アルゴリズム 58−7
(1997. 9. 19)

近接点を見つける最適な並列アルゴリズムとその応用

林達也 中野浩嗣
名古屋工業大学電気情報工学科

ステファンオラリウ
オールドドミニオン大学計算機科学学科

x 座標によりソートされた平面上の点の集合 P を考える。P に含まれる点 p が次の条件を満たすとき、p は近接点であると言う。x 軸上の点 q が存在し、P の他のどの頂点より p は q に近い。近接点問題とは集合 P が与えられたとときに、P の全ての近接点を求める問題である。本論文では、近接点問題を解く最適な逐次アルゴリズムと並列アルゴリズムを示す。逐次アルゴリズムの計算時間は $O(n)$ である。並列アルゴリズムは、Common-CRCW-PRAM で $O(\log\log n)$ 時間、$\frac{n}{\log\log n}$ プロセッサであり、EREW-PRAM で $O(\log n)$ 時間、$\frac{n}{\log n}$ プロセッサである。これらのアルゴリズムは仕事量・時間最適である。また、近接点問題のパターン解析、計算機幾何、画像処理などに対する応用を示す。

Optimal Parallel Algorithms for Finding Proximate Points, with Applications

Tatsuya Hayashi and Koji Nakano
Department of Electrical and Computer Engineering
Nagoya Institute of Technology
Showa-ku, Nagoya 466, Japan

Stephan Olariu
Department of Computer Science
Old Dominion University
Norfolk, VA 23529, USA

Consider a set P of points in the plane sorted by x-coordinate. A point p in P is said to be a *proximate point* if there exists a point q on the x-axis such that p is the closest point to q over all points in P. The *proximate points problem* is to determine all proximate points in P. We propose optimal sequential and parallel algorithms for the proximate points problem. Our sequential algorithm runs in $O(n)$ time. Our parallel algorithms run in $O(\log\log n)$ time using $\frac{n}{\log\log n}$ Common-CRCW processors, and in $O(\log n)$ time using $\frac{n}{\log n}$ EREW processors. We show that both parallel algorithms are work-time optimal; the EREW algorithm is also time-optimal. As it turns out, the proximate points problem finds interesting and highly nontrivial applications to pattern analysis, digital geometry, and image processing.

1 Introduction

Let P be a set of points in the plane sorted by x-coordinate. A point p in P is termed a *proximate point* if there exists a point q on the x-axis such that p is the closest point to q over all the points in P. The *proximate points problem* is to determine all proximate points in P. We propose optimal sequential and parallel algorithms for the proximate points problem. The sequential algorithm runs in $O(n)$ time, being therefore optimal. The parallel algorithms run in $O(\log\log n)$ time using $\frac{n}{\log\log n}$ Common-CRCW processors and in $O(\log n)$ time using $\frac{n}{\log n}$ EREW processors. Both these algorithms are work-time optimal; in fact, the EREW algorithm turns out to also be time-optimal. The proximate points problem has interesting and quite unexpected applications to problems in pattern recognition, digital geometry, shape analysis, compression, decomposition, and image reconstruction. We now summarize the main applications of our algorithm for the proximate points problem.

- A very simple algorithm for the convex hull of a set of n planar points sorted by x-coordinate running in $O(\log n)$ time using $\frac{n}{\log n}$ EREW processors and in $O(\log\log n)$ time using $\frac{n}{\log\log n}$ Common-CRCW processors. Our algorithm has the same performance as those of [1, 3, 2], being much simpler and more intuitive.

- Algorithms for the Voronoi map, the distance map, the maximal empty figure and the largest empty figure of a binary image of size $n \times n$. The *Voronoi map* assigns each pixel the position of the nearest black pixel. The *distance map* assigns each pixel the distance to the nearest black pixel. An *empty circle* of the image is a circle filled with white pixels. The *maximal empty circle* is an empty circle included in no other circle. The *largest empty circle* is an empty circle with the largest radius.

- A work optimal algorithm for the Euclidean distance map of a binary image of size $n \times n$ running in $O(\log\log n)$ time using $\frac{n^2}{\log\log n}$ Common-CRCW processors or in $O(\log n)$ time using $\frac{n^2}{\log n}$ EREW processors. We also show that the distance map of various metrics, including the well-known L_k metric ($k \geq 1$), can be computed in the same manner. Fujiwara *et al.* [5] presented a work-optimal algorithm running in $O(\log n)$ time and using $\frac{n^2}{\log n}$ EREW processors and in $O(\frac{\log n}{\log\log n})$ time using $\frac{n^2 \log\log n}{\log n}$ Common-CRCW processors. As we see it, our algorithm has three major advantages over Fujiwara's algorithm. First, the performance of our algorithm for the CRCW is superior; second, our algorithm applies to a large array of distance metrics; finally, our algorithm is much simpler and more intuitive.

- An algorithm for the maximal empty circle and for the largest empty circle of an $n \times n$ binary image running in $O(\log n)$ time using $\frac{n^2}{\log n}$ EREW processors and in $O(\log\log n)$ time using $\frac{n^2}{\log\log n}$ Common-CRCW processors. This algorithm is applicable to various other figures including circles, squares, diamonds, n-gons.

Due to page limitations, the applications to pattern analysis and image processing will be discussed in the journal version of this work.

2 The proximate points problem: a first look

For a point p in the plane we write $p = (x(p), y(p))$ in the obvious way. We let $d(p, q)$ denote the Euclidean distance between the points p and q.

Throughout this section we consider a set $P = \{p_1, p_2, \ldots, p_n\}$ of n points above the x-axis sorted by increasing x-coordinate. Let I_i, $(1 \leq i \leq n)$, be the locus of all the points q' on the x-axis for which $d(q', p_i) \leq d(q', p_j)$, i.e., $q' \in I_i$ if and only if p_i is the closest point to q' over all points in P. Elementary geometry confirms that every set I_i is an interval. Accordingly, $I_1, I_2, \ldots, I_n$ are the *proximate intervals* of P. Notice that some of these intervals may be empty. In case the interval I_i is non-empty, we say that p_i is a *proximate point* of P. A point q on the x-axis is the *boundary* between p_i and p_j if $d(p_i, q) = d(p_j, q)$. Figure 1 illustrates an example of proximate intervals and the Voronoi diagram of P. Clearly, the Voronoi diagram partitions the x-axis into proximate intervals. In Figure 1, p_1, p_2, p_4, p_6, and p_7 are proximate points; the others are not. Observe that the leftmost and rightmost points of P are always proximate points.

For three points p_i, p_j, p_k with $x(p_i) < x(p_j) < x(p_k)$, we say that p_j is *dominated* by p_i and p_k if p_j is not a proximate point of $\{p_i, p_j, p_k\}$. Note that one processor can determine in O(1) time whether p_j is *dominated* by p_i and p_k. A point p_i of P is a proximate point if and only if no pair of points in P dominates p_i.

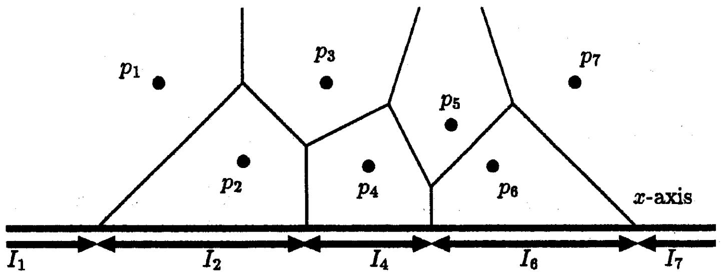

Figure 1: *Voronoi diagram and proximate intervals*

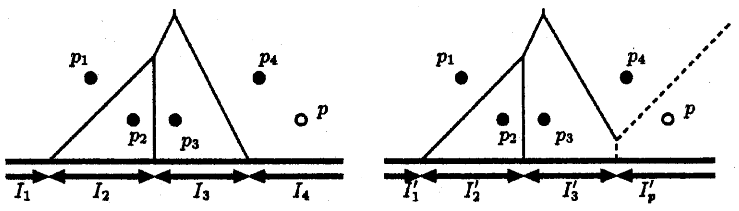

Figure 2: *Addition of* p *to* $P = \{p_1, p_2, p_3, p_4\}$.

Let P be as above and let p be a point to the right of P. We wish to compute the proximate intervals of $P \cup \{p\}$. Assume, wlog, that all points in P are proximate points, and let $I_1, I_2, \ldots, I_n$ be the proximate intervals of P. Further, let $I'_1, I'_2, \ldots, I'_n, I'_p$ be the proximate intervals of $P \cup \{p\}$ and refer to Figure 2. There exists a ***unique*** point p_i, called the *contact point* between P and p, such that

1. for every j, $(1 < j < i)$, p_j is not dominated by p_{j-1} and p. Moreover, $I'_j = I_j$ and p_j is a proximate point of $P \cup \{p\}$;
2. p_i is not dominated by p_{i-1} and p, and the boundary between p_i and p is in I_i; the left part of I_i, separated by the boundary, is I'_i. The right part of the x-axis is I'_p;
3. for every j, $(i < j \leq n)$, p_j is dominated by p_{j-1} and p. Moreover, I'_j is empty and p_j is not a proximate point of $P \cup \{p\}$.

Next, suppose that P is partitioned into subsets $P_L = \{p_1, p_2, \ldots, p_n\}$ and $P_R = \{p_{n+1}, p_{n+2}, \ldots, p_{2n}\}$. We are interested in updating the proximate intervals in the process or merging P_L and P_R. Let $I_1, I_2, \ldots, I_n$ and $I_{n+1}, I_{n+2}, \ldots, I_{2n}$ be the proximate intervals of P_L and P_R, respectively. We assume, wlog, that all these proximate intervals are nonempty. Let $I'_1, I'_2, \ldots, I'_{2n}$ be the proximate intervals of $P = P_L \cup P_R$, and refer to Figure 3. There exist ***unique*** proximate points $p_i \in P_L$ and $p_j \in P_R$, called the *contact points* between P_L and P_R, such that

1. for every k, $(1 < k < i)$, p_k is not dominated by p_{k-1} and p_j. Moreover, $I'_k = I_k$ and p_k is a proximate point of P;
2. p_i is not dominated by p_{i-1} and p_j, and the boundary between p_i and p_j is in I_i. The left part of I_i separated by the boundary is I'_i;
3. for every k, $(i < k \leq n)$, p_k is dominated by p_{k-1} and p_j. Moreover, the interval I'_k is empty and p_k is not a proximate point of P;
4. for every k, $(n < k \leq j)$, p_k is dominated by p_i and p_{k+1}. Moreover, the interval I'_k is empty and p_k is not a proximate point of P;
5. p_j is not dominated by p_i and p_{j+1}, and the boundary between p_i and p_j is in I_j. The right part of I_j separated by the boundary is I'_j;
6. for every k, $(j < k < 2n)$, p_k is not dominated by p_i and p_{k+1}. Moreover, $I'_k = I_k$ and p_k is a proximate point of P.

Next, using the two observations above, we propose a simple $O(n)$-time sequential algorithm for

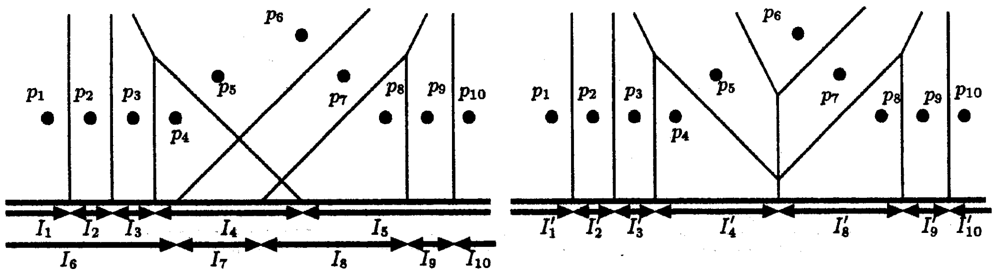

Figure 3: *Illustrating the merging of two sets of proximate intervals.*

finding the proximate points of a set $P = \{p_1, p_2, \ldots, p_n\}$ with $x(p_1) < x(p_2) < \cdots < x(p_n)$. The algorithm uses a stack that, when the algorithm terminates, contains all the proximate points in P.

Algorithm `Sequential-Proximate-Points`;

```
1 for $i := 1$ to $n$ do
2   while the stack has two or more points do
3     begin
4       $p' :=$ the top of the stack;
5       $p'' :=$ the next to the top of the stack;
6       if $p'$ is dominated by $p''$ and $p_i$ then
7         pop $p'$ from the stack
8       else exit while loop
9     end;
10    push $p_i$
11 end;
```

An easy inductive argument shows that at the end of the i-th iteration of the **for** loop, the stack contains all proximate points in $\{p_1, p_2, \ldots, p_i\}$. Once a point is removed from the stack it will never be considered again. Thus, we have

Lemma 2.1 *The task of finding the proximate points of a set of n points sorted by x-coordinate can be performed in $O(n)$ sequential time.*

3 Parallel algorithms for the proximate points problem

Consider a set $P = \{p_1, p_2, \ldots, p_n\}$ of points sorted by x-coordinate. For every point p_i we use three indices c_i, l_i, and r_i of p defined as:

1. $c_i = \max\{j \mid j \leq i$ and p_j is an proximate point $\}$;
2. $l_i = \max\{j \mid j < c_i$ and p_j is an proximate point $\}$;
3. $r_i = \min\{j \mid j > c_i$ and p_j is an proximate point $\}$.

Note that $l_i < c_i \leq i < r_i$ holds and there is no proximate point p_j such that $l_i < j < c_i$ or $c_i < j < r_i$. If $c_i = i$ then p_i is a proximate point, as shown in Figure 4.

Next, we are interested in finding the contact point between the set P and a point p to the right of P. We assume that for every i, $(1 \leq i \leq n)$, the indices c_i, l_i, and r_i are given and that m, $(m \leq n)$, processors are available.

Algorithm `Find-Contact-Point`

Step 1 Extract a sample $S(P)$ of size m consisting of the points $p_{c_1}, p_{c_{\frac{n}{m}+1}}, p_{c_{2\frac{n}{m}+1}}, \ldots$ in P. For every k, $(k \geq 0)$, check whether the point $p_{c_{k\frac{n}{m}+1}}$ is dominated by $p_{l_{k\frac{n}{m}+1}}$ and p, and whether $p_{r_{k\frac{n}{m}+1}}$ is dominated by $p_{c_{k\frac{n}{m}+1}}$ and p. If $p_{c_{k\frac{n}{m}+1}}$ is not dominated but $p_{r_{k\frac{n}{m}+1}}$ is dominated, then $p_{c_{k\frac{n}{m}+1}}$ is the desired contact point.

Step 2 Find k such that the point $p_{r_{k\frac{n}{m}+1}}$ is not dominated by $p_{c_{k\frac{n}{m}+1}}$ and p, and $p_{c_{(k+1)\frac{n}{m}+1}}$ is dominated by $p_{l_{(k+1)\frac{n}{m}+1}}$ and p.

Step 3 Execute recursively this algorithm for the set of points $P' = \{p_{r_{k\frac{n}{m}+1}}, p_{r_{k\frac{n}{m}+1}+1}, p_{r_{k\frac{n}{m}+1}+2}, \ldots, p_{l_{(k+1)\frac{n}{m}+1}}\}$ to find the contact point.

Since the set P' contains at most $l_{(k+1)\frac{n}{m}+1} - r_{k\frac{n}{m}+1} + 1 \leq \frac{n}{m} - 1$ points, the depth of the recursion is $O(\frac{\log n}{\log m})$. Thus, we have

Lemma 3.1 *The task of finding the contact point between a set P of n points in the plane sorted by x-coordinate and a point p to the right of P can be performed in $O(\frac{\log n}{\log m})$ time using m CREW processors.*

Next, consider two sets $P_L = \{p_1, p_2, \ldots, p_n\}$ and $P_R = \{p_{n+1}, p_{n+2}, \ldots, p_{2n}\}$ of points in the

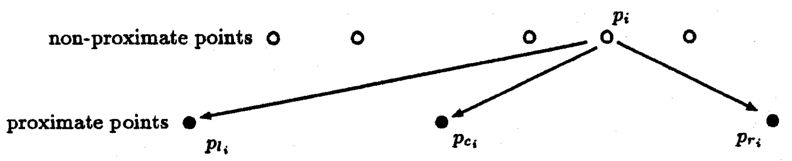

Figure 4: *Illustrating l_i, c_i, and r_i for p_i*

plane such that $x(p_1) < x(p_2) < \cdots < x(p_{2n})$. Assume that for every i the indices c_i, l_i and r_i are given and that m processors are available. The following algorithm finds the contact points of P_L and P_R.

Algorithm `Find-Contact-Points-Between-Sets`

Step 1 Extract $\sqrt{m}$ sample $S(P_L) = \{p_{c_1}, p_{c_{2\frac{n}{\sqrt{m}}+1}}, p_{c_{3\frac{n}{\sqrt{m}}+1}}, \ldots, \}$ from P_L. Using the algorithm `Find-Contact-Point` and $\sqrt{m}$ of the processors available, determine for each sample point $p_{c_{k\frac{n}{\sqrt{m}}+1}}$, the corresponding contact point $q_{c_{k\frac{n}{\sqrt{m}}+1}}$ in P_R.

Step 2 For each k, $(0 \le k \le \sqrt{m}-1)$, check whether the point $p_{c_{k\frac{n}{\sqrt{m}}+1}}$ is dominated by $p_{l_{k\frac{n}{\sqrt{m}}+1}}$ and $q_{c_{k\frac{n}{\sqrt{m}}+1}}$, and whether the point $p_{r_{k\frac{n}{\sqrt{m}}+1}}$ is dominated by $p_{c_{k\frac{n}{\sqrt{m}}+1}}$ and $q_{c_{k\frac{n}{\sqrt{m}}+1}}$. If $p_{c_{k\frac{n}{\sqrt{m}}+1}}$ is not dominated, yet $p_{r_{k\frac{n}{\sqrt{m}}+1}}$ is, output $p_{c_{k\frac{n}{\sqrt{m}}+1}}$ and $q_{c_{k\frac{n}{\sqrt{m}}+1}}$ as the desired contact points.

Step 3 Find k such that the point $p_{r_{k\frac{n}{\sqrt{m}}+1}}$ is not dominated by $p_{c_{k\frac{n}{\sqrt{m}}+1}}$ and $q_{c_{k\frac{n}{\sqrt{m}}+1}}$, yet $p_{c_{(k+1)\frac{n}{\sqrt{m}}+1}}$ is dominated by $p_{l_{(k+1)\frac{n}{\sqrt{m}}+1}}$ and $q_{c_{(k+1)\frac{n}{\sqrt{m}}+1}}$.

Step 4 Execute recursively this algorithm for $P'_L = \{p_{r_{k\frac{n}{\sqrt{m}}+1}}, p_{r_{k\frac{n}{\sqrt{m}}+1}+1}, p_{r_{k\frac{n}{\sqrt{m}}+1}+2}, \ldots, p_{l_{(k+1)\frac{n}{\sqrt{m}}+1}}\}$ and P_R to find the contact points.

By Lemma 3.1, Step 1 can be takes $O(\frac{\log n}{\log m})$ time on the CREW model. Steps 2 and 3 run, clearly, in $O(1)$ time. Since P'_L contains at most $\frac{n}{\sqrt{m}} - 1$ points, the depth of recursion is $O(\frac{\log n}{\log m})$. Thus, we have

Lemma 3.2 *The task of finding the contact points between the sets $P_L = \{p_1, p_2, \ldots, p_n\}$ and $P_R = \{p_{n+1}, p_{n+2}, \ldots, p_{2n}\}$ of points in the plane such that $x(p_1) < x(p_2) < \cdots < x(p_{2n})$ can be performed in $O(\frac{\log^2 n}{\log^2 m})$ time using m CREW processors.*

We now show how to compute the proximate points of a set P of n points in the plane sorted by x-coordinate in $O(\log\log n)$ time on the Common-CRCW. Assume that n processors are available. The idea is simple: first, we determine for every i, the indices c_i, l_i, and r_i. Next, we retain the points p_i for which $c_i = i$.

Algorithm `Find-Proximate-Points`

Step 1 Partition the set P into $n^{1/3}$ subsets $P_0, P_1, \ldots, P_{n^{1/3}-1}$ such that for every k, $(0 \le k \le n^{1/3}-1)$, $P_k = \{p_{kn^{2/3}+1}, p_{kn^{2/3}+2}, \ldots, p_{(k+1)n^{2/3}}\}$. For every point p_i in P_k, $(0 \le k \le n^{1/3}-1)$, determine the indices c_i, l_i, and r_i local to P_k.

Step 2 Compute the contact points of each pair of sets P_i and P_j, $(0 \le i < j \le n^{1/3}-1)$, using $n^{1/3}$ of the processors available. Let $q_{i,j} \in P_i$ denote the contact point between P_i and P_j.

Step 3 For every P_i, find the rightmost contact point p_{rc_i} among all the points $q_{i,j}$ with $j < i$ and find the leftmost contact point p_{lc_i} over all points $q_{i,j}$ with $j > i$. Clearly, $x(p_{rc_i}) = \max\{x(q_{i,j}) \mid j < i\}$ and $x(p_{lc_i}) = \min\{x(q_{i,j}) \mid j > i\}$.

Step 4 For each set P_i, the proximate points lying between rc_i and lc_i (inclusive) are proximate points of P. Update each c_i, l_i, and r_i.

Clearly, Step 2 runs in $O(\frac{\log^2 n^{2/3}}{\log^2 n^{1/3}}) = O(1)$ time. Step 3 runs in $O(1)$ time as well. The updating of the indices c_i, l_i, and r_i in Step 4 can be performed in $O(1)$ time. We only discuss c_i. In each P_i, the value of c_j, $(rc_i < j \le lc_i)$, is not changed. For all the points p_j with $lc_i < j$, the value of c_j must be changed to lc_i. For all points p_j with $j < rc_i$, the value of c_j is changed to lc_{i-1}, if P_{i-1} has an proximate point. However, if P_{i-1} does not contain a proximate points, we have to find the nearest subset that does. For this, first check whether each P_i has a proximate point. Next, we determine in O(1) time P_k such that $k = \max\{j \mid j < i$ and P_j contains a proximate point$\}$. Thus, Step 4 can be done in $O(1)$ time. Note that the depth of the recursion is $O(\log\log n)$. Thus, we have

Lemma 3.3 *An instance of size n of the proximate points problem can be solved in $O(\log\log n)$ time using n Common-CRCW processors.*

Next, we show that the number of processors can be reduced by a factor of $\log\log n$ without increasing the running time. The idea is as follows: begin by partitioning the set P into $\frac{n}{\log\log n}$ subsets $P_1, P_2, \ldots, P_{\frac{n}{\log\log n}}$ each of size $\log\log n$. Next, using algorithm `Sequential-Proximate-Points` find the proximate points within each subset in $O(\log\log n)$ sequential time and, in the process, remove from P all the points that are not proximate points. For every i, $(1 \leq i \leq \frac{n}{\log\log n})$, let $\{p_{i,1}, p_{i,2}, \ldots\}$ be proximate points in the set P_i.

At this moment, run `Find-Proximate-Point` on $P_1 \cup P_2 \cup \cdots \cup P_{\frac{n}{\log\log n}}$. Since n processors are needed to update the indices c_i, l_i, and r_i in $O(1)$, we will proceed slightly differently. The idea is the following: while executing the algorithm, some of the proximate points will cease to be proximate points. To maintain this information efficiently, we use *ranges* $[L_1, R_1], [L_2, R_2], \ldots, [L_{\frac{n}{\log\log n}}, R_{\frac{n}{\log\log n}}]$ such that for each P_i, $\{p_{i,L_i}, p_{i,L_i+1}, \ldots, p_{i,R_i}\}$ are the current proximate points. While executing the algorithm, P_i may contain no proximate points. To find the neighboring proximate points, we use the pointers $L'_1, L'_2, \ldots, L'_{\frac{n}{\log\log n}}$ and $R'_1, R'_2, \ldots, R'_{\frac{n}{\log\log n}}$ such that

- $L'_i = \max\{j \mid j < i$ and the set P_j contains proximate point,
- $R'_i = \min\{j \mid j > i$ and the set P_j contains proximate point.

By using this strategy, we can find the contact point between a point and P in $O(\frac{\log n}{\log m})$ time using m processors as discussed in Lemma 3.1. Thus, the contact points between two subsets can be found in the same manner as in Lemma 3.2. Finally, the task of updating L_i, R_i, L'_i, and R'_i in Step 4 can be done in $O(1)$ time by using $\frac{n}{\log\log n}$ processors. To summarize, we have the following result.

Theorem 3.4 *An instance of size n of the proximate points problem can be solved in $O(\log\log n)$ time using $\frac{n}{\log\log n}$ Common-CRCW processors.*

Finally, we show that `Find-Proximate-Points` can be implemented efficiently on the EREW-PRAM. For this recall that one step of an m-processor CRCW-PRAM can be simulated by an m-processor EREW-PRAM in $O(\log m)$ time [6]. Consequently, Steps 2, 3, and 4 can be performed in $O(\log n)$ time using n EREW processors, as the CRCW performs these steps in $O(1)$ time using n processors. Let $T_{EREW}(n)$ be the worst-case running time on the EREW. Then, the recurrence describing the EREW time complexity becomes $T_{EREW}(n) = T_{EREW}(n^{\frac{2}{3}}) + O(\log n)$, confirming that $T(n) \in O(\log n)$. Consequently, we have:

Lemma 3.5 *An instance of size n of the proximate points problem can be solved in $O(\log n)$ time using n EREW processors.*

Using, essentially, the same idea as for the CRCW-PRAM, we can reduce the number of processors by a factor of $\log n$ without increasing the computing time. Thus, we have

Theorem 3.6 *An instance of size n of the proximate points problem can be solved in $O(\log n)$ time using $\frac{n}{\log n}$ EREW processors.*

Further, it is not hard to see that Theorems 3.4 and 3.6 hold for the case of any L_k metric with $k \geq 1$.

4 Lower Bounds

The main goal of this section is to show that the running time of the Common CRCW algorithm for the proximate points problem cannot be improved while retaining work-optimality. This, in effect, will prove that our Common-CRCW algorithm is work-time optimal. We then show that our EREW algorithm is time-optimal.

The work-optimality of both algorithms is obvious; every point must be accessed to solve the proximate points problem, thus, $\Omega(n)$ work is required of any algorithm solving the problem. Our arguments rely, in part, on the following well known result [6, 7].

Lemma 4.1 *The task of finding the minimum (maximum) of n real numbers requires $\Omega(\log\log n)$ time on the CRCW provided that $n\log^{O(1)} n$ processors are available.*

Obviously, even if all the input numbers are non-negative, the task still requires $\Omega(\log\log n)$ time. Further, we rely on the following classic result of Cook *et al.* [4].

Lemma 4.2 *The task of finding the minimum (maximum) of n real numbers requires $\Omega(\log n)$ time on the CREW (therefore, also on the EREW) even if infinitely many processors are available.*

We shall reduce the task of finding the minimum of a collection A of n non-negative $a_1, a_2, \ldots, a_n$ to the proximate points problem. In other words, we will show that the minimum finding problem can be converted to the proximate points problem in $O(1)$ time.

For this purpose, let $a_1, a_2, \ldots, a_n$ be an arbitrary input to the minimum problem and refer to Figure 5. We construct a set $P = \{p_1, p_2, \ldots, p_{2n}\}$ of points in the plane by setting for every i, $(1 \leq$

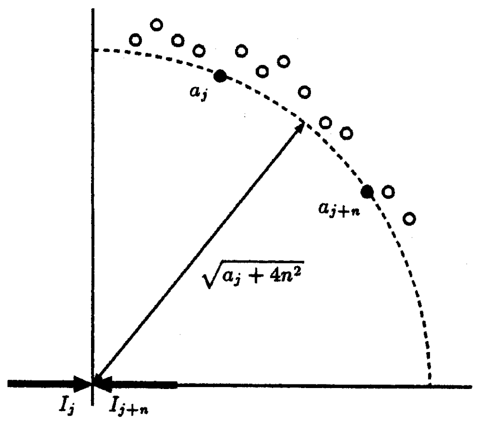

Figure 5: *Illustrating the construction of P*

$i \leq n$), $p_i = (i, \sqrt{a_i + 4n^2 - i^2})$ and $p_{i+n} = (i + n, \sqrt{a_i + 4n^2 - (i+n)^2})$. Notice that this construction guarantees that the points in P are sorted by x-coordinate and that for every i, $(1 \leq i \leq n)$, the distance between the point p_i and the origin is exactly $\sqrt{a_i + 4n^2}$. The set P that we just constructed has the following property.

Lemma 4.3 *a_j is the minimum of A if and only if both points p_j and p_{j+n} are proximate points of P.*

Proof. Assume that a_j is the minimum of A. Consider the circle C of radius is $\sqrt{a_j + 4n^2}$ centered at the origin. Clearly, the points a_j and a_{j+n} are on C, while all the other points are outside C. Therefore, there exists a small real number $\epsilon > 0$ such that a_j is the closest points of $(-\epsilon, 0)$ over all points in P, and a_{j+n} is the closest points of $(\epsilon, 0)$. Thus, both p_j and p_{j+n} are proximate points of P. Further, the proximate intervals for p_j and p_{j+n} are adjacent. Thus, no point p_i with $k < i < k+n$ can be a proximate point. □

Lemma 4.3 guarantees that we can determine the minimum of A once the proximate points of P are known. Now the conclusion follows immediately from Lemma 4.1. Thus, we have the following important result.

Theorem 4.4 *Any algorithm that solves an instance of size n of the proximate points problem on the CRCW must take $\Omega(\log\log n)$ time provided that $n\log^{O(1)} n$ processors are available.*

Using exactly the same construction we obtain the following lower bound for the CREW.

Theorem 4.5 *Any algorithm that solves an instance of size n of the proximate points problem on the CREW (also on the EREW) must take $\Omega(\log n)$ time even if an infinite number of processors are available.*

Obviously, the EREW algorithm presented in Section 3 solving the proximate points problem in $O(\log n)$ time and optimal work runs, within the same resource bounds, on the CREW-PRAM. By Theorem 4.4 the corresponding CREW algorithm is also time-optimal.

Further, in the case of the L_k metric, for every i, $(1 \leq i \leq n)$, the points $p_i = (i, \sqrt[k]{a_i + (2n)^k - i^k})$ and $p_{i+n} = (i+n, \sqrt[k]{a_i + (2n)^k - (i+n)^k})$ allow us to find the minimum of A. Thus, the above results hold for the L_k metric.

5 Application to the convex hull problem

Let $P = \{p_1, p_2, \ldots, p_n\}$ be a planar set of n points sorted by x-coordinate. We assume, with no loss of generality that for every i, $x(p_i)^2 \leq y(p_i)$. The line segment p_1p_n partitions the convex hull of P into the *lower hull*, lying below the segment, and the *upper hull*, lying above it. In this section we focus on the computation of the lower hull only, the computation of the upper hull being similar.

Referring to Figure 6 let $Q = \{q_1, q_2, \ldots, q_n\}$ be the set of n points obtained from P by setting for every i, $q_i = (x(p_i), \sqrt{y(p_i) - x(p_i)^2})$. The following result captures the relationship between P and Q.

Lemma 5.1 *For every j, $(1 \leq j \leq n)$, the point p_j is an extreme point of the lower hull of P if and only if q_j is a proximate point of Q.*

Proof. If $j = 1$ or $j = n$, then p_j is an extreme point and so q_j is a proximate point. Thus, the lemma is correct for $j = 1$ and $j = n$. For a fixed j, $2 \leq j \leq n-1$, let i and k be arbitrary indices such that $1 \leq i < j < k \leq n$. Let b_i and b_k be the boundaries between q_i and q_j and between q_j and q_k, respectively. Notice that $d(q_i, b_i) = d(q_j, b_i)$ implies that $(x(p_i) - x(b_i))^2 + y(p_i) - x(p_i)^2 = (x(p_j) - x(b_i))^2 + y(p_j) - x(p_j)^2$. Thus, we have $2x(b_i) = \frac{y(p_i)-y(p_j)}{x(p_i)-x(p_j)}$. Similarly, we have $2x(b_k) = \frac{y(p_k)-y(p_j)}{x(p_k)-x(p_j)}$.

Notice that the slopes of the segments p_ip_j and p_kp_j equal $2x(b_i)$ and $2x(b_k)$, respectively. It follows that the point p_j lies below the segment p_ip_k if and only if q_j is not dominated by q_i and q_k. In other words, the point p_j is an extreme point of the lower hull if and only if q_j is a proximate point of P. □

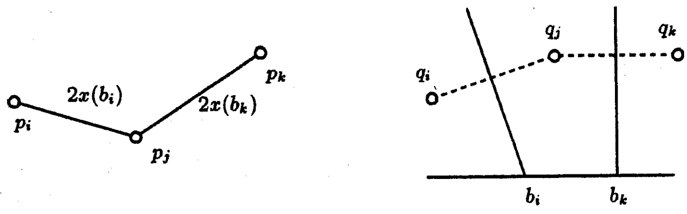

Figure 6: *Illustrating the transformation of P into Q*

Lemma 5.1 suggests the following algorithm for determining the extreme points of the lower hull of $P = \{p_1, p_2, \ldots, p_n\}$.

Algorithm `Find-Lower-Hull`

```
Step 1 Compute the minimum $Y = \min\{y(p_i)-x(p_i)^2 \mid 1 \leq i \leq n\}$ and construct the set $Q = \{q_1, q_2, \ldots, q_n\}$
       such that for every $i$, $(1 \leq i \leq n)$,
       $q_i = (x_i, \sqrt{y(p_i) - x(p_i)^2 - Y})$.
Step 2 Determine the proximate points of $Q$. Having done that, select $p_i$ as an extreme point in the
       lower hull whenever $q_i$ is a proximate point.
```

Clearly, the minimum Y is used to avoid that the argument of the square root be negative. The minimum finding can be done in $O(\log n)$ time using $\frac{n}{\log n}$ EREW processors and in $O(\log \log n)$ time using $\frac{n}{\log \log n}$ CRCW processors [6]. Step 2 can be completed by using the algorithms of Theorems 3.4 and 3.6. Thus, we have

Theorem 5.2 *The task of determining the convex hull of n points sorted by x-coordinate can be performed in $O(\log \log n)$ time using $\frac{n}{\log \log n}$ Common-CRCW processors and in $O(\log n)$ time using $\frac{n}{\log n}$ EREW processors.*

Acknowledgement

This work is supported in part by NSF grant CCR-9522093, by ONR grant N00014-97-1-0526, and by Grant-in-Aid for Encouragement of Young Scientists (08780265) from Ministry of Education, Science, Sports, and Culture of Japan

References

[1] O. Berkman, B. Schieber, and U. Vishkin. A fast parallel algorithm for finding the convex hull of a sorted point set. *Int. J. Comput. Geom. Appl.*, 6(2):231–241, June 1996.

[2] D. Z. Chen. Efficient geometric algorithms on the EREW PRAM. *IEEE Trans. Parallel Distrib. Syst.*, 6(1):41–47, January 1995.

[3] W. Chen, K. Nakano, T. Masuzawa, and N. Tokura. Optimal parallel algorithms for computing convex hulls. *IEICE Transactions*, J74-D-I(6):809–820, September 1992.

[4] S. A. Cook, C. Dwork, and R. Reischuk. Upper and lower time bounds for parallel random access machines without simultaneous writes. *SIAM J. Comput.*, 15:87–97, 1986.

[5] A. Fujiwara, T. Masuzawa, and H. Fujiwara. An optimal parallel algorithm for the Euclidean distance maps. *Information Processing Letters*, 54:295–300, 1995.

[6] J. JáJá. *An Introduction to Parallel Algorithms.* Addison-Wesley, 1992.

[7] L. G. Valiant. Parallelism in comparison problem. *SIAM J. Comput.*, 4(3):348–355, 1975.