An Improved Parallel Algorithm for a Geometric Matching Problem with Application to Trapezoid Graphs

Muhammad H. Alsuwaiyel
Department of Information and Computer Science
King Fahd University of Petroleum & Minerals
Dhahran 31261, Saudi Arabia
e-mail:suwaiyel@ccse.kfupm.edu.sa

Keywords: Parallel algorithms, Maximum matching, Circular-arc graphs, Trapezoid graphs.

Abstract. Let B be a set of n_b blue points and R a set of n_r red points in the plane, where $n_b + n_r = n$. A blue point b and a red point r can be matched if r dominates b, that is, if $x(b) \leq x(r)$ and $y(b) \leq y(r)$. We consider the problem of finding a maximum cardinality matching between the points in B and the points in R. We give an adaptive parallel algorithm to solve this problems that runs in $O(\log^2 n)$ time using the CREW PRAM with $O(n^{2+\epsilon}/\log n)$ processors for some $\epsilon, 0 < \epsilon < 1$. It follows that finding the minimum number of colors to color a trapezoid graph can be solved within these resource bounds.

1 Introduction

Let B be a set of n_b blue points and R a set of n_r red points in the plane, where $n_b + n_r = n$. A blue point b and a red point r can be matched if r dominates b, that is, if $x(b) \leq x(r)$ and $y(b) \leq y(r)$. We consider the problem of finding a maximum cardinality matching between the points in B and the points in R. In[3], it was shown that this problem can be solved in $O(\log^2 n)$ time using the CREW PRAM with $O(n^3/\log n)$ processors. It was used in [3] to solve the problem of finding a maximum clique in a circular-arc graph (which now can be solved in $O(\log n)$. In this paper, we first give a parallel divide-and-conquer algorithm for this problem that runs in $O(\log^3 n)$ time using the CREW PRAM with $O(n^2/\log n)$ processors, and thus reducing the number of processors by a

factor of n on the expense of increasing the running time by a factor of $\log n$. Next, we derive an adaptive parallel divide-and-conquer algorithm that runs in $O(\log^2 n)$ time using the CREW PRAM with $O(n^{2+\epsilon}/\log n)$ processors for some $\epsilon, 0 < \epsilon < 1$.

Let $G = (V, E)$ be an undirected graph where $V = \{v_1, v_2, \ldots, v_n\}$. Let $\mathcal{T} = \{T_1, T_2, \ldots, T_n\}$ be a set of n trapezoids whose upper and lower corners lie on two (infinite) horizontal lines L_t and L_b, respectively, such that $(v_i, v_j) \in E$ if and only if the intersection of T_i and T_j is nonempty. In this case, G is called a *trapezoid graph* and $\mathcal{T}$ a *geometric representation* of G. The class of trapezoid graphs was first introduced by Dagan, Golumbic and Pinter[4] in the context of the channel routing problem. The intention is that L_t and L_b define a channel, and $T_i \in \mathcal{T}$ corresponds to a *net* (see [4] for more details). Figure 1 shows an example of a trapezoid graph and a possible geometric representation as a set of trapezoids. In [4], it was also shown that the problem of finding the minimum number of colors to color a trapezoid graph can be solved in $O(n^2)$ sequential time in the worst case. This bound was improved later in [7] to $O(n \log n)$.

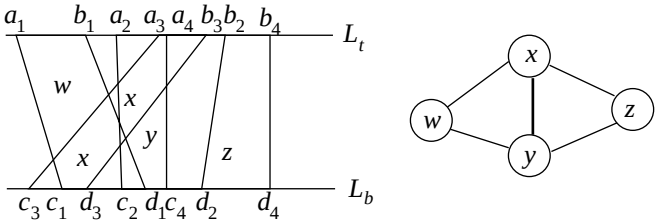

Figure 1: A trapezoid graph.

The first parallel algorithm for this problem was proposed in [6]. The algorithm runs in $O(\log^2 n)$ time using the CREW PRAM with $O(n^3/\log n)$ processors, and is based on an algorithm for the geometric matching problem presented in [3]. As the parallel time and number of processors of the algorithm in [6] are dominated by the matching step, and no other step takes more than $O(\log^2 n)$ time using $O(n^2)$ processors, it follows that the cost of finding the chromatic number of a trapezoid graph is at most that of the algorithm for the matching problem to be developed in Section 4.

2 Sequential algorithms

Consider the following greedy algorithm that finds a maximum matching[5]. Sort the red points in increasing x-coordinate, and let this sorted list be R_s. Scan the points in R_s in increasing order of their x-coordinates, each time matching the current red point r with the unmatched blue point of maximum y-coordinate that is dominated by r. We will call this method REDGREEDY.

Consider the following greedy algorithm, which we will call BLUEGREEDY. Sort the blue points in decreasing x-coordinate, and let this sorted list be B_s. Scan the points in B_s in decreasing order of their x-coordinates, each time matching the current blue point b with the unmatched red point of minimum y-coordinate that dominates b. The following theorem is easy to prove by induction.

Theorem 1 *Given a set of blue points B and a set of red points R in the plane, Algorithm* BLUEGREEDY *finds a maximum matching on $B \cup R$.*

Note that, although both BLUEGREEDY and REDGREEDY are optimal, they may give different matchings. In [3], a parallelization of REDGREEDY was used to extract the set of red points that can be matched. Theorem 1 implies that the set of blue points that can be matched can also be computed. Consequently, both algorithms will be used later to derive a parallel divide-and-conquer algorithm for solving the matching problem.

3 Identifying the set of matched points

In this section we briefly describe a method to identify those red points that can be matched using Algorithm REDGREEDY. This method is described in [3]. We then show that the same method can be used, with a minor modification, to identify those blue points that can be matched using Algorithm BLUEGREEDY. Assume that no two points have the same x-coordinate or y-coordinate. For a red point r, let $Reg(r)$ denote the region in the plane dominated by r, i.e., the southwest quadrant of the plane with origin at r. Define the *deficiency* of a point p, denoted by $Def(p)$, to be the number of red points minus the number of blue points in $Reg(r)$.

To find the red points that are matched by Algorithm REDGREEDY, we construct a weighted directed $(n_r+1)\times(n_r+1)$ grid graph by drawing for each red point the horizontal and vertical lines passing through it, and then adding the horizontal line at $-\infty$ and the vertical line at $-\infty$ (recall that we have assumed that no two points have the same x or y-coordinate). Each vertical edge is directed downward and is of weight 0, and each horizontal edge is directed from left to right and is of weight $Def(p) - Def(q)$, where p and q are, respectively, its right and left endpoints. Let s be the leftmost top vertex of the grid, and let $t_0, t_1, \ldots, t_{n_r}$ be the bottom vertices of the grid in left-to-right order. We will call such a grid a *red grid*. See Fig. 2 for an example.

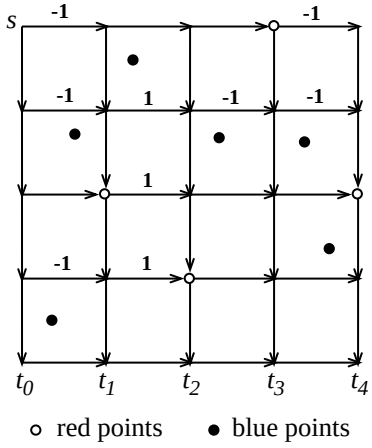

Figure 2: A 5×5 red grid on a set of four red points and six blue points.

In this example, the only nonzero weights are shown. The weight of a path is the sum of the weights of all its edges. For each $t_j, 0 \leq j \leq n_r$, define π_j to be a path of largest weight from s to t_j, and let w_j be the weight of π_j. The proof of the following lemma can be found in [3].

Lemma 1 *For all $j, 1 \leq j \leq n_r$, r_j is unmatched by Algorithm* REDGREEDY *if and only if in the red grid $w_j = w_{j-1} + 1$.*

For example, in Fig. 2, $w_0 = 0, w_1 = 0, w_2 = 1, w_3 = 1, w_4 = 1$. Hence, r_1, r_3 and r_4 are matched and r_2 is unmatched by Algorithm REDGREEDY.

The weights w_j of the paths $\pi_j, 0 \leq j \leq n_r$, can be computed in $O(\log^2 n)$ time using the CREW PRAM with $O(n^2/\log n)$ processors[1, 2]. Thus, we have the following theorem[3].

Theorem 2 *Given a set of n_r red points and a set of n_b blue points, the size of a maximum matching and the set of red points that can be matched using Algorithm* REDGREEDY *can be found in $O(\log^2 n)$ time using the CREW PRAM with $O(n^2/\log n)$ processors.*

Now consider negating the x and y-coordinates of all blue and red points. Specifically, let $B^- = \{(-x, -y) \mid (x, y) \in B\}$ and $R^- = \{(-x, -y) \mid (x, y) \in R\}$. Let the new red points play the role of the blue points and vice versa in Algorithm REDGREEDY. It is not hard to see that applying Algorithm REDGREEDY to the new sets B^- and R^- with the roles of red and blue reversed is the same as applying Algorithm BLUEGREEDY to the original sets B and R. Now construct a weighted directed $(n_b+1) \times (n_b+1)$ grid graph by drawing for each blue point in B^- the horizontal and vertical lines passing through it and adding the horizontal and vertical lines at $-\infty$ as before. Call this the *blue grid*. It is not hard to see that with the roles of blue and red reversed, Lemma 1 and Theorem 2 will apply and, consequently, we will be able to find the matched blue points with the same time complexity. Thus, we have

Lemma 2 *For all $j, 1 \leq j \leq n_b$, b_j is unmatched by Algorithm* BLUEGREEDY *if and only if in the blue grid $w_j = w_{j-1} + 1$.*

Theorem 3 *Given a set of n_r red points and a set of n_b blue points, the size of a maximum matching and the set of blue points that can be matched using Algorithm* BLUEGREEDY *can be found in $O(\log^2 n)$ time using the CREW PRAM with $O(n^2/\log n)$ processors.*

4 The algorithm

The matching algorithm in [3] finds a maximum matching in $O(\log^2 n)$ time using the CREW PRAM with $O(n^3/\log n)$ processors. For each point, the algorithm uses Theorem 2 to find the *size* of a maximum matching of the original

point set plus additional $O(n)$ red points, and based on this size, the blue point, if any, that is matched by Algorithm REDGREEDY is found.

In this section, we will exploit Theorem 3 to cut down on the number of processors used by the algorithm in [3] by a factor of n on the expense of increasing the running time by a factor of $\log n$. Later, we will refine the method to obtain an adaptive algorithm that runs in $O(\log^2 n)$ time using the CREW PRAM with $O(n^{2+\epsilon}/\log n)$ processors, for some appropriately chosen $\epsilon, 0 < \epsilon < 1$.
First, we present the simple algorithm. The algorithm uses the divide-and-conquer technique and is described as follows.
Preprocessing step. Use Theorem 2 to remove from R those red points that are unmatched. Let $|R| = m$. Use Theorem 3 to remove from B those blue points that are unmatched. Obviously, $|B| = m$. Sort both B and R in increasing order of their x-coordinates.

Algorithm MATCH1.

1. If $|R| \leq \log n$, then use the sequential algorithm REDGREEDY to find a maximum matching. Return the set of matched pairs.

2. Divide the set of red points in R into two subsets R_1 and R_2, where $|R|_1 = \lfloor m/2 \rfloor$ and $|R|_2 = \lceil m/2 \rceil$. Here the points in R_1 are to the left of the points in R_2.

3. Let $r \in R_1$ be the rightmost point in R_1, i.e., the one with largest x-coordinate. Let $B_1' = \{b \in B \mid x(b) \leq x(r)\}$.

4. Use Theorem 3 to find the set $B_1 \subseteq B_1'$ of blue points that can be matched with the set of red points in R_1. Let $B_2 = B - B_1$.

5. Apply Algorithm MATCH1 recursively on the two pairs (B_1, R_1) and (R_2, B_2).

6. Return the set of matched pairs in (B_1, R_1) and (B_2, R_2).

Figure 3 Illustrates the operation of the algorithm. In this figure, the four red points are divided into two groups of two points each. In Step5, the algorithm will recurse on the two pairs:

$$(\{b_1, b_2\}, \{r_1, r_2\}), \quad (\{b_3, b_4\}, \{r_3, r_4\}).$$

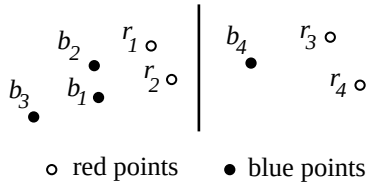

Figure 3: Example of Algorithm MATCH1.

Now we analyze the time complexity of the algorithm assuming that we have $O(n^2/\log n)$ processors. The preprocessing step takes $O(\log^2 n)$ time by Theorems 2 and 3. Step 1 takes $O(\log n \log\log n)$ time. Steps 2 and 3 take $O(1)$ time. By Theorem 3, the time taken by Step 4 is $O(\log^2 n)$. Finally, Step 6 takes $O(1)$ time. It follows that the running time of the algorithm is governed by the recurrence $T(n) = T(n/2) + O(\log^2(n))$ whose solution is $T(n) = O(\log^3 n)$. As to the number of processors, there are more processors than needed for the recursive calls.

Now we refine Algorithm MATCH1 to obtain an adaptive algorithm, which we will call MATCH2. Let $k = \lceil n^\epsilon \rceil$ for some appropriately chosen $\epsilon, 0 < \epsilon < 1$. In Step 2, instead of dividing the red points into two subsets, we will partition them into k subsets $R_1, R_2, \ldots, R_k$, of size $\lceil m/k \rceil$ each, except possibly the kth subset.

Modifying Steps 3 and 4 is not straightforward. In sequential terms, we may describe the refinement as follows. First, B_1 is computed and $B'_1 - B_1$ is added to B'_2. Next, B_2 is computed and $B'_2 - B_2$ is added to B'_3. We continue this way until finally B_{k-1} is computed and $B'_{k-1} - B_{k-1}$ is added to B'_k, which is exactly B_k. We proceed to implement this procedure in parallel as follows. For convenience, we will use the following notation. If $X_1, X_2, \ldots, X_j$ are j sets, then $X_{1,j}$ will denote their union. That is, $X_{1,j} = X_1 \cup X_2 \cup \ldots \cup X_j$. Let $R_{1,0} = B'_{1,0} = \{\}$. We start by computing the sets $R_{1,1}, R_{1,2}, \ldots, R_{1,k}, B'_{1,1}, B'_{1,2}, \ldots, B'_{1,k}$. Next, for $j = 1, 2, \ldots k$, we compute in parallel B_j as follows. We use Theorem 3 to compute $B_{1,j-1}$ from $B'_{1,j-1}$ and $R_{1,j-1}$. Then, we set $B''_j = B'_j \cup (B'_{1,j-1} - B_{1,j-1})$. Finally, we use Theorem 3 again to compute B_j from B''_j and R_j. After $B_1, B_2, \ldots, B_k$ have been computed, we use Algorithm MATCH2 recursively to

compute in parallel the matchings for the pairs $(B_1, R_1), (B_2, R_2), \ldots, (B_k, R_k)$. The results of these recursive calls constitute the desired matching. Algorithm MATCH2 is described more formally as follows.

Preprocessing step. Use Theorem 2 to remove from R those red points that are unmatched. Let $|R| = m$. Use Theorem 3 to remove from B those blue points that are unmatched. Obviously, $|B| = m$. Sort both B and R in increasing order of their x-coordinates. Compute $k = \lceil n^{\epsilon} \rceil$.

Algorithm MATCH2.

1. If $|R| \leq \log n$, then use the sequential algorithm REDGREEDY to find a maximum matching. Return the set of matched pairs.

2. Divide the set of red points in R into k subsets $R_1, R_2, \ldots, R_k$, of size $\lfloor m/k \rfloor$ each, except possibly the kth subset. Here the points in R_{j+1} are to the right of the points in $R_j, 1 \leq j < k$.

3. Let r_0 be the point $(-\infty, 0)$, and for $j = 1, 2, \ldots k$ let r_j be the rightmost point in R_j, i.e., the one with largest x-coordinate.

   For $j = 1, 2, \ldots k$, let $B'_j = \{b \in B \mid x(r_{j-1}) \leq x(b) \leq x(r_j)\}$.

4. For $j = 1, 2, \ldots k$, compute $R_{1,j} = R_1 \cup R_2 \cup \ldots \cup R_j$ and $B'_{1,j} = B'_1 \cup B'_2 \cup \ldots \cup B'_j$. Set $R_{1,0} = B'_{1,0} = \{\}$.

5. For $j = 1, 2, \ldots k$, do in parallel

   (a) Use Theorem 3 to compute $B_{1,j-1}$ from $B'_{1,j-1}$ and $R_{1,j-1}$.

   (b) Set $B''_j = B'_j \cup (B'_{1,j-1} - B_{1,j-1})$.

   (c) Use Theorem 3 to compute B_j from B''_j and R_j.

6. Use Algorithm MATCH2 recursively to compute in parallel the matchings for the pairs $(B_1, R_1), (B_2, R_2), \ldots, (B_k, R_k)$.

7. Return the set of matched pairs in $(B_1, R_1), (B_2, R_2), \ldots, (B_k, R_k)$.

Figure 4 Illustrates the operation of the algorithm.

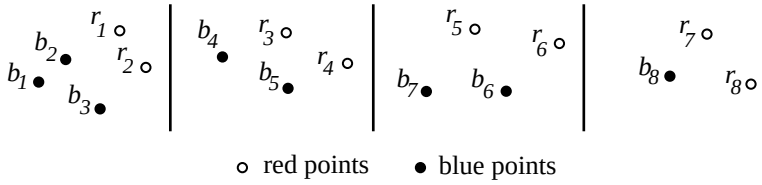

Figure 4: Example of Algorithm MATCH2.

In this figure, the eight red points are divided into four groups of two points each. The sets $B'_j, 1 \leq j \leq 4$, computed in Step 4 of the algorithm are

$$B'_1 = \{b_1, b_2, b_3\}, \quad B'_2 = \{b_4, b_5\}, \quad B'_3 = \{b_6, b_7\}, \quad B'_4 = \{b_8\}.$$

The sets computed in Step 5 of the algorithm starting from $j = 2$ are

$$\begin{array}{llll} j:1 & B_{1,0} = \{\}, & B''_1 = \{b_1, b_2, b_3\}, & B_1 = \{b_1, b_2\}. \\ j:2 & B_{1,1} = \{b_1, b_2\}, & B''_2 = \{b_3, b_4, b_5\}, & B_2 = \{b_4, b_5\}. \\ j:3 & B_{1,2} = \{b_1, b_2, b_4, b_5\}, & B''_3 = \{b_3, b_6, b_7\}, & B_3 = \{b_6, b_7\}. \\ j:4 & B_{1,3} = \{b_1, b_2, b_4, b_5, b_6, b_7\}, & B''_4 = \{b_3, b_8\}, & B_4 = \{b_3, b_8\}. \end{array}$$

Next, the algorithm will recurse on the following pairs:

$$\begin{array}{ll} (\{b_1, b_2\}, \{r_1, r_2\}), & (\{b_4, b_5\}, \{r_3, r_4\}), \\ (\{b_6, b_7\}, \{r_5, r_6\}), & (\{b_3, b_8\}, \{r_7, r_8\}). \end{array}$$

Now we analyze the complexity of Algorithm MATCH2. Let the running time of the algorithm be $T(n)$. The preprocessing step takes $O(\log^2 n)$ time by Theorems 2 and 3 using $O(n^2/\log n)$ processors. Step 1 takes $O(\log n \log\log n)$ time. Step 2 takes $O(1)$ time using k processors, where each processor needs to store the start and end indices. Step 3 takes $O(\log n)$ time using n processors. Step 4 takes $O(1)$ time using $O(kn) = O(n^{1+\epsilon})$ processors. By Theorem 3, Step 5 takes $O(\log^2 n)$ time using $O(kn^2/\log n) = O(n^{2+\epsilon}/\log n)$ processors. The cost of Step 6 is $T(n/k)$. Finally, Step 7 takes $O(1)$ time. Observe that for $j = 1, 2, \ldots, k$, $|B_j| = |R_j| = O(m/k) = O(n/k)$. Thus, the total number of processors needed for the recursive calls is $kO(k(n/k)^2/\log(n/k)) = O(n^2/\log n)$ processors. Consequently, the running time of the algorithm is governed by the recurrence $T(n) = T(n/k) + O(\log^2(n))$ whose solution is

$$T(n) = O(\log^2 n \log_k n) = O(\log^3 n/\log k) = O(\log^3 n/\log n^\epsilon) = O(\log^2 n).$$

It follows that the overall running time of the algorithm is $O(\log^2 n)$ using the CREW PRAM with $O(n^{2+\epsilon}/\log n)$, $0 < \epsilon < 1$. If, for example, we set $\epsilon = \lceil \log\log n / \log n \rceil$, the number of processors needed becomes $O(n^2)$.

References

[1] A. Aggarwal and J. Park, Notes on searching in multidimensional monotone arrays, *Proc.IEEE Symp. on Foundation of Computer Science* (1988) 497–512.

[2] M. J. Atallah, M. T. Goodrich and S. R. Kosaraju parallel algorithms for evaluating sequences of set–manipulation operations, *Proc. 3rd Aegean Workshop on Computing,* Lecture Notes in Computer Science **319** (1988) (Springer, Berlin, 1988) 1–10.

[3] S. K. Kim, A parallel algorithm for finding a maximum clique of a set of circular arcs of a circle, *Information Processing Letters* **34** (1990) 235–241.

[4] I. Dagan, M. C. Golumbic and R. Y. Pinter, Trapezoid graphs and their coloring, *Disc. Appl. Math.* **21** (1988) 35–46.

[5] T. Leighton and P. Shor, Tight bounds for minimax grid matching, with applications to the average case analysis of algorithms, *Proc. ACM Symp. on Theory of Computing* (1986) 91–103.

[6] Shin–ichi Nakayama and Shigeru Masuyama, A parallel algorithm for solving the coloring problem on trapezoid graphs, *Information Processing Letters* **62** (1997) 323–327.

[7] S. Felsner, R. Müller and L. Wernisch, Trapezoid graphs and generalizations, Geometry and Algorithms *Disc. Appl. Math.*, **74** (1997) 35–46.