1989

On the Parallel-Decomposability of Geometric Problems

Mikhail J. Atallah
Purdue University, mja@cs.purdue.edu

Jyh-Jong Tsay

Report Number:
89-873

ON THE PARALLEL-DECOMPOSABILITY OF GEOMETRIC PROBLEMS

Mikhail J. Atallah
Jyh-Jong Tsay

CSD-TR-873
March 1989
(Revised June 1991)

On the Parallel-Decomposability of Geometric Problems

Mikhail J. Atallah*
Department of Computer Science
Purdue University
West Lafayette, IN 47907

Jyh-Jong Tsay†
National Chung Cheng University
Institute of Computer Science
and Information Engineering
Chiayi, Taiwan 62107, ROC

Abstract

There is a large and growing body of literature concerning the solutions of geometric problems on mesh-connected arrays of processors. Most of these algorithms are optimal (i.e. run in time $O(n^{1/d})$ on a d-dimensional n-processor array), and they all assume that the parallel machine is trying to solve a problem of size n on an n-processor array. Here we investigate the situation where we have a mesh of size p and we are interested in using it to solve a problem of size $n > p$. The goal we seek is to achieve, when solving a problem of size $n > p$, the same speedup as when solving a problem of size p. We show that for many geometric problems, the same speedup can be achieved when solving a problem of size $n > p$ as when solving a problem of size p.

Key Word. Computational geometry, Mesh-connected arrays of processors, Parallel algorithms.

1 Introduction

Suppose we have a parallel machine, like a d-dimensional mesh-connected array of p processors, that can solve a problem of size p in time $O(p^{1/d})$ (this includes the time to input the data to the array as well as the actual computation time, a standard assumption in the literature of mesh-connected arrays of processors, and certainly a reasonable one for the case $d = 1$). Suppose such a parallel machine is attached to a conventional random access machine (RAM) that wishes to solve a problem of size $n > p$. We call such a machine a RAM/ARRAY(d) (see Figure 1). If the problem's sequential time complexity is, say, $\Theta(n \log n)$, then the p-processor array gives a factor of $s(p) = p^{1-1/d} \log p$ speedup *for a problem of size p*. However, if the RAM/ARRAY(d) is trying to solve a problem of size n,

*This author's research was supported by the Office of Naval Research under Contracts N00014-84-K-0502 and N00014-86-K-0689, the Air Force Office of Scientific Research under Grant AFOSR-90-0107, the National Science Foundation under Grant DCR-8451393, and the National Library of Medicine under Grant R01-LM05118.

†This author's research was partially supported by the Office of Naval Research under Contract N00014-84-K-0502, the Air Force Office of the Scientific Research under Grant AFOSR-90-0107, and the National Science Foundation under Grant DCR-8451393.

$n > p$, then it is not clear how it should use the p-processor array to achieve the factor of $s(p)$ speedup and obtain $O(n \log n/s(p))$ time performance. Actually, it is not even clear whether it is at all possible to maintain the $s(p)$ speeedup, i.e., whether the problem can be decomposed into p-sized subproblems in a way that maintains the $s(p)$ speedup. When this is possible for a certain problem, we say that the problem is *fully parallel-decomposable* for the RAM/ARRAY(d). Identifying the problems for which this optimal $O(n \log n/s(p))$ time can be achieved is an interesting question that has been answered in the affirmative for the problem of sorting [AFK88, AV88, BG90, LCW81] when $d = 1$. This result immediately implies an affirmative answer on a RAM/ARRAY(1) for some geometric problems that can be solved in linear time after a pre-processing sorting step, like the planar convex hull and maximal elements problems (this implicitly assumes that the p-processor array can be used for sorting, i.e., it is not restricted to just solving size-p instances of the problem considered). In this paper, we show that the answer is also affirmative for many geometric problems that are not known to be reducible to sorting. More specifically, we show that the $O(n \log n/s(p))$ time bound can indeed be achieved on a RAM/ARRAY(1) for the problems of computing the following :

- all nearest neighbors of a planar set of points,
- the measure and perimeter of the union of rectangles,
- visibility of a set of non-intersecting line segments from a point,
- 3-dimensional maxima,
- dominance counting between two sets of points (and hence the related problem of counting intersections between rectilinear rectangles).

(Note that, for $d = 1$, $s(p) = \log p$.) Essentially the same method as for the RAM/ARRAY(1) establishes that all these problems can be solved in $O(n \log n/s(p))$ on a RAM/ARRAY(d), with $s(p) = p^{1-1/d} \log p$.

Recall that in a mesh-connected processor array, each of the p processing elements has only $O(1)$ storage registers. This is a standard assumption in the literature of mesh-connected arrays and we shall not tamper with it. Allowing more memory in each array element gives rise to interesting issues that are not within the scope of this paper.

Let us remember that many existing parallel machines have a "front end" that is a conventional sequential computer and that the number of processors in the parallel machine

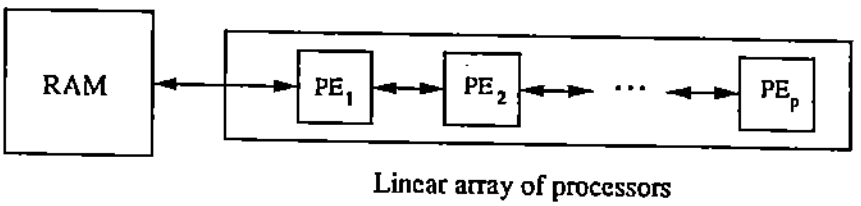

Figure 1: a RAM/ARRAY(1) machine

itself is typically the fixed number purchased rather than a function of the problem size n. This provides a justification for the model used in this paper.

This paper is organized as follows. Section 2 gives some preliminaries, Section 3 gives the detailed algorithms for all the problems we consider on a RAM/ARRAY(1), Section 4 gives a general approach to extend all the algorithms to a RAM/ARRAY(d), and Section 5 concludes.

We use p throughout to denote the number of processors of the attached processor array.

2 Preliminaries

In this section, we introduce some notation and definitions and review some known results which will be used later in our algorithms.

2.1 Notation

For any point q in the plane, we use $x(q)$ (resp., $y(q)$) to denote the x (resp., y) coordinate of q. If the point q is in three dimensional space $\Re^3$, we then use $z(q)$ to denote the z coordinate of q. For any rectangle r in the plane, whose edges are parallel to the two axes, we use $left(r)$ (resp., $right(r)$, $bottom(r)$, $top(r)$) to denote the left (resp., right, bottom, top) edge of r. We say that a line l in the plane is *horizontal* (resp., *vertical*) if it is parallel to the x (resp., y) axis. For any horizontal (resp., vertical) line segment u, we use $x(u)$ (resp., $y(u)$) to denote the x (resp., y) coordinate of u. Two line segments are said to intersect *properly* iff they intersect at other than their endpoints. In $\Re^3$, a *horizontal* (resp., *vertical*)

plane is one which is parallel to the xy (resp., yz) plane. For a set $S = \{s_1, s_2, \ldots, s_n\}$ of geometric objects (e.g., line segments or rectangles) in the plane, we say S is *monotone* with the x (resp., y) direction if, for any vertical (resp., horizontal) line l, l properly intersects at most one object in S. If S is a set of geometric objects in $\Re^3$, we say S is monotone with the x direction if, for any vertical plane E, E properly intersects at most one object in S.

2.2 Some useful known results

In [FJ82], Frederickson and Johnson established the following result. Given an $a \times b$ matrix whose columns are sorted, the kth smallest element can be selected in time $O(b + m\log(k/m))$, where $m = \min\{k, b\}$, if the matrix is already in the memory, or if any element of the matrix can be produced in constant time. This implies that the bth element can be selected from the matrix in $O(b)$ time. This selection algorithm has been used in [AFK88], and in the present paper too it is a crucial ingredient. However, the algorithms of this paper are far more intricate, and must use different partitioning and combining schemes than [AFK88].

Suppose we have $p^{1/k}$ sets $S_1, S_2, \ldots, S_{p^{1/k}}$ of line segments in the plane, where $k \geq 1$ is some constant. Each set S_i is monotone with the x direction and sorted by increasing x coordinate. For example, in Figure 2, $S_1 = \{u_1, u_2, u_3, \ldots\}$, $S_2 = \{v_1, v_2, v_3, \ldots\}$, and $S_3 = \{w_1, w_2, w_3, \ldots\}$. Let n be the total number of line segments in $\cup_{i=1}^{p^{1/k}} S_i$. Define a total order between the line segments by the x coordinates of their right endpoints, that is, for any two line segments u and v, $u < v$ iff the x coordinate of the right endpoint of u is less than that of the right endpoint of v. Let l_j be the vertical line defined by the x coordinate of the right endpoint of the (jp)th line segment in $\cup_{i=1}^{p^{1/k}} S_i$. In this paper, we frequently need to partition the set $\cup_{i=1}^{p^{1/k}} S_i$ into sets $G_1, G_2, \ldots, G_{n/p}$ and to create sets C_1, $C_2, \ldots, C_{n/p-1}$, such that G_j consists of the line segments whose right endpoints are to the right of line l_{j-1} and are on or to the left of line l_j, and C_j consists of the line segments which properly intersect line l_j (i.e., intersect it at other than their endpoints). In Figure 2, $G_1 = \{u_1, v_1, w_1\}$ and $C_1 = \{v_2, w_2\}$. Note that the size of each C_j is at most $p^{1/k} - 1$ since each S_i is monotone with the x direction.

Lemma 2.1 *Let the S_i's, G_i's and C_i's be defined as above. Then obtaining the G_i's and C_i's from the S_i's can be done in $O(n)$ time.*

Proof: To simplify the explanation, we first assume $k = 1$ and then extend to the case of

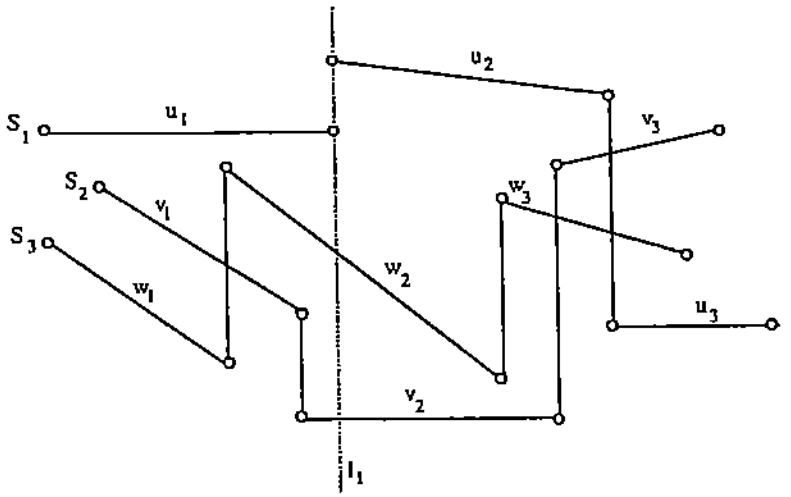

Figure 2: $G_1 = \{u_1, v_1, w_1\}$ and $C_1 = \{v_2, w_2\}$

$k > 1$. Assume $k = 1$. We have p sorted and monotone sets S_1,S_2,..., S_p. (This is the case which occurs when we solve problems on a RAM/ARRAY(1).) Treat each sorted set S_i as a sorted column and form a matrix of p columns. Then, in turn for $j = 1, \ldots, n/p$, we do the following (i)–(iv): (i) we select the pth smallest element in the matrix, (ii) we use the element so selected to obtain G_j, (iii) we implicitly delete the elements of G_j from each column of the matrix, and (iv) we obtain C_j. The selection of step (i) is done in $O(p)$ time using the algorithm in [FJ82] (it does not matter that the columns as handled by the algorithm may not be the same length, since the selection algorithm of [FJ82] does not examine any element beyond the pth smallest in any current column). The element selected in step (i) is clearly the (jp)th element in the original matrix (the matrix before any deletion from it). Because of this, performing step (ii) is trivial to do in $O(p + |G_j|)$ time: the contents of G_j are easily identified by examining each column of the matrix in turn. Step (iii) is done in $O(p)$ time simply by changing the index of the beginning of each column of the matrix (this implicit kind of deletion is possible because the elements to be deleted from a given column are contiguous and at the beginning of that column). For step (iv), note that an element of C_j coming from a column is necessarily the smallest remaining element in that column, and hence C_j has size at most p and can be constructed in $O(p)$ time.

If $k > 1$, then we run the same algorithm as the one for the case $k = 1$, except that we let $p' = p^{1/k}$ play the role of p. This gives sets $G'_1, G'_2, \ldots, G'_{n/p'}$ and $C'_1, C'_2, \ldots, C'_{n/p'-1}$. The

desired sets $G_1, G_2, \ldots, G_{n/p}$ and $C_1, C_2, \ldots, C_{n/p^{1-1}}$ are easily obtained with $O(n)$ extra work, since $G_i = \cup_{j=(i-1)p^{1-1/k}+1}^{ip^{1-1/k}} G'_j$ and $C_i = C'_{ip^{1-1/k}}$. □

Note that lemma 2.1 still holds if the geometric object in each S_i is an isothetic rectangle (one whose sides are parallel to the coordinate axes). In next section, we will use the partitioning scheme in lemma 2.1 to design $O(n \log n / \log p)$ time algorithms for many geometric problems on a RAM/ARRAY(1). A similar partitioning scheme which takes time $O(n/p^{1/d})$ will be presented later when we discuss the algorithms on a RAM/ARRAY(d) ($k = d + 1$).

A similar partitioning scheme has been used in one of the two approaches given in [AFK88] to merge p sorted lists of size n/p each in $O(n)$ time and hence obtain an $O(n \log n / \log p)$ time sorting algorithm on a RAM/ARRAY(1).

Lemma 2.2 *[AFK88] Given a set S of n numbers, we can sort the numbers in S in $O(n \log n / \log p)$ time on a RAM/ARRAY(1).*

Proof: We sketch the idea of [AFK88]. We first partition S into p subsets of size n/p each and recursively sort each subset one by one. After the recursive calls, we partition the p sorted lists into n/p groups of size p each as in Lemma 2.1 and sort each group in $O(p)$ time, using the p-processor array. The time complexity $T(n)$ of the above process thus satisfies the recurrence $T(n) = pT(n/p) + c_1 n$ if $n > p$, and $T(n) = c_2 n$ if $n \leq p$, where c_1, c_2 are constants. This implies that $T(n) = O(n \log n / \log p)$. □

3 RAM/ARRAY(1) algorithms

In this section, we present RAM/ARRAY(1) algorithms for the problems of computing all nearest neighbors of a set of planar points, the measure and perimeter of the union of rectangles, the visibility from a point, 3-dimensional maxima, dominance counting between two sets of points, and hence the related problem of counting intersections between rectilinear rectangles. All of the above problems have sequential time complexity $\Theta(n \log n)$ and will be solved in $O(n \log n / s(p))$ time on a RAM/ARRAY(1) (for all of the above problems, $s(p) = \log p$ on a RAM/ARRAY(1)). The RAM/ARRAY(1) algorithms which will be presented later follow the p-way divide-and-conquer paradigm as described for sorting in Lemma 2.2. That is, we will partition the problem into p subproblems of size n/p each, and recursively solve each subproblem one by one. After the p recursive calls, we then combine

the p subsolutions in $O(n)$ time to achieve the $O(n \log n / \log p)$ time performance. Our main contribution is in showing that the combining step can be done in $O(n)$ time with efficient use of the p-processor array. In particular, we will show that the combining step can be done by plane sweeps which consist of n/p sweep steps, each of which uses the p-processor array a constant number of times. Let C be the information maintained during the sweep. Each step of the sweep advances it past the next p elements, as follows: (i) we identify the group G of the next p elements, (ii) we solve a constant number of $O(p)$-sized problems, and (iii) we update C. Step (i) can be done in $O(p)$ time sequentially, using the selection algorithm of Frederickson and Johnson as in Lemma 2.1 (in order to enable us to use this selection scheme, a by-product of each recursive call is to sort its elements). For the various problems considered, when $p = 2$, it is usually already known (e.g., [Ben80, Gut84]) that $|C| = O(1)$ and hence steps (ii) and (iii) can be done in $O(1)$ time. In this section, we will show that step (ii) and (iii) can be done in $O(p)$ time, using the attached p-processor array. In particular, we show that $|C| = O(p)$ and that each step of the sweep can be reduced to solve a constant number of $O(p)$-sized problems. In the remainder of this section, we give the details of the algorithms for the problems considered.

3.1 All nearest neighbors

Given a set S of n points in the plane, the all nearest neighbors problem is to find a nearest neighbor $N(u)$ of every $u \in S$. This problem has many applications in answering basic proximity questions of sets of objects, and several authors have addressed the question of solving this problem on various kinds of parallel machine models [Cha84, CG88, JL90, MS89, SCL87, WW88]. Chazelle [Cha84], and, Shih *et al* [SCL87] gave $O(n)$ time systolic solutions on a linear systolic array of n processors. Jeong and Lee [JL90], and, Miller and Stout [MS89] independently solved the problem in $O(\sqrt{n})$ time on a 2-dimensional array of n processors. Cole and Goodrich [CG88] and Willard and Wee [WW88] gave $O(\log n)$ time algorithms on an n-processor PRAM. However, a simulation of these algorithms on our parallel machine, the RAM/ARRAY(d), fails to achieve the optimal $O(n \log n / s(p))$ time performance. In this subsection, we give an algorithm for this problem which runs in $O(n \log n / \log p)$ time on a RAM/ARRAY(1). The same idea can be used in solving this problem in $O(n \log n / s(p))$ time on a RAM/ARRAY(d), for $d > 1$ (this extension is given in Section 4).

In the remainder of this subsection, we give the algorithm for the all nearest neighbors

problem on a RAM/ARRAY(1). For simplicity, we assume that the x coordinates (resp., y coordinates) of points in S and the Euclidean distances between them are pairwise distinct. (Our algorithm can be easily modified to deal with the general case.) Throughout this subsection, we use $d(u, v)$ to denote the Euclidean distance between points u and v. Since sorting can be done in $O(n \log n / \log p)$ time on a RAM/ARRAY(1), we assume the points given in S are already sorted by increasing x coordinate.

3.1.1 Outline of the algorithm

The general idea of the algorithm for the all nearest neighbors problem is as follows. We first partition the problem into p subproblems of size n/p each, and recursively solve each subproblem one by one. Let us call the nearest neighbor of u returned by the recursive calls the *local neighbor* of u, and denote it by $N_1(u)$. Thus $N_1(u)$ is in the same partition as u and is closer to u than any other points in that partition. In the combining step, we find two *foreign neighbors* for each of the points: the *lower* (resp., *upper*) foreign neighbor of u is the point which is below (resp., above) u, is not in the same partition as u, and is closer to u than $N_1(u)$ and than any other points below u (hence neither foreign neighbor of u exists if u's nearest neighbor is in the same partition as u). What makes the combining step difficult is that the (upper or lower) foreign neighbor of a point in one partition is not necessarily in one of its adjacent partitions. How to capture this kind of foreign neighbors in $O(n)$ time on a RAM/ARRAY(1) is not clear from any of the previous algorithms. Below is a rough outline of the algorithm. The implementation of its various steps is given in the subsection that follows.

Algorithm NEAREST-NEIGHBORS

Input: A set S of n sorted points sorted by increasing x coordinate.

Output: For each $u \in S$, the point $N(u) \in S$ closest to it. Also, the elements of S sorted by decreasing y coordinate.

1. If $|S| \leq 5p$ then solve the problem in $O(|S|)$ time by direct use of the p-processor array. If $|S| > 5p$ then proceed to Step 2.

2. Partition S into p subsets $S_1, S_2, \ldots, S_p$ in left-to-right order by vertical cut-lines and recursively solve the problem on each S_i. The recursive call for S_i returns, for

every $u \in S_i$, the point of S_i that is closer to u than any other point in S_i. Call $N_1(u)$ such a restricted nearest neighbor of u. It also returns the elements of S_i sorted by decreasing y coordinate.

3. Perform a downward sweep of the points in S, during which a point $N_2(u)$ is found for each $u \in S$. $N_2(u)$ is defined as follows. Let $u \in S_i$. Then $N_2(u)$ is the point closest to u among all points that are in $S - S_i$, are below u, and are closer to u than the distance $d(u, N_1(u))$ (if no such point exists then $N_2(u) = \emptyset$).

4. Perform an upward sweep of the points in S, during which a point $N_3(u)$ is found for each $u \in S$. $N_3(u)$ is defined as follows. Let $u \in S_i$. Then $N_3(u)$ is the point closest to u among all points that are in $S - S_i$, are above u, and are closer to u than the distance $d(u, N_1(u))$ (if no such point exists then $N_3(u) = \emptyset$).

5. For each $u \in S$, compute $N(u)$, the point of S closest to u, by choosing one of the points $N_1(u)$, $N_2(u)$ and $N_3(u)$ which is closest to u.

6. Sort the elements in S by y coordinate by "merging" the p sorted lists returned by Step 1 in $O(n)$ time as in [AFK88].

End of Algorithm NEAREST-NEIGHBORS

Correctness of the algorithm would immediately follow if Steps 3 and 4 correctly compute $N_2(u)$ and $N_3(u)$, respectively. In order to achieve the $O(n \log n / \log p)$ time performance, it suffices to perform Steps 3 and 4 in $O(n)$ time, since the time complexity would then obey the recurrence $T(n) = pT(n/p) + c_1 n$ if $n > 5p$, and $T(n) = c_2 n$ if $n \leq 5p$, where c_1 and c_2 are constants. Since Steps 3 and 4 are symmetric, we only give the details of Step 3 and establish its correctness and its $O(n)$ time complexity.

3.1.2 Finding foreign neighbors

In this subsection, we show that Step 3 of algorithm NEAREST-NEIGHBORS can be performed in $O(n)$ time on a RAM/ARRAY(1). Recall that, at the beginning of Step 3, each point u knows its local nearest neighbor, $N_1(u)$. We compute $N_2(u)$, i.e. a lower foreign neighbor of u, for all $u \in S$ by performing a downward sweep, as follows.

In order to sweep the points and find $N_2(u)$ for every $u \in S$, we first partition S into n/p subsets $H_1, H_2, \ldots, H_{n/p}$ in top-to-bottom order by horizontal cut-lines (i.e., the points in

each H_j have larger y coordinates than those in H_{j+1}). It has been shown in [AFK88] and in our Lemma 2.1 that the partitioning process can be done in $O(n)$ time if the points in each S_i are available sorted by decreasing y coordinate (recall that a by-product of our algorithm is to sort the points by decreasing y coordinate). We then use these horizontal cut-lines as sweep lines to perform the downward sweep, using the p-processor array to achieve the $O(n)$ time performance for Step 3. The crucial observation is that, during the sweep, we only need to maintain a set of $O(p)$ points. Let C_j (which will be defined later) be some set of points which contains at least all the points $u \in S - H_j$ that have $N_2(u) \in H_j$. We will show later that, if we choose C_j in a suitable fashion, then the number of points in each C_j is at most $4p$ (specifically, at most 4 from each S_i). The downward sweep which finds $N_2(u)$ for every $u \in S$ consists of solving the all nearest neighbors problem on $H_j \cup C_j$, in turn for $j = 1, 2, \ldots, n/p$, by direct use of the p-processor array $O(n/p)$ times at a cost of $O(p)$ time each. (It takes time $O(p)$ for each $H_j \cup C_j$, since there are at most $5p$ points in each $H_j \cup C_j$.) We still have to define C_j, and to show that the C_j's can indeed be computed in $O(n)$ time.

We choose set C_j as follows. Let lines $l_0, l_1, \ldots, l_p$ be the vertical cut-lines in left-to-right order (i.e., $x(l_a) < x(l_b)$ if $a < b$) used in Step 2 to partition S (thus the points in S_i are to the right of l_{i-1} and to the left of l_i). Let $h_0, h_1, \ldots, h_{n/p}$ be the horizontal cut-lines in top-to-bottom order (i.e. $y(h_a) > y(h_b)$ if $a < b$) used in Step 3 to partition S (thus the points in H_j are below h_{j-1} and above h_j). Let point $q_{i,j}$ denote the intersection of lines l_i and h_j. We define C_j to be the set of points u such that u is above line h_{j-1} and, if $u \in S_i$, then the smaller of $d(u, q_{i-1,j-1})$ and $d(u, q_{i,j-1})$ is less than $d(u, N_1(u))$. In other words:

$$C_j = \cup_{i=1}^{p} \{u \in S_i \mid y(u) > y(h_{j-1}) \text{ and } d(u, N_1(u)) > \min\{d(u, q_{i-1,j-1}),\ d(u, q_{i,j-1})\}\}.$$

In the following two lemmas, we show that C_j contains at least all the points $u \in S - H_j$ that have $N_2(u) \in H_j$, and that the cardinality of C_j is at most $4p$. Furthermore, we show that C_{j+1} can be computed from $C_j \cup H_j$ in $O(p)$ time. Note that $h_1, h_2, \ldots, h_{n/p}$ were obtained in $O(n)$ time while we partitioned S into $H_1, H_2, \ldots, H_{n/p}$.

Let the S_i's, H_i's, C_i's and $q_{i,j}$'s be defined as above.

Lemma 3.1 *If $u \in S_i - H_j$ is a point with $N_2(u) \in H_j$, then u is in C_j.*

Proof: Suppose $u \in S_i - H_j$ is a point with $N_2(u) \in H_j$. Since $u \notin H_j$ and $y(u) > y(N_2(u))$, $y(u)$ is larger than $y(h_{j-1})$. WLOG, assume that $N_2(u) \in H_j$ is to the left of l_{i-1}. Consider

the triangle formed by points u, $N_2(u)$ and $q_{i-1,j-1}$. The edge with endpoints u and $N_2(u)$ is the longest one in this triangle. Thus, $d(u, N_1(u)) > d(u, N_2(u)) > d(u, q_{i-1,j-1}) \geq \min\{d(u, q_{i-1,j-1}),\ d(u, q_{i,j-1})\}$. Thus, u is in C_j. □

Lemma 3.2 *The cardinality of each C_j is at most $4p$.*

Proof: It suffices to prove that every S_i has at most four points that are in C_j. Since every point in C_j is above line h_{j-1}, we first prove that for each S_i there are at most two points $u \in S_i$ above h_{j-1} and having $d(u, N_1(u)) > d(u, q_{i-1,j-1})$. Assume u_1 and u_2 are two such points. The angle $u_1 q_{i-1,j-1} u_2$ must be larger than $\pi/3$, since otherwise we would have $d(u_1, u_2) < \max\{d(u_1, N_1(u_1)), d(u_2, N_1(u_2))\}$, contradicting the definition of $N_1(u_1)$ and $N_2(u_2)$. This implies that there are at most two such points. Similarly, we can prove that there are at most two points $u \in S_i$ above h_{j-1} and having $d(u, N_1(u)) > d(u, q_{i,j-1})$. □

A lemma to the above one was given in [WW88] to obtain an optimal PRAM algorithm for the all nearest neighbors problem. In that lemma, they show that when $p = 2$, $|C_j| \leq 8$.

Lemma 3.3 *C_{j+1} can be computed from $C_j \cup H_j$ in $O(p)$ time.*

Proof: Since $d(u, q_{i-1,j-1})$ (resp., $d(u, q_{i,j-1})$) is less than $d(u, q_{i-1,j})$ (resp., $d(u, q_{i,j})$) for every $u \in S_i$ and above line h_{j-1}, it is clear that $C_{j+1} \subseteq C_j \cup H_j$. Thus, we can identify the points in C_{j+1} by comparing $d(u, N_1(u))$ to $\min\{d(u, q_{i-1,j}),\ d(u, q_{i,j})\}$, for every $u \in H_j \cup C_j$, if $u \in S_i$, and this can be done in $O(p)$ time sequentially since there are only $O(p)$ points in $C_j \cup H_j$. Note that $C_1 = \emptyset$. □

We therefore perform Step 3 in $O(n)$ time by doing the following for $j = 1, 2, \ldots, n/p$ in turn: first, identifying the points in C_j, and then solving the all nearest neighbors problem on $H_j \cup C_j$ by direct use of the p-processor array. Step 4 can be done in $O(n)$ time similarly. This completes the sketch of the $O(n \log n / \log p)$ time algorithm on the RAM/ARRAY(1).

Theorem 3.1 *Given a set S of n points in the plane, for every $u \in S$, we can find a nearest neighbor $N(u)$ of u in $O(n \log n / \log p)$ time on a RAM/ARRAY(1).*

3.2 The measure and perimeter of the union of rectangles

Problems involving rectangles have many applications in VLSI design and pattern recognition. An important class of the rectangles are the isothetic ones, which are rectangles with sides parallel to the two coordinate axes. One of the most extensively studied

problems concerning isothetic rectangles is to compute the measure, i.e. the area, of the union of n isothetic rectangles. Many optimal $O(n \log n)$ sequential algorithms are known [Ben77, Gut84, LW80], and an optimal $O(\sqrt{n})$ time 2-dimensional mesh algorithm was given by Miller and Stout [MS89]. A linear time systolic solution on a linear array can be derived from a quadratic sequential algorithms which basically scans the rectangles from left to right. However, it is not clear whether or not any of the above algorithms can be implemented on a RAM/ARRAY(1) in $O(n \log n / \log p)$ time. In this section, we present an $O(n \log n / \log p)$ time algorithm on a RAM/ARRAY(1) to compute the measure of the union of n isothetic rectangles. Our algorithm can be easily modified to compute the perimeter, i.e. the length of the boundary, of their union.

3.2.1 Definitions and overview

Define a *left* (resp., *right*) representative of an isothetic rectangle to be its left (resp., right) vertical edge. Assume that a rectangle is "attached" to each of its two representatives, so that we can retrieve all the information about this rectangle from either of them. For any representative r, define $RECT(r)$ to be the isothetic rectangle attached to it. For any set R of representatives, define $RECT(R)$ to be the set $\{RECT(r) \mid r \in R\}$. Note that $|R|/2 \leq |RECT(R)| \leq |R|$ since every rectangle has at most two representatives. Define a *frame* to be an isothetic rectangular region in the plane. A representative r is contained in a frame f if the vertical edge r is contained completely in f. When we say that R is sorted from left to right, we mean that the vertical "edges" in R are sorted by increasing x coordinate.

To compute the area of the union of n isothetic rectangles, we first form a set R of $N = 2n$ representatives, consisting of both the left and right representatives of each given rectangle. We also select a frame f which encloses all the given rectangles (thus all the representatives in R). The problem then becomes that of computing the area of the intersection between the frame f and the union of rectangles in $RECT(R)$. Our algorithm actually has f and R as input. To simplify the exposition, we assume that the x coordinates (resp., y coordinates) of the vertical edges (resp., horizontal edges) of the rectangles in $RECT(R)$ are pairwise distinct.

To compute the intersection between the frame f and the union of rectangles in $RECT(R)$, we will partition the frame f into a collection $S_f(R) = \{s_1, s_2, \ldots, s_w\}$ of w

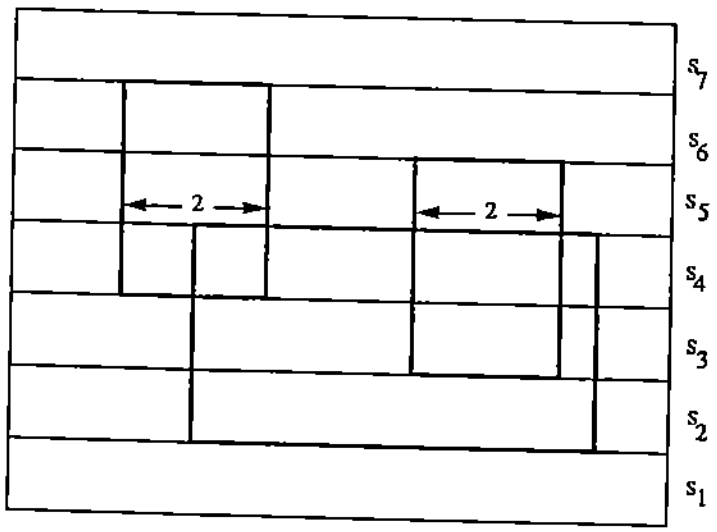

Figure 3: $S_f(R) = \{s_1, s_2, s_3, s_4, s_5, s_6, s_7\}$

($w \leq 2|R| + 1$) rectangular horizontal stripes and, for each stripe s_i in $S_f(R)$, we compute the intersection between s_i and the union of rectangles in $RECT(R)$ (see Figure 3). The partition is determined by the rectangles in $RECT(R)$ as each horizontal stripe boundary is along one horizontal edge, and each horizontal edge is contained in one horizontal stripe boundary. Note that the intersection between any stripe $s_i \in S_f(R)$ and the union of rectangles in $RECT(R)$ is defined by a set of disjoint x-intervals. For every $s_i \in S_f(R)$, we use $len_R(s_i)$ to denote the total length of the x-intervals which define the intersection between stripe s_i and the union of rectangles in $RECT(R)$. In Figure 3, $len_R(s_5) = 2 + 2 = 4$. Given $S_f(R)$ and $len_R(s_i)$ for every $s_i \in S_f(R)$, it is obvious that the area of the union of rectangles can be computed in $O(n)$ time. Our algorithm computes and returns $S_f(R)$ and $len_R(s_i)$ for all $s_i \in S_f(R)$.

The idea of partitioning an enclosing frame into stripes and computing the intersection of each stripe and the union of the given rectangles, in order to compute the measure of their union, was first used by Güting in [Gut84] to develop the first optimal divide-and-conquer algorithm for this problem. Our algorithm involves substantially different techniques in the combining step. We next show how to construct $S_f(R)$, and compute $len_R(s_i)$ for all $s_i \in S_f(R)$ in time $O(n \log n / \log p)$ on a RAM/ARRAY(1).

3.2.2 The algorithm

The initial call to the recursive procedure MEASURE-OF-UNION gives it as input a frame f containing all of the n rectangles we wish to consider, together with a set R containing the $2n$ representatives of these rectangles. Thus initially we have $|R| = 2|RECT(R)|$, and every rectangle of $RECT(R)$ is completely contained in f. However, these two conditions are *not* maintained through the recursion. What is maintained is the condition that every rectangle in $RECT(R)$ has at least one representative contained in f, and that R contains all the representatives of $RECT(R)$ that are enclosed in f (thus a rectangle of $RECT(R)$ can have one of its representatives outside of f, in which case that representative is not in R). Thus the relationship $|RECT(R)| \leq |R| \leq 2|RECT(R)|$ is maintained. Our measure of problem size shall be $|R|$ rather than $|RECT(R)|$. Let $|R| = N$.

Since sorting can be done in time $O(N \log N / \log p)$ on a RAM/ARRAY(1), we assume the input representatives R are already sorted by increasing x-coordinate. For every representative r, we use $left(r)$ (resp., $right(r)$, $top(r)$, $bottom(r)$) to denote the left (resp., right, top, bottom) edge of the rectangle attached to r. Since $RECT(r)$, for $r \subset f$, need not be contained in f, the definition of $S_f(R)$ needs a minor modification : the stripes in it are now defined by the horizontal lines through the endpoints of the vertical edges in R.

Algorithm MEASURE-OF-UNION

Input: A frame f (a rectangle) and a collection R of vertical line segments contained in f. To each such line segment r is "attached" a rectangle $RECT(r)$ such that r is one of the two representatives of $RECT(r)$, and the other representative of $RECT(r)$ is also in the set R iff that representative is also contained in f. Let $|R| = N$.

Output: The set of stripes $S_f(R)$ sorted by y coordinates, and arrays $rect_R$ and len_R such that, for any $s \in S_f(R)$, $rect_R(s)$ is the rectangle in $RECT(R)$ whose bottom edge coincides with the bottom boundary of s (if any), and $len_R(s)$ is the length of the x-intervals formed by the intersection of s and the union of rectangles in $RECT(R)$.

1. If $N \leq 10p$ then solve the problem in $O(N)$ time by direct use of the p-processor array [MS89].

2. Take p vertical lines such that there is exactly one of them between the Ni/pth and $(Ni/p + 1)$th representatives in R, for $i = 1, 2, \ldots, p$. These vertical lines induce a

partition of the set R into sets R_1, R_2, ..., R_p of size N/p each, and of the frame f into frames f_1, f_2, ..., f_p, in left-to-right order. Note that the representatives in R_j are contained in frame f_j, but the rectangles of $RECT(R_j)$ may extend outside of f_j. Solve each subproblem defined by R_j and f_j recursively. The recursive call for R_j and f_j returns $S_{f_j}(R_j)$ sorted by increasing y coordinate, and arrays $rect_{R_j}$ and len_{R_j}.

3. Perform an upward "sweep" of the stripes in $\cup_{j=1}^{p} S_{f_j}(R_j)$ (and hence the rectangles in $RECT(R)$) with jumps of p stripes at a time, to construct $S_f(R)$, and compute $rect_R(s)$ and $len_R(s)$ for every $s \in S_f(R)$. The details of this are explained below.

End of Algorithm MEASURE-OF-UNION

To establish the correctness and $O(N \log N / \log p)$ time complexity of the above algorithm, we show how to perform the upward sweep of Step 3 in $O(N)$ time. The idea is that, in the ith jump of the sweep, we construct $X_i \subset S_f(R)$, and compute $rect_R(s)$ and $len_R(s)$ for every $s \in X_i$, where X_i is the set of stripes in $S_f(R)$ corresponding to that ith jump (i.e. between the $(i-1)$th and ith sweep lines). Meanwhile, we maintain the set C_i of stripes in $\cup_{j=1}^{p} S_{f_j}(R_j)$ which intersect the ith sweep line properly, and an array $span_i(1:p)$ where $span_i(j)$ is the maximum y coordinate of the top edges of the rectangles which intersect properly the ith sweep line and span subframe f_j (a rectangle r *spans* a frame f_j if the x-interval of r contains completely the x-interval of f_j). We next give the details of the sweep.

We define a total order among stripes by the y coordinates of their bottom boundaries in increasing order. Let m be the cardinality of $\cup_{j=1}^{p} S_{f_j}(R_j)$. Let l_i be the horizontal line coinciding with the bottom boundary of the (ip)th stripe in $\cup_{j=1}^{p} S_{f_j}(R_j)$ (i.e. l_i is the ith sweep line). The lines l_0, l_1, ..., $l_{m/p}$, $l_{m/p+1}$ partition the set $\cup_{j=1}^{p} S_{f_j}(R_j)$ into sets G_1, G_2, ..., $G_{m/p}$ such that G_i is the group of stripes in $\cup_{j=1}^{p} S_{f_j}(R_j)$ whose bottom boundaries are on or above line l_{i-1} and below line l_i. (Assume l_0 is the line $y = -\infty$ and $l_{m/p+1}$ is the line $y = y(top(f))$.) In Figure 4, $G_3 = \{t_2, t_3, t_7, t_{11}\}$. Since each $S_{f_j}(R_j)$ is sorted and monotone (i.e. for any horizontal line, there is at most one stripe in $S_{f_j}(R_j)$ which intersects this line properly), the partitioning process can be done in $O(m) = O(N)$ time as in lemma 2.1. We then perform the upward sweep using lines l_1, l_2, ..., $l_{m/p+1}$ as sweep lines. When we move the sweep line from l_{i-1} to l_i, we compute X_i, C_i, and $len_R(s)$ and $rect_R(s)$ for every $s \in X_i$, where X_i is the set of stripes in $S_f(R)$ which are between lines l_{i-1} and l_i,

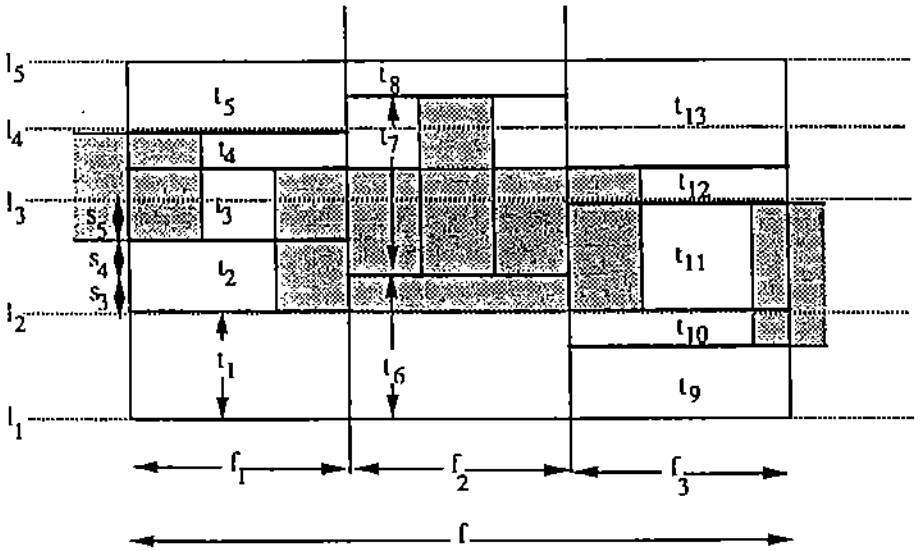

Figure 4: $G_3 = \{t_2, t_3, t_7, t_{11}\}$, $C_3 = \{t_3, t_7\}$, $X_3 = \{s_3, s_4, s_5\}$

C_i is the set of stripes in $\cup_{j=1}^{i} G_j$ which intersect line l_i properly (hence $|C_i| \leq p$). We also maintain an array $span_i(1:p)$ such that the value of $span_i(j)$ is the maximum y coordinate of the top edges of rectangles which intersect line l_i properly and span the x-interval of the frame f_j. In Figure 4, $X_3 = \{s_3, s_4, s_5\}$, $C_3 = \{t_3, t_7\}$ and $span_3(2)$ is the y coordinate of the top edge of the rectangle spanning f_2. Note that $|X_i| \leq p+1$ and $|C_i| \leq p-1$, for $1 \leq i \leq m/p$. Since $|G_i \cup C_{i-1}| \leq 2p$, it is clear that set C_i and array $span_i(1:p)$ can be computed in $O(p)$ time sequentially or using the attached array constant times. In the following lemma, we show that, given G_i, C_{i-1} and array $span_{i-1}(1:p)$, we can compute set X_i and $len_R(s)$, for every $s \in X_i$, in $O(p)$ time on a RAM/ARRAY(1). Once $S_f(R)$ is computed, we can easily compute $rect_R(s)$ for every $s \in S_f(R)$.

Lemma 3.4 *Given G_i, C_{i-1} and array $span_{i-1}(1:p)$, we can compute set X_i and $len_R(s)$ for every $s \in X_i$, in $O(p)$ time on a RAM/ARRAY(1).*

Proof: To construct X_i and compute $len_R(s)$ for every $s \in X_i$, we solve an instance of the measure of union problem of $O(p)$ rectangles. These $O(p)$ rectangles are defined as follows.

1. Consider the rectangles in $RECT(R)$ which have one of their horizontal edges between

h_1 and h_2. For each of these rectangles, we exclude its portions which are not between l_{i-1} and l_i and its portions which do not (horizontally) span any subframe (the portions so excluded have already been taken care of by the recursive calls). There are no more than $2p$ rectangles so created.

2. We then consider the stripes in $G_i \cap C_{i-1}$. For each such stripe s, we create a new rectangle r which is a portion of s, shares the horizontal boundaries with s, and has its width equal to $len_{R_i}(s)$ (it can be located anywhere in s, and its purpose is to encapsulate the portions taken care of by the recursive calls). For each created rectangle, exclude its portions which are not between l_{i-1} and l_i. There are no more than $2p$ rectangles so created.

3. We then create p rectangles from array $span_{i-1}$ as follows. For each $span_{i-1}(j)$, we create a new rectangle whose x interval is the same as that of subframe f_j, and whose y interval is from $y(l_{i-1})$ to $\min\{span_{i-1}(j), y(l_i)\}$. There are p rectangles so created.

Let R' be the set the representatives of the rectangles created as above and f' be the portion of f between l_{i-1} and l_i. It is clear that $X_i = S_{f'}(R')$. We next show that, for every $s \in X_i$, $len_R(s) = len_{R'}(s)$. Let $len_R(s \cap f_j)$ represent the intersection of $s \cap f_j$ with the union of the rectangles of $RECT(R)$, and $len_{R-R_j}(s \cap f_j)$ represent the intersection of $s \cap f_j$ with the union of the rectangles of $RECT(R) - RECT(R_j)$. Let us consider $len_{R-R_j}(s \cap f_j)$. If $len_{R-R_j}(s \cap f_j) = 0$, i.e. $s \cap f_j$ does not intersect the union of the rectangles of $RECT(R) - RECT(R_j)$, then $len_R(s \cap f_j) = len_{R_j}(s \cap f_j)$ and hence $len_R(s \cap f_j)$ is defined by the intersection of $s \cap f_j$ with the rectangle of $RECT(R')$ that is created from the stripe of $S_{f_j}(R_j)$ containing $s \cap f_j$. If $len_{R-R_j}(s \cap f_j) > 0$, then $s \cap f_j$ is covered completely by at least one rectangle of $RECT(R) - RECT(R_j)$ since none of its representatives is in R_j. Since that rectangle must intersect the region between l_{i-1} and l_i and must span the x interval of f_j, $s \cap f_j$ is covered completely by the one of $RECT(R')$ created either from that rectangle or from the value of $span_{i-1}(j)$. Thus, $len_R(s \cap f_j) = len_{R'}(s \cap f_j)$. Therefore, $len_R(s) = len_{R'}(s)$ for every $s \in X_i$. We thus compute X_i and $len_R(s)$ for every $s \in X_i$ by computing the measure of the union of rectangles of $RECT(R')$. Since $|R'| \leq 10p$, the measure of the union of rectangles of $RECT(R')$ can be computed in $O(p)$ time, using the p-processor array a constant number of times. □

Therefore, the overall time complexity, $T(N)$, of algorithm MEASURE-OF-UNION satisfies the recurrence $T(N) = pT(N/p) + c_1 N$ if $N > 10p$, and $T(N) = c_2 N$ if $N \leq 10p$,

where c_1 and c_2 are constants. This implies $T(N) = O(N \log N / \log p) = O(n \log n / \log p)$. We therefore have the following theorem.

Theorem 3.2 *Given a set R of N representatives of isothetic rectangles and a frame f containing these given representatives, algorithm MEASURE-OF-UNION computes the area of the intersection between the union of rectangles in $RECT(R)$ and the frame f in time $O(N \log N / \log p)$ on a RAM/ARRAY(1).*

3.3 3-dimensional maxima

Let $P = \{p_1, p_2, \ldots, p_n\}$ be a set of points in $\Re^3$. For simplicity, we assume the x coordinates (resp., y, z coordinates) of the points in P are pairwise distinct. For any two point p_i and p_j, p_i is *dominated* by p_j if $x(p_i) < x(p_j)$, $y(p_i) < y(p_j)$ and $z(p_i) < z(p_j)$. A point $p_i \in P$ is said to be a *maximum* if it is not dominated by any other point in P. The 3-dimensional maxima problems, then, is to compute the set, $M(P)$, of the maxima in P. Define $D(p_i)$ as the region dominated by point p_i, *i.e.* $D(p_i) = \{(x, y, z) \mid x < x(p_i), y < y(p_i), z < z(p_i)\}$. $M(P)$ then is the set of points in P which are not in the region $D(P) = \cup_{i=1}^{n} D(p_i)$.

Recall that a plane H is *horizontal* if H is parallel to xy plane. To compute $M(P)$, we first partition the points in P into subsets $P_1, P_2, \ldots, P_p$ of size n/p each in top-to-bottom order, using horizontal cut-planes $H_0, H_1, \ldots, H_p$ (i.e. P_i is the set of points between H_{i-1} and H_i). We then solve the subproblem defined by each P_i recursively. The recursive call for P_i returns $M(P_i)$ and $R(P_i)$, where $R(P_i)$ is a description of the part of $D(P_i)$ below plane H_i (see Figure 5). Note that $R(P_i)$ may contain points in P but not in P_i. To compute $M(P)$, we must remove the points in $\cup_{j=1}^{p} M(P_j)$ which are in region $\cup_{j=1}^{p} R(P_j)$. The idea for doing this is based on the observation that each $R(P_i)$ can be represented by a monotone chain of $O(|P_i|)$ horizontal (i.e. perpendicular to the yz plane) line segments. The property of monotonicity ensures us that $\cup_{i=1}^{p} R(P_i)$ can be partitioned into $O(n/p)$ chunks of size $O(p)$ each, using n/p vertical cut-planes in $O(n)$ time as in Lemma 2.1. (The size of each chunk is the number of line segments used to describe it).

Observation 3.1 *Let H_i, P_i and $R(P_i)$ be defined as above. Then, $R(P_i)$ can be represented by a set of no more than $|P_i|$ line segments that form a monotone chain.*

Proof: Let $P_i = \{v_1, v_2, \ldots, v_w\}$. Let $L_i = \{l_1, l_2, \ldots, l_w\}$ where l_j is the projection on H_i of the horizontal half line $\{(x, y, z) \mid x \leq x(v_j),\ y = y(v_j)\ and\ z = z(v_j)\}$. As in Figure

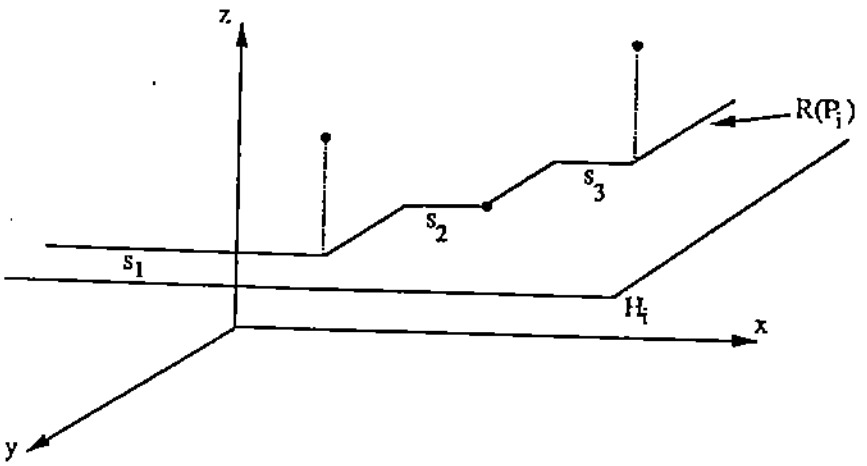

Figure 5: $R(P_i) = \{s_1, s_2, s_3\}$

5, the boundary of the region $R(P_i)$ is defined by the boundary of the visible region on H_i defined by set L_i and the point $(0, -\infty, z(H_i))$. Since this visible region can be represented by a monotone chain, $R(P_i)$ can be represented by a monotone chain. The number of line segments in that chain is no more than $|P_i|$. □

We then have the following $O(n \log n / \log p)$ algorithm. Since sorting can be done in $O(n \log n / \log p)$ time, we assume the input points in P are already sorted by decreasing z coordinate.

Algorithm MAXIMA

Input: A set of points $P = \{p_1, p_2, \ldots, p_n\}$ in $\Re_3$ sorted by decreasing z coordinate. For simplicity, assume the x (resp., y, z) coordinates of the points in P are pairwise distinct.

Output: The set $M(P) = \{q_1, q_2, \ldots, q_u\}$ of maxima in P sorted by increasing x coordinate, and the region $R(P) = \{s_1, s_2, \ldots, s_v\}$ which is sorted and forms a monotone chain.

1. If $n \leq 3p$ then solve the problem by direct use of the p-processor array in $O(n)$ time.

2. Partition the set P into p subsets P_1, P_2, ..., P_p of size n/p each in top-to-bottom order, using horizontal (i.e. parallel to the xy plane) cut-planes. Note that the points in P_i have larger z coordinate than those in in P_j if $i < j$. Recursively solve the

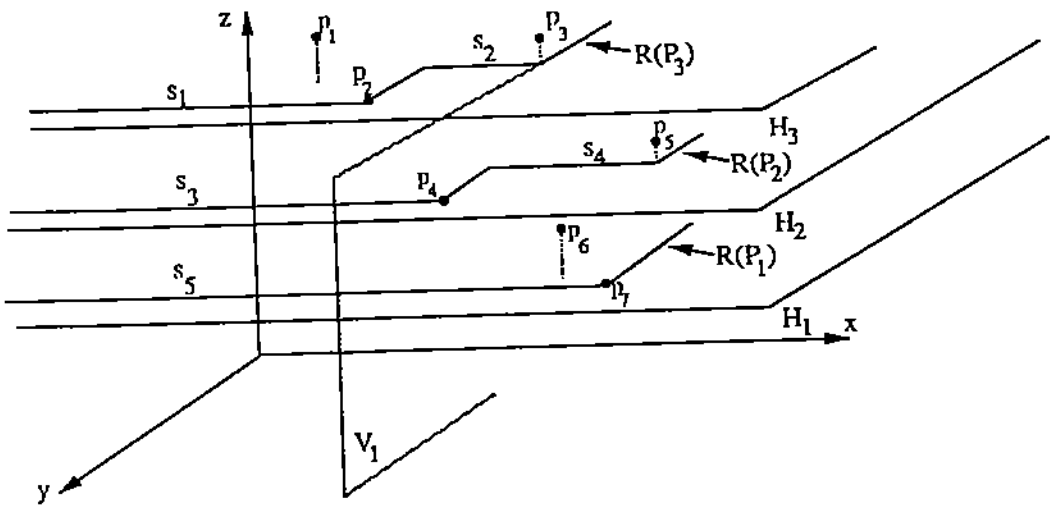

Figure 6: $G_1 = \{s_1, s_2, s_3\}$, $C_1 = \{s_4, s_5\}$ and $Q_1 = \{p_1, p_2, p_3, p_4\}$

subproblem defined by each P_i one by one. The recursive call for each P_i returns $M(P_i)$ and $R(P_i)$.

3. Partition the space into $O(n/p)$ chunks using $O(n/p)$ vertical (i.e. parallel to the yz plane) cut-planes such that the part of the region $\cup_{j=1}^{p} R(P_j)$ in each chunk is represented by $O(p)$ line segments. For each chunk, do the following two steps : (i) identify those line segments in $\cup_{j=1}^{p} R(P_j)$ and points in $\cup_{j=1}^{p} M(P_j)$ which are in that chunk, (ii) compute $M(P)$ and $R(P)$ by computing the subsolutions in each chunk (i.e. the portion of $M(P)$ and $R(P)$ in that chunk). The details of this step is explained below.

End of Algorithm MAXIMA

The details of Step 3 are as follows. We define a total order among the line segments in $\cup_{j=1}^{p} R(P_j)$ by the x coordinates of their right endpoints. Let $m = |\cup_{j=1}^{p} R(P_j)|$, and note that $m \leq n$. Let V_i be the vertical plane defined by the x coordinate of the right endpoint of the (ip)th line segment in $\cup_{j=1}^{p} R(P_j)$ ($V_0 = \{(x, y, z) \mid x = -\infty\}$). Given vertical planes $V_0, V_1, \ldots, V_{m/p}$, we partition the set $\cup_{j=1}^{p} R(P_j)$ into sets $G_1, G_2, \ldots, G_{m/p}$ and create sets $C_1, C_2, \ldots, C_{m/p}$ such that G_i consists of the line segments whose right endpoints are to the right of plane V_{i-1} and on or to the left of plane V_i, and C_i consists of the line segments

that intersect plane V_i properly. In Figure 6, $G_1 = \{s_1, s_2, s_3\}$ and $C_1 = \{s_4, s_5\}$. Since each $R(P_i)$ is sorted and monotone, all the G_i's and C_i's can be obtained in $O(m) = O(n)$ time as in lemma 2.1. Note that $|G_i| = p$ and $|C_i| \leq p$, and $G_i \cup C_i$ contains all the lines segments which represent the part of $\cup_{j=1}^{p} R(P_j)$ between planes V_{i-1} and V_i. We next identify the subset Q_i of $\cup_{j=1}^{p} M(P_j)$ that is between V_{i-1} and V_i for $i = 1, 2, \ldots, n/p$, and then for each Q_i, we form $\lceil |Q_i|/p \rceil$ instances of $O(p)$-sized problems (each instance consists of $G_i \cup C_i$ and no more than p points from Q_i). Each such instance is solved in $O(p)$ time, using the p-processor array a constant number of times. To partition $\cup_{j=1}^{p} M(P_j)$ into Q_1, $Q_2, \ldots, Q_{m/p}$, we do the following: (i) form a sorted list Q by merging $M(P_1)$, $M(P_2), \ldots,$ $M(P_{n/p})$, (ii) view Q as a sequence of blocks of size p each, and, for each block of Q, do binary searches to partition that block if its points belong to more than one Q_is. The total time for (i) is $O(n)$ by Lemma 2.1, and the total time for (ii) is $O((n \log p)/p)$ since there are only $O(n/p)$ binary searches and each of them takes $O(\log p)$ time.

Therefore, the time complexity of algorithm MAXIMA is $O(n \log n / \log p)$.

Theorem 3.3 *Given $P = \{p_1, p_2, \ldots, p_n\}$ of points in $\Re^3$, algorithm MAXIMA computes the maxima in P in time $O(n \log n / \log p)$ on a RAM/ARRAY(1).*

3.4 Visibility from a point

Given a set of line segments $S=\{s_1, s_2, s_3, \ldots, s_n\}$, which are opaque and do not intersect except possibly at their endpoints, and a point v in the plane, the visibility problem is to determine the region of the plane visible from v. WLOG, we assume v is the point $(0, -\infty)$.

Optimal mesh algorithm for this problem can be easily derived and is omitted. In this section, we show how to to solve a problem of size $n > p$ in time $O(n \log n / \log p)$ on a RAM/ARRAY(1).

Observe that the upper boundary of the visible region of S is a chain monotone with the x direction. In algorithm VISIBILITY, a visible region will be represented by a sorted and monotone set of line segments which describes the upper boundary of this visible region.

Algorithm VISIBILITY

Input: $S = \{s_1, s_2, \ldots, s_n\}$ is a set of nonintersecting line segments except possibly at their endpoints. For simplicity, we assume the x coordinates of distinct endpoints are pairwise distint.

Output: A sorted and monotone set $R = \{t_1, t_2, \ldots, t_w\}$ of line segments which describes the upper boundary of the visible region, where $1 \leq w \leq 2n$.

1. If $n \leq 2p$ then solve this problem in $O(n)$ time by direct use of the p-processor array.

2. Partition the set S into sets S_1, S_2, ..., S_p of size n/p each and recursively solve the visibility problem defined by each S_i. The recursive call for S_i returns R_i, the visibility of S_i. Note that $|R_i| \leq 2n/p$.

3. Partition the set $\cup_{j=1}^{p} R_p$ into $O(n/p)$ subsets of size $O(p)$ each using $O(n/p)$ vertical lines. (A line segment may be in several subsets if it crosses several regions separated by these vertical lines.) The part of R between two consecutive vertical lines is computed by solving the visibility problem defined by the set of lines segments between them, by using the p-processor array a constant number of times, as explained below.

End of Algorithm VISIBILITY

To establish the $O(n \log n / \log p)$ time complexity of the above algorithm, we show how to perform Step 3 in $O(n)$ time on a RAM/ARRAY(1). The idea is based on the property of the monotonicity of each R_i. Let $m = \sum_{i=1}^{p} |R_i|$, and note that $m \leq 2n$. Define a total order between line segments by the x coordinates of their right endpoints. Let l_i be the vertical line defined by the x coordinate of the right endpoint of the (ip)th line segment in $\cup_{j=1}^{p} R_j$, for $i = 1, 2, \ldots, m/p$. We then compute the visible region R by computing the part of R between l_{i-1} and l_i, for $i = 1, 2, \ldots, m/p$, as follows. Given lines $l_1, l_2, \ldots, l_{m/p}$, we partition the set $\cup_{j=1}^{p} R_j$ into sets G_1, G_2, ..., $G_{m/p}$ and create sets C_1, C_2, ..., $C_{m/p}$, such that G_i is composed of the line segments whose right endpoint is to the right of line l_{i-1} and on or to the left of line l_i, and C_i is composed of the line segments which properly intersect line l_i. (Assume l_0 is the line $x = -\infty$ and $C_{m/p} = \emptyset$.) The monotonicity of each sorted set R_i implies that the partitioning and creation can be done in $O(m) = O(n)$ time as in lemma 2.1, and that $|C_i| \leq p$ for each $1 \leq i \leq n/p$. Note that $G_i \cup C_i$ contains all the line segments in $\cup_{j=1}^{p} R_j$ which intersect the region between lines l_{i-1} and l_i. We then compute the part of R between lines l_{i-1} and l_i by solving the visibility problem defined by $G_i \cup C_i$ in $O(p)$ time on the p-processor array. Step 3 of algorithm VISIBILITY therefore takes $O(n)$ on a RAM/ARRAY(1). Therefore, the overall time complexity of algorithm VISIBILITY is $O(n \log n / \log p)$.

Theorem 3.4 *Given a set of line segments $S = \{s_1, s_2, \ldots, s_n\}$ in the plane, which are opaque and do not intersect except possibly at their endpoints, algorithm VISIBILITY computes the visibility of S from $p = (0, -\infty)$ in time $O(n \log n / \log p)$ on a RAM/ARRAY(1).*

3.5 Dominance counting between two sets

Given two sets of points $P = \{p_1, p_2, \ldots, p_m\}$ and $Q = \{q_1, q_2, \ldots, q_n\}$ in the plane, the dominance counting problem is to determine, for every point $p_i \in P$, the number of points in Q which are dominated by p_i. (Recall that, for any two points u and v, u is dominated by v if $x(u) < x(v)$ and $y(u) < y(v)$.) In the algorithm DOMCOUNT, we show how to solve the problem in $O(N \log N / \log p)$ time on a RAM/ARRAY(1), where $N = m + n$. For simplicity, we assume the x (resp., y) coordinates of points in $P \cup Q$ are pairwise distinct. Since sorting can be done in $O(N \log N / \log p)$ time, we assume also that the points in P (resp., Q) are sorted by increasing x coordinate.

Algorithm DOMCOUNT:

Input: Two sets of points $P = \{p_1, p_2, \ldots, p_m\}$ and $Q = \{q_1, q_2, \ldots, q_n\}$ in the plane sorted by increasing x coordinate. For simplicity, we assume the x (resp., y) coordinates of points in $P \cup Q$ are pairwise distinct.

Output: A list of points $X = \{u_1, u_2, \ldots, u_{m+n}\}$, which are the points in $P \cup Q$ sorted by increasing y coordinates. For each point $u \in X$, we also have a count $C(u)$, which is the number of points in Q dominated by u.

1. Merge the points in P and Q into one set $V = \{v_1, v_2, \ldots, v_N\}$, sorted by increasing x coordinate, where $N = m + n$. Also, we "mark" each point in V if it came from Q. The count $C(v)$, for every $v \in V$, is initialized to 0.

2. If $N \leq p$ then solve the problem in $O(N)$ time by direct use of the p-processor array. This is straightforward and omitted.

3. Using p vertical lines, partition the points in V into sets V_1, V_2, ..., V_p of size N/p each in left-to-right order. The points in V_i are to the left of those in V_j if $i < j$. For each i, recursively solve the problem on V_i. The recursive call for V_i returns a sorted list X_i and a count $C_i(u)$ for every $u \in X_i$, where X_i is the list of points in V_i sorted by increasing y coordinate, and $C_i(u)$ is the number of points in $Q \cap V_i$ that are dominated by u.

4. Compute X and $C(u)$ for every $u \in X$ by using the information returned by the p recursive calls of Step 3. The details of this step are explained below.

End of Algorithm DOMCOUNT

The details of Step 4 are based on the following observation. Let H_j be the subset of points of V whose y coordinate is larger than the $(j-1)p$th and less than or equal to the (jp)th y coordinate of points in V. Suppose v is a point in $V_i \cap H_j$ and let $D(v)$ denote the set of points in Q which are dominated by v. The set $D(v)$ is the union of three disjoint sets $D(v) \cap V_i$, $D(v) \cap (H_j - V_i)$ and $D(v) \cap (\cup_{k=1}^{i-1} V_k) - H_j$ (see Figure 7). Let $S(v)$, $W(v)$ and $SW(v)$ denote the respective cardinalities of these three sets $D(v) \cap V_i$, $D(v) \cap (H_j - V_i)$ and $D(v) \cap (\cup_{k=1}^{i-1} V_k) - H_j$. Note that $S(v)$ was already computed by the recursive call for V_i. In the combining step, we need to compute $W(v)$ and $SW(v)$, for every point $v \in V$.

In order to compute $W(v)$ and $SW(v)$, we do the following operations. As in lemma 2.1, in $O(n)$ time, we partition the set $\cup_{i=1}^{p} X_i$ into sets $H_1, H_2, \ldots, H_{n/p}$, using the horizontal lines defined by the pth, $2p$th,...,nth smallest y coordinates of points in $\cup_{i=1}^{p} X_i$. Note that the points in H_i are above those in H_j if $i > j$. For each point $v \in H_j$, we also mark it with the index of the set X_i (hence the set V_i) which v came from. Let $Count_j(i)$ be the number of points in $Q \cap V_i$ which are below all the points in H_j. The array $Count_j(1:p)$ can be computed while the set of points in H_j are identified in the partitioning step. Observe that, if v is a point in $H_j \cap V_i$, $SW(v)$ is the value $\sum_{k=1}^{i-1} Count_j(k)$. Given array $Count_j(1:p)$, the value of $SW(v)$ for all points $v \in H_j$ can be computed in $O(p)$ time, by using the p-processor array to compute $\sum_{k=1}^{l-1} Count_j(k)$ for all l, $1 \leq l \leq p$. Thus, the value of $SW(v)$ for all $v \in V$ can be computed in $O(p(n/p)) = O(n)$ time totally.

To compute $W(v)$ for all $v \in H_j$, we use the p-processor array to solve the dominance counting problem on H'_j where H'_j is the union, over all $i \in \{1, 2, \ldots, p\}$, of the projections of all $u \in H_j \cap V_i$ on the $(i-1)$th vertical cut-line (separating V_{i-1} from V_i). This can be done in $O(p)$ time since $|H'_j| = |H_j| = p$.

To sort the points of $P \cup Q$, we sort each H_i in $O(p)$ time, for $i = 1, 2, \ldots, n/p$, using the p-processor array. Therefore, the total combining time of algorithm DOMCOUNT is $O(n)$ time. The overall time complexity of algorithm DOMCOUNT thus is $O(n \log n / \log p)$.

Theorem 3.5 *Given two sets of points in the plane, $P = \{p_1, p_2, \ldots, p_n\}$ and $Q = \{q_1, q_2, \ldots, q_m\}$, algorithm DOMCOUNT computes, for every point $v \in P$, the number*

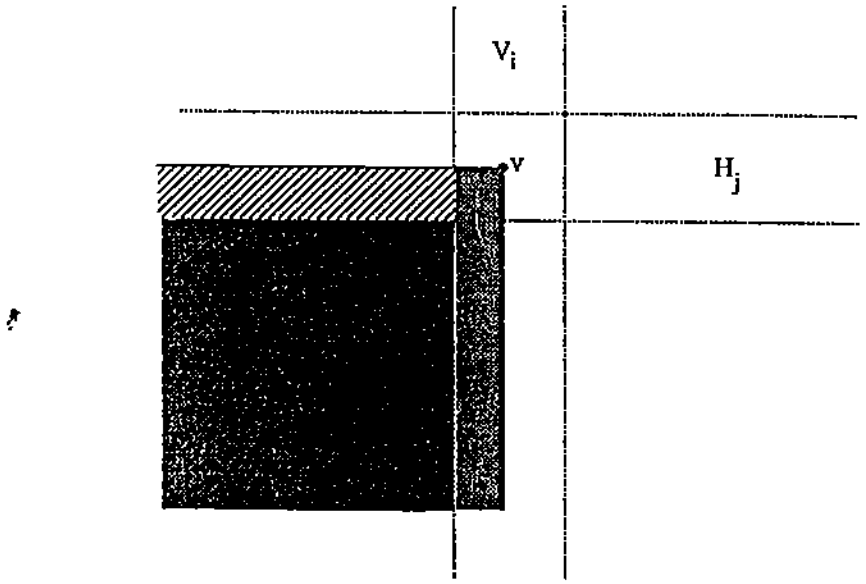

Figure 7: $D(v) = (D(v) \cap V_i) \cup (D(v) \cap (H_j - V_i)) \cup (D(v) \cap (\cup_{k=1}^{i-1} V_k) - H_j)$

of points in Q which are dominated by v, in $O(N \log N / \log p)$ time on a RAM/ARRAY(1), where $N = n + m$.

There are several problems can be reduced to this problem. The multiple range counting and intersection counting of rectilinear segments are two examples [Goo87]. Thus, we have the following corollaries.

Corollary 3.1 *Given a set P of m points in the plane and a set R of n isothetic rectangles, we can compute, for each rectangle, the number of points interior to it, in $O(N \log N / \log p)$ time on a RAM/ARRAY(1), where $N = m + n$.*

Corollary 3.2 *Given a set S of n rectilinear segments in the plane, we can compute, for each line segment, the number of segments intersecting it properly, in $O(n \log n / \log p)$ time on a RAM/ARRAY(1).*

4 Generalization to a RAM/ARRAY(d)

In this section, we show how to design $O(n \log n/(p^{1-1/d} \log p))$ time algorithms on a RAM/ARRAY(d) for all the problems we already solved in $O(n \log n / \log p)$ time on a RAM/ARRAY(1). Instead of giving the detailed algorithms for each problem, we sketch the general approach. We assume a d-dimensional array of p processors can solve a problem

of size p in $O(p^{1/d})$, an assumption that is true for all the problems we considered. We assume also that the input/output of p elements into/from the array can be done in $O(p^{1/d})$ time (this is a standard assumption in literature of mesh-connected array of processors and we shall not tamper with it).

Recall that in all of our $O(n \log n / \log p)$ algorithms on a RAM/ARRAY(1), we partition the problems into p subproblems and recursively solve each problem and then combine the subsolutions in $O(n)$ time. If we want to use the same paradigm to design $O(n \log n/(p^{1-1/d} \log p))$ algorithm on a RAM/ARRAY(d) ($d > 1$), the combining step has to be done in $O(n/p^{1-1/d})$ time. We give a general approach as follows. Instead of partitioning the problem into p subproblems, we partition the problem into $p^{1/(d+1)}$ subproblems and recursively solve them. To show how to combine the subsolutions in $O(n/p^{1-1/d})$ time, we show that, given $p^{1/(d+1)}$ sorted lists, the group of the first p elements in their union can be identified and transferred to the array in $O(p^{1/d})$ time.

Recall that the selection algorithm in [FJ82] selects the pth element from an $m \times p^{1/(1+d)}$ column sorted array in time $O(p^{1/(d+1)} + p^{1/(d+1)} \log(p/p^{1/(d+1)}))$, which is $O(p^{1/(d+1)} \log p)$. This implies that the pth elements in the given $p^{1/(d+1)}$ sorted lists can be selected in $O(p^{1/d})$ time. Given the pth element, we identify and transfer the group of the first p elements to the array in $O(p^{1/d})$ time as follows. We do a binary search in each of the $p^{1/(d+1)}$ sorted lists to identify the group of the first p elements. This takes $O(\log p)$ time for each sorted list. We then can identify the group of the first p elements in $O(p^{1/(d+1)} \log p)$ time. The RAM then issues $p^{1/(d+1)}$ I/Os (one for each sorted list) to send the identified p elements to the array. Thus, the decomposition as in Lemma 2.1 can be done in $O(n/p^{1-1/d})$ time on a RAM/ARRAY(d).

This gives us a general approach to generalize our $O(n \log n / \log p)$ time algorithms on a RAM/ARRAY(1) to a RAM/ARRAY(d) ($d > 1$). And the resulting recurrence for the time complexity $T(n)$ is:

$$\begin{array}{ll} T(n) = p^{1/(d+1)} T(n/p^{1/(1+d)}) + c_1 n/p^{1-1/d}, & \text{if } n > p \\ T(n) = c_2 n^{1/d}, & \text{if } n \leq p, \end{array}$$

where c_1, c_2 are constants. Therefore, $T(n) = O(dn \log n/(p^{1-1/d} \log p))$, which is $O(n \log n/(p^{1-1/d} \log p))$ when d is a constant.

5 Further Remarks

The question of whether the speedup of $p^{1-1/d}\log p$, that the d-dimensional array makes possible for a problem of size p, can be carried over to larger problems is really dealing with the fundamental issue of the *parallel-decomposability* of the problem at hand: given that a problem of size p can be solved on a parallel machine P faster by a factor of (say) $s(p)$ than on a RAM alone, then that problem is $s(p)$-*parallel-decomposable* for P if the problem can be decomposed in such a way that the $s(p)$ speedup also holds when the RAM/P combination is solving a problem too large to fit in P. Mueller's paper [Mue87] was a pioneering one in that respect. This paper is another step in that direction, in that we were able to show that some geometric problems are $s(p)$-parallel-decomposable for a d-dimensional array of p processors where $s(p) = p^{1-1/d}\log p$. This question remains open for many other classical geometric problems, in particular, the general trapezoidal decomposition, Voronoi diagram and three dimensional convex hull problems. All of them can be solved optimally on an n-processor array [JL90, TA90].

It is well known that there are close connections between the work on parallel-decomposability and the work on I/O complexity [AV88, HK81]. In the study of I/O complexity, we are given a sequential computer which has a small main memory and a large secondary storage, and we are interested in solving problems of arbitrarily large size. The input of the problem is initially stored in the secondary storage and the output has to be written in the secondary storage. The limitation that the size of the main memory is small, is similar to the limitation that the size of the attached parallel machine is small. The major concern in the study of I/O complexity is to minimize the amount of I/O between the main memory and the secondary storage. To achieve the best I/O performance, the algorithm is allowed arbitrarily long computation times for scheduling the I/Os (only the amount of I/O matters). On the other hand, the time to decompose the computation into subcomputations and to schedule the subcomputations must be counted in the study of the parallel-decomposability (i.e., it has to be of $O(Seq(n)/s(p))$). In [Tsa90], we have shown that techniques presented in this paper for the study of parallel-decomposability can be used to achieve I/O performance which is known to be optimal for sorting [AV88]. The techniques can also be used to achieve linear speedup for the geometric problems we considered here, on several Hypercube related computers which consist of p processors each containing $O(n/p)$ local memory, provided that $n > p^{1+\epsilon}$ for some constant $\epsilon > 0$ (see [Tsa90] for the

details). The same speedup has been previously achieved for sorting [AH88, CS88].

References

[AFK88] M. J. Atallah, G. N. Frederickson, and S. R. Kosaraju. Sorting with efficient use of special-purpose sorters. *Information Processing Letters*, 27:13–15, 1988.

[AH88] A. Aggarwal and M.-D. Huang. Network complexity of sorting and graph problems and simulating CRCW PRAMS by interconnection networks. In *Lecture Notes in Computer Science, 319: VLSI Algorithms and Architectures, 3rd Aegean Workshop on Computing, AWOC 88*, pages 339–350, New York, 1988. Springer-Verlag.

[AV88] A. Aggarwal and J.S. Vitter. The Input/Output complexity of sorting and related problems. *Communications of the ACM*, 31:1116–1127, September 1988.

[Ben77] J. Bentley. Algorithms for Klee's rectangle problems. Carnegie-Mellon University, CS Dept., unpublished notes, 1977.

[Ben80] J. Bentley. Multidimensional divide-and-conquer. *Communications of the ACM*, 23:214–229, April 1980.

[BG90] R. Beigel and J. Gill. Sorting n objects with a k-sorter. *IEEE Transactions on Computers*, 1990.

[CG88] R. Cole and M.T. Goodrich. Optimal parallel algorithms for polygon and point-set problems. In *Proceedings of the Fourth Annual Symposium on Computational Geometry*, pages 201–210, June 1988.

[Cha84] B. Chazelle. Computational geometry on a systolic chip. *IEEE Transactions on Computers*, C-33(9):774–785, September 1984.

[CS88] R. Cypher and J. L. C. Sanz. Optimal sorting on feasible parallel computers. In *International Conference in Parallel Processing*, 1988.

[FJ82] G. N. Frederickson and D. B. Johnson. The complexity of selection and ranking in $X + Y$ and matrices with sorted columns. *Journal of Computer and System Sciences*, 24:197–208, 1982.

[Goo87] M. T. Goodrich. *Efficient Parallel Techniques for Computational Geometry.* PhD thesis, Purdue University, 1987.

[Gut84] R. H. Guting. Optimal divide-and-conquer to compute measure and contour for a set of iso-rectangles. *Acta Informatica*, 21:271–291, 1984.

[HK81] J.-W. Hong and H. T. Kung. I/O complexity: The red-blue pebble game. In *Proceedings of the 13th Annual ACM Symposium on Theory of Computing*, pages 326–333, 1981.

[JL90] C. S. Jeong and D. T. Lee. Parallel geometric algorithms on a mesh connected computer. *Algorithmica*, 1990.

[KLP75] H. T. Kung, F. Luccio, and F. P. Preparata. On finding the maxima of a set of vectors. *Journal of the ACM*, 22(4):469–476, 1975.

[LCW81] D. T. Lee, H. Chang, and C. K. Wong. An on-chip compare steer bubble sorter. *IEEE Transactions on Computers*, C-30(6):396–405, 1981.

[LP84] D. T. Lee and F. P. Preparata. Computational geometry—a survey. *IEEE Transactions on Computers*, C-33(12):872–1101, 1984.

[MS89] R. Miller and Q. F. Stout. Mesh computer algorithms for computational geometry. *IEEE Transactions on Computers*, January 1989.

[Mue87] H. Mueller. Sorting numbers using limited systolic coprocessors. *Information Processing Letters*, 24:351–354, 1987.

[OY88] M. Overmars and C. K. Yap. New upper bounds in Klee's measure problem. In *Proceedings of the 29th Annual IEEE Symposium on Foundations of Computer Science*, pages 550–556, 1988.

[PS85] F. P. Preparata and M. I. Shamos. *Computational Geometry.* Springer-Verlag, 1985.

[SCL87] Z.-C. Shih, G.-H. Chen, and R. C. T. Lee. Systolic algorithms to examine all pairs of elements. *Communications of the ACM*, 30(2):161–167, February 1987.

[TA90] J.-J. Tsay and M. J. Atallah. Multisearch techniques of hierarchical DAGs on Mesh-Connected Computers, with applications. Technical Report TR-1008, Department of Computer Science, Purdue University, 1990.

[Tsa90] J.-J. Tsay. *Techniques for Solving Geometric Problems on Mesh-Connected Computers.* PhD thesis, Purdue University, 1990.

[LW80] J. van Leeuwen and D. Wood, The measure problem for rectangular ranges in d-space. *J. of Algorithms*, 10:51–56, 1980.

[WW88] D. E. Willard and Y. C. Wee. Quasi-valid range querying and its implementations for nearest neighbor problem. In *Proceedings of the Fourth Annual Symposium on Computational Geometry*, pages 34–43, June 1988.