JOURNAL OF ALGORITHMS 12, 375–392 (1991)

Parallel Recognition of the Consecutive Ones Property with Applications

LIN CHEN*

Department of Computer and Information Science, Ohio State University, Columbus, Ohio 43210

AND

YAACOV YESHA†

Department of Computer Science, University of Maryland Baltimore County, Baltimore, Maryland 21228 and University of Maryland, Institute for Advanced Computer Studies, College Park, Maryland 20742

Received August 22, 1988; revised August 1990

Given a (0, 1)-matrix, the problem of recognizing the consecutive 1's property for rows is to decide whether it is possible to permute the columns such that the resulting matrix has the consecutive 1's in each of its rows. In this paper, we give the first NC algorithm for this problem. The algorithm runs in $O(\log n + \log^2 m)$ time using $O(m^2 n + m^3)$ processors on Common CRCW PRAM, where $m \times n$ is the size of the matrix. The algorithm can be extended to detect the circular 1's property within the same resource bounds. We can also make use of the algorithm to recognize convex bipartite graphs in $O(\log^2 n)$ time using $O(n^3)$ processors, where n is the number of vertices in a graph. We further show that the maximum matching problem for arbitrary convex bipartite graphs can be solved within the same complexity bounds, combining the work by Dekel and Sahni, who gave an efficient parallel algorithm for computing maximum matchings in convex bipartite graphs with the condition that the neighbors of each vertex in one vertex set of a bipartite graph occur consecutively in the other vertex set. This broadens the class of graphs for which the maximum matching problem is known to be in NC. © 1991 Academic Press, Inc.

1. INTRODUCTION

Testing for the consecutive 1's property for a (0, 1)-matrix is an important problem. The problem asks, for a given (0, 1)-matrix, whether it is

*Supported in part by the Office of Research and Graduate Studies of Ohio State University.

†Supported in part by the National Science Foundation under Grant DCR-8606366 at Ohio State University.

0196-6774/91 $3.00
Copyright © 1991 by Academic Press, Inc.
All rights of reproduction in any form reserved.

possible to permute the columns of the matrix such that the resulting matrix has consecutive 1's in each of its rows. If so, we are also interested in knowing how the columns should be permuted. Fulkerson and Gross [9] gave the first polynomial time sequential algorithm. The problem has also been studied by Eswaran [8]. Booth and Lueker [1] obtained the most efficient sequential algorithm. The problem is closely related to the consecutive retrieval property [11] and finds applications in many fields such as genetics, archaeology, information science, operational research, and so on. Readers are referred to Golumbic [12] and Roberts [21] for more details. In this paper, we give, for the first time, an NC algorithm which recognizes the consecutive 1's property for a (0, 1)-matrix and, if possible, transforms the matrix into one with consecutive 1's in each of its rows or columns. NC is the class of problems which can be solved in polylogarithmic time using a polynomial number of processors. Such an algorithm is called an NC algorithm.

Based on a characterization of the circular 1's property by Tucker [24], we can obtain an NC algorithm for recognizing the circular 1's property. Our algorithm can also be used to recognize the class of convex bipartite graphs, which has been studied by Lipski and Preparata [19] and Dekel and Sahni [7]. Lipski and Preparata gave the sequential algorithm for the maximum matching problem on convex bipartite graphs. Dekel and Sahni gave the parallel algorithm for the maximum matching problem, assuming in the input, the bipartite graph matrix has consecutive 1's in each of its rows. Note that two definitions of convex bipartite graphs in the two papers mentioned above are not equivalent. The class of "convex bipartite graphs" defined by Dekel and Sahni is smaller, since the 1's in each row of the matrix for a "convex bipartite graph" are always consecutive, according to their definition. In this paper, we study the larger class of convex bipartite graphs. So far, it is still open whether the maximum matching problem is in NC or not. Karp *et al.* [16] showed that the problem can be solved by a randomized algorithm in polylogarithmic time using a polynomial number of processors. Galil [10] has provided a very good survey on the maximum matching problem. Combining the work by Dekel and Sahni, we can show that the maximum matching problem for the (larger) class of convex bipartite graphs is in NC. This broadens the class of graphs for which the maximum matching problem is known to be in NC.

This paper is organized as follows. The next section introduces many terminologies. Section 3 deals with some useful properties of graphs and matrices. Section 4 treats at length the parallel recognition of the consecutive 1's property. And finally, Section 5 discusses the applications, primarily the recognition of convex bipartite graphs and finding their maximum matching.

2. Preliminaries

In this section we provide the reader with the terminology needed to discuss the problem. Many terms such as graphs, trees are standard. Readers may see Golumbic [12] and Liu [20] for the definitions.

An $n \times n$ matrix A is called an identity matrix if for any element a_{ij} of A,

$$a_{ij} = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{otherwise.} \end{cases}$$

We use I to denote an identity matrix. A matrix is called a permutation matrix if it can be obtained from an identity matrix by 0 or more interchanges of rows. We use $I(i, j)$ to represent a matrix obtained by interchanging row i and row j of the identity matrix.

A $(0, 1)$-matrix is said to have the consecutive 1's property for rows (columns) if its columns (rows) can be permuted such that the 1's in each row (column) of the resulting matrix occur consecutively. A $(0, 1)$-matrix is said to have the circular 1's property for rows (columns) if its columns (rows) can be permuted such that the 1's in each row (column) of the resulting matrix occur in a circular consecutive order. We can also define these properties in terms of permutation matrix. For example, we say a $(0, 1)$-matrix M has the consecutive 1's property for rows if there exists a permutation matrix P such that the 1's in each row of MP occur consecutively.

For a $(0, 1)$-matrix M, it is convenient, and imposes no loss of generality in studying the consecutive 1's property, to assume that row vector $x_i \neq 0$ for $i = 1, \ldots, n$, and that $x_i \neq x_j$ for $i \neq j$. In the sequel, we will make these assumptions unless otherwise stated and refer to such a matrix M as proper.

A graph is often represented by a matrix. Let $G = (V, E)$ be a graph whose vertices have been (arbitrarily) ordered $v_1, v_2, \ldots, v_n$. The adjacency matrix $M = (m_{ij})$ of G is an $n \times n$ matrix with entries

$$m_{ij} = \begin{cases} 1 & \text{if } \{v_i, v_j\} \text{ or } (v_i, v_j) \in E \\ 0 & \text{otherwise.} \end{cases}$$

A matching M of graph G is a subset of edge set E such that no two edges in M have a common endpoint. A maximum matching M of a graph G is a matching such that no matching of G has a size greater than $|M|$.

A graph is bipartite if its vertices can be partitioned into two sets such that each edge is between a vertex in one set and a vertex in the other. A bipartite graph is often represented by a $(0, 1)$-matrix with rows corresponding to the vertices in one vertex set and columns corresponding to

the vertices in the other. Such a matrix is often referred to as a bipartite graph matrix in this paper. If we can order the vertices of a bipartite graph such that the matrix has the consecutive 1's in each of its rows, then the graph is called a convex bipartite graph. The graph corresponding to the resulting matrix is called an ordered convex bipartite graph.

Vector a and vector b over $\{0, 1\}$ are said to intersect if their inner product $a \cdot b > 0$. They overlap if they intersect and $a \cdot b < \min(a \cdot a, b \cdot b)$. If $a \cdot b = b \cdot b$, we say that vector a contains vector b. Hence, for any two row vectors r_1 and r_2 of a $(0, 1)$-matrix M, r_1 contains r_2 implies $r_2 \cdot r_2 = r_1 \cdot r_2 < r_1 \cdot r_1$, because we assume that the matrix M is proper.

An undirected graph $S(M) = (V, E)$ is called an intersection graph of a $(0, 1)$-matrix M if there exists a bijection f: $V \rightarrow$ the set of row vectors of M such that for any two distinct vertices v_i and v_j in V, $\{v_i, v_j\} \in E$ if and only if $f(v_i)$ and $f(v_j)$ intersect. Similarly, an undirected graph $L(M) = (V, E)$ is called an overlap graph of a bipartite graph matrix M if there exists a bijection f: $V \rightarrow$ the set of row vectors of M such that for any two vertices v_i and v_j in V, $\{v_i, v_j\} \in E$ if and only if $f(v_i)$ and $f(v_j)$ overlap.

Hereafter, we will use the vertex of an overlap graph to indicate a row vector of the corresponding $(0, 1)$-matrix, and vice versa, if no confusion can arise.

A directed graph $T(M) = (V, E)$ is called the component graph of a bipartite graph matrix M, if there exists a bijection f: $V \rightarrow$ the set of the components of the overlap graph such that for any two distinct vertices v_i and v_j in V, $(v_i, v_j) \in E$ if and only if there exists a row vector x_p in the component corresponding to v_i and a row vector x_q in the component corresponding to v_j such that x_p contains x_q.

A directed graph $C(M) = (V, E)$ is called a containment graph of a $(0, 1)$-matrix M if there exists a bijection f: $V \rightarrow$ the set of row vectors of M such that for any two distinct vertices v_i and v_j in V, $(v_i, v_j) \in E$ if and only if $f(v_i)$ contains $f(v_j)$.

EXAMPLE 1. Suppose we have a $(0, 1)$-matrix M with 9 rows and 9 columns,

$$M = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \end{bmatrix}.$$

Figures 1, 2, 3, and 4 are its corresponding intersection graph, overlap graph, containment graph, and component graph, respectively.

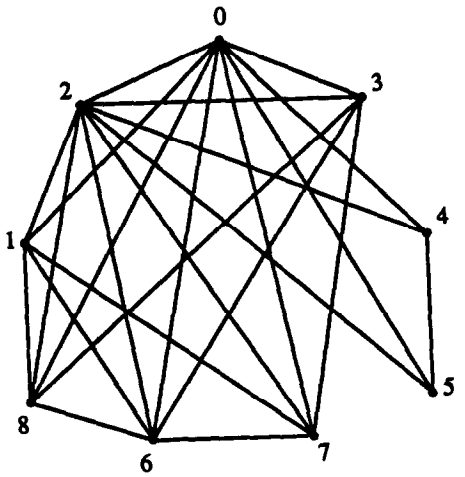

FIG. 1. The intersection graph.

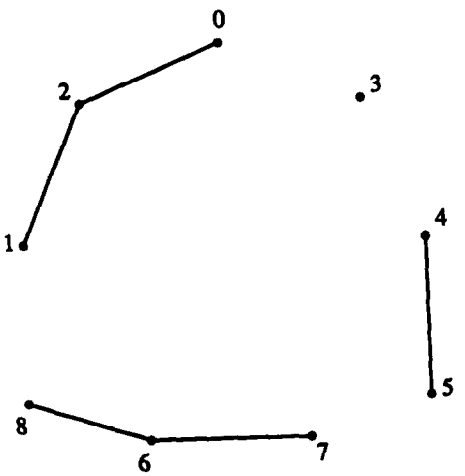

FIG. 2. The overlap graph.

Note that every vertex of a component graph corresponds to a component of the overlap graph, which in turn corresponds to some row vectors of the (0, 1)-matrix. Those row vectors constitute a submatrix. We will refer to such submatrices as components too.

If A is a matrix with consecutive 1's in each of its rows, then PA is, too, where P is any permutation matrix. So is AQ, where Q is $I(1, n)I(2, n-1)\cdots I(n/2, n-n/2+1)$. We call Q the queue matrix.

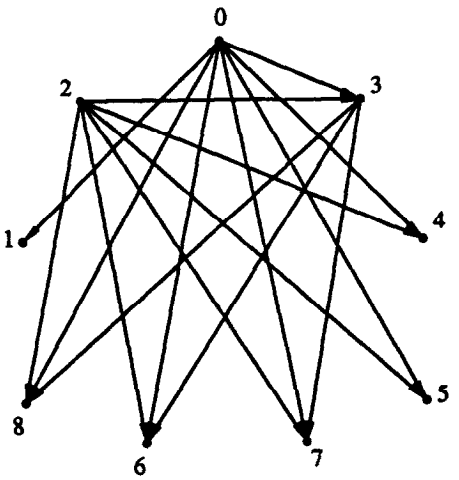

FIG. 3. The containment graph.

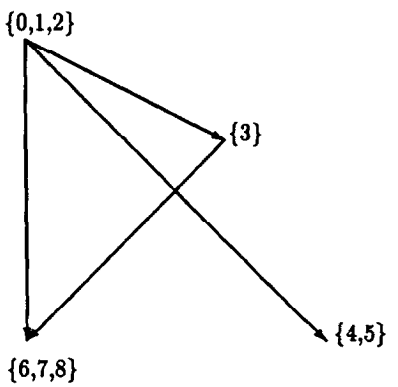

FIG. 4. The component graph.

The parallel computation model used in this paper is a parallel random access machine (PRAM for short). On this model, several processors can read from a memory cell at the same time. However, no concurrent write is allowed unless several processors attempt to write the same value into a memory cell. This model is often referred to as the common concurrent read concurrent write (CRCW) PRAM. We use as subroutines some existing algorithms implemented on some weaker PRAM models, but obviously those algorithms also work on common CRCW PRAM. We shall mention them later in this paper.

3. Some Useful Properties

In this section we present and investigate some properties of certain graphs and matrices which are helpful in the design of our parallel algorithm.

LEMMA 1. *The containment graph $C(M) = (V, E)$ of a $(0,1)$-matrix M is a cyclic and transitive.*

Proof. Immediate from the definition. □

Below we refer to a graph obtained by deleting from a containment graph all the arcs implied by transitivity as a reduced containment graph. Fulkerson and Gross have obtained the following similar result:

LEMMA 2. *The component graph $T(M) = (V, E)$ of a $(0,1)$-matrix M is acyclic and transitive. Moreover, every vertex has at most one immediate predecessor.*

From the above lemma, we can easily see that, if we delete all the arcs implied by transitivity, we get a forest. We call the forest a reduced component graph.

Next we present a theorem, due to Fulkerson and Gross, which is crucial in justifying the correctness of the sequential algorithm.

THEOREM 1. *Let A and B be $(0,1)$-matrices satisfying $AA^T = BB^T$. Then either both A and B have the consecutive 1's property for rows or neither does. Moreover, if A and B have the same number of columns and A has the consecutive 1's property for rows, then there is a permutation matrix P such that $B = AP$.*

Proof. See [9]. □

Below we give a theorem which helps us to obtain the parallel algorithm.

THEOREM 2. *If a $(0,1)$-matrix has the consecutive 1's property for rows, and if the overlap graph is connected, then the matrix which can be obtained by permuting the columns and which has consecutive 1's in each of its rows is unique up to reversing the order of columns.*

Proof. Apply induction on the number of rows of the matrix. If $n = 1$ or $n = 2$, it is trivially true. Suppose it is true for $n = k$ $(k > 1)$. Consider the case $n = k + 1$. Let the overlap graph be denoted by G. Permute the columns of the matrix so that the resulting matrix, denoted by M, has consecutive 1's in each of its rows. Since G is connected, there exists at least one vertex v such that $G - \{v\}$ is also connected. Delete the corresponding row from the matrix M. The resulting matrix, M', also has

consecutive 1's in each of its rows. By the induction hypothesis, M' is unique up to reversing the order of columns. Now look at M. Since G is connected, there exists v', v'' such that $\{v', v\}$ and $\{v', v''\}$ are edges of G. Observe that the inner products of $v \cdot v', v' \cdot v''$, and $v'' \cdot v$ are not affected by the permutation of columns. The consecutive 1's for v' and v'' in M determines the unique location for the sequence of 1's corresponding to v. This completes the proof. □

4. The Consecutive Ones Property

In this section, we exhibit the parallel algorithm for recognizing the consecutive 1's property. The algorithm gives a useful tool for solving many other problems.

The following procedure ConsecRow(M, M', ok) decides whether the given matrix M has the consecutive 1's property for rows. If so, the procedure transforms M into a matrix M' which has consecutive 1's in each of its rows, and sets ok to *true*; otherwise, it returns with ok = false.

Procedure ConsecRow(M, M', ok).

1. Construct the overlap graph and containment graph from matrix M;
2. Get the reduced containment graph;
3. Get the connected components of the overlap graph;
4. Get the reduced component graph from the results of steps 2 and 3; comment: the resulting graph is a forest.
5. For every connected component of the overlap graph, get a spanning tree T;
6. Get the component trees from the reduced component graph;
7. For every tree T, call Consec(T, MX, yes); Comment: If the submatrix corresponding to T does not have the consecutive 1's property for rows, Consec(T, MX, yes) will return with yes = false; otherwise yes = true and MX is a matrix obtained from the matrix corresponding to T by permuting its columns so that the 1's in each row of MX appear consecutively.
8. If all the connected components of the overlap graph have the consecutive 1's property, then proceed with ok = true; otherwise quit with ok = false; Comment: the matrix has consecutive 1's property if and only if all the components have consecutive 1's property.
9. For every component tree Tr, call Arrange(Tr, Matrix); Comment: Arrange(Tr, Matrix) will return in Matrix an ordered convex bipartite graph corresponding to Tr.

10. Combine the matrix for every component tree into one matrix M'. End ConsecRow.

We now analyze the time and the number of processors needed by procedure ConsecRow and go into more details whenever appropriate. Assume the input matrix M is of size $m \times n$.

Step 1 can be done in $O(1)$ time as long as there are enough processors. For example, it can be done in the following way:

1. for $0 < i, j \leq m$ pardo
   $l_{ij} := 0$; $c_{ij} := 0$; $d_{ij} := 0$; $s_{ij} := 0$;
2. for $0 < p, q \leq m, 0 < j \leq n$ pardo
   if $m_{pj} = 1$ and $m_{qj} = 0$, then $d_{pq} := 1$;
   if $m_{pj} = 1$ and $m_{qj} = 1$, then $s_{pq} := 1$;
3. for $0 < p, q \leq m$ and $p < q$ pardo
   if $d_{qp} = 1$ and $d_{qp} = 1$ and $s_{pq} = 1$ then $\{l_{pq} := 1; l_{qp} := 1\}$;
   if $d_{qp} = 1$ and $d_{qp} = 0$ then $c_{pq} := 1$;
   if $d_{qp} = 0$ and $d_{qp} = 1$ then $c_{qp} := 1$.

s_{ij} is used to represent intersection relation. After the second step, $s_{ij} = 1$ if and only if row i and row j intersect. The resultant overlap graph is $(l_{ij})_{m \times m}$ in the form of adjacent matrix, and the containment graph $(c_{ij})_{m \times m}$. Substep 1 needs constant time and m^2 processors. Substep 2 can be done in $O(1)$ time with $O(m^2 n)$ processors. The last substep requires $O(1)$ time and $O(m^2)$ processors. Therefore, step 1 can be done in $O(1)$ time with $O(m^2 n)$ processors.

Step 2 can be done in $O(1)$ time with $O(m^3)$ processors. Note that for every arc (or directed edge) $\{a, c\}$ implied by transitivity, there exist two arcs $\{a, b\}$, $\{b, c\}$. So removing an arc implied by transitivity will destroy a triangle. Therefore, we can obtain the reduced containment graph as follows:

1. for $0 < i, j \leq m$ pardo, $r_{ij} := c_{ij}$;
2. for $0 < i, j, k \leq m$ pardo, if $c_{ij} = 1$ and $c_{jk} = 1$ then $r_{ik} := 0$.

The reduced containment graph is $(r_{ij})_{m \times m}$. The second substep is the dominating step. It can be done in $O(1)$ time with $O(m^3)$ processors.

The next step can be done using the existing algorithm. An NC algorithm for finding the connected components of a graph was presented in Hirschberg *et al.* [15]. Chin *et al.* [5] have improved the result. They have shown that, among other things, finding the connected components and spanning forest of a graph can be done in $O(\log^2 n)$ time using $O(n^2/\log^2 n)$ processors on concurrent read exclusive write (CREW) PRAM. So this step can be done in $O(\log^2 m)$ time with $O(m^2/\log^2 m)$ processors.

Step 4 is to get the reduced component graph. We want to transform all the vertices of a reduced containment graph in the same connected component of the overlap graph into a single vertex of a reduced component graph. This only needs $O(1)$ time and $O(m)$ processors.

Step 5 can also be done using the existing algorithm within the same resource bounds as the third step [5]. Step 6 is to get the component trees. All we need to do is to get the connected components of the reduced component graph. So we can do this step in the same way as in the third step.

Step 7 is not as easy as the previous steps. It is the trickiest part of the algorithm. We do this step in the following manner: Procedure Consec first computes the size of the intersection of every two row vectors. Then it invokes a recursive procedure Re-con and gets a matrix from that procedure. For the resulting matrix, the 1's occur consecutively in each row. Obviously, it has the consecutive 1's property. Finally it checks whether the size of the intersection of every two row vectors of the resultant matrix is the same as that of the input matrix. By Theorem 1, we know the input matrix has the consecutive 1's property if the size of every corresponding pairwise intersection is the same for the two matrices. The procedure can be described as follows:

```
PROCEDURE Consec(T, MX, yes).
Compute the size of the intersection of every two row vectors;
Re-con(T, MX);
If size of every pairwise intersection unchanged
  then yes := true
  else yes := false
End Consec.

PROCEDURE Re-con(T, MX).
If |T| < c then obtain MX directly
 else Begin
        Get a balanced decomposition (V1, V2) of T;
        Re-con(V1, MX1);
        Re-con(V2, MX2);
      Merge MX1, MX2 into MX
      End
End Re-con.
```

The first step of Procedure Consec is to compute the size of the intersection of every two row vectors. That is, for any two row vectors r_i and r_j, compute $r_i \cdot r_j$. In case $i = j$, $r_i \cdot r_j$ is the number of 1's in the row vector. For two n-dimension $(0, 1)$-vectors a and b, we can compute the

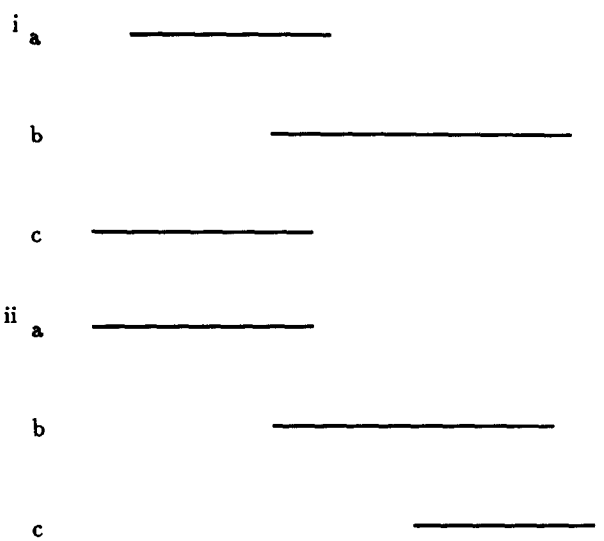

FIG. 5. The relation between intervals: (i) a and c are on the same side of b; (ii) a and c are on the different sides of b.

intersection as follows. Obtain another vector c with $c[i] = a[i] \wedge b[i]$, for $0 < i \leq n$. Then compute $\sum_{i=1}^{n} c[i]$ by applying the existing algorithm for prefix computation, which can be done in $O(\log n)$ time using $O(n/\log n)$ processors on EREW PRAM [18, 17], or in $O(\log n/\log\log n)$ time using $n \log\log n/\log n$ processors on common CRCW PRAM [6]. These algorithms are cost optimal in the sense that the processor–time product equals the time lower bound for the sequential algorithm. Note that there are $O(m^2)$ pairwise intersections. So the first step of Consec can be done in $O(\log n)$ time using $O(m^2 n/\log n)$ processors on EREW PRAM or in $O(\log n/\log\log n)$ time using $O(m^2 n \log\log n)$ processors on common CRCW PRAM.

Now we take a close look at Procedure Re-con. In case $|T| < c$, we can get MX row by row. Each row can be regarded as an interval. For any three intervals a, b, and c, if a and b overlap and b and c overlap, then we can decide whether a and c should be on the same side of b or not (see Fig. 5). If $a \cdot c < \min\{a \cdot b, b \cdot c\}$, then they should be put at the different sides of b. Otherwise, they should be put at the same side of b. Suppose we want to get a row corresponding to c, and we have known that c should be put at the right-hand side of b. Using a technique as in Shiloach and Vishkin [22], we can get the last point p of interval b in constant step. Since we know the length l of interval c, i.e., $c \cdot c$, and the size s of the intersection between b and c, i.e., $b \cdot c$, we know immediately

that interval c is from point $p - s + 1$ to point $p - s + l$. It is not difficult to see that we can handle the case $|T| < c$ in $O(1)$ time with $O(n)$ processors.

Next consider merging MX_1 and MX_2 into MX. Note that MX_1 and MX_2 are both matrices with consecutive 1's in each of its rows. According to Theorem 2, we may need to reverse the order of columns of one matrix before merging. Notice that two matrices have one interval in common. We use b to denote that interval. In MX_1 there exists an interval a such that a and b overlap. Likewise, there exists an interval c in MX_2 such that b and c overlap. By looking at these three intervals and comparing the values of $a \cdot b$, $b \cdot c$, and $a \cdot c$, we can decide whether a and c are at the correct side of b. If not, on one of the two matrices, call it X, we need to perform the transformation $X' = XQ$. After that, we can do the actual merging easily. This merging step can be done in constant time with $O(mn)$ processors. Notice that the resulting matrix is guaranteed to be a matrix with consecutive 1's in each of its rows, though the original matrix may not have the consecutive 1's property. The last step of procedure Consec will decide whether the resulting matrix can be obtained by permuting the columns of the original matrix.

He and Yesha [13] have designed an algorithm to decompose a tree into two subtrees of approximately the same size. A decomposition of tree T can be denoted as (v, T_1, T_2), where $V(T_1) \cap V(T_2) = \{v\}$ and $E(T_1) \cap E(T_2) = \emptyset$. A decomposition (v, T_1, T_2) is called a balanced decomposition if $n/3 \leq |T_1| \leq 2n/3$, where $n = |T|$. Sometimes we simply use (T_1, T_2) to denote a decomposition of a tree. He and Yasha [14] have shown that if the input is given by the decomposition tree form, many graph problems can be solved efficiently in parallel. He and Yasha's algorithm for finding a balanced decomposition runs in $O(\log n)$ time with $O(n)$ processors on exclusive read exclusive write (EREW) PRAM. Their decomposition algorithm turns out to be very useful. It directly supports divide-and-conquer methodology, which is one of the most important algorithm (both sequential and parallel) design techniques. We believe that the algorithm is useful in solving many other problems.

Since the number of recursions in Procedure Re-con is bounded by $O(\log m)$ and a balanced decomposition can be found in $O(\log m)$ time, Procedure Re-con can be done in $O(\log^2 m)$ time. The processor requirement is $O(mn)$.

The last step of procedure Consec is to do some checking. The amount of resources needed is the same as in the first step of Procedure Consec. Therefore, the total time needed by step 7 is $O(\log n + \log^2 m)$, and the total processor requirement is $O(m^2 n)$.

Step 8 is trivial. Its resource requirements are dominated by the other steps. For step 9, we adopt divide-and-conquer strategy. Following is the

outline of Procedure Arrange:

```
PROCEDURE Arrange(Tr, Matrix).
If |Tr| < c then obtain Matrix directly
 else Begin
        Get a balanced decomposition (Tr_1, Tr_2) of Tr;
        Arrange(Tr_1, Matrix_1);
        Arrange(Tr_2, Matrix_2);
        Merge Matrix_1, Matrix_2 into Matrix
      End
End Arrange.
```

Similar to the analysis of step 7, we can show that step 9 can be done in $O(\log^2 m)$ with $O(mn)$ processors. For the last step, it is not difficult to see that the time needed is $O(\log m)$, and the number of processors needed is $O(mn)$.

From the above analysis, we can conclude that the overall time complexity for Procedure ConsecRow is $O(\log n + \log^2 m)$. The total number of processors needed is $O(m^2 n + m^3)$. Notice that Procedure ConsecRow can be made faster. Some steps can be done simultaneously, for instance, steps 2 and 3, steps 4 and 5. However, the time complexity remains of the same order of magnitude. Notice that we assume the input to Procedure ConsecRow is a proper matrix. In case some rows only consist of 0's, we can simply discard those rows. If several rows are exactly the same, we can save one of them and delete the others. After the transformation, we duplicate the corresponding rows. All these can be done in constant time with no more processors. Hence we have proved the following theorem.

THEOREM 3. *Testing if a matrix M has the consecutive* 1*'s property for rows can be done in* $O(\log n + \log^2 m)$ *time using* $O(m^2 n + m^3)$ *processors, where* $m \times n$ *is the size of M.*

5. APPLICATIONS

We first show that, building on the NC algorithm for recognizing the consecutive 1's property, we can give an NC algorithm for recognizing the circular 1's property. Tucker [24] once discovered a nice characterization for the circular 1's property:

THEOREM 4. *Matrix M has the circular* 1*'s property for rows if and only if M' has the consecutive* 1*'s property for rows, where M' is the matrix obtained from M by complementing those rows with a* 1 *in the kth column* (*k chosen arbitrarily*).

Therefore, to decide whether a matrix M has the circular 1's property, it suffices to test M' for the consecutive 1's property. It is easy to see that the complexity bounds are dominated by testing for the consecutive 1's property. Hence we have the following theorem.

THEOREM 5. *Testing if a matrix M has the circular 1's property for rows, and if so, transforming the matrix into one with circular consecutive 1's in each of its rows can be done in* $O(\log n + \log^2 m)$ *time using* $O(m^2 n + m^3)$ *processors, where* $m \times n$ *is the size of M.*

Our algorithm has been applied to solving various graph problems. We have given first NC algorithms which test for bipartite permutation graphs and isomorphism of bipartite permutation graphs, and which find a Hamiltonian path and cycle [4], and first NC algorithms which recognize several circular-arc graphs [2, 3] and many other problems.

Now let us consider the problem of recognizing convex bipartite graphs and finding the transformation. First we describe how to recognize bipartite graphs. Without loss of generality, we assume the input is a connected graph. The idea of recognition is to try to put all the vertices into two sets such that there is no edge between two vertices of one set. If we are unable to do this, then by the definition of bipartite graphs, we can conclude that the graph is not a bipartite graph. We first get the spanning tree of the input graph. Since every tree is a bipartite graph, we can put all the vertices into two sets such that every edge connects two vertices from different sets. Then we check if there is any edge of the original graph between two vertices of one set. If so, the graph is not a bipartite graph; otherwise, it is. Tarjan and Vishkin [23] developed the Euler tour technique on trees. They showed that computing a rooted tree and the level of each vertex in the rooted tree can be done in $O(\log n)$ time using $O(n)$ processors on EREW PRAM. Then two vertex sets can be trivially constructed from the level numbers of the vertices and the bipartiteness can be checked easily. The dominating step is to obtain the spanning tree, which takes $O(\log^2 n)$ time and $n^2/\log^2 n$ processors. Now we only need to consider the problem of deciding whether a bipartite graph is a convex bipartite graph or not.

By definition, we know that a bipartite graph is a convex bipartite graph if and only if the bipartite graph matrix corresponding to each connected component has the consecutive 1's property for rows or columns.

Based on *ConsecRow*, we can design an NC algorithm which recognizes convex bipartite graphs. Here we assume the input graph is a bipartite graph and the input is a bipartite graph matrix B. The following procedure *Recog* will set *yes* to *true* if and only if the input graph is a convex bipartite graph.

PROCEDURE Recog(B, yes).
1. Obtain the connected components of B;
2. For each connected component C do in parallel
begin
ConsecRow(C, C', ok);
$D := C^T$;
ConsecRow(D, D', ok');
If $ok = false \wedge ok' = false$ then set yes to *false* and return
end;
3. set yes to *true*
End Recog.

Let $m \times n$ be the size of matrix B. Then both m and n are of the order $O(N)$, where N is the number of the vertices of the corresponding bipartite graph. It is now easy to see that the above algorithm runs in $O(\log^2 N)$ time with $O(N^3)$ processors, where N is the number of vertices of the input graph. Suppose the input graph has k connected component, where $k > 0$. There are n_i vertices in each connected component i. So $\sum_{i=1}^{k} n_i = n$, where n is the number of vertices of the input graph. For each connected component i, the time needed is $O(\log^2 n_i)$ and processor requirement is $O(n_i^3)$. So the overall time complexity is $O(\max_{i=1}^{k} \log^2 n_i)$, which is $O(\log^2 n)$. The total processor requirement is:

$$\begin{aligned} O(n_1^3) + O(n_2^3) + \cdots + O(n_k^3) &\le cn_1^3 + cn_2^3 + \cdots + cn_k^3 \\ &\le c(n_1 + n_2 + \cdots + n_k)^3 \\ &= O(n^3) \end{aligned}$$

This result can be summarized as the following theorem.

THEOREM 6. *A convex bipartite graph can be recognized in $O(\log^2 N)$ time with $O(N^3)$ processors on CRCW PRAM.*

Now let us consider the problem of finding a maximum matching of a convex bipartite graph. For the sake of simplicity, we will assume the input graph is connected. If the graph is not connected, the solution is simply the combination of solutions of all the connected components.

We may obtain a maximum matching of a convex bipartite graph by transforming the input graph into an ordered convex bipartite graph, and then using Dekel and Sahni's algorithm for finding a maximum matching in an ordered convex bipartite graph. But even if we know the maximum matching of the ordered convex bipartite graph, we may not know the maximum matching of the input graph. To tackle this difficulty, we

establish the correspondence between the edges of the graph before transformation and those after transformation. Suppose a bipartite graph matrix has the consecutive 1's property for rows. We permute the columns and get a convex bipartite graph. Then we try to obtain the correspondence by comparing the columns of the new and old matrices since every column of the new matrix is also a column of the old matrix and vice versa. But sometimes a matrix may have several identical columns, so before the transformation we add some rows to make every column distinct. Each of these rows appended consists of one 1 and several 0's. It is not difficult to see that this operation does not affect the consecutive 1's property for rows. In this way, we are able to establish the mapping of the edges. After we get the mapping, we delete the appended rows and invoke Dekel and Sahni's procedure.

Next we give an example to show how to add some rows to make every column of a matrix unique.

EXAMPLE 3. Suppose we have the following matrix:

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 & 0 \end{bmatrix}.$$

The first three columns of the matrix are identical. So we append two rows to the matrix, and the resulting matrix is:

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix}.$$

Now every column of the matrix is distinct. Note that the number of rows appended is always less than the number of the columns. □

A variation of this method, as one referee points out, is to simply append an $n \times n$ identity matrix below the matrix B.

The following algorithm finds a maximum matching in a convex bipartite graph. We assume the input is a bipartite graph represented by a bipartite graph matrix. If the graph is not a convex bipartite graph, the algorithm gives a message which indicates so. In our algorithm, we use as a subroutine *DSMatching* which is Dekel and Sahni's algorithm for finding a maximum matching in an ordered convex bipartite graph.

PROCEDURE MaxMatching(B, M).

1. $C := B$;

2. Append some rows, if necessary, so that every column of C becomes distinct;

3. ConsecRow(C, C', ok);
4. If ok then begin
   Compare the columns of C and C' and obtain the mapping F;
   Delete from C' the rows appended;
   DSMatching(C', M');
   Obtain M from M' and F, then return
   end
5. $C := B^T$;
6. Append some rows, if necessary, so that every column of C becomes distinct;
7. ConsecRow(C, C', ok);
8. If ok then begin
   Compare the columns of C and C' and obtain the mapping F;
   Delete from C' the rows appended;
   DSMatching(C', M');
   Obtain M from M' and F, then return
   end
   else return with the message "the graph is not convex"

End MaxMatching.

Since Dekel and Sahni's algorithm for finding a maximum matching in an ordered convex bipartite graph runs in $O(\log^2 N)$ time with $O(N)$ processors on CREW PRAM, we can get a maximum matching of a convex bipartite graph in $O(\log^2 N)$ time with $O(N^3)$ processors. Thus we have obtained the following theorem.

THEOREM 7. *A maximum matching of a convex bipartite graph can be found in $O(\log^2 N)$ time with $O(N^3)$ number of processors on a concurrent read concurrent write PRAM.*

ACKNOWLEDGMENTS

We are grateful to many people who helped us during this research. In particular, we thank the anonymous referees for their many comments and suggestions which were very helpful in correcting some errors and improving the presentation of this paper.

REFERENCES

1. K. S. BOOTH AND G. S. LUEKER, Testing for the consecutive ones property, interval graphs, and graph planarity using PQ-tree algorithms, *J. Comput. System Sci.* **13** (1976), 335–379.

2. L. CHEN, Efficient parallel algorithms for several intersection graphs, *in* "Proceedings, 22nd Int'l Symp. on Circuits and Systems," pp. 973–976, IEEE, New York, 1989.
3. L. CHEN, NC algorithms for circular-arc graphs, *in* "Proceedings, Workshop on Algorithms and Data Structures" (F. Dehne, J.-R. Sack, and N. Santoro, Eds.), Lecture Notes in Computer Science, Vol. 382, pp. 291–302, Springer-Verlag, New York/Berlin, 1989.
4. L. CHEN AND Y. YESHA, Fast parallel algorithms for bipartite permutation graphs, *in* "Proceedings, 26th Annu. Allerton Conf. on Communication, Control, and Computing, University of Illinois at Urbana-Champaign, 1988," 264–273.
5. F. Y. CHIN, J. LAM, AND I. CHEN, Efficient parallel algorithms for some graph problems, *Comm. ACM* **25** (1982), 659–665.
6. R. COLE AND U. VISHKIN, Faster optimal parallel prefix sums and list ranking, *Inform. and Comput.* **81**, No. 3 (1989), 334–352.
7. E. DEKEL AND S. SAHNI, A parallel matching algorithm for convex bipartite graphs and applications to scheduling, *J. Parallel Distrib. Comput.* **1** (1984), 185–205.
8. K. P. ESWARAN, Faithful representation of a family of sets by a set of intervals, *SIAM J. Comput.* **4** (1975), 56–68.
9. D. R. FULKERSON AND O. A. GROSS, Incidence matrices and interval groups, *Pacific J. Math.* **15** (1965), 835–855.
10. Z. GALIL, Efficient algorithms for finding maximum matching in graphs, *Comput. Surveys* **18**, No. 1 (1986), 23–38.
11. S. P. GHOSH File organization: The consecutive retrieval property, *Commun. ACM* **15** (1972), 802–808.
12. M. C. GOLUMBIC, "Algorithmic Graph Theory and Perfect Graphs," Computer Science and Applied Mathematics Series, Academic Press, New York, 1980.
13. X. HE AND Y. YESHA, Parallel recognition and decomposition of two terminal series parallel graphs, *Inform. and Comput.* **75** (1987), 15–38.
14. X. HE AND Y. YESHA, A nearly optimal parallel algorithm for constructing depth first spanning trees in planar graphs, *SIAM J. Comput.* **17** (1988), 486–491.
15. D. S. HIRSCHBERG, A. K. CHANDRA, AND D. V. SARWATE, Computing connected components on parallel computers, *Comm. ACM* **22** (1979), 461–464.
16. R. M. KARP, E. UPFAL, AND A. WIGDERSON, Constructing a perfect matching is in random NC, *Combinatorica* **6**, No. 1 (1986), 35–48.
17. C. P. KRUSKAL, L. RUDOLPH, AND M. SNIR, The power of parallel prefix, *IEEE Trans. Comput.* **C-34** (1985), 965–968.
18. R. E. LADNER AND M. J. FISCHER, Parallel prefix computation, *Assoc. Comput. Mach.* **27** (1980), 831–838.
19. W. LIPSKI, JR. AND F. P. PREPARATA, Efficient algorithms for finding maximum matchings in convex bipartite graphs and related problems, *Acta Inform.* **15**, No. 4 (1981), 329–346.
20. C. L. LIU, "Elements of Discrete Mathematics," 2nd ed., McGraw–Hill, New York, 1985.
21. F. S. ROBERTS, "Discrete Mathematical Models with Applications to Social, Biological, and Environmental Problems," Prentice-Hall, Englewood Cliffs, NJ, 1976.
22. Y. SHILOACH AND U. VISHKIN, Finding the maximum, merging, and sorting in a parallel computation model, *J. Algorithms* **2** (1981), 88–102.
23. R. E. TARJAN AND U. VISHKIN, An efficient parallel biconnectivity algorithm, *SIAM J. Comput.* **14**, No. 4 (1985), 862–874.
24. A. C. TUCKER, Characterizing circular-arc graphs, *Bull. Amer. Math. Soc.* **76** (1970), 1257–1260.