Nordic Journal of Computing
Vol 6, pp 422-445, 1999

NC Coloring Algorithms for Permutation Graphs

Maria Andreou[‡] and Stavros D. Nikolopoulos[*]

[‡]*Department of Computer Engineering and Informatics, University of Patras, GR-26500 Patras, Greece.*
[*]*Department of Computer Science, University of Ioannina, GR-45110 Ioannina, Greece.*
emails: mandreou@ceid.upatras.gr, stavros@cs.uoi.gr

Abstract — We show that the problem of coloring a permutation graph of size n can be solved in $O(\log n \log k)$ time using $O(kn^2 / \log k \log^2 n)$ processors on the CREW PRAM model of computation, where $1 < k < n$. We estimate the parameter k on random permutation graphs and show that the coloring problem can be solved in $O(\log n \log\log n)$ time in the average-case on the CREW PRAM model of computation with $O(n^2)$ processors. Our computational strategy goes as follows: Given a permutation π or its corresponding permutation graph $G[\pi]$, we first construct a directed acyclic graph $G^*[\pi]$ using certain combinatorial properties of π, and then compute longest paths in the directed acyclic graph using divide-and-conquer techniques. We show that the problem of coloring a permutation graph $G[\pi]$ is equivalent to finding longest paths in its acyclic digraph $G^*[\pi]$. The best-known parallel algorithms for the same problem run in $O(\log^2 n)$ time using $O(n^3 / \log n)$ processors on the CREW PRAM model of computation.

CR Classification: F.2.2, G.2.2

Key words: parallel algorithms, perfect graphs, permutation graphs, coloring problem, directed acyclic graphs, longest paths.

1. Introduction

An *undirected graph* is a pair $G = (V, E)$, where V is a finite set of n elements called *vertices* and E is a set of m unordered vertex pairs called *edges*. An undirected graph G with vertices numbered from 1 to n; that is, $V = \{1, 2, ..., n\}$, is called a *permutation graph* if there exists a permutation $\pi = [\pi_1, \pi_2, ..., \pi_n]$ on $N = \{1, 2, ..., n\}$ such that,

$$(i, j) \in E \Leftrightarrow (i - j)\,(\pi^{-1}(i) - \pi^{-1}(j)) < 0$$

for all $i, j \in N$, where π_i^{-1}, denoted here as $\pi^{-1}(i)$, is the index of the element i in π [5]. Given a permutation π on $N = \{1, 2, ..., n\}$, we shall say, hereafter, that it is of length n, and we shall denote its corresponding permutation graph as $G[\pi]$.

Many researchers have been devoted to the study of permutation graphs (see [1, 2, 4, 10, 13, 14, 16]). Although many sequential algorithms have been proposed for such graphs, few parallel algorithms have been appeared in the literature. We mention here some of them: Due to work of Helmbold and Mayr [6] and Kozen *et. al.* [9], the problem of recognizing permutation graphs was shown to be in the *NC* class. Helmbold and Mayr presented a parallel algorithm that recognizes a permutation graph in $O(\log^3 n)$

time using $O(n^4)$ processors on a CRCW PRAM model of computation. They also proposed parallel algorithms for the weighted clique problem and the coloring problem which run in $O(\log^3 n)$ time using $O(n^4)$ processors on the same model of computation. Moreover, given a permutation graph, their algorithms can construct the permutation that represents the permutation graph.

Our objective is to study the coloring problem on permutation graphs. Recently, Yu and Chen [15] proposed a technique that transfers the coloring problem into the largest-weight path problem. Specifically, their technique, first, transforms a permutation graph (combinatorial object) into a set of planar points (geometrical object), then constructs an acyclic directed graph by exploiting the domination relation within the geometric object and, finally, solves the largest-weight path problem on the acyclic directed graph. The parallel algorithm they proposed solves the coloring problem in $O(\log^2 n)$ time with $O(n^3 / \log n)$ processors on the CREW PRAM, or in $O(\log n)$ time with $O(n^3)$ processors on a CRCW PRAM model of computation. Moreover, Yu, Tseng and Chang [17] proposed a parallel algorithm that solves the maximum-weight independent set problem on permutation graphs in $O(\log^2 n)$ time with $O(n^3 / \log n)$ processors on the CREW PRAM. Thus, it can also solve the coloring problem within the same time and processor bounds.

In this paper, we present a parallel algorithm for solving the coloring problem on permutation graphs which runs in $O(\log n \log k)$ time using $O(kn^2 / \log k \log^2 n)$ processors on the CREW PRAM model of computation, where $1 < k < n$. Our algorithm transforms the coloring problem into the longest path problem on directed acyclic graphs. Specifically, given a permutation π of length n, the algorithm directly constructs an acyclic digraph $G^*[\pi] = (V^*, E^*)$ using certain combinatorial properties of π and then computes the length of the longest paths from a specific vertex $s \in V^*$ to every vertex $v \in V^*$ using divide-and-conquer techniques. We show that there is an one-to-one correspondence between the length of the longest path from s to a vertex v in $G^*[\pi]$ and the color of vertex v in the graph $G[\pi]$. Moreover, we compute the average-case performance of our algorithm by estimating the parameter k which appears in its time and processor complexity. We show that the problem of coloring a permutation graph of size n can be solved in $O(\log n \log\log n)$ time in the average-case on the CREW PRAM model of computation with $O(n^2)$ processors.

The paper is organised as follows: In Section 2, we establish the notation and terminology we shall use throughout the paper. In Section 3, we describe a method that transforms a given permutation π into an acyclic directed graph $G^*[\pi]$, and we show that coloring a permutation graph $G[\pi]$ is equivalent to finding the length of longest paths on its acyclic directed graph $G^*[\pi]$. In Section 4, we show the way we can efficiently construct the graph $G^*[\pi]$. In Section 5, we propose two parallel algorithms for solving the longest path problem on the acyclic directed graph $G^*[\pi]$. Section 6 addresses the average-case behaviour of our coloring algorithms, while Section 7 concludes the paper.

The model of parallel computation used in this paper is the well-known Concurrent-Read, Exclusive-Write PRAM model (CREW PRAM) [7, 11].

2. Definitions

A *coloring* of a graph is an assignment of colors to its vertices so that no two adjacent vertices have the same color. The set of all vertices with any one color is independent and is called a *color class*. The *coloring problem* is to color a graph with the smallest possible (i.e., minimum) number of colors (see Jensen and Toft [8]).

Permutations may be represented in many ways. The most straightforward is simply a rearrangement of the numbers 1 through n, as in the following example where $n = 9$.

index	1	2	3	4	5	6	7	8	9
permutation	8	3	2	7	1	9	6	5	4

Suppose π is a permutation of length n. Let us think of π as the sequence $[\pi_1, \pi_2, ..., \pi_n]$, so, for example, the permutation $\pi = [8, 3, 2, 7, 1, 9, 6, 5, 4]$ has $\pi_1 = 8$, $\pi_2 = 3$, etc. Notice that $(\pi^{-1})_i$, denoted here as $\pi^{-1}(i)$, is the position of element i in the sequence π. In our example $\pi^{-1}(8) = 1$, $\pi^{-1}(3) = 2$, etc.

Let π be a permutation on $N = \{1, 2, ..., n\}$. An element i of the sequence π is said to be *dominated* by the element j (or j *dominates* i) if $i < j$ and $\pi^{-1}(i) > \pi^{-1}(j)$. An element i is said to be *directly dominated* by the element j (or j *directly dominates* i) if i is *dominated* by j and there exists no element k in π such that i is dominated by k and k is dominated by j. In the permutation given above, elements 1, 6, 5, 4 are dominated by 7 and elements 1, 6 are directly dominated by 7. We shall use, hereafter, the notation *D-dominates* and *D-dominated* for the terms "directly dominates" and "directly dominated", respectively. The *domination set* (*D-domination set*) of an element i of the permutation π is the set which contains all the elements of π that dominate (D-dominate) i.

Given a permutation π of length n, its corresponding permutation graph $G[\pi] = (V, E)$ can be constructed as follows: $G[\pi]$ has vertices numbered from 1 to n; two vertices are joined by an edge if the larger of their corresponding numbers is to the left of the smaller in permutation π—that is, $G[\pi]$ has n vertices $v_1, v_2, ..., v_n$, and m edges such that $(i, j) \in E$ if and only if $(i - j)\,(\pi^{-1}(i) - \pi^{-1}(j)) < 0$.

Let $G[\pi] = (V, E)$ be a permutation graph. A vertex $v_j \in V$ is said to be a neighbour (D-neighbour) of vertex v_i if j dominates (D-dominates) i in π. For example, in the graph $G[\pi]$ of Figure 1 vertices v_8, v_7, v_9, v_6 are neighbours of v_5, while vertex v_6 is D-neighbour of v_5.

We conclude this section with some graph-theoretic notation employed in this paper. A *path* P in a graph $G = (V, E)$ is a sequence of vertices $[v_0, v_1, ..., v_k]$ such that $(v_{i-1}, v_i) \in E$, $i = 1, 2, ..., k$; P is a path from v_0 to v_k of *length* k; P is directed or undirected depending on whether G is directed or undirected graph. The path P is a *simple path* if $v_0, ..., v_{k-1}$ are distinct and $v_1, ..., v_k$ are distinct and all edges on P are distinct. A simple path $[v_0, v_1, ..., v_k]$ is a *cycle* if $v_0 = v_k$; otherwise it is *noncyclic*. A *directed acyclic graph* (dag) is a digraph with no cycles.

3. Problem Transformation and Solution

We have referred to the problem of coloring a graph as one of trying to assign particular colors to its vertices so that no two adjacent vertices have the same color. Moreover, the number of colors used must be as small as possible. The key to the solution is to find the color classes of the graph—that is, the classes of vertices that can be colored with the same color. To this end, one can think of transforming the graph into another combinatorial object (e.g., tree, directed graph, etc.) and, then, solving a particular problem on this object (e.g., vertices lying in the same level of the tree, vertices having the same distance from a particular vertex, etc.) which gives the solution to the coloring problem.

In this work, we use a strategy to transform a permutation graph $G[\pi]$ into a directed acyclic graph $G^*[\pi]$ using certain combinatorial properties of π. Then, we solve the coloring problem on $G[\pi]$ by solving the longest path problem on $G^*[\pi]$. In particular, given a permutation π (or its corresponding graph), we construct the acyclic directed graph $G^*[\pi]$ by exploiting the D-domination relation, as follows:

(i) for every element i in permutation π, add the vertex v_i to $G^*[\pi]$—that is, $v_i \in V^*$;

(ii) compute the D-domination set of each element i in π;

(iii) if p is D-dominated by i then add the edge $<v_i, v_p>$ to $G^*[\pi]$—that is, $<v_i, v_p> \in E^*$;

(iv) add a dummy vertex s in V^* and set $<s, v_i> \in E^*$ for every v_i with $indegree(v_i) = 0$;

Figure 1 shows a permutation $\pi = [8, 3, 2, 7, 1, 9, 6, 5, 4]$ and its corresponding permutation graph $G[\pi]$, while Figure 2 shows the result of transforming the permutation graph $G[\pi]$ into a directed acyclic digraph using the transformation strategy described above. The resulting graph is the directed acyclic digraph $G^*[\pi]$.

index	1	2	3	4	5	6	7	8	9
permutation	8	3	2	7	1	9	6	5	4

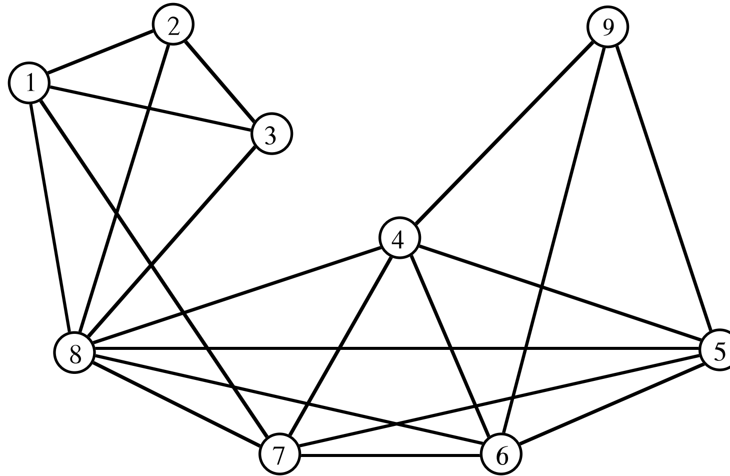

Fig. 1: A permutation $\pi = [8, 3, 2, 7, 1, 9, 6, 5, 4]$, and its corresponding permutation graph $G[\pi]$.

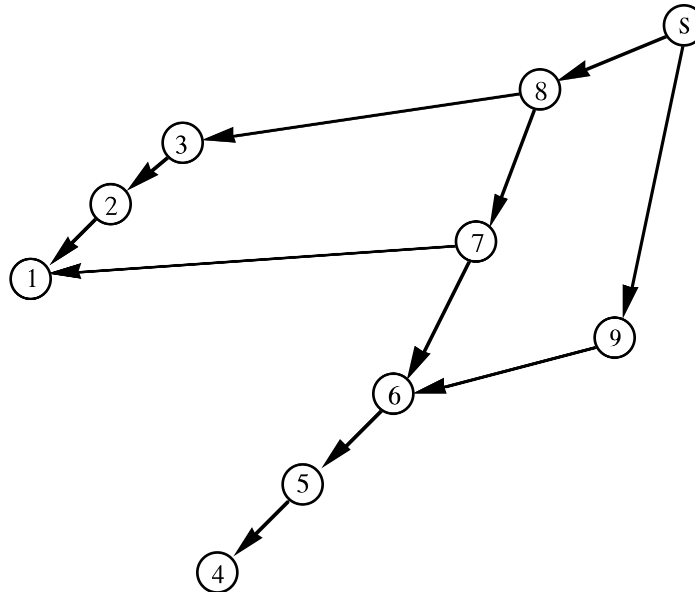

Fig. 2: The directed acyclic graph $G^*[\pi]$ obtained from permutation π.

We next show that there is an one-to-one correspondence between the length of the longest path from s to a vertex v in $G^*[\pi]$ and the color of vertex v in $G[\pi]$. To this end, we include here some results of Golumbic [5; Corollary 7.4].

Lemma 3.1 Let π be a permutation of length n. The following numbers are equal:

(i) the chromatic number of $G[\pi]$;

(ii) the length of a longest decreasing subsequence of π;

Moreover, it has been shown that the decreasing (resp. increasing) subsequences of π and the cliques (resp. independent sets) of $G[\pi]$ are in one-to-one correspondence [5].

Based on these results we can easily show that the problem of coloring a permutation graph $G[\pi]$ can be solved by computing the length $lp(v_i)$ of the longest path from s to v_i in digraph $G^*[\pi]$, $1 \le i \le n$. It is also easy to see that the chromatic number of $G[\pi]$ equals to $max\{lp(v_i) \mid outdegree(v_i) = 0$ and $1 \le i \le n\}$.

Having shown the relation between the coloring problem on a permutation graph $G[\pi]$ and the longest path problem on $G^*[\pi]$, we are in a position to formulate a parallel algorithm for solving the coloring problem on permutation graphs. It consists of three steps:

Algorithm `Coloring`:

Input : A permutation π and its corresponding graph $G[\pi]=(V, E)$.

Output : The color of each vertex $v_i \in V$, $i=1, 2, ..., n$.

begin

1. Transform the permutation graph $G[\pi]=(V, E)$ into a dag graph $G^*[\pi]=(V^*, E^*)$ as follows:
   1.1. $V^* \leftarrow V$;
   1.2. Edge $<v_p, v_i> \in E^*$ if and only if p *D-dominates* i;
   1.3. Add a dummy vertex s in V^* and set: $<s, v_i> \in E^*$ for every v_i with $indegree(v_i) = 0$;
2. For each vertex $v_i \in V^* - \{s\}$ compute the length $lp(v_i)$ of the longest path from s to v_i;
3. Set $color(v_i) \leftarrow lp(v_i)$, $i=1, 2, ..., n$.

end.

In step 3, the algorithm colors the vertices of graph G with χ colors, where χ is the chromatic number of $G[\pi]$ or, equivalently, the maximum length between all the lengths of longest paths in $G^*[\pi]$, where $\chi \le n$. Vertices v_i and v_j are colored with the same color if and only if the longest paths from s to v_i and from s to v_j have the same length. The correctness of the algorithm is established through Theorem 3.1. Its proof relies on the way we construct the graph $G^*[\pi]$ and the results of [5].

Theorem 3.1 Given a permutation π of length n, the algorithm `Coloring` correctly solves the coloring problem on its corresponding permutation graph.

4. Construction of the Directed Acyclic Graph $G^*[\pi]$

As we saw previously, the directed acyclic graph $G^*[\pi]$ is constructed by exploiting the *D-domination* relation on permutation π. Therefore, there is a need for computing the D-domination set for every element in π. It is obvious that the D-domination set of an element is a subset of its domination set. So,

we can first compute the domination set for every element i in permutation π, and then select from it the elements that constitute the D-domination set.

We can easily show that the domination set of an element, say i, of a permutation π, is simply the set which contains all the elements that are greater than i and lying on the left of the element i in π (see the definition of the domination set in Section 2). Towards the computation of the domination set of the element i, we define two arrays D'_i and D_i of lengths $n+1$, where n is the number of vertices in the permutation graph $G[\pi]$; that is,

D'_i array of length $n+1$; it contains all the elements of π that dominate i; that is, $D'_i\,[\pi^{-1}(j)] = j$ if $i < j$ and $\pi^{-1}(i) > \pi^{-1}(j)$; otherwise $D'_i\,[\pi^{-1}(j)] = 0$, $1 \leq j \leq n$. We set $D'_i[0] = s$, where s denotes here the number $n + 1$.

D_i array of length $n + 1$; it contains the elements of D'_i in consecutive memory locations.

The computation of each array D_i, $1 \leq i \leq n$, can be done independently, and therefore in parallel. Obviously, the computation of D_i can be done in $O(\log n)$ time using $n / \log n$ processors on the EREW PRAM model. The problem of storing all the non-zero elements of an array of length n into consecutive memory locations is equivalent to the problem of computing the prefix-sums. It is well-known that the prefix-sums of n elements can be computed in $O(\log n)$ time using $n / \log n$ processors on the EREW PRAM model.

In order to compute all the arrays D_i, $1 \leq i \leq n$, in $O(\log n)$ time on the EREW PRAM model, there is a need for copying in the shared-memory n times the n elements of π. This computation can be done in $O(\log n)$ using $n^2 / \log n$ processors on such a model. Therefore, the computation of each D_i, $1 \leq i \leq n$, can be done in $O(\log n)$ using $n^2 / \log n$ processors on the EREW PRAM model.

Hereafter, we shall refer to the parallel algorithm which computes the domination set for every element of permutation π as `Domination` algorithm. Thus, the following theorem holds.

Theorem 4.1 Given a permutation π of length n, the algorithm `Domination` computes the dominated set of each element of π in $O(\log n)$ time using $n^2 / \log n$ processors on the EREW PRAM model of computation.

Remark 4.1 Recall that, j is an element of D'_i if $i < j$ and $\pi^{-1}(i) > \pi^{-1}(j)$. It is obvious that the elements of π that dominate i are at most n. Actually, these elements are at most δ, where δ is the degree of the permutation graph $G[\pi]$.

Having computed the domination set of each element i of π, let us now show the way we can compute the D-domination set of the element i, $1 \leq i \leq n$. Datta *et. al.* [3] have presented a parallel algorithm for the D-domination problem which runs in $O(\log n)$ time with $O(n + k / \log n)$ processors on the CREW PRAM model. The value of k can be as large as $\Omega(n^2)$. Here, we present a simple parallel algorithm for the same problem which runs in $O(\log n)$ time using $n^2 / \log n$ processors on a less powerful model of computation—that is, the EREW PRAM model.

Based on the definition of the D-domination relation of two elements of a permutation π, we can easily show that the last element of the array D_i, say j, directly dominates the elements i, $1 \leq i \leq n$. Moreover, the D-domination set contains the element j and every other element z of the array D_i such that $z < j$ and there is no element z' smaller than both z and j—that is, $z' < z$ and $z' < j$, and $\pi^{-1}(z) < \pi^{-1}(z')$ $< \pi^{-1}(j)$. Therefore, it is easy to see that we can compute the D-domination set of the element i, say DD_i, by computing the suffix minima of the array D_i, $1 \leq i \leq n$. Recall that $D_i\,[0] = s$, for every $i = 1, 2, ..., n$; the symbol s denotes the number $n+1$.

Next, we list a parallel algorithm that computes the array DD_i of length n+1. Recall that the array DD_i contains all the elements of D_i that directly dominate i, $1 \le i \le n$.

Algorithm `D-domination`

Input : The set D_i of all the elements which dominate i, $1 \le i \le n$;

Output : The set DD_i of all the elements which D-dominate i, $1 \le i \le n$;

begin

1. Set $Y_i \leftarrow D_i$, $1 \le i \le n$;
2. If Y_i is in increasing order, then go to step 6;
3. Compute the suffix minima of Y_i (see [7]);
4. Set $DD_i \leftarrow Y_i$, $1 \le i \le n$;
5. For every i, $1 \le i \le n$, do in parallel
   for every j, $0 \le j \le n$-1, do in parallel
   if $Y_i[j] = Y_i[j+1]$ then $DD_i[j] \leftarrow 0$;
6. Store all the non-zero elements from array DD_i, $1 \le i \le n$, into consecutive memory locations (see Step 2 of the algorithm `Domination`);

end.

Let us now show step-by-step the computation of the D-domination set of an element of the permutation $\pi = [8, 3, 2, 7, 1, 9, 6, 5, 4]$ using the algorithm `D-domination`; we choose the element 1. This computation goes as follows: Steps 1 & 2: $Y_1 \leftarrow D_1 = [s, 8, 3, 2, 7]$; Step 3: $Y_1 \leftarrow [2, 2, 2, 2, 7]$; Step 4: $DD_1 \leftarrow [2, 2, 2, 2, 7]$; Step 5: $DD_1 \leftarrow [0, 0, 0, 2, 7]$; Step 6: $DD_1 \leftarrow [2, 7]$.

The correctness of the algorithm is based on the previous discussion and is established through the following lemma.

Lemma 4.1 Algorithm `D-domination` correctly computes the D-domination set for each element of a permutation π of length n.

Let us now analyse the computational complexity of the algorithm `D-domination` using the EREW PRAM model. We shall obtain the overall complexity by computing the complexity of each step separately.

The algorithm consists of 5 steps: *Step 1*: The assignment operation on n^2 elements takes $O(1)$ time by using n^2 processors or $O(\log n)$ time by using $n^2 / \log n$ processors. *Step 2*: The operation of testing whether a sequence of n elements is in increasing order can be executed in $O(\log n)$ time with $n / \log n$ processors. Therefore, this step is executed in $O(\log n)$ time with $n^2 / \log n$ processors. *Step 3*: This step computes the suffix minima of a sequence of length n. It is well-known that the range-minima problem on a sequence of n elements can be solved in $O(\log n)$ time using $n / \log n$ processors on the EREW PRAM model [7, 11]. Thus, step 3 can be executed in $O(\log n)$ time with $n^2 / \log n$ processors. *Step 4*: This step has the same time and processor complexity as step 1. *Step 5*: Obviously, this step can be executed in $O(\log n)$ time with $n^2 / \log n$ processors. *Step 6*: The problem of storing all the non-zero elements of an array of length n into consecutive memory locations can be computed in $O(\log n)$ time using $n / \log n$ processors (see Step 2 of algorithm `Domination`).

Taking into consideration the time and processor complexity of each step of the algorithm, as well as the complexity of copying the dominated set D_i of each element of π, we conclude that the algorithm `D-domination` can be executed in $O(\log n)$ time using $n^2 / \log n$ processors.

The algorithm D-domination takes as input the dominated set D_i of each element of π. We have shown that D_i can be computed by algorithm Domination in $O(\log n)$ time using $n^2 / \log n$ processors (see Theorem 4.1). Thus, we obtain the following theorem.

Theorem 4.2 Given a permutation π of length n, the algorithm D-domination computes the direct dominated set of each element of π in $O(\log n)$ time using $n^2 / \log n$ processors on the EREW PRAM model of computation.

The algorithms Domination and D-domination actually compute the matrices D and DD, respectively, which both are of length $n \times n+1$. We shall refer to theses matrices as *domination matrix* and *D-domination matrix*, respectively. Figure 3 shows the matrices D and DD for the sample permutation π of length 9. Recall that, we denote by s the number $n+1$.

D :

	0	1	2	3	4	5	6	7	8	9
1	s	8	3	2	7					
2	s	8	3							
3	s	8								
4	s	8	7	9	6	5				
5	s	8	7	9	6					
6	s	8	7	9						
7	s	8								
8	s									
9	s									

DD :

	0	1	2	3	4	5	6	7	8	9
1	2	7								
2	3									
3	8									
4	5									
5	6									
6	7	9								
7	8									
8	s									
9	s									

Fig. 3: The matrices D and DD of $\pi = [8, 3, 2, 7, 1, 9, 6, 5, 4]$.

We have already shown the way we can construct the directed acyclic graph $G^*[\pi]$ using the D-domination relation of π. Next, we give a more formal listing of the parallel algorithm for constructing the directed graph $G^*[\pi]$.

Algorithm D-domination-to-dag:

Input : The set DD_i of all the points which D-dominates i, $1 \leq i \leq n$;

Output : The directed acyclic graph $G^*[\pi]$;

begin

1. Construct the dag $G^*[\pi] = (V^*, E^*)$, with vertex set $V^* = \{v_1, v_2, ..., v_n\}$, as follows:
   For every $i = 1, 2, ..., n$, do in parallel
   for every $j = 1, 2, ..., n$, do in parallel
   if $DD_i [j] = p > 0$ then add an arc from vertex v_p to vertex v_i, i.e., $<v_p, v_i> \in E^*$;
2. Add a vertex s in $G^* = (V^*, E^*)$ and arcs $<s, v_i> \in E^*$ for every vertex v_i with *indegree*$(v_i) = 0$;
   That is, vertex s is now the only vertex in G^* whose indegree equals 0;

end.

We can easily compute the time and processor complexity of the algorithm D-domination-to-dag on the CREW PRAM model. The algorithm consists of two steps: *Step 1*: It can be executed in $O(\log n)$ time using $n^2 / \log n$ processors. *Step 2*: To find a vertex in $G^*[\pi]$ with indegree zero is equivalent to find a row D_i of the $n \times n+1$ matrix D having all its entries equal 0. Therefore, this step can be executed

in $O(\log n)$ time using $n^2 / \log n$ processors.

From the above step-by-step analysis, we obtain that the algorithm `D-domination-to-dag` is executed in $O(\log n)$ time using $n^2 / \log n$ processors on the EREW PRAM model.

The algorithm `D-domination-to-dag` takes as input the D-domination matrix DD. We have shown that, given a permutation π, the D-domination matrix can be computed by the algorithm `D-domination` in $O(\log n)$ time using $n^2 / \log n$ processors. Therefore, we have the following result.

Theorem 4.3 Given a permutation π of length n, the algorithm `D-domination-to-dag` constructs the directed acyclic graph $G^*[\pi]$ in $O(\log n)$ time using $n^2 / \log n$ processors on the EREW PRAM model of computation.

5. The Main Results

Having defined the directed acyclic graph $G^*[\pi] = (V^*, E^*)$ of a given permutation π and shown the one-to-one correspondence between the coloring problem on $G[\pi]$ and the longest path problem on $G^*[\pi]$, we are in a position to formulate a parallel algorithm for computing the length of the longest paths from vertex $s \in V^*$ to every vertex $v \in V^*$.

5.1 Longest paths in $G^*[\pi]$

While it is normally most convenient to represent permutations as we have been doing—as rearrangements of the numbers 1 through n—many other ways to represent permutations are often appropriate. A useful one is the two-dimensional representation of a permutation: the permutation $\pi_1 \pi_2 \ldots \pi_n$ is represented by labeling the point at row i and column π_i with the number π_i for each i (or equivalently, it is represented by labeling the point at row $\pi^{-1}(i)$ and column i with the number i for each i), $1 \le i \le n$. We delete the node s and all the arrows of Figure 4; then the resulting figure shows the two-dimensional representation of the permutation $\pi = [8, 3, 2, 7, 1, 9, 6, 5, 4]$. We extend the notion of two-dimensional representation so that for any permutation π we define the two-dimensional representation of its directed acyclic graph $G^*[\pi] = (V^*, E^*)$. Figure 4 shows such a representation for the graph $G^*[\pi]$, where $\pi = [8, 3, 2, 7, 1, 9, 6, 5, 4]$.

Consider now the problem of finding the length of the longest path between every pair of vertices of the directed acyclic graph $G^*[\pi]$. Recall that the previous best-known parallel algorithms for the coloring problem require $O(\log^2 n)$ time and $O(n^3 / \log n)$ processors on the CREW PRAM model and mainly use transformation techniques [15, 17]. In this paper, we use a simple divide-and-conquer strategy to solve the same problem. It is well-known that the divide-and-conquer technique consists of three steps and its success depends on whether or not we can perform the first step (decomposition of the input graph into similar structured graphs of almost equal sizes) and the third step (combine or merge the solutions of the different subproblems into a solution for the overall problem) efficiently.

Let $\pi = [\pi_1, \pi_2, \ldots, \pi_n]$ be a permutation of length n and let $G^*[\pi]$ be the acyclic directed graph of $G[\pi]$. For simplicity, we assume that n is a power of 2. We partition the elements of π into two subsets $A = \{\pi_1, \pi_2, \ldots, \pi_{n/2}\}$ and $B = \{\pi_{(n/2)+1}, \pi_{(n/2)+2}, \ldots, \pi_n\}$ or, equivalently, we decompose the graph $G^*[\pi]$ into two subgraphs $G^*[\pi, A]$ and $G^*[\pi, B]$, where $G^*[\pi, A]$ (resp. $G^*[\pi, B]$) is the subgraph of G^* induced by A (resp. B). The parallel algorithm we develop in this section consists of recursively determining the longest path between every pair in $G^*[\pi, A]$ and $G^*[\pi, B]$, and then combining the solutions of the two subproblems to obtain the solution for the overall problem; that is, the longest path

between every pair in $G^*[\pi]$. Before proceeding, we define some key sets of edges and vertices of the graph $G^*[\pi]$, with respect to subsets A and B.

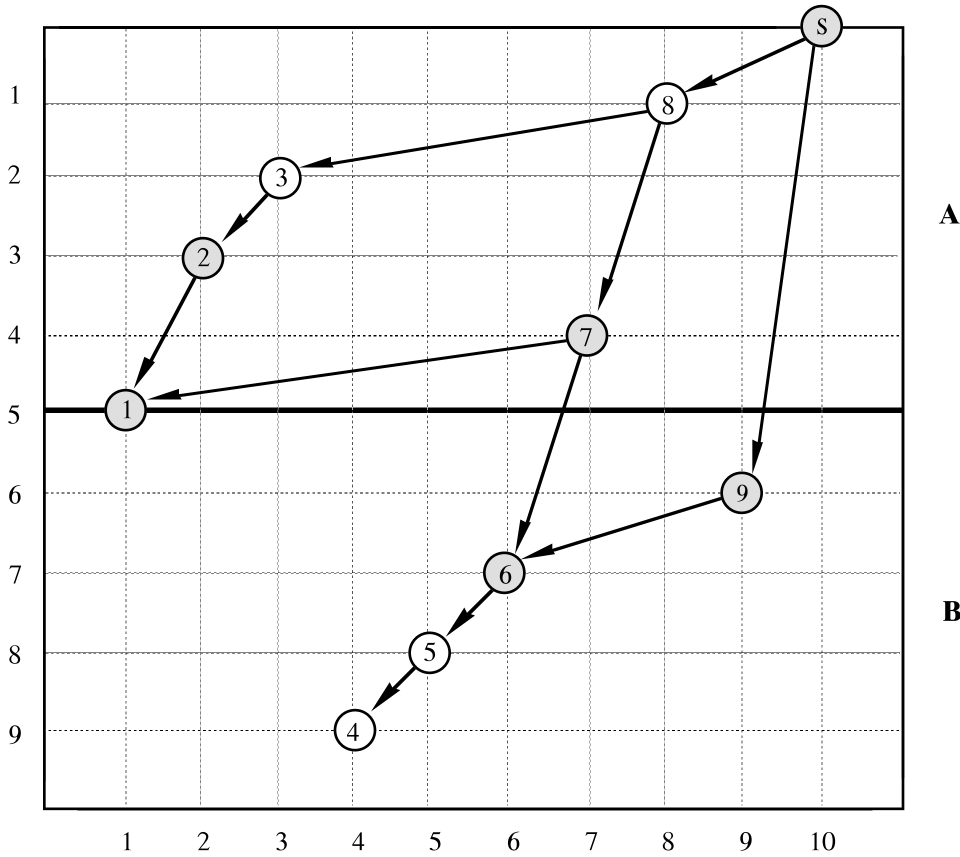

Fig. 4: The two-dimensional representation of the directed acyclic graph $G^*[\pi]$, where π = [8, 3, 2, 7, 1, 9, 6, 5, 4].

Definition 5.1: The *cut-edge set* $CE(A, B)$ of $G^*[\pi]$ is defined to be the set of all the edges <*u, v*> of $G^*[\pi]$ such that u is a vertex of $G^*[\pi, A]$ and v is a vertex of $G^*[\pi, B]$. An edge of the set $CE(A, B)$ is called cut-edge.

Definition 5.2: The *lower-vertex set* $LV(A)$ of the graph $G^*[\pi, A]$ is defined to be the set of all the vertices u of $G^*[\pi, A]$ which form a cut-edge in $G^*[\pi]$. Similarly, the *upper-vertex set* $UV(B)$ of the graph $G^*[\pi, B]$ is defined to be the set of all the vertices v of $G^*[\pi, B]$ which form a cut-edge in $G^*[\pi]$.

Let us illustrate with a help of an example. Consider the two-dimensional representation of the graph $G^*[\pi]$, where π = [8, 3, 2, 7, 1, 9, 6, 5, 4], and let A = {*s*, 8, 3, 2, 7} and B = {1, 9, 6, 5, 4}; see Figure 4. In this case, the cut-edge set $CE(A, B)$ is the set of edges {<2, 1>, <7, 1>, <7, 6>, <*s*, 9>}, while the lower-vertex and upper-vertex sets of the graph $G^*[\pi]$ are the sets $LV(A)$ = {2, 7, *s*} and $UV(B)$ = {1, 6, 9}, respectively.

In addition to the above, we define a global array L_i of length n+1 for each vertex v_i of the graph $G^*[\pi]$, where n+1 is the number of vertices in $G^*[\pi]$; that is,

L_i array of length n+1; it contains for each vertex v_j the length of a longest path from v_j to v_i, $1 \leq i \leq n$. Obviously, the length of a longest path from v_j to v_i is at most n or, equivalently, the elements which dominate the element i are at most n (at most n-1 elements plus the element s);

Hereafter, the terms "the vertex v_i" of the graph $G^*[\pi]$ and "the element i" of the permutation π will be used equivalently.

We proceed now to formulate a parallel algorithm that computes the longest path between every pair of vertices of the directed acyclic graph $G^*[\pi]$. The algorithm takes as input the graphs $G^*[\pi, A]$ and $G^*[\pi, B]$, and the array L_i of length n+1 for every vertex v_i of $G^*[\pi]$. Initially, $L_i[j]$ equals the length of the longest path from v_j to v_i if (i) such a path exists and (ii) both vertices v_j, v_i belong to either $G^*[\pi, A]$ or $G^*[\pi, B]$; otherwise, $L_i[j]$ equals zero. The algorithm, which we call `M-Longest-Paths`, consists of the following three main steps:

1. Find the lower-vertex set $LV(A)$ of $G^*[\pi, A]$ and the upper-vertex set $UV(B)$ of $G^*[\pi, B]$;

2. Update L_i for every vertex $v_i \in UV(B)$ using information of the set $LV(A)$; Now, $L_i[j]$ equals the length of the longest path from v_j to v_i if such a path exists and vertex v_j belongs to $A \cup UV(B)$;

3. Update L_i for every vertex $v_i \in B$ using information of the set $UV(B)$; Now, $L_i[j]$ equals the length of the longest path from v_j to v_i if such a path exists and vertex v_j belongs to $A \cup B$;

In order to illustrate the workings of the algorithm `M-Longest-Paths`, we present with a help of an example the steps of the algorithm on three vertices of $G^*[\pi, B]$; we choose the vertices v_6 and v_9 of the set $UV(B)$ and the vertex v_4 of the set $B - UV(B)$. Figure 5 shows the arrays L_6 and L_9 after the execution of steps 1 and 2, as well as the array L_4 after the execution of steps 1 and 3. Moreover, the same figure shows the arrays $LL_{4,6}$ and $LL_{4,6}$ which are used for the computation of the array L_4 in step 3. The array $LL_{4,6}$ (resp. $LL_{4,9}$) contains the length of the longest path from $v_j \in A$ to v_4 passing through vertex v_6 (reps. v_9). The same figure also shows the initialization (step 1) of the array L_7; that is, the length of the longest paths from s to v_7 and from v_8 to v_7.

We next give a more formal listing of the algorithm `M-Longest-Paths` which is based on a simple divide-and-conquer strategy.

Algorithm `M-Longest-Paths`

Input : The all pairs longest paths in graphs $G^*[\pi, A]$ and $G^*[\pi, B]$;

Output : The all pairs longest paths in graph $G^*[\pi]$;

begin

1. Find the lower-vertex set $LV(A)$ of $G^*[\pi, A]$ and the upper-vertex set $UV(B)$ of $G^*[\pi, B]$;
2. For every vertex $v_i \in UV(B)$ do in parallel
   - 2.1 find the vertices $v_j \in LV(A) \cap DD_i$—that is, the vertices v_j such that $<v_j, v_i> \in E^*$;
   - 2.2 compute the array $L_i[1..n]$ as follows:
     $L_i[q] \leftarrow 1 + max_{j \in LV(A) \cap DDi} (L_j[q])$, for every $p \in A \cap D_i$;
     Initially, $L_i[i] \leftarrow 0$ and $L_i[q] \leftarrow 0$ for every q such that there is no directed path from q to i in graph $G^*[\pi, B]$;
3. For every vertex $v_i \in B$ do in parallel
   - 3.1 find the vertices $v_k \in UV(B) \cap D_i$—that is, the vertices v_k such that there exists a directed path from $v_k \in UV(B)$ to v_i in graph $G^*[\pi, B]$;
   - 3.2 for every $v_k \in UV(B) \cap D_i$ compute the array $LL_{i,k}[q]$ as follows:
     $LL_{i,k}[q] \leftarrow L_k[q] + L_i[k]$, for every $q \in D_k$;
   - 3.2 compute the array $L_i[1..n]$ as follows:
     $L_i[q] \leftarrow max\big(L_i[q], max_{k \in UV(B) \cap Di} (LL_{i,k}[q])\big)$, $1 \le q \le n$;

end.

We are now ready to show that the algorithm `M-Longest-Paths` works correctly. We prove the following theorem.

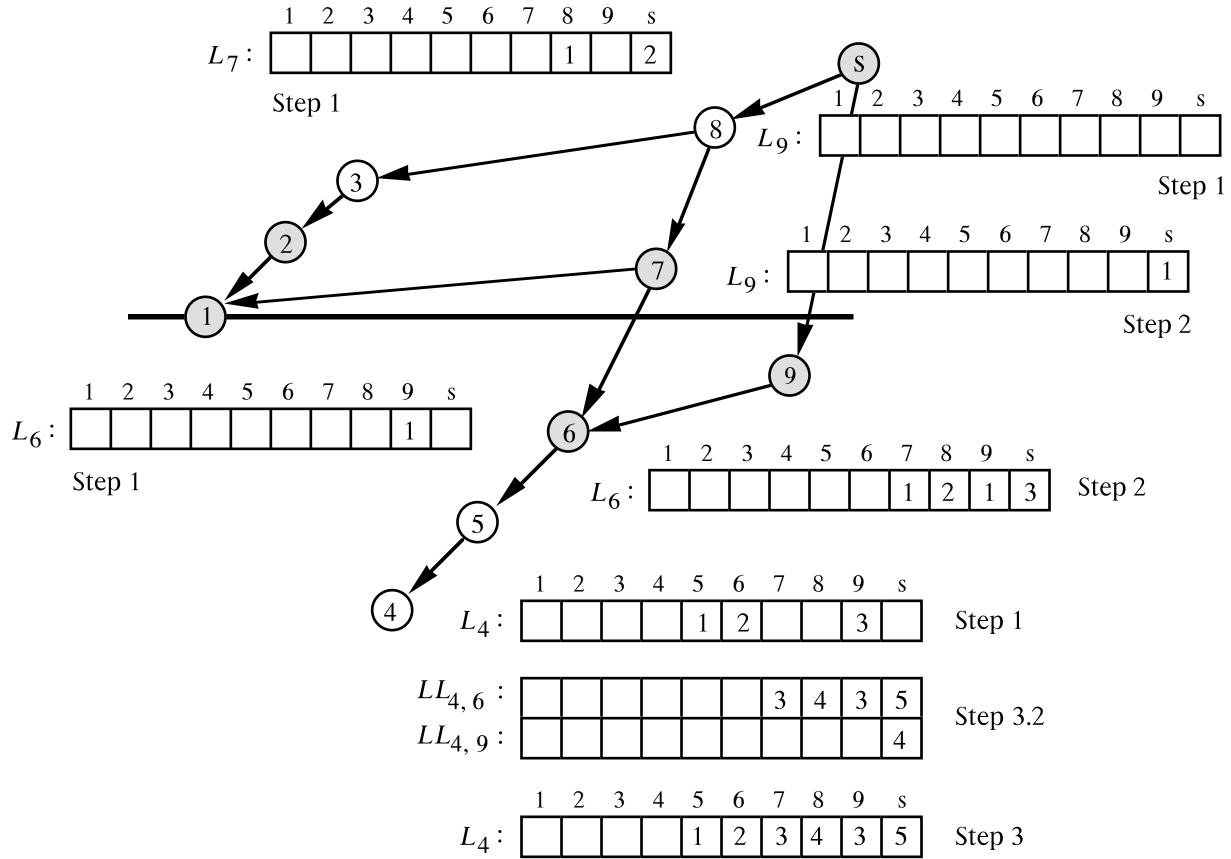

Fig. 5: Some intermediate results during the execution of the algorithm `M-Longest-Paths`.

Theorem 5.1 Given the directed acyclic graph $G^*[\pi]$ of a permutation π, and the length of the longest path between every pair of vertices in graphs $G^*[\pi, A]$ and $G^*[\pi, B]$, the algorithm `M-Longest-Paths` correctly computes the length of the longest path between every pair of vertices in $G^*[\pi]$.

Proof. Let v_i be a vertex of the set $UV(B)$ and let v_j be a vertex of the set A. Initially, the array L_i (resp. L_j) contains the length of the longest path $p = [u, ..., v_i]$ (resp. $p = [u, ..., v_j]$) for every $u \in B$ (resp. A).

By definition, the vertex v_i forms a cut-edge in $G^*[\pi]$—that is, there exists a vertex v_k in A such that $<v_k, v_i> \in E^*$. Therefore, it is easy to see that every path from v_j to v_i passes through a vertex $v_k \in LV(A)$. We consider all the paths from v_j to v_i of the form $p = [v_j, ..., v_k, v_i]$, where $v_k \in LV(A)$. Then, it is obvious that the length of the longest path from v_j to v_i passing through a vertex $v_k \in LV(A)$ and having the property $<v_k, v_i> \in E^*$ can be computed as follows: $L_i\,[j] = 1 + max\ \{L_k\,[j] \mid v_k \in LV(A)$ and path $p = [v_j, ..., v_k, v_i]$ exists$\}$. Thus, after the execution of the Step 2 of the algorithm the array L_i contains the length of the longest path from vertex $v_j \in A$ to vertex $v_i \in UV(B)$. We point out, here, that there is a case where this may not be true; consider the case where there exists a vertex $w \in UV(B)$ such that w D-dominates $v_i \in UV(B)$ and a path of the form $p = [v_j, ..., w, v_i]$ (see in Figure 5 the vertices v_6 and v_9). This ambiguity is cleaned up in the last step of the algorithm by considering not only the vertices in B-$UV(B)$ but also the vertices in $UV(B)$.

Let us now concentrate on the operations performed in Step 3. Let v_i be a vertex of the set B and let u_k be a vertex of the set $UV(B)$. Initially, the array L_i contains the length of the longest path

$p = [u, ..., v_i]$, for every $u \in B$. Our goal is to compute the length of the path $p = [v_j, ..., v_i]$, where $v_j \in A$. Again, it is easy to see that every path from v_j to v_i passes through a vertex $v_k \in UV(B)$—that is, it has the form $p = [v_j, ..., v_k, ..., v_i]$. Thus, we can compute the length of the longest path from v_j to v_i passing through a vertex $v_k \in UV(B)$ as follows: $L_i[j] = max\{L_i[k] + L_k[j] \mid v_k \in UV(B)$ and path $p = [v_j, ..., v_k, ..., v_i]$ exists$\}$. Thus, after the execution of the Step 3 of the algorithm the array L_i contains the length of the longest path from every vertex $v_j \in A$ to vertex $v_i \in B$. Thus, the theorem is proved. □

Let us now compute the complexity of the algorithm `M-Longest-Paths` using the CREW PRAM parallel model of computation. The time and processor complexity of the algorithm is analyzed as follows:

Step 1: The first step computes the arrays $LV(A)$ and $UV(B)$. We can test whether or not a vertex $u \in UV(B)$ as follows: if $DD_u[0] \in A$ and $u \in B$ then $u \in UV(B)$, $n/2 \leq u \leq n$. This computation can be done in $O(1)$ time with n processors on the EREW PRAM. In a similar way we can compute the vertex set $LV(A)$.

Step 2: This step consists of two substeps 2.1 and 2.2, which are executed for every vertex $v_i \in UV(B)$, $n/2 \leq i \leq n$. Hereafter, $k = max\{|LV(A)|, |UV(B)|\}$. *Substep 2.1.* Since DD_i is sorted in increasing order, we can compute the set $LV(A) \cap DD_i$ in $O(\log k)$ time with $n > |DD_i|$ processors on the CREW PRAM using sequential binary search. (Notice that, since DD_i is sorted in increasing order and $v_q \in LV(A) \cap DD_i$ for every $q < p$, we can also compute the set $LV(A) \cap DD_i$ in $O(1)$ time with $n > |DD_i|$ processors on the CREW PRAM model, in the case where $v_p \in LV(A) \cap DD_i$.) *Substep 2.2.* Obviously, the computation of the array $L_i[1..n]$ can be done in $O(\log k)$ time with $(k/\log k)n$ processors on the CREW PRAM model. Therefore, the whole step is executed in $O(\log k)$ time with $O(k\, n^2 / \log k)$ processors.

Step 3: It consists of three substeps which are executed for every vertex $v_i \in B$, $n/2 \leq i \leq n$. *Substep 3.1.* This substep is executed in $O(\log k)$ time with $n > |D_i|$ processors (see complexity of substep 2.1). *Substep 3.2.* It is easy to see that it can be executed in $O(1)$ time with kn processors. *Substep 3.3.* The complexity of this substep is the same as that of substep 2.2; that is, it takes $O(\log k)$ time with $O(kn/\log k)$ processors on the CREW PRAM model. Therefore, step 3 takes $O(\log k)$ parallel time using $O(k\, n^2 / \log k)$ processors.

Taking into consideration the time and processor complexity of each step of the algorithm, we can state the following result.

Theorem 5.2 Given the directed acyclic graph $G^*[\pi]$ of a permutation π and the length of the longest path between each pair of vertices of the graphs $G^*[\pi, A]$ and $G^*[\pi, B]$, the algorithm `M-Longest-Paths` computes the length of the longest path between each pair of vertices of the graph $G^*[\pi]$ in $O(\log k)$ time using $O(kn^2 / \log k)$ processors on the CREW PRAM model of computation, where $1<k<n$.

Remark 5.1 Combining the divide-and-conquer strategy with algorithm `M-Longest-Paths`, we obtain that the length of the longest path between each pair of vertices in graph $G^*[\pi]$ can be computed in $O(\log n \log k)$ time using $O(kn^2 / \log k)$ processors on the CREW PRAM model, where $1<k<n$.

5.2 Longest paths from a specific vertex in $G^*[\pi]$

Consider again the problem of finding the length of the longest path between every pair of vertices in the directed acyclic graph $G^*[\pi]$. The parallel algorithm `M-Longest-Paths` we developed so far takes $O(\log n \log k)$ time and uses $O(kn^2 / \log k)$ processors, where $1<k<n$. Thus, its cost and therefore its efficiency is better than that of the previously proposed algorithms for the same problem [15, 17]. However, it is well-known that in some cases we can decrease the number of processors of an algorithm

by performing a reallocation of the computation. In this section, we modify the algorithm `M-Longest-Paths` so that we achieve better efficiency by reducing the number of processors without increasing the time complexity.

Before we address the reallocation of the computation of the algorithm `M-Longest-Paths`, let us first recall the main result of Section 3: the color of a vertex u of a permutation graph $G[\pi]$ is equal to the length of the longest path from s to u in graph $G^*[\pi]$. Based on this result, it is obvious that instead of solving the all pair longest path problem in $G^*[\pi]$, as we do with algorithm `M-Longest-Paths`, we can solve the single-source longest path problem which is obviously computationally easier.

In the remainder of this section, we present a parallel algorithm which computes the length of the longest paths from the vertex s to every other vertex v_i in graph $G^*[\pi]$. Again, for simplicity, we assume that n is a power of 2.

The overall strategy for solving the problem of finding the length of the longest paths from vertex s to every other vertex v_i in graph $G^*[\pi]$ is outlined next.

1. Partition the graph $G^*[\pi]$ into $\log n$ subgraphs $G^*[\pi, A_1]$, $G^*[\pi, A_2]$, ..., $G^*[\pi, A_{\log n}]$;

2. Find the length of the longest path between every pair of vertices in each one of the graphs $G^*[\pi, A_1]$, $G^*[\pi, A_2]$, ..., $G^*[\pi, A_{\log n}]$ using the algorithm `M-Longest-Paths`; Moreover, compute the length of the longest paths from vertex s to every other vertex $u \in A_i$, $1 \le i \le n$. Obviously, if $lp(u)$ indicates the length of the longest path from s to u, then $lp(u) = 0$ for every $u \in A_i$, $2 \le i \le n$;

3. Find the length $lp(u)$ of the longest path from s to every vertex $u \in A_i$, $2 \le i \le n$, by applying the algorithm `M-Longest-Paths` on the graphs $G^*[\pi, A_1 \cup ... \cup A_{i-1}]$ and $G^*[\pi, A_i]$;

Next, we state more formally the algorithm which is based in the above strategy. We call it `S-Longest-Paths`.

Algorithm `S-Longest-Paths`

Input : The directed acyclic graph $G^*[\pi]$;

Output : The lengths of the longest paths from s to every vertex v_i in $G^*[\pi]$, $1 \le i \le n$;

begin

1. Partition the graph $G^*[\pi]$ into $\log n$ subgraphs $G^*[\pi, A_1]$, $G^*[\pi, A_2]$, ..., $G^*[\pi, A_{\log n}]$;
2. For $i \leftarrow 1, 2, ..., \log n$, do
   - 2.1 compute the array $L_u[1..n]$ for every $u \in A_i$—that is, the length of all pair longest paths in subgraph $G^*[\pi, A_i]$;
   - 2.2 set $lp(u) \leftarrow L_i[s]$, for every $u \in A_i$;
3. For $i \leftarrow 2, 3, ..., \log n$, do
   - 3.1 compute the vertex sets $UV(A_i)$ and $LV(A_1 \cup ... \cup A_{i-1})$;
   - 3.2 for every vertex $v_k \in UV(A_i)$ do in parallel
     - 3.2.1 for every vertex $v_j \in LV(A_{i-1}) \cap DD_k$ do in parallel
       $lp(v_k) \leftarrow 1 + max_{j \in LV(Ai-1) \cap DDk} (L_j[s]))$;
   - 3.3 for every vertex $v_k \in A_i$ do in parallel
     - 3.3.1 find the vertices $v_j \in UV(A_i) \cap D_k$—that is, the vertices v_j such that there is a directed path from $v_j \in UV(A_i)$ to v_k in $G^*[\pi, A_i]$;
     - 3.3.2 $lp(v_k) \leftarrow max_{j \in UV(Ai) \cap Dk} (L_k[j] + L_j[s])$;

end.

The correctness of the algorithm `S-Longest-Paths` is established through Theorem 5.1; it proves the correctness of the algorithm `M-Longest-Paths`.

Let us now compute the time and processor complexity of the algorithm `S-Longest-Paths` using the CREW PRAM model of computation. We shall obtain the overall complexity by computing the complexity of each step separately. The algorithm consists of three steps.

Step 1: We can partitioned the graph $G^*[\pi]$ into $\log n$ subgraphs $G^*[\pi, A_i]$ in $O(1)$ time using $\log n$ processors, $1 \le i \le \log n$. Obviously, each of the $\log n$ subgraphs $G^*[\pi, A_i]$ contains $n / \log n$ vertices.

Step 2: It consists of two substeps 2.1 and 2.2, which are executed $\log n$ times and compute the length of the longest path between every pair of vertices in subgraphs $G^*[\pi, A_i]$, $1 \le i \le \log n$ (the second substep also computes the length of the longest paths from s to every other vertex in $G^*[\pi, A_i]$). *Substep 2.1*. The array $L_u[1..n]$ can be computed using the algorithm `M-Longest-Paths` for every $u \in A_i$. Thus, substep 2.2 is executed in $O(\log k)$ time using $O(k\, n^2 / \log k \log^2 n)$ processors, where $k < n$. *Substep 2.2*. It is easy to see that it is executed in $O(1)$ time with $n / \log n$ processors. Therefore, the whole step can be executed in $O(\log n \log k)$ time with $O(kn^2 / \log k \log^2 n)$ processors on the CREW PRAM model.

Step 3: The sequential for-loop is repeated $\log n$-1 times. The substep 3.1 is executed in $O(1)$ time with n processors (see complexity of substep 2.1 of algorithm `M-Longest-Paths`). Having computed the complexity of algorithm `M-Longest-Paths`, we can easily see that the substeps 3.2 and 3.3 are executed in $O(\log k)$ time with $O(k\, n / \log k \log n)$ processors. (Recall that by algorithm `S-Longest-Paths` we compute the length of the longest paths from s to v_i, $1 \le i \le n$. That is the reason why the number of processors is not $O(kn^2 / \log k \log^2 n)$, $k < n$.) Thus, step 3 is executed in $O(\log n \log k)$ time using $O(kn / \log k \log n)$ processors on the CREW PRAM model of computation.

Taking into consideration the time and processor complexity of each step of the algorithm, we can state the main result of this work.

Theorem 5.3 Given the directed acyclic graph $G^*[\pi]$ of a permutation π, the length of the longest paths from vertex s to every other vertex can be computed in $O(\log n \log k)$ time using $O(kn^2 / \log k \log^2 n)$ processors on the CREW PRAM model of computation, where $1<k<n$.

5.3 The Coloring algorithm

The algorithm `Coloring`, which we have presented and proved its correctness in Section 2, incorporates all the operation described in the previous algorithms. In particular, all the operations of step 1 are executed in $O(\log n)$ time with $n^2 / \log n$ processors on the EREW PRAM model using the algorithm `D-domination-to-dag` (see Theorem 4.3), while all the operations of step 2 are executed in $O(\log n \log k)$ time with $O(kn^2 / \log k \log^2 n)$ processors on the CREW PRAM model using the algorithm `S-Longest-Paths` (see Theorem 5.3). Obviously, step 3 is executed in $O(\log n)$ time using $n / \log n$ processors on the EREW PRAM model.

Taking into consideration the time and processor complexity of the algorithms `D-domination-to-dag` and `S-Longest-Paths`, as well as the time and processor complexity of the step 3 of the algorithm `Coloring`, we present the following result.

Theorem 5.4 The coloring problem on permutation graphs of size n can be solved in $O(\log n \log k)$ time using $O(kn^2 / \log k \log^2 n)$ processors on the CREW PRAM model of computation, $1 < k < n$.

6. Average-Case Analysis

We have proposed two parallel algorithms for solving the longest path problem on acyclic digraphs or, equivalently, for solving the coloring problem on permutation graphs, and presented them in Sections

5.1 and 5.2, respectively. Our approach to the analysis of the proposed coloring algorithms is concentrated on determining their computational complexity—that is, the worst-case performance of the algorithms. There is also another well-known approach to the analysis of algorithms which concentrates on characterizing the performance of algorithms by determining their average-case or expected performance.

In this section, we consider the average-case analysis of our coloring algorithms—that is, we compute their time and processor complexities using random permutations of length n as an input model [12]. More precisely, we focus on estimating the parameter k in the case where the input is a random permutation graph. Notice that the time and processor complexities of our coloring algorithms have been computed as a function of n and k.

Definition 6.1: Let π be a permutation of length n. A left-to-right maximum is an index i with $\pi_j < \pi_i$ for all $j < i$. We use the notation $\lambda(\pi)$ to refer to the number of left-to-right maxima in a permutation π.

The first element in every permutation is a left-to-right maximum; if the largest element is the first, then it is the only left-to-right maximum. There are two left-to-right maxima in the sample permutation, at positions 1 and 6. Left-to-right minima as well as right-to-left maxima and minima are defined analogously.

Many properties of permutations have been studied and reported in the combinatorics literature (see [12]; Chapter 6). It has been shown that the number of left-to-right maxima in a permutation π is $\lambda(\pi) = \sim \ln n$ on the average. This average-case result is applied in any subsequence of a random permutation. We point out that the natural logarithm $\ln n$ and the binary logarithm $\log n$ are related by a constant factor, so we can refer to either as being $O(\log n)$ in the complexity analysis.

Based on the above results, we next estimate some parameters of the graph $G^*[\pi]$ and, consequently, we compute the average-case performance of our coloring algorithms. Let v_i be a vertex of $G^*[\pi]$, where π is a random permutation of length n. We construct a subsequence $\Pi(i)$ of π by the following rule: Remove from $\pi' = [\pi_{i+1}, \pi_{i+2}, ..., \pi_n]$ all the elements π_j which are greater than π_i, $1 \le i \le n$-1. In the sample permutation, $\Pi(7) = [1, 6, 5, 4]$. It is then easy to show that the number of left-to-right maxima in the permutation $\Pi(i)$ is equal to the outdegree of vertex v_i. Thus, $outdegree(v_i) = O(\log n)$ for every vertex v_i in $G^*[\pi]$. In a similar way we can show that $indegree(v_i) = O(\log n)$ for every vertex v_i in $G^*[\pi]$. Thus, we report the following result.

Lemma 6.1 Let $G^*[\pi]$ be the acyclic digraph of a random permutation π of length n. Then $\delta(G^*) = O(\log n)$, where $\delta(G^*)$ is the degree of $G^*[\pi]$.

Let π be a random permutation of length n and let i be an arbitrary integer in the range 2 to n-1. We shall determine the average number of vertices in the sets $LV(A_{i+1})$ and $UV(A_{i+1})$, $2 \le i \le n$-1. Hereafter, $A_i = [\pi_1, \pi_2, ..., \pi_i]$ and $A_{i+1} = [\pi_{i+1}, \pi_{i+2}, ..., \pi_n]$.

We compute the left-to-right maxima in A_i—say, $lr\text{-}max(A_i) = [x_1, x_2, ..., x_m]$. Since we consider here the average-case, we assume that every element x of $lr\text{-}max(A_i)$ is equally likely to be in any particular position in $A_i = [\pi_1, \pi_2, ..., \pi_i]$. Without loss of generality we suppose that $x_1, x_m \in CE(A_i, A_{i+1})$; that is, both x_1 and x_m form cut-edges with some elements of A_{i+1}. Then, every left-to-right maximum in A_i belongs to $LV(A_i)$ and every other element w in $LV(A_i)$ satisfies $\pi^{-1}(x_1) < \pi^{-1}(w) < \pi^{-1}(x_m)$.

We have seen that the number of elements of the set $lr\text{-}max(A_i)$ is $\lambda(A_i) = O(\log n)$ on the average. Moreover, we have assumed that all possible positions for the elements of $lr\text{-}max(A_i)$ are equally likely in A_i (the probability that an element x_p of $lr\text{-}max(A_i)$ belongs in any one specific position in A_i is $1/i$).

Then, it is not unreasonable to expect that the position of the element x_p in A_i to be close to $r\, i$ / log$n$, for r = 1, 2, ..., log$n$.

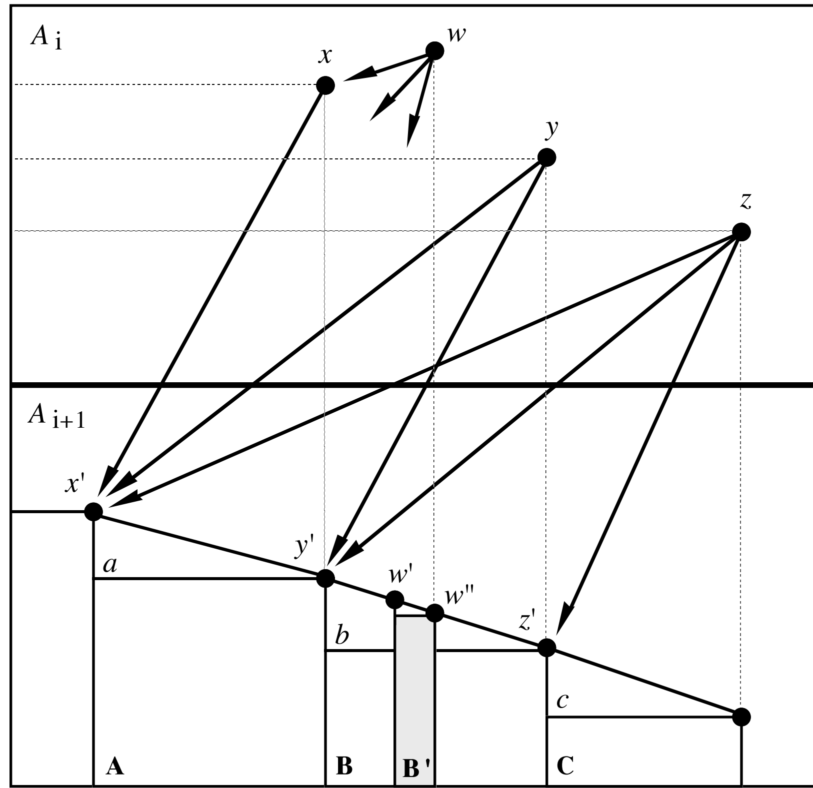

Fig. 6: The two-dimensional representation of a random permutation π and the positions of some elements of π which form cut-edges with respect to a partition.

Next, for convenience, we consider the first three element of *lr-max*(A_i)—that is, $x_1 = x$, $x_2 = y$ and $x_3 = z$ (see Figure 6). Let x', y', z' be three elements in A_{i+1} which form a cut-edge with the elements x, y, z respectively. Then, by Lemma 6.1, we have that the number of elements in A_{i+1} which form a cut-edge with x is log$n$; that is $N(x, A_{i+1})$ = log$n$. It is easy to see that the elements which form a cut-edge with x and belong to the area a (see Figure 6), also form a cut-edge with y. Therefore, the number of new elements in A_{i+1} (the elements in the area b), which form a cut-edge with y is log$n$ / 2 on the average; that is $N(y, A_{i+1})$ = log$n$ / 2. Following the same argument, we can say that the number of new elements in A_{i+1} (the elements in the area c), which form a cut-edge with z is log$n$ / 3; that is $N(z, A_{i+1})$ = log$n$ / 3, and so on. Thus, the number of elements in A_{i+1} which form a cut-edge with at least an element in *lr-max*(A_i) can be computed as follows:

$$
\begin{aligned}
N(lr\text{-}max(A_i), A_{i+1}) &= \log n + \log n / 2 + \log n / 3 + \ldots + 1 \\
&= \log n\, (1 + 1/2 + 1/3 + \ldots + 1/\log n) = \log n\, H_{\log n} \\
&= \log n\, \big(\ln(\log n) + O(1)\big) \\
&= O(\log n \, \log\log n).
\end{aligned}
$$

Suppose now that there exists an element w in A_i such that $\pi^{-1}(x) < \pi^{-1}(w) < \pi^{-1}(y)$. Moreover, we suppose that $\pi^{-1}(w) = \lceil (\pi^{-1}(x) + \pi^{-1}(y)) / 2 \rceil$. Note that the pair (w, x) forms an edge in A_i; otherwise, $w \in lr\text{-}max(A_i)$. Let $S(x, y) = [y', ..., w', w'', ..., z']$ be the sequence of all the elements in A_{i+1} which form a

cut-edge with y and have index in π greater than $\pi^{-1}(x)$. Then, it is easy to show that the sequence $S(x, y)$ has $\Theta(\log\log n)$ elements on the average. Therefore, the number of elements that form a cut-edge with both w and y is $\log\log n / 2$, and thus the number of new elements in A_{i+1} which form a cut-edge with w is $\log n - (\log\log n / 2)$. In this case, it is obvious that all the new elements are in the area B' (see Figure 6). Consequently, we suppose that there exists an element p in A_i such that $\pi^{-1}(w) < \pi^{-1}(p) < \pi^{-1}(y)$ and $\pi^{-1}(p) = \lceil (\pi^{-1}(w) + \pi^{-1}(y)) / 2 \rceil$. Then, it is easy to see that the number of elements which form a cut-edge with both p and y is $\log\log n / 4$ if the pair (p, w) forms an edge in A_i; otherwise, this number is $(3/4)\log\log n$. In our analysis, we suppose that (p, w) forms an edge in A_i. Thus, the number of new elements in A_{i+1} which form a cut-edge with p is $\log n - (\log\log n / 4)$. Following the same argument, we conclude that the number $N(W(x, y), A_{i+1})$ of new elements in A_{i+1} is:

$$\begin{aligned} N(W(x, y), A_{i+1}) &= \log n - (\log\log n / 2) + \log n - (\log\log n / 4) + \ldots + \log n - (\log\log n / \log n) \\ &= \log n \log\log n - ((\log\log n / 2) + (\log\log n / 4) + \ldots + (\log\log n / \log n)) \end{aligned}$$

Thus, the number of elements in A_{i+1} which form a cut-edge with at least an element in A_i—that is, the number of elements in the set $UV(A_{i+1})$, can be computed as follows:

$$|UV(A_{i+1})| = N(lr\text{-}max(A_i), A_{i+1}) + \log n \, N(W(x, y), A_{i+1}) = O(\log^2 n \log\log n)$$

Following a similar way, we can show that the number of elements in the set $LV(A_i)$ is $O(\log^2 n \log\log n)$ on the average.

In Section 5 we presented a parallel algorithm which solves the coloring problem on permutation graphs in $O(\log n \log k)$ time using $O(kn^2/\log k \log^2 n)$ processors on the CREW PRAM model of computation, where $k = max\{k_1, k_2, ..., k_{\log n - 1}\}$ and $k_i = max\{|LV(A_i)|, |UV(A_{i+1})|\}$, $i = 1, 2, ..., \log n - 1$. Thus, the following theorem applies.

Theorem 6.1 The problem of coloring a permutation graph of size n can be solved in $O(\log n \log\log n)$ time in the average-case on the CREW PRAM model of computation with $O(n^2)$ processors.

7. Concluding Remarks

In this paper we study the problem of coloring permutation graphs. We propose an efficient parallel algorithm which colors a permutation graph in $O(\log n \log k)$ time using $O(kn^2 / \log k \log^2 n)$ processors on the CREW PRAM model of computation, where n is the number of vertices in the permutation graph and $1 < k < n$. Moreover, we estimate the parameter k using random permutations and show that the proposed algorithm requires $O(\log n \log\log n)$ time and $O(n^2)$ processors in the average-case on the CREW PRAM model of computation.

Our algorithm takes as input a permutation graph $G^*[\pi]$ and directly constructs a directed acyclic graph $G^*[\pi]$ using combinatorial properties on π. Then, it solves the coloring problem on the permutation graph $G[\pi]$ by solving the longest path problem on the constructed acyclic digraph $G^*[\pi]$ using divide-and-conquer techniques.

We point out that our algorithm incorporate all the features that allow it to be executed in a less powerful computational model—that is, the EREW PRAM model, with slight modifications (this would add a factor of $f(n)$ to the processor analysis). Another point we should stress concerns the transformation strategy. Our algorithm transforms the given permutation graph into a directed acyclic graph which has much less edges than the input permutation graph. Notice that, all the transformation

graphs which appeared in previous algorithms have the same number of edges which equals the number of edges of the input permutation graph. Moreover, to the best of our knowledge no parallel algorithm has been published using divide-and-conquer techniques for the same problem.

In closing, we point out that we are now looking into the coloring problem using the Lattice representation of a permutation [12] and the relationship between permutations and binary search trees.

References

[1] M.J. Atallah, G.K. Manacher and J. Urrutia, Finding a minimum independent dominating set in a permutation graph, *Discrete Applied Mathematics*, vol. 21, pp. 177-183, 1988.

[2] A. Brandstadt and D. Kratsch, On domination problems for permutation and other graphs, *Theoretical Computer Science*, vol. 54, pp. 181-198, 1987.

[3] A. Datta, A. Maheshwari and J-R. Sack, Optimal Parallel Algorithms for Directed Dominance Problems, *Nordic Journal of Computing*, vol. 3, pp. 72-88, 1996.

[4] M. Farber and J.M. Keil, Domination in permutation graphs, *Journal of Algorithms*, vol. 6, pp. 309-321, 1985.

[5] M.C. Golumbic, *Algorithmic Graph Theory and Perfect Graphs*, Academic Press, Inc., New York, 1980.

[6] D. Helmbold and E.W. Mayr, Applications of parallel algorithms to families of perfect graphs, *Computing*, vol. 7, pp. 93-107, 1990.

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

[8] T.R. Jensen and B. Toft, *Graph Coloring Problems*, John Wiley & Sons, Inc., 1995.

[9] D. Kozen, U.V. Vazirani and V.V. Vazirani, NC algorithms for comparability graphs, interval graphs, and testing for unique perfect matching, *Proc. of the 5th Conference on Foundation of Software Technology and Theoretical Computer Science*, New Delhi, pp. 498-503, 1985.

[10] A. Pnueli, A. Lempel and S. Even, Transitive orientation of graphs and identification of permutation graphs, *Canadian J. Math.*, vol. 23, pp. 160-175, 1971.

[11] J. Reif (editor), *Synthesis of Parallel Algorithms*, Morgan Kaufmann Publishers, Inc., San Mateo, California, 1993.

[12] R. Sedgewick and P. Flajolet, *An Introduction to the Analysis of Algorithms*, Addison-Wesley, 1996.

[13] J. Spinrad, On comparability and permutation graphs, *SIAM Journal on Computing*, vol. 14, pp. 658-670, 1985.

[14] K.H. Tsai and W.L. Hsu, Fast algorithms for the dominating set problem on permutation graphs, *Lecture Notes in Computer Science: Algorithms*, vol. 450, pp. 109-117, 1990.

[15] C-W. Yu and G-H. Chen, Parallel algorithms for permutation graphs, *BIT*, vol. 33, pp. 413-419, 1993.

[16] C-W. Yu and G-H. Chen, Generate all maximal independent sets in permutation graphs, *Intern. J. Computer Math.*, vol. 47, pp. 1-8, 1993.

[17] M-S. Yu, L.Y Tseng and S-J. Chang, Sequential and parallel algorithms for the maximum-weight independent set problem on permutation graphs, *Inform. Process. Lett.*, vol. 46, pp. 7-11, 1993.