# Computing Intersections and Arrangements for Red-Blue Curve Segments in Parallel[1]

(Extended Abstract)

Christine Rüb

Max-Planck-Institut für Informatik, Im Stadtwald, W-6600, Saarbrücken, Germany
email: rueb@mpi-sb.mpg.de

**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 $A$ (resp., $B$) intersect. In this paper we give a parallel CREW-algorithm for reporting all points of intersection between segments in $A$ and segments in $B$, and for constructing the arrangement defined by the segments in $A \cup B$.

## 1. Introduction

Computing points of intersection and arrangements for lines or segments in the plane is a well known problem in computational geometry and has attracted a lot of attention (cf., e.g., [CE88], [G89], [G91], [MS88], [R92]). Here we consider a variant of this problem where the input consists of a set $A$ of "red" and a set $B$ of "blue" non-intersecting "well-behaved" curve segments in the plane and the machine model is the CREW-PRAM. These problems have applications, e.g., in computer graphics and computer aided design.

The work performed by our algorithms depends on the complexity of "elementary operations", 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. Here we assume that two segments intersect only a constant number of times. Then all points 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. In the full version of this paper (cf. [R92a]) we also consider the case where "elementary operations" are more time consuming. In this case the performed work increases to $O(n \log n + m(k+p))$ where $m$ is the maximal number of intersections between two segments. The running time of the best known sequential algorithm for this problem (cf. [MS88]) is $O(n \log n + k)$.

Our algorithms do not need to know the number $k$ of points of intersection in advance. Rather, $k$ is determined during the execution of the algorithms and additional processors are requested if necessary. This is done only a constant number of times.

We are not aware of any other parallel algorithm for these problems. However, there exist several algorithms for (simpler) variants of them. Goodrich showed how to construct the arrangement defined by a set $A$ of *vertical* and a set $B$ of *horizontal straight line segments* in time $O(\log n)$ using $n + k/\log n$ processors on a CREW-PRAM (cf. [G89]). Rüb showed (cf. [R90], [R92]) how to solve the red-blue intersection *reporting* problem for straight line segments with arbitrary slopes within the same time and processor bounds. (This does not include the construction of the arrangement.) The same result was claimed by Goodrich et al (cf. [GSG90]), although their proof seems to be incomplete. The problems of reporting all points of intersection between $n$ *arbitrary straight line segments* and constructing the arrangement defined by them in parallel, were considered in [G89], [R92], and [CCT91], and the problem of constructing the arrangement defined by $n$ *straight lines* in parallel was considered in [ABB90], [G91], and [HJW90].

The basic ideas of our algorithms are as follows. We use the plane-sweep tree, i.e., an extension of the segment tree, that is well suited for intersection problems in parallel. The plane-sweep tree divides each segment into $O(\log n)$ fragments. For each such fragment we first compute which other fragments are intersected by it and then distribute the thus found points of intersection equally among the processors to actually report them. To construct the arrangement we could now simply sort, for each segment, all points of intersection that lie on it according to their x-coordinates.

[1] This work was supported by the DFG, SFB 124, TP B2, VLSI Entwurfsmethoden und Parallelität.

This would lead to a performed work of $\Omega(n \log n + k \log n)$. We show how to compute the sorted list of intersections on each segment spending less work by using the fact that these sortings are not independent.
This paper is organized as follows: Section 2 contains some basic definitions, and Section 3 contains the algorithms.

## 2. Basic Definitions

The segments that we consider in this paper are "well-behaved", i.e., continuous and x-monotone, curve segments. In the remainder of this paper we assume, for ease of explanation, that no two segments overlap. We assume that the following functions can be evaluated in constant time by a single processor: $Y(p,x)$, where $Y(p,x)$ is the y-value of segment $p$ at x-coordinate $x$, *intersect*$(p,q)$ that is true iff segments $p$ and $q$ intersect, and *IntPoint*$(p,q)$ that returns an arbitrary point of intersection between segments $p$ and $q$.

**Definition** *plane-sweep tree*
*Let $S = \{l_1, ..., l_n\}$ be a set of curve segments and let $U = \{x_1 < x_2 < ... < x_r\} \subseteq \mathbb{R}$, called the universe, contain the x-coordinates of all endpoints of segments in $S$.*
*A plane-sweep tree PST for $S$ with universe $U$ consists of a balanced binary tree with $2r+1$ leaves. Each node $v$ has associated with it a vertical strip $\Pi_v$: the leaves from left to right are associated with the strips $(-\infty, x_1) \times (-\infty, +\infty), [x_1, x_1] \times (-\infty, +\infty), ..., (x_r, +\infty) \times (-\infty, +\infty)$, and every internal node is associated with the union of the strips of its children. 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: $W(v) = \{l \in S | l$ has an endpoint in $\Pi_v$ and does not span $\Pi_v\}$, $H(v) = \{l \in S | l$ spans $\Pi_v$ but not $\Pi_{parent(v)}\}$. The segments in $H(v)$ are sorted according to their y-coordinates at the left boundary of $\Pi_v$.*
**Notation:** Let $N(v)$ be some set of segments assigned to a node $v$ of a plane-sweep tree. Then $\widetilde{N}(v) = \{l \cap \Pi_v | l \in N(v)\}$. We call the elements of $\widetilde{N}(v)$ *fragments* of the segments in $N(v)$.

It is easy to see that a plane-sweep tree for a set $S$ of size $n$ and a universe of size $r$ has a size of $O(n \log r)$. The following lemma demonstrates how we can use this tree to compute segment intersections.

**Lemma 1**
Let $S$ be a set of segments in the plane and let $U$ contain the x-coordinates of their endpoints. Let $PST$ be a plane-sweep tree for $S$ with universe $U$. Suppose that a segment $l \in S$ intersects a segment $q \in S$ at a point $d$.
Then there exists exactly one node $v$ in $PST$ where $d \in \Pi_v$ and either (i) $l \in H(v)$ and $q \in H(v)$, or (ii) $l \in H(v)$ and $q \in W(v)$, or (iii) $l \in W(v)$ and $q \in H(v)$.

## 3. The Algorithms

Our algorithms are based on lemma 1. This means that we proceed in two or three steps as follows. In **step 1** we build up a plane-sweep tree $PST$ for $A \cup B$ whose universe consists of the x-coordinates of all endpoints of segments in $A \cup B$. 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.,). This can be done in time $O(\log n)$ by $n$ processors (cf. [R92]). In **step 2** we compute all points of intersection (cf. Section 3.1), and in **step 3** we construct the arrangement (cf. Section 3.2), using the information gathered in step 2.

### 3.1 Computing the Points of Intersection

We compute all points of intersection by first determining, for all nodes $v$ and each fragment $l$ at $v$, which segments in $H^A(v)$ ($H^B(v)$, resp.,) it intersects, and then distributing the points of intersection equally among the processors to actually report them.

*Computing the intersected segments*
We show here how to compute the intersected segments for all fragments of segments in $B$ in time $O(\log n)$ using $n$ processors. For each node $v$ in $PST$ and each fragment $l \in \widetilde{H}^B(v) \cup \widetilde{W}^B(v)$,

let $low(l)$ ($high(l)$, resp.,) be the rank of the highest (lowest, resp.,) fragment in $\widetilde{H}^A(v)$ that lies entirely below (above, resp.,) $l$. For all $\widetilde{H}$-fragments these ranks can be computed in time $O(\log n)$ by $n$ processors with the help of merging. For the $\widetilde{W}$-fragments they can be computed with the help of fractional cascading. We show here how to compute $low(l)$ for all fragments $l \in \widetilde{W}(v)$ and all nodes $v$ in 3 steps as follows.

**Step 1:** First we turn $PST$ together with the $H^A$–sequences into a fractional cascading data structure. This can be done in time $O(\log n)$ by $n$ processors (cf. [ACG89]). Now each node $v$ in $PST$ has assigned a sequence $M(v)$ such that given the position of an element $x$ in $M(v)$, a single processor con compute the position of $x$ in $H^A(v)$ and in $M(parent(v))$ in constant time.

**Step 2:** In this step we compute, for all nodes $v$ and all fragments $l \in \widetilde{H}^B(v)$, the nearest neighbour from below of $l$ in $M(v)$ with the help of merging. We can store this information for each segment in an array of length $2depth(PST)$.

**Step 3:** Now we assign one processor to each segment $l \in B$ that computes $low(l_v)$ for all fragments $l_v$ of $l$ where $l_v \in \widetilde{W}^B(v)$ for a node $v$. It does this by simultaneously following the two paths of nodes such that $l$ is contained in their $W$-sets upwards. When moving from a node $v$ to its parent $w$, we distinguish two cases. In **case 1** $l_v = l_w$. Then the processor assigned to $l$ can compute the nearest neighbour from below of $l_w (= l_v)$ in $M(w)$ and $low(l_w)$ in time $O(1)$.

In **case 2** $l_v \neq l_w$. 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 $l \in H^B(u)$. 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 $M(w)$ 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.

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

***Computing all points of intersection***

All points of intersection can now be computed using $p \leq n + k \log n$ processors and $O(n \log n + k)$ work.

## 3.2 Constructing the Arrangement

We show here how to construct the arrangement after all points of intersection have already been computed. 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 $l$. According to the three clauses in lemma 1, there exist three cases for a segment $l$ and a point of intersection $s$ on $l$. Following clause (iii), 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 3.2.1 we show how to compute $L^{WH}(l)$ for all segments $l$ using $O(n \log n + k)$ work and $p \leq n + k/\log n$ processors. These lists will then guide the computation of all neighbours using $O(n \log n + k)$ work and $p \leq n/\log n + k/\log^2 n$ processors, as shown in Section 3.2.2.

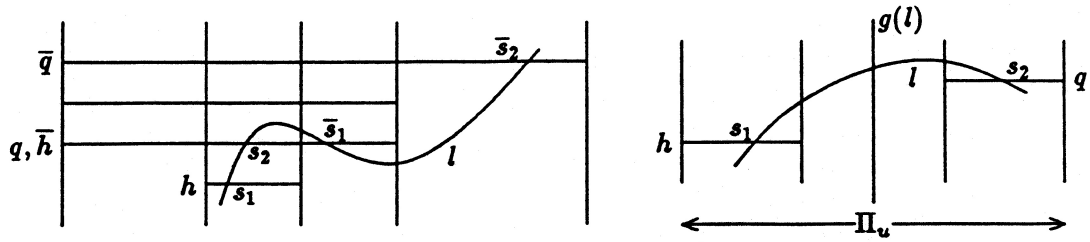
### 3.2.1 Computing $L^{WH}(l)$

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 $l \in A$ and for each point of intersection in $L^{WH}(l)$, its neighbours in $L^{WH}(l)$ and then using list ranking to obtain all $L^{WH}$–lists. The latter step can be executed in time $O(\log n)$ by $n + k/\log n$ processors (cf. [AM91]), so let us concentrate on the first one. To compute the neighbours we proceed in two steps as follows.

**Step 1:** This is a preprocessing step. In it we compute, for each segment $q \in B$ and each node $v$ where $q \in H^B(v) \cup W^B(v)$, the ***nearest neighbours of $q$ above $v$***, i.e., 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. This can be done in time $O(\log n)$ with $n$ processors.

**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 $h \in \widetilde{H}^B(v)$, and let $s_2$ lie on $q \in \widetilde{H}^B(w)$. W.l.o.g., let $depth(v) \geq depth(w)$. 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. 1a), and in **case 3** $w$ is not an ancestor of $v$ (cf. Fig. 1b). We deal with the 3 cases in reversed order.




a) $s_1$ and $s_2$ comply to case 1, $\overline{s}_1$ and $\overline{s}_2$ comply to case 2

b) $s_1$ and $s_2$ comply to case 3

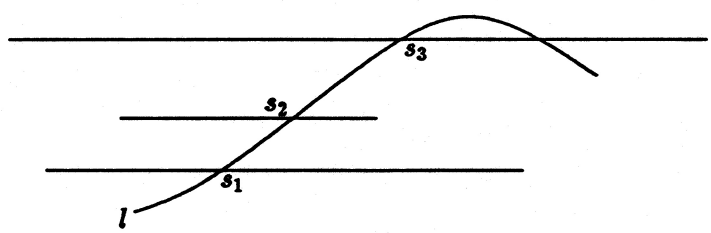
**Fig. 1**

**Case 3:** This case can occur at most once for each segment $l \in A$, and all neighbours complying with it can be computed in time $O(\log n)$ by altogether $n + k/\log n$ processors.

**Case 2:** 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 $\widetilde{H}^B(v)$, and only $l$'s left endpoint is contained in $\Pi_v$. We assign one processor to each sement that computes all such pairs of intersection by walking upwards in *PST*. Altogether this can be done using $O(n \log n + k)$ work and $p \leq n + k/\log n$ processors.

**Case 1:** We will use the fact that under the above conditions, $q$ is a nearest neighbour of $h$ above $v$. We compute the neighbours in 2 steps. In **step 1** we compute, for all $l \in A$ and all $s \in L^{WH}(l)$, "candidates" $cand_l(s,l)$ and $cand_r(s,l)$ for $s$'s left (right, resp.,) neighbour in $L^{WH}(l)$, and in **step 2** we actually compute the neighbours.

**Step 1:** For each segment $l \in A$, we assign one processor to each point of intersection in $L^{WH}(l)$. Let $s \in L^{WH}(l)$ lie on $h \in \widetilde{H}^B(v)$. The processor assigned to $s$ examines $h$'s nearest neighbours $h_1$ and $h_2$ above $v$ and assigns the points of intersection between $l$ and $h$, $h_1$, and $h_2$ that are nearest to $s$ to $cand_l(s,l)$ ($cand_r(s,l)$, resp.,). After this we assign one processor to each pair of points of intersection complying with case 2 or case 3 that replaces their candidates appropriately.




$cand_r(s_1,l) = s_3$ at the beginning. When the neighbour of $s_2$ from the left has been computed, $cand_r(s_1,l)$ is replaced by $s_2$.

**Fig. 2**

**Step 2:** In this step we compute all neighbours. We do this in $depth(PST) - 1$ steps, one for each level of *PST* except the root, moving from the leaves upwards. For each $i$, $1 \leq i \leq depth(PST)$, let $k_i$ be the sum of the lengths of all lists $L^{WH}(l,q)$ of intersections between $l \in A$ and $q \in \widetilde{H}^B(v)$ for a node $v$ at depth $i$.

**Step 2.i,** $1 \leq i \leq depth(PST) - 1$: Let $depth(i) = depth(PST) + 1 - i$. We assign one processor to each point of intersection $s \in L^{WH}(l,q)$ where $l \in A$ and $q \in \widetilde{H}^B(v)$ for a node $v$ at depth $depth(i)$. The processor assigned to $s$ examines $cand_l(s,l)$ and tests whether $s$ lies between $cand_l(s,l)$ and $cand_r(cand_l(s,l))$. If so, it sets $cand_r(cand_l(s,l))$ to $s$ (cf. Fig. 2). A similar rule is applied to $cand_r(s,l)$ and $cand_l(cand_r(s,l))$.

Thus step 2 can be executed in time $O(\sum_{i=1}^{depth(PST)} \lceil k_i \log n/k \rceil) = O(\log n)$ by $n + k/\log n$ processors. The correctness of the two steps can be proved by induction on $i$.

### 3.2.2 Computing all neighbours

In this section we show how to compute the neighbours of all vertices in the arrangement of $A \cup B$ using $O(n \log n + k)$ work and $p \leq n/\log n + k/\log^2 n$ processors. 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 $h$ where $s \in \Pi_v$. 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 $\widetilde{H}^B(v)$, sorted according to their y-coordinates. In **step 1** we compute the list $Copy(v)$ for each node $v$ in $PST$. Additionally we compute, for each node $v$, each segment $l \in Copy(v)$ and each fragment $q \in \widetilde{H}^B(v)$ that is intersected by $l$ all points of intersection between $l$ and $q$. With the help of the $L^{WH}$–lists this can be done in time $O(\log n)$ using $n + k/\log n$ processors.
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 $l \in W^B(v) \cup Copy(v)$, and $s$ lies on a fragment in $\widetilde{H}^B(v)$, 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 the same technique as in Section 3.2.1 to compute all neighbours. I.e., we want to identify, for all segments $l \in A$, all pairs $(s_1, s_2)$ of neighbours in $L(l)$ where $s_1$ lies on $h \in \widetilde{H}^B(v)$, $s_2$ lies on $q \in \widetilde{H}^B(w)$ and $s_1 \notin \Pi_w$ or $s_2 \notin \Pi_v$. (Otherwise $h$ is a neighbour of $q$ above $w$ or $q$ is a neighbour of $h$ above $v$.) This will be done in **step 2**. In **Step 3** we then employ the same technique as for the computation of the $L^{WH}$–lists to compute all neighbouring points of intersection.

**Step 2:** 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 that $s_2 \notin \Pi_v$ or $s_1 \notin \Pi_w$ need our attention. We distinguish 2 cases. In **case 1** $l \in W^A(v)$ and $l \in W^A(w)$, i.e., $s_1$ and $s_2$ are neighbours in $L^{WH}(l)$, and in **case 2** $l \in Copy(v)$ or $l \in Copy(w)$. To compute the neighbours that comply with case 2 we use the fact that in this case $s_1$ is extremal among the points of intersection of $l$ with fragments in $\widetilde{H}^B(v)$, or $s_2$ among those with fragments in $\widetilde{H}^B(w)$.
Thus we define, for each node $v$, two sorted (according to y-coordintes) lists $Search_r(v)$ and $Search_l(v)$ of $H^A$–segments that "search" for neighbours for a point of intersection on them lying 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 $\widetilde{W}^B(v) \cup \widetilde{H}^B(v)$, and is associated with the segment in $W^B(v) \cup H^B(v)$ that contains this intersection.
Now let us investigate how the *Search*–lists can help us to find all required neighbours. Remember that $l \in A$ and $s_1$ and $s_2$ are neighbours in $L(l)$ where $s_1$ lies on $h \in H^B(v)$ and $s_1 \in \Pi_v$, and $s_2$ lies on $q \in H^B(w)$ and $s_2 \in \Pi_w$. W.l.o.g. we assume that $l \in Copy(v)$ 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 $\widetilde{H}^B(v) \cup \widetilde{W}^B(v)$ and thus $l \in Search_r(v)$ and $l$'s entry in $Search_r(v)$ represents $s_1$. Let $u$ be the ancestor of $v$ where $l \in H^A(v)$. We distinguish two cases.
**Case 2.1:** $s_1$ is the rightmost point of intersection on $l$ in $\Pi_u$. Then $l \in Search_r(u)$ and $l$'s entry in this list represents $s_1$, and $l \notin Search_r(parent(u))$.

**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 $Search_r(u)$ 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 $q$ is a nearest neighbour of $h$ above $u$.

**Case 2.2.2:** $l$'s entry in $Search_r(u)$ does not represent $s_1$. Let $z$ be the highest ancestor of $v$ where $l$'s entry in $Search_r(z)$ represents $s_1$. Then $l \in Search_l(sibling(z))$, or $l \in Copy(parent(z))$. 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 $Search_l(sibling(z))$, and $s_4$ the point of intersection between $l$ and the fragments in $\widetilde{H}^B(parent(z))$ 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. 3).
Thus given the *Search*–lists we can compute all neighbours that comply with case 2 using additional work $O(n \log n + k)$. This leaves the computation of the *Search*–lists.

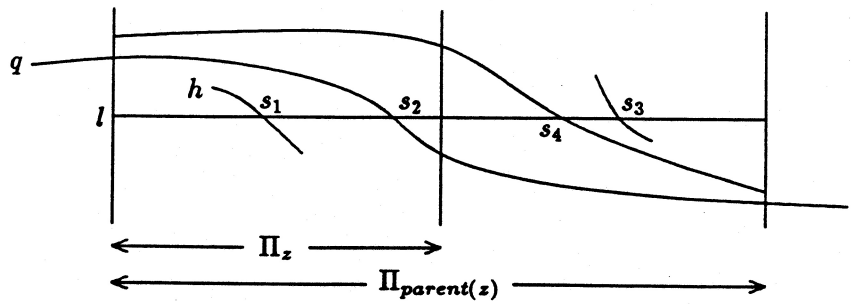



**Fig. 3**

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 $Search_r(v)$ ($Search_l(v)$, resp.,) be divided into maximal sublists where the elements in a sublist are associated with the same segment in $W^B(v) \cup H^B(v)$. We store in the compressed version of $Search_r(v)$ ($Search_l(v)$, resp.,), called $CSearch_r(v)$ ($CSearch_l(v)$, resp.,), only the lowest and highest element in each sublist. It can be shown that the size of all *CSearch*–lists in $PST$ is bounded by $O(n \log n)$.
To compute the missing information, we proceed as follows. First we compute the lists $CSearch_r(v)$ and $CSearch_l(v)$ for each node $v$ in $PST$ walking upwards in the tree. While doing this we also gather the information needed for cases 2.1 and 2.2.1. Afterwards we then compute the information needed for case 2.2, what amounts to expanding parts of the *CSearch*–lists. All this can be done in $O(n \log n + k)$ work using $p \leq n/\log n + k/\log^2 n$ processors. We omit the details.

## 6. References


[ABB90] R. Anderson, P. Beame, E. Brisson, *" Parallel Algorithms for Arrangements"*, 1990 ACM Symp. on Parallel Algorithms and Architectures, 298-306

[ACG89] M. Atallah, R. Cole, M. Goodrich, *"Cascading Divide-and-Conquer: A Technique for Designing Parallel Algorithms"*, SIAM J. Comput., Vol 18, No 3, 1989, 499–532

[AM91] R.J. Anderson, G.L. Miller, *Deterministic Parallel List Ranking*, Algorithmica, Vol. 6, 1991, 859-868

[CE88] B. Chazelle, H. Edelsbrunner, *"An Optimal Algorithm for Intersecting Line Segments in the Plane"*, 29th FOCS, 1988, 590-600

[CCT91] K.L. Clarkson, R. Cole, R.E. Tarjan, *"Randomized Parallel Algorithms for Trapezoidal Diagrams"*, Proc. 7th ACM Symposium on Computational Geometry, 1991, 152-161

[G89] M. Goodrich, *"Intersecting Line Segments in Parallel with an Output-Sensitive Number of Processors"*, 1989 ACM Symp. on Parallel Algorithms and Architectures, 127–136

[G91] M. Goodrich, *"Constructing Arrangements Optimally in Parallel"*, 1991 ACM Symp. on Parallel Algorithms and Architectures, 169-179

[GSG90] M.T. Goodrich, S.B. Shauck, S. Guha, *"Parallel Methods for Visibility and Shortest Path Problems in Simple Polygons"*, Proc. 6th ACM Symposium on Computational Geometry, 1990, 73–82

[HJW90] T. Hagerup, H. Jung, E. Welzl, *"Efficient Parallel Computation of Arrangements of Hyperplanes in d Dimensions"*, 1990 ACM Symp. on Parallel Algorithms and Architectures, 290-297

[MS88] H.G. Mairson, J. Stolfi, *"Reporting and Counting Intersections Between Two Sets of Line Segments"*, NATO ASI Series, Vol F40, Theoretical Foundations of Computer Graphics and CAD, ed. R.A. Earnshaw, Springer 1988, 307–325

[R90] Ch. Rüb, *"Parallele Algorithmen zum Berechnen der Schnittpunkte von Liniensegmenten"*, PhD Thesis, Saarbrücken, 1990

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

[R92a] Ch. Rüb, *Computing Intersections and Arrangements for Red-Blue Curve Segments in Parallel*, Technical Report No. MPI-I-92-108, Saarbrücken, 1992