EFFICIENT PARALLEL SOLUTIONS TO GEOMETRIC PROBLEMS

Mikhail J. Atallah †
Michael T. Goodrich

Department of Computer Science
Purdue University
West Lafayette, IN 47907

ABSTRACT

This paper presents new algorithms for solving geometric problems on a shared memory parallel computer, where concurrent reads are allowed but no two processors can simultaneously attempt to write in the same memory location. The algorithms are new and are quite different from the known sequential algorithms. One of our results is an $O(\log n)$ time, $O(n)$ processor algorithm for the convex hull problem. Another result is an $O(\log n \log\log n)$ time, $O(n)$ processor algorithm for the problem of selecting a closest pair of points among n input points.

Key Words. Parallel algorithms, computational geometry, convex hull problem, closest pair problem.

1. Introduction

Since they involve asking basic questions about sets of points, lines, polygons, etc, geometric problems arise often in many applications [7]. We are interested in finding parallel algorithms solving some of these problems which are efficient both in terms of their running time and in the number of processors used. Efficient sequential algorithms for solving geometric problems often use the divide-and-conquer paradigm: to solve a problem of size n solve two sub-problems of size $n/2$, and then "marry" the results of these two recursive calls. Unfortunately, trying to "parallelize" known sequential algorithms often yields suboptimal parallel solutions (actually this seems to be a rule rather than an exception). Such is the case for the convex hull and the closest pair problems. Indeed, the efficient parallel algorithms we give for solving these problems turn out to be quite different from the known sequential algorithms.

Throughout this paper, the computational model used is the shared memory model in which concurrent reads are allowed, but no two processors should attempt to simultaneously write in the same memory location. We henceforth refer to this model as the CREW PRAM (Concurrent Read Exclusive Write Parallel RAM). Using this model of parallel computation, we are interested in achieving the highest speed-up using only $O(n)$ processors (this restriction on the numbers of processors is crucial, since the problems we consider can trivially be solved in logarithmic time if the number of processors used were of no concern, e.g. $O(n^2)$). In reference [4], Chow has given an $O(\log^2 n)$ time, $O(n)$ processor parallel convex hull algorithm. This paper improves on that result by giving an $O(\log n)$ time, $O(n)$ processor parallel algorithm for that problem. Our algorithm is optimal with respect to both the time and number of processors, since this problem has an $\Omega(n \log n)$ time sequential lower bound [10], and an obvious $\Omega(\log n)$ time lower bound for the parallel machine considered in this paper. Another problem we address is that of finding the closest pair among a set of n input points. We give an algorithm which solves the closest pair problem in $O(\log n \log\log n)$ time using $O(n)$ processors by exploiting a technique similar to the one we used to solve the convex hull problem. Our algorithm for the closest pair is far more complex than that for the convex hull, and therefore we begin by presenting the latter.

To simplify the exposition, we assume that no three points are collinear and that the points have distinct x (resp. y) coordinates (our results can easily be modified for the general case). We also frequently make use of the known result that, on this model of parallel computation, n objects can be sorted in $O(\log n)$ time by an n processor machine [6].

2. Convex Hull

Given n points in the plane, the convex hull problem is that of finding which of these points belong to the perimeter of the smallest convex region containing all n points. This problem has applications in many fields, including computer graphics, computer vision, and statistics [7]. As mentioned earlier, the convex hull problem has an $\Omega(n \log n)$ time sequential lower bound [10], and this bound is achievable [5], [9].

Many authors have addressed the question of finding parallel solutions to this problem. Chazelle [2] shows how to solve the problem on a linear array of processors in $O(n)$ time. Miller and Stout, in reference [8], present an $O(\sqrt{n})$ time solution on an n node mesh-connected computer. Although both of these algorithms are suited for the computational models for which they were designed, implementing them on a CREW PRAM would lead to sub-optimal algorithms. The only known previous parallel algorithm solving this problem on a CREW PRAM is due to Chow [4], and runs in $O(\log^2 n)$ time using $O(n)$ processors. In this section we present a new parallel algorithm which solves the convex hull problem in $O(\log n)$ time on a CREW PRAM with $O(n)$ processors. As mentioned earlier, our algorithm is optimal (to within a constant).

We first present some definitions and observations. Let R be a set of points in the plane. We denote a clock-wise listing of the points which belong to the convex hull of R by $CH(R)$. Let u and v be the points of R with the smallest and largest x-coordinate, respectively. Clearly, u and v are both in $CH(R)$. They divide $CH(R)$ into two sets: an upper hull, consisting of points from u to v, inclusive, in the clock-wise

† This research was supported by the Office of Naval Research under contract N00014-84-K-0502.

listing of $CH(R)$, and a lower hull, consisting of points from v to u, inclusive. We denote a clock-wise listing of the points in the upper hull of R by $UH(R)$, and a similar listing of the points in the lower hull by $LH(R)$. Given a set S of n points in the plane the following algorithm will compute $CH(S)$.

Algorithm CH:

Input: A set S of n points in the plane.

Output: The list $CH(S)$. That is, the points of the convex hull of S listed in clock-wise order.

Method: The main idea of our algorithm is to divide the problem into $\sqrt{n}$ subproblems of size $\sqrt{n}$ each, solve the subproblems recursively in parallel, and combine the solutions to the subproblems quickly (that is, in $O(\log n)$ time) and with a linear number of processors.

Step 1. Sort the n points by x-coordinate, and partition S into sets $R_1, R_2, \ldots, R_{\sqrt{n}}$, each of size $\sqrt{n}$, divided by vertical cut-lines, such that R_i is left of R_j if $i < j$ (see Figure 2.1).

Step 2. Recursively solve the convex hull problem for each R_i, $i \in \{1, 2, \ldots, \sqrt{n}\}$, in parallel. After this parallel recursive call returns we will have $CH(R_i)$ for each R_i.

Step 3. Find the convex hull of S by computing the convex hull of the union of the $\sqrt{n}$ convex polygons $CH(R_1), \ldots, CH(R_{\sqrt{n}})$. This is done using algorithm COMBINE which will be described later in this section.

End of algorithm CH.

Theorem: Algorithm CH finds the convex hull of a set of n points in the plane in $O(\log n)$ time on a CREW PRAM with $O(n)$ processors.

Proof: We give this proof assuming that algorithm COMBINE (used in Step 3) is correct and takes $O(\log n)$ time and $O(n)$ processors. (This will be justified once we describe algorithm COMBINE later in this section.) That Step 1 can be done in $O(\log n)$ time and $O(n)$ processors follows from the results of reference [6]. Thus the running time, $T(n)$, of the algorithm can be expressed in the recurrence relation $T(n) = T(\sqrt{n}) + b \log n$, which is $O(\log n)$. The number of processors needed, $P(n)$, satisfies the recurrence $P(n) = \max\{\sqrt{n}P(\sqrt{n}), cn\}$, which is $O(n)$. This completes the proof, subject to the already stated assumption about Step 3 and algorithm COMBINE (yet to be described). ∎

The rest of this section deals with the problem of implementing Step 3 of algorithm CH in time $O(\log n)$ and with $O(n)$ processors. This is done by using algorithm COMBINE, described below. For convenience, we choose to describe algorithm COMBINE for the problem of computing the upper hull, since that of computing the lower hull is entirely symmetrical. In the algorithm description, when we talk about the "upper" common tangent between $CH(R_i)$ and $CH(R_j)$, we mean the common tangent such that both $CH(R_i)$ and $CH(R_j)$ are below it. Also, when we say that a point p is "to the left" of another point q, what we mean is that the x-coordinate of p is less than that of q.

Algorithm COMBINE:

Input: The collection of convex polygons $\{CH(R_1), CH(R_2), \ldots, CH(R_{\sqrt{n}})\}$. Recall that these input polygons are separated by vertical lines, and that none of them has more than $\sqrt{n}$ vertices. Also recall that $CH(R_i)$ is to the left of $CH(R_j)$ if $i < j$.

Output: The upper convex hull $UH(S)$ of the vertices of the union of the $CH(R_i)$'s.

Method: The main idea is to find, in parallel for each $CH(R_i)$, which of its vertices are on $UH(S)$. This is done by assigning $\sqrt{n}$ processors to each $CH(R_i)$ and having each of these processors compute the upper common tangent between $CH(R_i)$ and one of the other input polygons. The details follow.

Step 1. In parallel for each $i \in \{1, 2, \ldots, \sqrt{n}\}$ use $\sqrt{n}$ processors to find those points of $CH(R_i)$ which belong to $CH(S)$ by doing the following:

Step 1.1 Find the $\sqrt{n} - 1$ upper common tangents between $CH(R_i)$ and the remaining $\sqrt{n} - 1$ other input polygons. Let T_{ij} denote the upper common tangent between $CH(R_i)$ and $CH(R_j)$, where T_{ij} is represented by its point of contact with $CH(R_i)$ and its point of contact with $CH(R_j)$. A tangent T_{ij} is easily computed in $O(\log n)$ time by one processor, using the techniques described in reference [3]. Therefore all of $T_{i1}, \ldots, T_{i,\sqrt{n}}$ can be computed in $O(\log n)$ time by the $\sqrt{n}$ processors assigned to $CH(R_i)$.

Step 1.2. Let V_i be the tangent with smallest slope in $\{T_{i1}, \ldots, T_{i,i-1}\}$ (i.e., V_i is the smallest-slope tangent which "comes from the left" of $CH(R_i)$), and let W_i be the tangent with largest slope in $\{T_{i,i+1}, \ldots, T_{i,\sqrt{n}}\}$ (i.e., W_i is the largest-slope tangent which "comes from the right" of $CH(R_i)$). Let v_i be the point of contact of V_i with $CH(R_i)$, and let w_i be the point of contact of W_i with $CH(R_i)$. Both V_i and W_i can be found in $O(\log n)$ time by the $\sqrt{n}$ processors assigned to $CH(R_i)$.

Step 1.3. Since neither V_i nor W_i can be vertical, they intersect and form an angle (with interior pointing upward). If this angle is less than 180° (as in Figure 2.2), then none of the points of $UH(R_i)$ belong to $UH(S)$. Otherwise, (as in Figure 2.3) all the points from v_i to w_i, inclusive, belong to $UH(S)$.

Step 2. Step 1 has computed, for every $i \in \{1, \ldots, \sqrt{n}\}$, all the points of $CH(R_i)$ which belong to $UH(S)$ (possibly none). This step compresses each of these lists into one list to get $UH(S)$. This can be done in $O(\log n)$ time and $O(n)$ processors (e.g., by using parallel sorting).

End of algorithm COMBINE.

That COMBINE runs in time $O(\log n)$ and $O(n)$ processors should be clear from the comments made in the algorithm description. The correctness of COMBINE depends on the correctness of Step 1.3. The correctness of Step 1.3 for the case depicted in Figure 2.2. follows from the fact that,

in that case, the straight-line segment joining the other endpoints of V_i and W_i (shown dashed in Figure 2.2) is entirely above $CH(R_i)$; hence, no vertex of $CH(R_i)$ can belong to $UH(S)$. The correctness of Step 1.3 for the case depicted in Figure 2.3 follows from the fact that all the points of the other $CH(R_j)$'s are below V_i and W_i. This proves the correctness of algorithm COMBINE.

The next section deals with the closest-pair problem.

3. Closest Pair

Given n points in the plane, the closest pair problem is that of choosing two of them that are closest (i.e., the distance between them is smallest). This problem has applications in answering basic proximity questions of sets of objects, such as monitoring airplanes in air-traffic control. We are not aware of any previous work done in finding parallel solutions to this problem. A trivial $O(\log n)$ time parallel algorithm exists, but it requires a quadratic number of processors. Here we are investigating what speed can be achieved with only $O(n)$ processors. Parallelizing what seems to be the most promising sequential algorithm, by Bentley and Shamos [1], on $O(n)$ processors only leads to an $O(\log^2 n)$ time algorithm. Applying a technique similar to the one we used in the convex hull problem, we show how to solve the problem in $O(\log n \log\log n)$ time using $O(n)$ processors on a CREW PRAM.

As in our solution to the convex hull problem, we will be dividing the input set of points into $\sqrt{n}$ subsets divided by vertical cut-lines. Let $R_1, \ldots, R_{\sqrt{n}}$ be these subsets in left-to-right order, i.e. R_i is left of R_j if $i < j$. We define the *region-width* of a point set R_i to be the distance between the cut-lines separating R_i from R_{i-1} and R_{i+1}, respectively. Note: the region-width of R_1 and $R_{\sqrt{n}}$ is defined to be ∞. We present the closest pair algorithm CP below.

Algorithm CP:

Input: A set S of n points in the plane.

Output: A closest pair of points in S.

Method: Before giving the details, we present an overview of the various stages of the algorithm. First, we partition S into $\sqrt{n}$ sets, of size $\sqrt{n}$ each, using vertical cut-lines, and recursively solve the closest pair problem for each. This gives us a closest pair of points not separated by a cut-line. For the combining step to run quickly (i.e., in $O(\log n)$ time) there should not be more than a constant number of cut-lines which are "close" to one another. Since this may not presently be the case, we do not perform a combining step at this point. Instead, we re-partition S using the COALESCE algorithm, presented later, which results in a better distribution of the remaining vertical cut-lines. The COALESCE method works by removing cut-lines which divide point sets with small region-widths, thereby coalescing the two sets into one. Even after coalescing, we still do not combine the subproblems, because in removing a cut-line we coalesce previously solved subproblems into conglomerates which must now be re-solved. Consequently, for each conglomerate point set, we use the $\sqrt{n}$ divide-and-conquer technique again, dividing the conglomerate horizontally this time, and solving each of the resulting horizontally divided sets recursively. We divide the conglomerate point sets horizontally, because, as we will see, it guarantees that cut-lines will be far enough from each other so as to allow for a fast combining step. Since we have already forced the vertical cut-lines to be distributed nicely, we are now ready to combine the solutions to the subproblems: first combining the solutions to the horizontally divided sets, and then combining the solutions to the vertically divided sets. This combining step is done using the DIST algorithm, presented later. A detailed description of the algorithm follows. Let I denote the index set $\{1, 2, \ldots, \sqrt{n}\}$.

Step 1. Partition S into point sets $R_1, R_2, \ldots, R_{\sqrt{n}}$, each of size $\sqrt{n}$, separated by vertical cut-lines, such that R_i is left of R_j if $i < j$ (see Figure 2.1).

Step 2. Recursively solve the closest pair problem for each R_i, $i \in I$, in parallel. After the parallel recursive call returns we will have a closest pair of points, (p_i, q_i), for each R_i. Let δ_i be the distance between p_i and q_i.

Step 3. Find the minimum δ_i value, and call it δ. Let (p, q) be the pair of points associated with the selected (minimum) δ_i value.

Step 4. Repartition S into $\{R'_1, R'_2, \ldots, R'_l\}$, $l \leq \sqrt{n}$ so that there is never more than 2 vertical cut-lines which are within δ of each other. This is done by using the COALESCE algorithm, presented later on. Let I' denote $\{1, 2, \ldots, l\}$.

Comment: The new partition is created by coalescing some adjacent point sets by removing cut-lines which separate sets with region-width less than δ. Algorithm COALESCE will give the details of how this is done.

Step 5. In parallel, for each $i \in I'$ find a closest pair (p'_i, q'_i) in R'_i by doing the following:

Step 5.1. If R'_i is one of the original point sets, say R_k (i.e., R'_i was not created by coalescing any of the original point sets), then set δ'_i to δ_k, set (p'_i, q'_i) to (p_k, q_k), and go to Step 6.

Step 5.2. Sort the points in R'_i by y-coordinate and partition R'_i into point sets $r_1, r_2, \ldots, r_{\sqrt{n_i}/2}$, separated by horizontal cut-lines, each of size $2\sqrt{n_i}$ (where $n_i = |R'_i|$), and such that r_j is below r_k if $j < k$ (see Figure 3.1). Let J_i denote $\{1, 2, \ldots, \sqrt{n_i}/2\}$.

Step 5.3. Recursively solve the closest pair problem for each r_j, $j \in J_i$, in parallel. After the parallel recursive call returns we will have a closest pair of points (u_j, v_j), for each r_j. Let ε_j be the distance between u_j and v_j.

Step 5.4. Find the minimum ε_j value, and call it ε. Let (u, v) be the pair of points associated with the selected ε_j value.

Comment: (u, v) is a closest pair in R'_i not separated by a horizontal cut-line.

Step 5.5. Combine the solutions to the r_j's to find a clos-

est pair (p_i', q_i') in R_i'. This is done by using the DIST algorithm presented later in this section. Let δ_i' be the distance between p_i' and q_i'.

Step 6. Find the minimum δ_i' value, and call it δ'. Let (p', q') be the pair associated with the selected δ_i' value.

Comment: (p', q') is a closest pair in S not separated by a vertical cut-line.

Step 7. Combine the solutions to the R_i' sets to find a closest pair of points in S. This too is done by using the DIST algorithm.

End of algorithm CP.

Theorem: Algorithm CP finds a closest pair of n points in the plane in $O(\log n \log\log n)$ time on a CREW PRAM with $O(n)$ processors.

Proof: Suppose, for the time being, that the algorithms COALESCE and DIST work correctly (in lemmas 3.1 and 3.2 we will prove this). Then the correctness of algorithm CP follows from a straight-forward induction on n.

We now turn to the time complexity of CP. It easy to see that steps 1, 3, and 6 run in $O(\log n)$ time and $O(n)$ processors each, since they only involve sorting or finding the minimum of $O(n)$ values. Let $T_C(n)$ be the time to re-partition the $\sqrt{n}$ point sets using the COALESCE algorithm (Step 4). Let $T_D(n)$ be the time to perform the combining algorithm DIST (Step 7). Finally, let $S(n_i)$ be the time to perform Step 5 for R_i', where $n_i = |R_i'|$. We can then characterize the running time, $T(n)$, of the algorithm in the recurrence relation

$$T(n) = T(\sqrt{n}) + T_C(n) + \max_{i \in I'}\{S(n_i)\} + T_D(n) + b_1 \log n.$$

We will show in lemma 3.1 that $T_C(n)$ is $O(\log n)$. We will show, in lemma 3.2, that $T_D(n)$ is $O(\log n)$, since the COALESCE algorithm forces the maximum number of vertical cut-lines which are within δ of one another to be at most 2. Thus, the new recurrence becomes $T(n) = T(\sqrt{n}) + \max_{i \in I'}\{S(n_i)\} + b_2 \log n$.

To characterize $S(n_i)$, consider the running of Step 5 for some $i \in I'$. If R_i' was one of the original point sets (the condition of step 5.1) then $S(n_i)$ is $O(1)$. So we have yet to consider the case when R_i' was created by coalescing 2 or more of the original point sets. Notice that steps 5.2 and 5.4 run in $O(\log n_i)$ time, since they only involve sorting and minimizing. Step 5.5 is a call to algorithm DIST to combine the solutions to the $\sqrt{n_i}/2$ subproblems. Thus, $S(n_i) = T(2\sqrt{n_i}) + T_D(n_i) + b_3 \log n_i$. To prove that $T_D(n_i)$ is $O(\log n_i)$ we need to show that the maximum number of horizontal cut-lines which are within ϵ of one another is constant for the point set R_i'. Since $\epsilon \leq \delta$, it is enough to show this for δ.

Claim: There are no more than 3 horizontal cut-lines which are within δ of one another in any point set R_i' which was created from horizontal point sets, $i \in I'$.

Proof: Suppose there are 4 horizontal cut-lines within δ of each other in some R_i'. Let $Q = r_j \cup r_{j+1} \cup r_{j+2}$, $j \in J_i$, be the set of points which are bounded by these lines. Let d be the number of original point sets which were coalesced to create R_i'. Then $n_i = |R_i'| = d\sqrt{n}$, and $|r_j| = 2\sqrt{n_i} = 2d^{1/2}n^{1/4}$, for all $j \in J_i$. Since $\sqrt{n} \geq d$, $|r_j| \geq 2d$, for all $j \in J_i$. Each of the d original point sets must have had region-width less than δ to have been coalesced. (This fact will become obvious in our discussion of the algorithm COALESCE.) Thus, since the value δ was found by solving the closest pair problem for each original point set, there can be at most 4 points in Q for any of the d original point sets which were coalesced to form R_i' (see Figure 3.2). Thus, $|Q| \leq 4d$. But since Q contains 3 r_j's, $|Q| \geq 6d$. This is obviously a contradiction. ∎ (of Claim.)

Thus, Step 5.5 runs in $O(\log n_i)$ time, and $S(n_i) = T(2\sqrt{n_i}) + b_4 \log n_i$. Therefore, since $n_i \leq n$, $T(n) \leq T(\sqrt{n}) + T(2\sqrt{n}) + b_5 \log n$, which implies that $T(n)$ is $O(\log n \log\log n)$.

To prove that the number of processors needed is $O(n)$ we will consider each part of the algorithm separately. As already mentioned, the number of processors for steps 1,3 and 6 is $O(n)$. Let $P_C(n)$ be the number of processors needed for the COALESCE algorithm to re-partition the $\sqrt{n}$ point sets (Step 4), and let $P_D(n)$ be the number of processors needed for the combining algorithm DIST (Step 7). Finally, let $Q(n_i)$ be the number of processors needed to solve Step 5 for a point set R_i', $i \in I'$. Then the total number of processors needed satisfies the recurrence relation

$$P(n) = \max\{c_1 n,\ \sqrt{n}P(\sqrt{n}),\ P_C(n),\ \sum_{i \in I'} Q(n_i),\ P_D(n)\}.$$

In lemmas 3.1 and 3.2 we will show that both $P_C(n)$ and $P_D(n)$ are $O(n)$. Concentrating on $Q(n_i)$, note that the sorting and min-finding steps in 5.1, 5.2, and 5.4 need only $O(n_i)$ processors, as does the call to algorithm DIST in Step 5.5. Thus, $Q(n_i) = \max\{c_2 n_i,\ \frac{1}{2}\sqrt{n_i}P(2\sqrt{n_i})\}$. Using the fact that $\sum_{i \in I'} n_i = n$, we get, by induction, that $P(n)$ is $O(n)$. ∎

We now need to show that the algorithms DIST and COALESCE work within the claimed time and processor bounds. We start with DIST. Recall that this is the combining algorithm used in algorithm CP. Simply stated, the problem it solves is the following: given a collection of point sets separated by parallel cut-lines and a closest pair of points not separated by a cut-line, find a closest pair of points in the union of all the point sets. For convenience, we describe the algorithm for the case when the cut-lines are vertical, since the case when they are horizontal is entirely symmetrical.

Algorithm DIST:

Input: A collection of point sets $\{R_1, R_2, \ldots, R_k\}$, separated by vertical cut-lines $L_1, L_2, \ldots, L_{k-1}$, such that R_i is left of R_j if $i < j$, and L_i is the line that separates R_i and R_{i+1}. We are also given (u, v), a closest pair of points not separated by a cut-line in H, where $H = R_1 \cup R_2 \cup \cdots \cup R_k$. Moreover, there is a constant c such that there are no more than c cut-lines which are within δ of each other, where δ is the distance between u and v.

Output: A closest pair of points in H.

Method: Before giving the details we present a brief overview

of the algorithm. The algorithm is based on the fact that the only way a closest pair of points in H can be closer than (u, v) is if there is a pair of points which are separated by at least one cut-line and closer than δ to one another. In parallel for each point $p \in H$, we find all points which could possibly be within δ of p and are separated from p by a cut-line. Call this set of points $N(p)$. As we will show, there will only be $O(1)$ such points in $N(p)$ for any p. So, in $O(1)$ time we can then find the point closest to p in $N(p)$. Call the distance between p and this point $d(p)$. Then, by taking the minimum of all the $d(p)$'s and comparing it to δ, we can find a closest pair of points in H.

To make the description clearer we make the following definitions. We define the *left δ-window* of R_i, denoted X_i, to be the set of points in R_i which are within δ of L_{i-1} (i.e., R_i's left cut-line). We define the *right δ-window* of R_i, denoted Y_i, symmetrically, to be the set of points in R_i which are within δ of L_i (i.e., R_i's right cut-line). Note that R_1 has no left window and that R_k has no right window. Let I denote $\{1, 2, \ldots, k\}$, and let $n_i = |R_i|$.

Step 1. In parallel for each $i \in I$ sort the points of X_i, the left δ-window of R_i, by y-coordinate. Also do this for the points in Y_i, the right δ-window of each R_i.

Step 2. In parallel for every point $p \in H$ find $d_Y(p)$, the distance between p and, if it exists, a closest point in the set of all points which could possibly be within δ of p and are separated from p by a cut-line to the right of p, by doing the following:

Step 2.1. Let R_i be the point set containing p. If p is not in Y_i, the right δ-window of R_i, then set $d(p)$ to ∞ and go to Step 3, because there can be no point right of p which is closer than δ to p and separated from p by a cut-line.

Step 2.2. For each left δ-window X_j which is right of R_i (i.e., $i < j \leq k$) such that L_{j-1} (R_j's left cut-line) is within δ of L_i (R_i's right cut-line), perform a binary search on X_j, selecting all points with y-coordinate within δ of p's y-coordinate (see Figure 3.3). Call this set of points $N_Y(p)$. Note that there can be at most c such X_j's. Also, the intersection of $N_Y(p)$ and any X_j contains at most 6 points (see Figure 3.4). Therefore, the size of $N_Y(p)$ is $O(1)$.

Step 2.3. If $N_Y(p) \neq \emptyset$, then find a point $q_Y(p)$ in $N_Y(p)$ which is closest to p, and let $d_Y(p)$ be the distance between p and $q_Y(p)$. If $N_Y(p) = \emptyset$, then set $d_Y(p)$ to ∞.

Step 3. Perform the procedure analogous to Step 2 to find $d_X(p)$ and, if $d_X(p) < \infty$, $q_X(p)$ for each $p \in H$. This can be done by substituting each instance of the word "left" by "right," each "X" by "Y," and vice-versa, in Step 2.

Step 4. Let $d(p)$ be the smaller of $d_Y(p)$ and $d_X(p)$, and, if $d(p) < \infty$, let $q(p)$ be the closer to p of $q_X(p)$ and $q_Y(p)$. Find the minimum $d(p)$ value (for all $p \in H$), and call this value δ'. If $\delta' < \infty$, then let $(p, q(p))$ be the pair of points associated with this value (i.e., the distance between p and $q(p)$ is δ'. A closest pair of points in H is $(p, q(p))$ if $\delta' < \delta$, otherwise it is (u, v).

End of algorithm DIST.

Lemma 3.1: Suppose we are given a collection of point sets $\{R_1, R_2, \ldots, R_k\}$ separated by $k-1$ vertical cut-lines, and a closest pair of points (u, v) not separated by a cut-line. Moreover, suppose there is a constant c such that there are no more than c cut-lines which are within δ of one another. Then the algorithm DIST will find a closest pair of points in H in $O(\log n)$ time on a CREW PRAM with $O(n)$ processors, where $H = R_1 \cup R_2 \cup \cdots \cup R_k$, and $n = |H|$.

Proof: We first prove that algorithm DIST is correct. Let δ be the distance between u and v. Since we are given a closest pair of points not separated by a cut-line as input (namely, (u, v)), either (u, v) is a closest pair in H or there is a pair of points separated by a cut-line and closer than (u, v). Let $S(\delta, p)$ denote the 2δ by 2δ square centered at p. For all the L_k distance metrics the δ-neighborhood of a point p is a subset of $S(\delta, p)$. Thus, if we are searching for points in H which could possibly be closer than δ to a point p, as in Steps 2 and 3, we can restrict our attention to points which are in $S(\delta, p) \cap H$. Notice that from the constructions of steps 2 and 3, if a point is contained in $S(\delta, p) \cap H$, it is also contained in either $N_Y(p)$, $N_X(p)$, or R_i, where $p \in R_i$. Thus, if there are any points closer than δ to p and separated from p by a cut-line, they will be discovered in steps 2 and 3. There can be no points closer than δ to p in R_i. Therefore, in Step 4, when we find a closest pair from among all pairs found in steps 2 and 3, we get a closest pair separated by a cut-line. Comparing this pair to (u, v) correctly gives a closest pair of points in H.

We have yet to show that the algorithm runs within the specified time and processor bounds. Step 1 can be done in $O(\log n)$ time on $O(n)$ processors, since it involves sorting n_i elements for all $i \in I$, in parallel, where $n = \sum_{i \in I} n_i$. Similarly, Step 4 runs in $O(\log n)$ time on $O(n)$ processors, since it involves finding the minimum of as many as n elements. Step 2 is performed in parallel for all $p \in H$; hence, its running time is the maximum time spent for any $p \in H$. The maximum number of times we can perform the binary search in Step 2.2 for any p is clearly c, the maximum number of cut-lines which are within δ of one another. Since each binary search runs in $O(\log n)$ time, and c is a constant, the total running time for Step 2.2 is $O(\log n)$. In Step 2.3, there will never be more than a constant number of points (6) selected from any δ-window, because δ is the distance between a closest pair of points not separated by a cut-line, and there are no cut-lines in δ-windows (see Figure 3.4). Also, there are at most c point sets which intersect $N_Y(p)$ for any p. Thus, there are at most $O(1)$ points in $N_Y(p)$ for any p; hence, Step 2.3 runs in $O(1)$ time. Summing the times for the substeps of Step 2 gives us that Step 2 runs in $O(\log n)$ time. Similarly for Step 3. Thus the algorithm DIST runs in $O(\log n)$ time.

Turning to the processor bounds, notice that we dedi-

cate one processor to at worst every point in H in Step 2; hence, will need $O(n)$ processors for this step in the worst case. Similarly for Step 3. We have already observed that Steps 1 and 4 require $O(n)$ processors. Thus, algorithm DIST requires $O(n)$ processors. ∎

Since the algorithm DIST requires that there is a constant c such that there are no more than c cut-lines which are within δ of one another, we need to have some way of taking a general collection of point sets and turning it into one for which this is true. Given any collection of point sets divided by vertical lines, as in the beginning of the algorithm CP, the algorithm COALESCE will produce a collection such that there are no more than 2 cut-lines within δ of one another.

Algorithm COALESCE:

Input: The collection of point sets $\Re = \{R_1, R_2, \ldots, R_{\sqrt{n}}\}$ separated by vertical cut-lines. Recall that R_i is left of R_j if $i < j$, and that when COALESCE is called CP has already computed δ.

Output: A collection $\Re' = \{R'_1, R'_2, \ldots, R'_l\}, l \leq \sqrt{n}$, in which there is never more than 2 cut-lines which are within δ of one another..

Method: We divide the problem into two subproblems, solve the the re-partioning problem recursively, and combine the two subproblems in constant time. For simplicity of expression let $k = \sqrt{n}$.

Step 1. Let $\Re = \{R_1, R_2, \ldots, R_k\}$. Divide the collection $\Re$ into two contiguous collections $\Re_1 = \{R_1, \ldots, R_{k/2}\}$ and $\Re_2 = \{R_{k/2+1}, \ldots, R_k\}$ ($\Re_1$ being to the left of $\Re_2$).

Comment: $R_{k/2}$ has the same region-width in $\Re_1$ as it does in $\Re$. Similarly, $R_{k/2+1}$ has the same region-width in $\Re_2$ as it does in $\Re$.

Step 2. Recursively re-partition $\Re_1$ and $\Re_2$ in parallel. After the parallel recursive call returns there will be no two adjacent point sets in $\Re_1$, or in $\Re_2$, which both have region-width less than δ.

Step 3. If the region-width of the rightmost point set in $\Re_1$ and the region-width of the leftmost point set in $\Re_2$ are both less than δ, then coalesce them into one point set by removing the cut-line between them. Otherwise, do nothing.

End of algorithm COALESCE.

Lemma 3.2: Given the collection of point sets $\Re = \{R_1, R_2, \ldots, R_{\sqrt{n}}\}$ and the real number $\delta > 0$, the algorithm COALESCE constructs a collection $\Re' = \{R'_1, R'_2, \ldots, R'_l\}$, $l \leq \sqrt{n}$, such that there are at most 2 cut-lines which are within δ of one another, and runs in $O(\log n)$ time on a CREW PRAM with $O(n)$ processors.

Proof: The correctness of this algorithm is easily justified by a simple inductive argument based on n. To see than this re-partitioning algorithm runs in the specified time bound notice that, since Step 1 and 3 run in $O(1)$ time, the time required for DIST, $T(n)$, satisfies the recurrence $T(n) = T(n/2) + b$, which implies that $T(n)$ is $O(\log n)$. For the processor bound, $P(n)$, notice that Steps 1 and 3 can easily be done with $O(n)$ processors, while Step 2 needs $2P(n/2)$ processors. Thus, the number of processors required satisfies the recurrence $P(n) = \max\{2P(n/2), cn\}$, which implies that $P(n)$ is $O(n)$. ∎

The main result of this section is an algorithm (CP) to solve the the closest pair problem on n points in $O(\log n \log\log n)$ time on a CREW PRAM with $O(n)$ processors. Our algorithm will work with any of the L_k distance metrics.

4. Conclusion

We gave efficient parallel algorithms for solving two geometric problems: We have shown how to solve the convex hull problem in $O(\log n)$ time and the closest pair problem in $O(\log n \log\log n)$ time on a CREW PRAM with $O(n)$ processors. We suspect that the technique we used will be helpful in tackling other geometric problems as well.

Acknowledgment

We would like to thank Greg Shannon for his comments and careful reading of an earlier draft of this work.

References

[1] J. L. Bentley and M. I. Shamos, "Divide-And-Conquer in Multidimensional Space," *Symp. on Theory of Comp.*, 1976, pp. 220–230.

[2] B. Chazelle, "Computational Geometry on a Systolic Chip," *IEEE Trans. on Computers*, Vol. C-33, No. 9, September, 1984, pp. 774–785.

[3] B. Chazelle and D. Dobkin, "Detection is Easier than Computation," *Symp. on Theory of Comp.*, 1980, pp. 146–153.

[4] A. Chow, "Parallel Algorithms for Geometric Problems," PhD Thesis, University of Illinois, Urbana-Champain, December, 1981.

[5] R. L. Graham, "An Efficient Algorithm for Determining the Convex Hull of a Finite Planar Set," *Inform. Processing Letters*, Vol. 1, pp. 132–133, 1972.

[6] T. Leighton, "Tight Bounds on the Complexity of Parallel Sorting," *IEEE Trans. on Computers*, vol. C-34, no. 4, April 1985, Pp. 344–354.

[7] D. T. Lee and F .P. Preparata, "Computational Geometry—A Survey," *IEEE Trans. on Computers*, Vol. C-33, No. 12, December, 1984, pp. 1072–1101.

[8] R. Miller and Q .F. Stout, "Computational Geometry on a Mesh-Connected Computer," *Proc. of 1984 Int. Conf. on Parallel Processing*, pp. 66–73.

[9] F. P. Preparata, "An Optimal Real-Time Algorithm for Planar Convex Hulls," *Comm. ACM*, Vol. 22, No. 7, July 1979, pp. 402–405.

[10] A. C. Yao, "A Lower Bound to Finding Convex Hulls," *J. ACM*, Vol. 28, 1981, pp. 780–787.

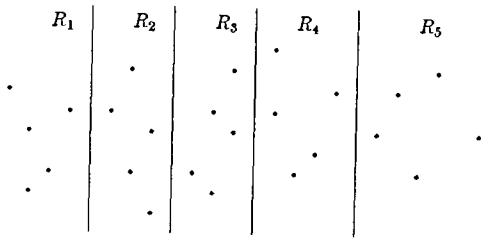

Figure 2.1: A partitioning of S into $\sqrt{n}$ sets, an example with $n = 25$.

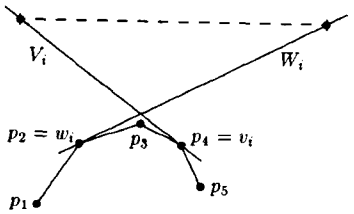

Figure 2.2: An illustration of the case when none of $UH(R_i)$ are in $UH(S)$, because V_i and W_i form an angle which is less than 180°.

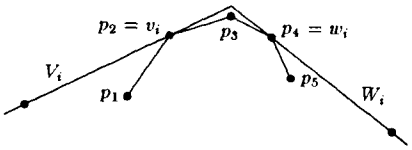

Figure 2.3: The points p_2, p_3, and p_4 are in $UH(S)$, because V_i and W_i form an angle which is at least 180°.

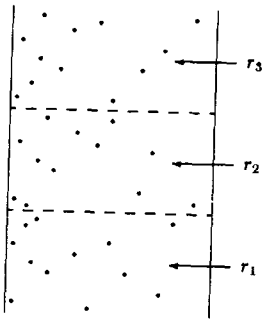

Figure 3.1: A horizontal partitioning of a point set R'_i with $n_i = 36$.

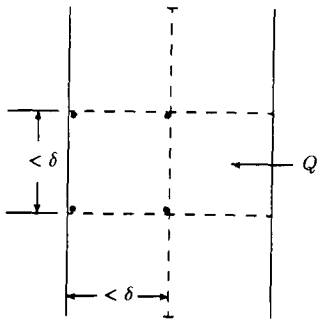

Figure 3.2: The upper bound on number of points in Q which were all in one of the origingal point sets is 4.

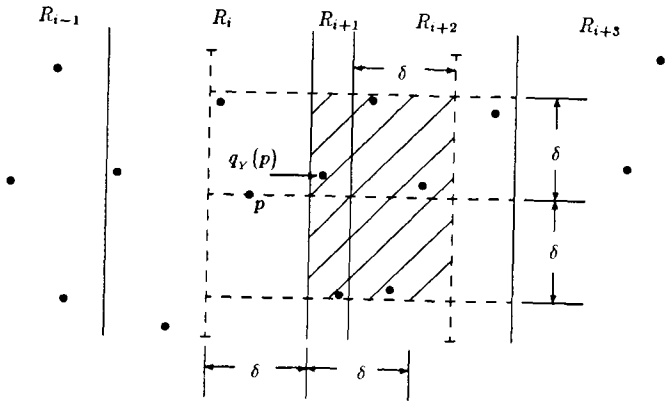

Figure 3.3: The binary search is for finding the set of points $N_Y(p)$ in the shaded region. In this case, the processor for p need only examine the left δ-windows of R_{i+1} and R_{i+2}. The point labeled $q_Y(p)$ is the closest point to p in $N_Y(p)$.

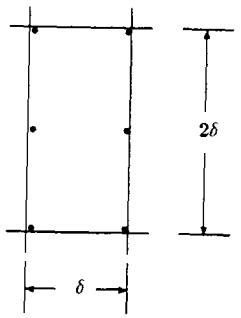

Figure 3.4: The upper bound on number of points selected from one δ-window is 6.