COAST Tech Report 97-22

PARALLEL ALGORITHMS FOR LONGEST INCREASING CHAINS IN THE PLANE AND RELATED PROBLEMS

by Mikhail Atallah, Danny Z. Chen, and
Kevin S. Klenk

Center for Education and Research in
Information Assurance and Security,
Purdue University, West Lafayette, IN 47909

Parallel Processing Letters Vol. 9 No. 4 (1999) 511–520

PARALLEL ALGORITHMS FOR LONGEST INCREASING CHAINS IN THE PLANE AND RELATED PROBLEMS

MIKHAIL J. ATALLAH*, DANNY Z. CHEN† and KEVIN S. KLENK†

Received February 1999
Revised November 1999
Accepted by I. Stojmenovic

ABSTRACT

Given a set S of n points in the plane such that each point in S is associated with a nonnegative weight, we consider the problem of computing the single-source longest increasing chains among the points in S. This problem is a generalization of the planar maximal layers problem. In this paper, we present a parallel algorithm that computes the single-source longest increasing chains in the plane in $O(\log^2 n)$ time using $O(n^2/\log^3 n)$ processors in the CREW PRAM computational model. We also solve a related problem of computing the all-pairs longest paths in an n-node weighted planar st-graph, in $O(\log^2 n)$ time using $O(n^2/\log n)$ CREW PRAM processors. Both of our parallel algorithms are improvement over the previously best known results.

1 Introduction

A point p in the plane is said to *dominate* another point q if and only if $x(p) \geq x(q)$ and $y(p) \geq y(q)$, where $x(p')$ and $y(p')$ respectively denote the x- and y-coordinates of a point p'. Let S be a set of n points in the plane such that each point p in S is associated with a nonnegative weight $w(p)$, and let $\sigma = (p_1, p_2, \ldots, p_k)$ be a sequence of distinct points in S. The sequence σ is *increasing* if and only if p_i dominates p_{i-1} for all i, $1 < i \leq k$. We also call σ an increasing chain. The *length* of σ is the sum of the weights of the points in σ. The chain σ between p_1 and p_k is *longest* if no other p_1-to-p_k increasing chain passing through the points in S has a length greater than σ. In this paper, we study the problem of computing in parallel longest increasing chains in S and some related problems.

The notions of dominance between points and of increasing chains are useful in many probems in computational geometry, graph theory, scheduling, and economics, including problems on independent dominating sets in permutation graphs and problems on increasing subsequences of a sequence of numbers (cf. [5] for more on these). The problem of computing a longest increasing subsequence of a number sequence, for instance, has appeared in a number of areas, and there are $O(n \log n)$ time sequential algorithms for this problem (see [9–11] among others). In particular, increasing subsequence problems occur in the context of circle graphs, circular-arc graphs, interval graphs, and

*Dept. of Computer Sciences, Purdue University, West Lafayette, IN 47907, USA, E-mail: mja@cs.purdue.edu.
This author gratefully acknowledges the support of the COAST Project at Purdue University and its sponsors, in particular Hewlett Packard, DARPA, and the National Security Agency.
†Dept. of Computer Science and Engineering, University of Notre Dame, Notre Dame, IN 46556-5637, USA, E-mail: {chen,kklenk}@cse.nd.edu.
This research of this author was supported in part by the National Science Foundation under Grant CCR-9623585.

permutation graphs (see [3] for references). In this paper, we shall also exploit a connection between increasing chains and paths in st-graphs.

For sake of simplicity, we shall henceforth refer to all increasing chains simply as *chains.* Also, we will focus on computing the *lengths* of the longest chains/paths. Our algorithms can be easily modified to generate the actual longest chains/paths as well, by using standard techniques.

Some interesting work has been done on solving various chain problems. Atallah and Kosaraju [5] presented optimal sequential $O(n \log n)$ time algorithms for several problems related to planar chains. Atallah and Chen [3] gave a sequential $O(n^2)$ time algorithm for the unweighted planar all-pairs longest chains and a parallel algorithm for the weighted planar all-pairs version in $O(\log^2 n)$ time using $O(n^2/\log n)$ CREW PRAM processors.

The *single-source* longest chains problem is that of finding longest chains from a fixed *source* point of S to all other points of S. The single-source problem is in fact a generalization of the maximal layers problem, which computes the *maximal layers* of the points in S as follows: Find all the points of S that are not dominated by any other point of S, call these *layer-1 points*, and remove them from S; produce points of subsequent layers by repeating this process on S, until S becomes empty. Optimal sequential $O(n \log n)$ time solutions for the planar maximal layers problem easily follow from the known algorithms for the longest increasing subsequence [9–11]; an $O(n \log n)$ time algorithm for the 3-D version of the problem was recently given by Atallah, Goodrich, and Ramaiyer [4]. Aggarwal and Park [1] gave a parallel algorithm for the planar maximal layers problem that takes $O(\log^2 n)$ time and $O(n^2/\log n)$ CREW PRAM processors.

This paper extends Atallah and Chen's parallel approach [3] to computing the planar single-source longest chains. One of the ways in which this extension differs from [3] is that it involves a new *implicit spreading* idea (to be described in Section 2) that helps us overcome the difficulty of extending the solution for a smaller size problem to one for a larger size problem (this need did not occur in [3]). Another idea involves a different kind of partitioning scheme, leading to a logarithmic number of iterations that, while performed in sequence, individually involve parallelism in the way they are performed. Our parallel single-source algorithm takes $O(\log^2 n)$ time using $O(n^2/\log^3 n)$ processors. Our solution, when applied to the (simpler) planar maximal layers problem, improves the processor bound of Aggarwal and Park [1] by a $\log^2 n$ factor while retaining the same time bound.

The computational model that we use is the CREW PRAM [13]. Recall that the CREW PRAM is a synchronous parallel model in which each processor can access any memory location in constant time. It allows simultaneous accesses to the same memory location by multiple processors only if all such accesses are for reading data only.

We also show a new application of the parallel all-pairs longest chains algorithm by Atallah and Chen [3], to computing the all-pairs *longest paths* in weighted planar st-graphs. Briefly, a planar st-graph is a planar directed acyclic graph with exactly one source s and exactly one sink t that is embedded in the plane such that s and t are both on the boundary of the outer face [16]. Planar st-graphs have been used in many applications: computational geometry, graph drawing, motion planning, partial orders, planar graph embedding, and VLSI layout (see [16] for references).

We are not aware of any previous parallel algorithm for computing the all-pairs longest paths in weighted planar st-graphs. Of course, one could attempt to apply known parallel algorithms for *shortest paths* in directed graphs to solve this st-graph problem [8, 12, 15]. In particular, Cohen [8] gave algorithms that compute shortest paths in planar directed graphs, in $O(\log^4 n)$ time and $O(n^2/\log^3 n)$ CREW PRAM processors.

Our algorithm takes advantage of the underlying geometry of a planar st-graph to enables us to compute the all-pairs longest paths in an n-node weighted planar st-graph in $O(\log^2 n)$ time using $O(n^2/\log n)$ CREW PRAM processors. Specifically, our computation is performed by reducing the all-pairs longest paths in a planar st-graph to that of the all-pairs longest chains in the plane.

2 Preliminaries

As in [3], our solution is based on fast matrix multiplications of particular types of matrices (specifically, *monotone* matrices) in the $(\max, +)$ closed semi-ring, i.e., $(M' \times M'')(i,j) = \max_k\{M'(i,k) + M''(k,j)\}$. All of our matrix multiplications are of this form.

Atallah and Chen [3] considered the problem of computing an $n \times n$ matrix D of the lengths of the longest chains between each pair of points in S. Thus, $D(p,q)$ gives the length of a longest p-to-q chain, for any $p, q \in S$. The computation of D is based on the following observation.

Lemma 1 ([3]) *Let V_l, V_m and V_r be three vertical lines with $x(V_l) < x(V_m) < x(V_r)$. Let S_l (resp., S_r) be the set of points in S whose x-coordinates are $\geq x(V_l)$ (resp., $\geq x(V_m)$) and $\leq x(V_m)$ (resp., $\leq x(V_r)$). Let the set X_l (resp., X_r) contain the horizontal projections of the points of S_l (resp., S_r) onto V_l (resp., V_r), and X_m contain the horizontal projections of the points of $S_l \cup S_r$ onto V_m (see Figure 1). Let the weights of the points in X_l (resp., X_m, X_r) be all zero. Let $\Omega = S_l \cup S_r \cup X_l \cup X_m \cup X_r$. Then for every increasing chain C through the points in Ω from a point $p \in X_l$ to a point $q \in X_r$, $y(p) \leq y(q)$, there is a p-to-q increasing chain C' through Ω such that C' is at least as long as C and C' goes through some point $w \in X_m$.*

Let $M[A, B]$ denote a matrix that contains the lengths of longest chains starting from a point in the set A, ending at a point in the set B, and passing through a particular set of points. Lemma 1 implies that $M[X_l, X_r] = M[X_l, X_m] \times M[X_m, X_r]$. Atallah and Chen [3] also showed that the matrices $M[X_l, X_r]$, $M[X_l, X_m]$, and $M[X_m, X_r]$ have special properties that enable fast matrix multiplications (by using the monotone matrix searching algorithms in [1,2]). This is summarized in the next lemma.

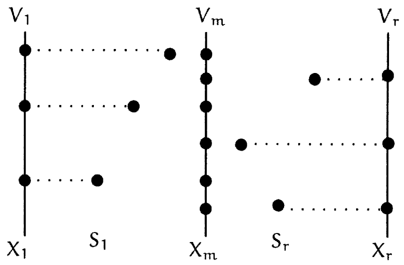

Figure 1: Illustrating Lemma 1.

Lemma 2 ([3]) *Let V_l, V_m, V_r, X_l, X_m, X_r, and Ω be defined as in Lemma 1. Let the point set X_l (resp., X_m, X_r) be ordered by increasing y-coordinates along V_l (resp., V_m, V_r). Assume that the size $|X_l|$ of X_l is proportional to $|X_r|$. Then, given the matrices $M[X_l, X_m]$ and $M[X_m, X_r]$, the matrix $M[X_l, X_r]$ can be computed in $O(\log |X_l|)$ time and $O(|X_l|^2/\log |X_l|)$ CREW PRAM processors.*

Lemma 2 was the basis for a two-phase algorithm for computing the all-pairs planar longest chains in [3], which we shall extend. We sketch below some needed structures of our algorithm.

Let $S = \{p_1, p_2, \ldots, p_n\}$. Without loss of generality (WLOG), we assume that $x(p_1) < x(p_2) < \cdots < x(p_n)$. Let $V_0, V_1, \ldots, V_n$ be vertical lines such that $x(V_0) < x(p_1), x(p_n) < x(V_n)$, and $x(p_i) < x(V_i) < x(p_{i+1})$ for all $i \in \{1, 2, \ldots, n-1\}$. Let T be a complete n-leaf binary tree. For the i-th leaf v of T in the left-to-right order, associate with v the region I_v of the plane that is between V_{i-1} and V_i. For each internal node v of T, associate with v the region I_v consisting of the union of the regions of its children. That is, if v has children u and w, then $I_v = I_u \cup I_w$. The tree T is called the *computation tree* on the point set S.

Let v be a node of T. Suppose that the left (resp., right) boundary of I_v is V_i (resp., V_j), and let $S_v = S \cap I_v$. Let L_v (resp., R_v) be the set consisting of the horizontal projections of S_v onto V_i (resp., V_j). If v is an internal node of T with left child u and right child w, then let Y_v denote $R_u \cup L_w$, i.e., the horizontal projections of the points of S_v onto the vertical line that separates the region I_u from the region I_w. The weights of all projection points are zero.

Atallah and Chen's algorithm proceeds in the following two phases. In Phase 1, start at the leaves and go up the tree T level by level, computing, at each level, the $M[L_v, R_v]$ matrices for nodes v of T, which contain the lengths of all the L_v-to-R_v longest chains (these chains begin on L_v and end on R_v, possibly going through points in S_v along the way). For an internal node v of T, $M[L_v, R_v]$ is computed from the two matrices $M[L_v, Y_v]$ and $M[Y_v, R_v]$ based on Lemma 2. Note that when the computation reaches v, only the matrix $M[L_u, R_u]$ (resp., $M[L_w, R_w]$) is available from its left child u (resp., right child w). The matrices $M[L_v, Y_v]$ and $M[Y_v, R_v]$ need to be obtained from $M[L_u, R_u]$

and $M[L_w, R_w]$, respectively. $M[L_v, Y_v]$ is computed, in $O(\log n)$ time and $O(|S_v|^2/\log n)$ processors, from $M[L_u, R_u]$ by the following *spreading procedure* (the computation of $M[Y_v, R_v]$ is similar): (1) Compute, for every point $z \in L_v$ (resp., Y_v), the lowest (resp., highest) point $l(z)$ (resp., $h(z)$) such that $l(z) \in L_u$ (resp., $h(z) \in R_u$) and that $y(l(z)) \geq y(z)$ (resp., $y(h(z)) \leq y(z)$), and (2) for every pair of points p and q such that $p \in L_v$ and $q \in Y_v$, do the following: If $y(p) \leq y(l(p)) \leq y(q)$, then let $M[L_v, Y_v](p, q) = M[L_u, R_u](l(p), h(q))$; otherwise, let $M[L_v, Y_v](p, q) = 0$.

The matrices $M[L_v, Y_v]$, $M[Y_v, R_v]$, and $M[L_v, R_v]$ are stored at v (even after the computation has reached higher level nodes of T); these matrices are useful in Phase 2. Phase 1 is accomplished in $O(\log^2 n)$ time, $O(n^2/\log^2 n)$ processors, and $O(n^2)$ space.

Phase 2 is a top-down computation, starting at the root of T and going downward to the leaves, one level at a time. This phase uses the information produced in Phase 1 to obtain the lengths of the all-pairs longest chains. In particular, for every pair of nodes u, w at the same level of T such that u is to the left of w, it computes the matrix $M[R_u, L_w]$.

Note that, although the above spreading procedure *explicitly* obtains the matrix $M[L_v, Y_v]$ from the matrix $M[L_u, R_u]$ in [3], we in this paper choose to represent $M[L_v, Y_v]$ *implicitly*. That is, the matrix $M[L_v, Y_v]$ can be fully described by $M[L_u, R_u]$ and the two sorted lists L_v and Y_v. To obtain this representation of $M[L_v, Y_v]$, we only need to perform Step 1 of the spreading procedure (but not Step 2). This takes $O(\log(|L_v| + |Y_v|))$ time and $O((|L_v| + |Y_v|)/\log(|L_v| + |Y_v|))$ processors. After that, useful information about the matrix $M[L_v, Y_v]$ is readily available. For example, every entry $M[L_v, Y_v](p, q)$ can be computed in $O(1)$ time and one processor from $M[L_u, R_u]$, as in Step 2 of the above spreading procedure. We call this the *implicit spreading procedure*. The implicit spreading procedure is important to our single-source algorithm in the next section.

3 Single-source longest chain algorithm

This section presents the $O(\log^2 n)$ time, $O(n^2/\log^3 n)$ processor algorithm for computing the single-source longest chains in the plane. WLOG, we assume that the source point p^* dominates all other points in S. We also assume that we have already sorted the points in S by their x-coordinates and by their y-coordinates, in $O(\log n)$ time and $O(n)$ processors [13].

The all-pairs longest chain algorithm in [3] relies heavily on the multiplication of two $m \times m$ matrices via monotone matrix searching. For our single-source computation, we often rely on multiplying an $m \times m$ matrix with an $m \times 1$ matrix. The following lemma is for this purpose:

Lemma 3 *Let V_l and V_r be two vertical lines such that $x(V_l) < x(V_r)$. Let S' be the set of points of S that are between V_l and V_r, with $m = |S'|$. Let X_l (resp., X_r) be the set of horizontal projection points of S' onto V_l (resp., V_r) that are ordered by increasing y-coordinates and whose weights are*

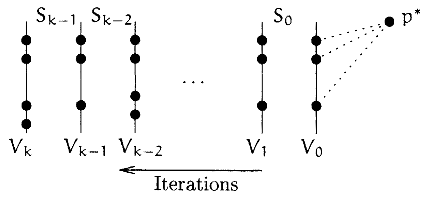

Figure 2: Illustrating the single-source longest chain algorithm.

all zero. Then, given the $m \times m$ *matrix* $M[X_l, X_r]$ *and the* $m \times 1$ *matrix* $M[X_r, p^*]$ *(for lengths of the longest chains through the points of* S*), the* $m \times 1$ *matrix* $M[X_l, p^*]$ *can be computed in* $O(\log m)$ *time and* $O(m)$ *EREW PRAM processors.*

Proof: This is an easy adaptation of Lemma 2 to the single-source situation. In this case, multiplying an $m \times m$ length matrix by an $m \times 1$ matrix is the main operation, which can be done by the $O(\log m)$ time, $O(m)$ processor algorithm for searching an $m \times m$ monotone matrix in [6]. □

Our algorithm below for computing the single-source longest chains in the plane works in an iterative fashion, taking advantage of the monotonicity shown in Lemmas 1 and 3.

1. Partition the set $S - \{p^*\}$ of $n-1$ points into k subsets of (roughly) size $n/\log n$ each, by using $k = \log n$ vertical lines $V_0, V_1, \ldots, V_k$, in the *right-to-left* order (see Figure 2). WLOG, we assume that no point of S is on any line V_i and that the source point p^* is the only point of S to the right of the line V_0.

2. Let S_i be the subset of the points in S that lie between the vertical lines V_i and V_{i+1}. Project horizontally the points of S_i onto V_i and V_{i+1} (we call these projection points the *boundary points* for S_i). Let the weights of all the projection points be zero.

3. Compute the lengths of the longest chains from the projection points on V_i to the projection points on V_{i+1} (through the points of S_i).

4. For $i = 0, 1, \ldots, k-1$, iteratively compute the lengths of the longest chains from the source point p^* to the boundary points on each vertical line V_i.

5. Compute the lengths of the longest chains from p^* to the points in every subset S_i.

We now discuss the details of the steps of the above algorithm. Steps 1 and 2 can be done easily in $O(\log n)$ time using $O(n)$ processors. Hence we only need to focus on Steps 3, 4, and 5.

The computation of the longest chain lengths in Step 3 from the projection points on V_i to the projection points on V_{i+1} can be performed in parallel for every $i = 0, 1, \ldots, k-1$, by using Phase 1 of the algorithm in [3]. We run Phase 1 of [3] on each point set S_i, generating and maintaining a $|S_i|$-leaf computation tree T_i on S_i, together with all the length matrices produced during this process. Let $R(S_i)$ (resp., $L(S_i)$) be the set of projection points of S_i onto V_i (resp., V_{i+1}). Then after Step 3, the length matrix $M[L(S_i), R(S_i)]$ is available at the root node of the tree T_i. Each tree T_i and the length matrices stored at its nodes are useful in the subsequent steps of the algorithm. Phase 1 of the algorithm in [3] takes $O(\log^2 m)$ time, $O(m^2/\log^2 m)$ processors, and $O(m^2)$ space on a set of m points. Thus, the computation on each point set S_i (whose size is $O(n/\log n)$) in Step 3 takes $O(\log^2 n)$ time, $O(n^2/\log^4 n)$ processors, and $O(n^2/\log^2 n)$ space. Summing over all the $k = \log n$ sets S_i, this step takes $O(\log^2 n)$ time, $O(n^2/\log^3 n)$ processors, and $O(n^2/\log n)$ space.

Step 4 computes the longest chain lengths from the boundary points of each point set S_i to the source point p^*. This computation is done *iteratively*, as follows. Let P_i be the set of the horizontal projection points of all the points in S onto the vertical line V_i (thus $|P_i| = n$). Then for every $i = 0, 1, \ldots, k-1$, compute, by using the implicit spreading procedure in Section 2, the (implicitly represented) length matrix $M[P_{i+1}, P_i]$ of size $n \times n$ from the $(n/\log n) \times (n/\log n)$ matrix $M[L(S_i), R(S_i)]$ (from Step 3, $M[L(S_i), R(S_i)]$ is already available). This takes $O(\log n)$ time and $O(n/\log n)$ processors for each i, and $O(\log n)$ time and $O(n)$ processors for all the $k = \log n$ instances. (Note that, if we had used an *explicit* representation for every $n \times n$ matrix $M[P_{i+1}, P_i]$, then it would have taken $O(\log n)$ time and $O(n^2/\log n)$ processors to obtain each such matrix, and $O(\log n)$ time and $O(n^2)$ processors to obtain all the $\log n$ such matrices, clearly too expensive an approach!) Next, the following iterative procedure is performed:

> First, let the length matrix $M[P_0, p^*]$ be such that every entry $M[P_0, p^*](p, p^*) = w(p^*)$, where $w(p^*)$ is the weight of p^*. This is because there is no point of $S - \{p^*\}$ to the right of the line V_0 (see Figure 2). For any $i \in \{0, 1, \ldots, k-2\}$, once the matrix $M[P_i, p^*]$ is available, compute $M[P_{i+1}, p^*]$ by multiplying the $n \times n$ matrix $M[P_{i+1}, P_i]$ with the $n \times 1$ matrix $M[P_i, p^*]$, in $O(\log n)$ time and $O(n)$ processors based on Lemma 3.

Finally, extract the matrix $M[R(S_i), p^*]$ from $M[P_i, p^*]$, for every $i \in \{0, 1, \ldots, k-1\}$. Since there are $k - 1 = O(\log n)$ iterations to perform, Step 4 takes altogether $O(\log^2 n)$ time and $O(n)$ processors.

For Step 5, recall that for every $i \in \{0, 1, \ldots, k-1\}$, we have maintained (in Step 3) the computation tree T_i on the point set S_i together with a collection of length matrices stored at the nodes of T_i. These length matrices were computed on S_i by using Phase 1 of the algorithm in [3]. Step 5 uses these matrices to compute the longest chain lengths from p^* to all the points in every S_i.

In Step 5, a top-down computation is performed on each tree T_i. For an internal node v of T_i, let v be associated with the region I_v of the plane that is between the vertical lines $H_l(v)$ and $H_r(v)$. For example, the root of T_i is associated with the region bounded by the vertical lines V_i and V_{i+1}.

Let L_v (resp., R_v) be the set consisting of the horizontal projections of $S \cap I_v$ onto $H_l(v)$ (resp., $H_r(v)$). Let $H_m(v)$ be the vertical line separating the regions I_u and I_w, where u and w are the left and right children of v, and let Y_v be the set consisting of the horizontal projections of $S \cap I_v$ onto $H_m(v)$. Suppose that the top-down computation now reaches the node v and assume that the length matrix $M[R_v, p^*]$ is already available (note that, initially, the matrix $M[R(S_i), p^*]$ is available from Step 4). At v, the matrix $M[Y_v, p^*]$ is first computed; this is done by multiplying the matrix $M[Y_v, R_v]$ (available from Step 3) with the matrix $M[R_v, p^*]$, in $O(\log |R_v|)$ time and $O(|R_v|)$ processors based on Lemma 3. The matrix $M[R_u, p^*]$ is then extracted from $M[Y_v, p^*]$ and the matrix $M[R_w, p^*]$ is extracted from $M[R_v, p^*]$. After that, the computation is carried out recursively at each child of v.

T_i has $O(\log n)$ levels and each level of T_i uses $O(\log n)$ time and altogether $O(|S_i|)$ processors to compute. Hence, over all $k = \log n$ point sets S_i, Step 5 takes $O(\log^2 n)$ time and $O(n)$ processors.

Summing over all steps, our parallel single-source algorithm presented takes $O(\log^2 n)$ time, $O(n^2/\log^3 n)$ CREW PRAM processors, and $O(n^2/\log n)$ space. As mentioned before, this algorithm also solves the maximal layers problem in the same complexity bounds (with the weights of all the points of S being a unit). In comparison, Aggarwal and Park's CREW PRAM algorithm for the maximal layers problem [1] takes $O(\log^2 n)$ time, $O(n^2/\log n)$ processors, and $O(n^2)$ space.

Remark: We should point out that there is a trade-off between the time and processor/space bounds in our single-source algorithm, based on parameter k. By using a larger value of k, one can reduce the processor/space bounds at the expense of a larger time bound. For example, if we choose $k = \log^2 n$ (instead of $\log n$), then the time bound increases to $O(\log^3 n)$ while the processor (resp., space) bound decreases to $O(n^2/\log^5 n)$ (resp., $O(n^2/\log^2 n)$). The *time* $\times$ *processors* product of the algorithm with $k = \log^2 n$ is hence a factor of $\log n$ smaller than the one with $k = \log n$.

4 All-pairs longest paths in planar st-graphs

This section presents an $O(\log^2 n)$ time, $O(n^2/\log n)$ CREW PRAM processor algorithm for computing the all-pairs longest paths in a weighted planar st-graph G.

A *planar* st*-graph* is a planar directed acyclic graph with exactly one source s and exactly one sink t, that is embedded in the plane such that s and t are both on the boundary of the outer face. A *weighted planar* st*-graph* is a planar st-graph such that each of its edges e is associated with a nonnegative weight $w(e)$. As in [16], we assume that the input graph representation for the weighted planar st-graph G is already embedded (i.e., for each vertex v of G, the cyclical ordering of the neighboring vertices of v in the embedding is given).

As mentioned earlier, we would like to use the all-pairs longest chains result of [3] to compute the all-pairs longest paths in a weighted planar st-graph. To do this, we need to reduce the graph problem to the geometric problem. Two problems need to be solved for this reduction: (1) We need to map the st-graph G onto a point set S in the plane in such a way that all directed paths in G are preserved by the dominance relation among the points in S. (2) We need to convert the weight information of the edges in G to the weights of the points in S.

Our approach depends on the following lemma:

Lemma 4 ([7, 14, 16]) *Let* G *be an embedded planar* st*-graph with* n *vertices. There exist two total orders on the vertices of* G*, denoted by* $<_l$ *and* $<_r$*, such that for any two vertices* u *and* v *of* G*, there is a directed path from* u *to* v *in* G *if and only if* $u <_l v$ *and* $u <_r v$*. Furthermore, the orders* $<_l$ *and* $<_r$ *can be computed in* $O(\log n)$ *time using* $O(n/\log n)$ *EREW PRAM processors.*

A parallel algorithm for Lemma 4 can be found in [16]. The total orders shown in Lemma 4 allow us to solve our first problem, which is to reduce a planar st-graph to a point set in the plane. Given a planar st-graph with n vertices, we simply obtain the two orderings $<_l$ and $<_r$ by Lemma 4. These orderings map the vertices of the graph onto points in the plane, with one ordering specifying the x-coordinates of the points and the other ordering specifying the y-coordinates of the points.

We also need to solve the second problem, by putting the weights of the edges in a graph onto the vertices in another "equivalent" graph. This is done as follows: (*a*) Let the weights of all (original) vertices of the st-graph G be zero; (*b*) for each (original) edge $e = (u, v) \in G$, insert an "artificial vertex" z such that z splits e into two edges (u, z) and (z, v), and let $w(z) = w(e)$; (*c*) let the weights of all edges in the (new) graph be zero. This process clearly creates a new planar st-graph G' with $O(n)$ weighted vertices (and hence $O(n)$ zero-weight edges), such that all longest paths in G' between the original vertices of G are equivalent to their longest paths in G. It is easy to convert G into G' in $O(\log n)$ time and $O(n/\log n)$ processors.

Therefore, our reduction from the weighted planar st-graph G to a point set S in the plane consists of the following two steps: (*i*) Create G' from G, and (*ii*) map G' onto a set S of $O(n)$ weighted points (i.e., the weight of each point in S is the same as that of its corresponding vertex in G'). This reduction clearly takes $O(\log n)$ time and $O(n/\log n)$ processors.

Once we obtain the set S of $O(n)$ weighted points in the plane, we compute the all-pairs longest chains passing through the points in S, by using the all-pairs algorithm in [3]. By Lemma 4, the directed paths between the vertices in G' correspond precisely to the dominance relation between the points in S. Hence the lengths of the all-pairs longest paths in G' (and hence in G) can be easily obtained from the lengths of the all-pairs longest chains passing through the points in S.

Our time and work bounds are dominated by Atallah and Chen's all-pairs longest chains algorithm [3]. Thus, our algorithm takes $O(\log^2 n)$ time and $O(n^2/\log n)$ CREW PRAM processors.

References

[1] A. Aggarwal and J. Park. Notes on searching in multidimensional monotone arrays. In *Proceedings of the 29th Annual Symposium on Foundations of Computer Science*, pages 497–512. IEEE, 24–26 Oct. 1988.

[2] A. Apostolico, M. J. Atallah, L. L. Larmore, and S. McFaddin. Efficient parallel algorithms for string editing and related problems. *SIAM J. Comput.*, 19(5):968–988, Oct. 1990.

[3] M. J. Atallah and D. Z. Chen. Computing the all-pairs longest chains in the plane. *International Journal of Computational Geometry & Applications*, 5(3):257–271, 1995.

[4] M. J. Atallah, M. T. Goodrich, and K. Ramaiyer. Biased finger trees and three-dimensional layers of maxima. In *Proceedings of the Tenth Annual Symposium on Computational Geometry*, pages 150–159. ACM, 6–8 June 1994.

[5] M. J. Atallah and S. R. Kosaraju. An efficient algorithm for maxdominance, with applications. *Algorithmica*, 4:221–236, 1989.

[6] M. J. Atallah and S. R. Kosaraju. An efficient parallel algorithm for the row minima of a totally monotone matrix. *Journal of Algorithms*, 13(3):394–413, Sept. 1992.

[7] G. Birkhoff. Lattice theory. *American Mathematical Society Colloquium Publications*, 25, 1979.

[8] E. Cohen. Efficient parallel shortest-paths in digraphs with a separator decomposition. *Journal of Algorithms*, 21(2):331–357, Sept. 1996.

[9] R. B. K. Dewar, S. M. Merritt, and M. Sharir. Some modified algorithms for Dijkstra's longest upsequence problem. *Acta Inf.*, 18(1):1–15, 1982.

[10] E. W. Dijkstra. Some beautiful arguments using mathematical induction. *Acta Inf.*, 13(1):1–8, 1980.

[11] M. L. Fredman. On computing the length of longest increasing subsequences. *Discrete Mathematics*, pages 29–35, 1975.

[12] Y. Han, V. Y. Pan, and J. H. Reif. Efficient parallel algorithms for computing all pair shortest paths in directed graphs. *Algorithmica*, 17(4):399–415, Apr. 1997.

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

[14] T. Kameda. On the vector representation of the reachability in planar directed graphs. *Inf. Process. Lett.*, 3(3):75–77, Jan. 1975.

[15] P. N. Klein and S. Subramanian. A linear-processor polylog-time algorithm for shortest paths in planar graphs. In *Proceedings of the 34th Annual Symposium on Foundations of Computer Science*, pages 259–270. IEEE, 3–5 Nov. 1993.

[16] R. Tamassia and J. S. Vitter. Parallel transitive closure and point location in planar structures. *SIAM J. Comput.*, 20(4):708–725, Aug. 1991.