{Search}_1(v)| = O(|W^B(v)| + |H^B(v)|)$.

Proof:
Let, w.l.o.g., $\text{Search}_r(v) = l_1, ..., l_s$, and let $l_i$ be associated with a segment $h_1$ and $l_{i+1}$ with a segment $h_2$, $h_1 \neq h_2$ and $1 \leq i \leq s - 1$. We distinguish 2 cases.

![Fig. 13]

Case 1: $l_{i+1}$ is not intersected by $h_1$, or $l_i$ is not intersected by $h_2$. Assume the latter.
Then $l_{i+1}$ is the lowest element in $\text{Search}_v(v)$ that is intersected by $h_2$ (cf. Fig. 13), and the minimum of $h_2$ lies between $l_i$ and $l_{i+1}$. This can happen only once for each segment in $W^B(v) \cup H^B(v)$.

**Case 2:** $l_i$ and $l_{i+1}$ are both intersected by $h_1$ and $h_2$ (cf. Fig. 14). We distinguish two cases.

**Case 2.1:** The index $i$ is maximal such that this situation occurs for $h_1$ (cf. Fig. 14a). Obviously this can happen only $O(|W^B(v)| + |H^B(v)|)$ times altogether.

**Case 2.2:** The index $i$ is not maximal (cf. Fig. 14b). Then this situation cannot occur again for $h_2$. W.l.o.g. we show that it cannot occur to the right of the point of intersection $s$ between $h_2$ and $l_{i+1}$. First we observe that each segment in $\text{Search}_v(v)$ that is intersected by $h_2$ is intersected by $h_1$ further to the left (cf. Fig. 14b). Let $h'_1$ be a segment in $W^B(v) \cup H^B(v)$ such that this situation occurs again for $h_2$, to the right of $s$ and where $h_1$ is replaced by $h'_1$. Then $h'_1$ starts above $l_{i+1}$ and, moreover, to the right of $h_2$. This is a contradiction.

**Fig. 14**

To compute the missing information, we proceed as follows. First we compute the lists $C\text{Search}_v(v)$ and $C\text{Search}_l(v)$ for each node $v$ in PST. While doing this we also gather the information needed for cases 2.1 and 2.2.1. Afterwards we compute the information needed for case 2.2, which amounts to expanding parts of the $C\text{Search}$-lists.

**Computing the $C\text{Search}$-lists**

Though the $C\text{Search}_v$- and $C\text{Search}_l$-lists are computed simultaneously, we show here only how to compute $C\text{Search}_v(v)$ for all nodes $v$ in PST using $O(n \log n + t(m)(k+p))$ work and $p \leq n/\log n + k/\log^2 n$ processors. To do this we make use of the fact that for an inner node
the elements in $\text{Search}_v(v)$ are contained in the $\text{Search}$-lists of its children, or in $\text{Copy}(v)$. This leads to $1 + \text{depth}(\text{PST})$ steps as follows.

First we compute, for all nodes $v$ in $\text{PST}$, the sorted list $\text{Add}_v(v)$ that contains all segments in $\text{Copy}(v)$, and associate each segment in $\text{Add}_v(l)$ with the segment in $H^B(v)$ that intersects it in $\Pi_v$ farthest to the right. This can be done in time $O(\log n)$ by $n + k/\log n$ processors for all nodes $v$ in $\text{PST}$. Note that the fragments in $\text{Add}_v(v)$ that are associated with a fixed segment in $H^B(v)$ form a continuous sublist of $\text{Add}_v(v)$. In the following $\text{depth}(\text{PST})$ steps we compute the $\text{CSearch}_v$-lists, walking upwards through $\text{PST}$.

Step 1, $1 \leq i \leq \text{depth}(\text{PST})$: Let $\text{depth}(i) = \text{depth}(\text{PST}) + 1 - i$. In this step we compute the list $\text{CSearch}_v(v)$ for each node $v$ in $\text{PST}$ at depth $\text{depth}(i)$. We show how this can be done using $O(n + k)$ work and $p \leq (n + k)/\log n$ processors, where $k_i$ is the sum of the sizes of all $\text{Add}$-lists at depth $\text{depth}(i)$.

If $v$ is a leaf, then $\text{Search}_v(v) = \text{Add}_v(v)$ and we only need to compress the $\text{Add}$-lists. This can be done in time $O(\log n)$ by $(n + k_i)/\log n$ processors for all nodes $v$ at depth $\text{depth}(i)$.

So let $v$ be an inner node with left child $u$ and right child $w$. Then obviously $\text{Search}_v(v)$ contains all segments in $\text{Search}_u(u)$ ($\text{Search}_w(v)$, resp.) that are not contained in $H^A(u)$ ($H^A(w)$, resp.), and all segments in $\text{Add}_v(v)$. Thus we first compute the compressed representations $\text{CSearch}'_v(u)$ of $\text{Search}_v(u) = \{l \in \text{Search}_v(u) | l \notin H^A(u)\}$ and $\text{CSearch}'_v(w)$ of $\text{Search}_v(w)$, defined analogously, (cf. step (1.1)), and then unite $\text{Search}_v(u)$, $\text{Search}_v(w)$, and $\text{Add}_v(v)$ (cf. step (1.2) and step (1.3)).

Step (1.1): In this step we compute $\text{CSearch}'_v(u)$ for all nodes $u$ at depth $\text{depth}(i) + 1$.

Let $u$ be a node at depth $\text{depth}(i) + 1$. We want to delete all segments from $\text{Search}_u(u)$ that are contained in $H^A(u)$. Since we deal with the compressed representations of the $\text{Search}$-lists, only the case that an element in $H^A(u)$ is the lowest or highest element of one of the sublists of $\text{Search}_v(u)$ needs our attention. W.l.o.g., let $l \in H^A(u)$ be the highest element of a sublist of $\text{Search}_v(u)$, and let $l$ be associated with the segment $h$ (cf. Fig. 15).

Then we replace $l$ by its nearest neighbour $q$ from below above $parent(u)$, if $q$ is intersected by $h$ and not contained in another sublist. If no such segment $q$ exists, i.e. this sublist is now empty, we replace $q$ by a dummy symbol. Afterwards we delete all dummy symbols, and then merge all adjacent sublists that are now associated with the same segment in $W^B(u) \cup H^B(u)$, both with the help of a parallel prefix computation.

Now we are ready to compute the information needed for cases 2.1 and 2.2.1 (cf. discussion above). Namely, we compute for each node $u$ at depth $\text{depth}(i) + 1$ the sorted list $\text{Search}_u(u)/\text{Search}_v(u)$ by merging $H^A(u)$ and $\text{CSearch}_v(v)$, and compute for each entry in this list the point of intersection represented by it with the help of the appropriate $\text{IntPoint}$-lists. This can be done using $O(n + t(m)(k + p))$ work and $p \leq (n + k)/\log n$ processors for all levels of the tree together.
Step (1.2): In this step we unite $CSearch_r(u)$ and $CSearch_r(w)$ for all siblings $u$ and $w$ at depth $\text{depth}(i)+1$. To do this we make use of the following observation: Let $v$ be a node at depth $\text{depth}(i)$ with left child $u$ and right child $w$, let $Search'_r(v)$ be the union of $Search'_r(u)$ and $Search'_r(w)$, and let $CSearch'_r(v)$ be the compressed representation of $Search'_r(v)$. Let $q \in Search'_r(v)$ (cf. Fig. 16). If $q \in Search'_r(w)$, then $\text{Seg}(q, Search'_r(v)) = \text{Seg}(q, Search'_r(w))$, and if $q \in Search'_r(v) \setminus Search'_r(w)$, then $\text{Seg}(q, Search'_r(v)) = \text{Seg}(q, Search'_r(u))$ This means that we have to split some of the sublists of $Search'_r(u)$, namely those that contain elements in $Search'_r(w)$ and elements in $Search'_r(v) \setminus Search'_r(w)$.

Thus we can compute $CSearch'_r(v)$ as follows. First we merge $CSearch'_r(u)$ and $CSearch'_r(w)$, using the y-coordinates of the segments. This is possible since all segments span $\Pi_v$. Let $L$ be a sublist of $Search'_r(u)$. We distinguish three cases.

Case 1. $L$ lies between two sublists of $Search'_r(w)$: Then $\text{Seg}(l, Search'_r(v)) = \text{Seg}(l, Search'_r(u))$ for all $l \in L$, and $L$ is a sublist of $Search'_r(v)$ or one of its neighbours in $Search'_r(w)$ is associated with the same segment.

Case 2. $L$ is contained in a sublist of $Search'_r(w)$: nothing needs to be done.

Case 3. Neither case 1 nor case 2. Then at least one endpoint of a sublist of $Search'_r(w)$ lies in $L$. Thus let $q \in CSearch'_r(w)$ be a segment that is contained in $L$, let, w.l.o.g., $q$ be the highest element of a sublist in $Search'_r(w)$, and let $l$ be the segment associated with
the segments in $L$. To decide whether $L$ has to be split at $q$ it is sufficient to compute $q$'s neighbour $h$ from above in $Search_r(u)$: $L$ has to be split at $q$ iff $h$ is not already a starting point of a sublist in $Search_r(w)$. The segment $h$ can be computed as follows. Since $q$ is intersected by $l$, $l \in H^B(u) \cup W^B(u)$, and $q$ spans $\Pi_v$, $q \in W_r(l)$ or $q \in W_l(l)$. Assume that $q \in W_l(l)$, and let $g$ be $q$'s neighbour from above in $W_1(l)$. Then $h = g$ if $g$ spans $\Pi_v$, and $h$ is the nearest neighbour from above of $g$ above $v$ else. We can therefore compute $h$ in $O(1)$ time.

Thus all new sublists can be computed in time $O(\log n)$ by $n/\log n$ processors for all nodes at depth $\text{depth}(i)$. To obtain $CSearch''(v)$ for all nodes $v$ we now have to unite those sublists that are neighboured and associated with the same segment in $B$. Thus step (i.2) can be executed in time $O(\log n)$ by $n/\log n$ processors.

To be able to compute all neighbours that comply with case 2.2.2, we additionally store, for each node $v$ at depth $\text{depth}(i)$ with left son $u$ and right son $w$, the segments in $Search'_r(u)$ ($Search'_r(w)$, resp.) that are contained in $Search'_r(w)$ ($Search'_r(u)$, resp.), again in compressed representation. As shown above, this information can be computed by $n/\log n$ processors in time $O(\log n)$ for all nodes at depth $\text{depth}(i)$. Below we show how to expand these lists.

Figure 17

Step (i.3): In this step we unite $CSearch''(v)$ and $Add_r(v)$ to obtain $CSearch_r(v)$ for all nodes $v$ at depth $\text{depth}(i)$. Let $v$ be a node at depth $\text{depth}(i)$. First we compute the sublist $Add'_r(v)$ of $Add_r(v)$ that contains all segments $l \in Add_r(v)$ where $\text{Seg}(l, Search_r(v)) = \text{Seg}(l, Add_r(v))$ (cf. Fig. 17). To do this we merge $CSearch''(v)$ and $Add_r(v)$ to determine for each segment $l$ in $Add_r(v)$ whether it is contained in $Search''(v)$ and if so, $\text{Seg}(l, Search''(v))$. After this we determine for each segment $l \in Add_r(v) \cap Search''(v)$ in time $O(1)$ whether it intersects $\text{Seg}(l, Add_r(v))$ or $\text{Seg}(l, Search''_r(v))$ furthest to the right in $\Pi_v$. Thus $Add'_r(v)$ can be computed in time $O(\log n)$ using $(n + k_i)/\log n$ processors for all nodes $v$ at depth $\text{depth}(i)$ together.
Next the segments in $Add_v(v) \cap Search_v(v) = Del_v(v)$ have to be deleted from $Search_v(v)$, i.e., sublists in $Search_v(v)$ have to be split or deleted (cf. Fig. 17). To do this we compute for each segment $l \in Del_v(v)$ its nearest neighbours in $Search_v(v)$. This can be done using the following observation: Let $l \in Del_v(v)$, let $h = Seg(l, Add_v(v))$, and let, w.l.o.g., $q$ be $l$'s neighbour from above in $Search_v(v)$. If $l \in H^A(v)$, then $q$ is either $l$'s neighbour from above in $CSearch(v)$, or $l$'s nearest neighbour from above above $v$. If $l \notin H^A(v)$, then $l \in W_1(h) \cup W_2(h)$ and $q$ is either $l$'s neighbour from above in $CSearch_v(v)$, or a neighbour of $l$ in $W_1(h)$ or $W_2(h)$, or the nearest neighbour from above above $v$ of this.

After this we insert the segments in $Add_v(v)$ into $CSearch_v(v)$ at the appropriate places and compress this list to obtain $CSearch_v(v)$. Thus $CSearch_v(v)$ can be computed in time $O(\log n)$ by $(n + k_i)/\log n$ processors for all nodes $v$ at depth $\text{depth}(i)$ together.

To be able to compute all neighbours that comply with case 2.2.2 (cf. discussion above) we additionally compute for each segment in $Del_v(v)$ the point of intersection represented by its entry in $Search_v(v)$. With the help of the appropriate $IntPoint$-lists this can be done using $O(n + t(m)(k + p))$ work and $p \leq (n + k)/\log n$ processors for all levels of $PST$ together.

Thus all $CSearch$-lists together can be computed using $O(\sum_{i=1}^{\text{PST}} n + k_i) = O(n \log n + k)$ work and $p \leq n/\log n + k/\log^2 n$ processors, and the information needed for cases 2.1 and 2.2.1 can be computed using $O(n \log n + t(m)(k + p))$ work and $p \leq n/\log n + k/\log^2 n$ processors. This leaves the expansion of the $Search$-lists.

**Computing $Search_v^l(a) \cap Search_v^l(b)$.**

We show how to compute, for all left nodes $a$ with sibling $b$, all segments in $Search^l_v(a) \cap Search^l_v(b)$, using $CSearch^l_v(a) \cap CSearch^l_v(b)$ and the $LWH$-lists and $O(n + t(m)(k + p))$ work and $p \leq (n + k)/\log n$ processors. For each left node $a$ with sibling $b$ $Search^l_v(a) \cap Search^l_v(b)$ is given by a set of sublists, where the elements of a sublist are associated with the same segment in $B$, and we know its lowest and its highest element. Let $L$ be such a sublist of $Search^l_v(a) \cap Search^l_v(b)$ for siblings $a$ and $b$, let $h_1$ ($h_2$, resp.) be the segment associated with $L$ in $Search^l_v(a)$ ($Search^l_v(b)$, resp.), let $l_{\text{low}}$ ($l_{\text{high}}$, resp.) be the lowest (highest, resp.) segment in $L$, and let $\tilde{L}$ be the sorted list of the points of intersection represented by the segments in $L$ in $Search^l_v(a)$. Given $l_{\text{low}}$ and $l_{\text{high}}$, we want to compute all segments in $L$ with the help of $LWH^l_v(a)$. To do this we first compute the points of intersection $s_1$ and $s_2$ represented by $l_{\text{low}}$'s ($l_{\text{high}}$'s, resp.) entry in $Search^l_v(a)$ in time $O(t(m))$ with a single processor. To compute $L$, we use the following observation: $L$ contains all segments $q$ in $A$ where $LWH^l_v(h_1)$ contains a point of intersection between $q$ and $h_1$ that lies between $s_1$ and $s_2$, where $q \in H^A(v)$ for an ancestor $v$ of $a$, and $q$ is not contained in another sublist (cf. Fig. 18). Also, all points of intersection in $\tilde{L}$ lie either on descending parts of $h_1$, or on ascending parts of $h_1$, or $h_2$ "embraces" the right endpoint of $h_1$ (cf. Fig. 18), i.e., there exist two segments that contribute to $LWH^l_v(h_1)$ such that $h_1$'s right endpoint lies between them, $h_2$ intersects them to the right of $h_1$, and a curved half line emanating from $h_1$'s...
right endpoint to the right and not intersecting any segment in $A$ intersects $h_2$ further to the right.

The marked points of intersection find neighbours on $h_2^2$.

$h_2^2$ and $h_4^2$ contribute to ascending lists, $h_1^2$ to a descending, and $h_3^2$ to a mixed list.

**Fig. 18**

Thus we divide all sublists into the three classes ascending, descending, and mixed. Moreover, let $L'$ be a sublist in the same class as $L$ where an element in $L'$ lies between $s_1$ and $s_2$. Then all points of intersection in $L'$ lie between $s_1$ and $s_2$.

We show here how to compute all decreasing sublists with the help of list ranking. First we delete all points of intersection lying on increasing parts of a curve, and introduce some dummy points of intersection into the $L^{WH}$-lists: Whenever two adjacent points of intersection in a $L^{WH}$-list are extreme points of different sublists, we insert a dummy element between them. Next we compute, for each element in each $L^{WH}$-list, a candidate for its left neighbour in the sublist that contains it, namely the next point of intersection to the left that is not contained in another sublist. This is its left neighbour in the $L^{WH}$-list, if this is not the rightmost point of intersection of another sublist, or the left neighbour of the corresponding left endpoint else. Then we compute for each point of intersection the sublist that contains it with the help of list ranking, and then delete those points not fulfilling the appropriate conditions. Thus all required $Search$-lists can be expanded using $O(n + t(m)(k + p))$ work and $p \leq (n + k / \log n) / \log^* n$ processors.

Now we have shown how to compute all information needed to compute neighbours according to case 2 and thus the remaining pairs in the $Pre$-lists. After this we can apply lemma 7 to compute all neighbours of vertices in the arrangement.

**Acknowledgements**

I wish to thank Torben Hagerup for helpful discussions.
6. References


[R92] Ch. Rüb, "Parallel Line Segment Intersection Reporting", Algorithmica, to appear