1989

An Optimal Parallel Algorithm for the Minimum Circle-Cover Problem

Mikhail J. Atallah
Purdue University, mja@cs.purdue.edu

Danny Z. Chen

Report Number:
88-813

AN OPTIMAL PARALLEL ALGORITHM FOR THE MINIMUM CIRCLE-COVER PROBLEM

Mikhail J. Atallah
Danny Z. Chen
CSD-TR-813
September 1989
(Revised April 1989)

An Optimal Parallel Algorithm for the Minimum Circle-Cover Problem

Mikhail J. Atallah*
Danny Z. Chen
Computer Sciences Department, Purdue University
West Lafayette, In 47907

Abstract

Given a set of n circular-arcs, the problem of finding a minimum number of circular-arcs whose union covers the whole circle has been considered both in sequential and parallel computational models. Here we present a parallel algorithm in the EREW PRAM model that runs in $O(\log n)$ time using $O(n)$ processors if the arcs are not given already sorted, and using $O(n/\log n)$ processors otherwise. Our algorithm is optimal since the problem has an $\Omega(n \log n)$ lower bound for the unsorted arcs case, and an $\Omega(n)$ lower bound for the sorted arcs case. The previous best known parallel algorithm runs in $O(\log n)$ time using $O(n^2)$ processors, in the worst case, in the CREW PRAM model.

Keywords: Analysis of algorithms, circle-cover, combinatorial problems, computational geometry, parallel processing.

1 Introduction

Let $S = \{A_1, A_2, \ldots, A_n\}$ be a set of n circular-arcs on a circle C. The *minimum circle-cover problem* is to find a minimum number of circular-arcs whose union covers C. This problem was considered in references [1,6], where one practical application was mentioned. Another possible application is in scheduling workers so that at least one worker is on the job at any time (the circle then represents the 24 hours in one day, each circular-arc represents the period of time during which a particular worker is willing to work, and the goal is to use as few different workers as possible). In this paper, we present an efficient parallel algorithm for the minimum circle-cover problem.

Lee and Lee [6] gave an $O(n \log n)$ time sequential algorithm for unsorted S and a linear time algorithm for sorted S, and they showed that their algorithms are optimal. Bertossi [1], later,

*This author's research was supported by the Office of Naval Research under Grants N00014-84-K-0502 and N00014-86-K-0689, and the National Science Foundation under Grant DCR-8451393, with matching funds from AT&T.

provided a parallel algorithm in the CREW PRAM computational model (recall that this is the model where the processors, which operate synchronously, share a common memory, and are allowed to concurrently read from the same memory cell but are not allowed to simultaneously write to the same memory cell). The algorithm in [1] runs in $O(\log n)$ time using $O(n^2/\log n + qn)$ processors, where $q-1$ is the minimum number of arcs crossing any point of the circle. In the worst case, Bertossi's algorithm uses $O(n^2)$ processors, and its *time* × *processor* product is $O(n^2 \log n)$.

The computational model for our algorithm is the EREW PRAM model (Exclusive Read Exclusive Write Parallel Random Access Machine), which differs from the CREW PRAM in that even concurrent reading from a memory cell is disallowed. Thus, this computational model is less powerful than that in [1]. Our algorithm runs in $O(\log n)$ time using $O(n)$ processors for unsorted S and using $O(n/\log n)$ processors for sorted S. Therefore, our algorithm is obviously optimal since its time and processor products match the lower bounds given in [6].

In the next section, we give the notations and some preliminary observations, and we outline the main steps of the algorithm. Section 3 describes the preprocessing steps, and Section 4 presents the algorithm.

2 Preliminaries

The input to the algorithm consists of a set $S = \{A_1, A_2, \ldots, A_n\}$, where each A_i is a circular-arc on a circle C (a circular-arc on C is a contiguous portion of the circumference of C). Each A_i is specified by an ordered pair $[x_i, y_i]$, where x_i and y_i are the two endpoints of A_i, such that A_i is drawn by moving the pen clockwise from x_i to y_i; point x_i is then called the *first endpoint* of A_i, and y_i is called the *second endpoint* of A_i. Without loss of generality, we assume that no single arc A_i covers the whole circle C. To avoid cluttering the exposition, we also assume that no two input arcs have the same endpoint (i.e., the $2n$ endpoints are distinct). Our algorithm can easily be modified for the general case. We sort the $2n$ endpoints of the arcs in S such that we encounter the endpoints in increasing order if we start at x_1 and travel along C in the clockwise direction. This sorting can be done in $O(\log n)$ time using $O(n)$ processors in the EREW PRAM model [2]. From now on, we assume that the $2n$ endpoints in S are available in this sorted order. We also assume that the arcs in S have been relabeled such that $i < j$ implies that x_i occurs before x_j in the sorted array of endpoints. This relabeling is easily implemented by a parallel prefix computation (for the sake of completeness, the definition of parallel prefix is reviewed below).

Lemma 2.1 *Let M be an array of elements, $M = \{m_1, m_2, \ldots, m_n\}$. Let* parallel prefix *be the problem of computing all the partial sums $s_i = m_1 \oplus m_2 \oplus \cdots \oplus m_i$, where $\oplus$ is an associative operation. The parallel prefix problem can be solved in $O(\log n)$ time using $O(n/\log n)$ processors on an EREW PRAM.*

Proof: See [4,5]. □

If $A_i \cap A_j = A_j$, and $i \neq j$, we say that A_j is *contained* in A_i. It is easy to see that those arcs that are contained in some other arcs can be ignored, since there is always a minimum cover for C that does not use any of them. Thus in the rest of this section, we assume that no arc in S is contained in some other arc (we call this the *noncontainment property*). A preprocessing procedure for removing all contained arcs in S is given later, in Section 3. The following was observed in [6] (for notational convenience, $y_{n+1} = y_1$).

Lemma 2.2 *If the noncontainment property holds, then for any $i \in \{1, 2, \ldots, n\}$, if we start at y_i and move clockwise along C, then the first y_j $(j \neq i)$ that we encounter is y_{i+1}.*

Proof: Suppose not, i.e., suppose we encounter a y_j with $j \neq i+1$. Then we distinguish two cases: (i) if a clockwise trip from x_i to y_j encounters x_j then A_j is contained in A_{i+1} and this contradicts the noncontainment property; (ii) if a clockwise trip from x_i to y_j does not encounter x_j then A_i is contained in A_j and this also contradicts the noncontainment property. □

Therefore, in the clockwise direction of C, the next x_j encountered after x_i is x_{i+1}, and the next y_j encountered after y_i is y_{i+1}.

As in [6], we define a function SUC: $S \rightarrow S$, called *successor*, as follows: $SUC(A_i) = A_j$, if and only if x_j is the last of the x_k's encountered by a clockwise sweep from x_i to y_i (possibly it is x_i itself). Note that this implies that $SUC(A_i)$ and A_i overlap. If $SUC(A_i) = A_i$ then we can obviously conclude that there is no cover for C from the arcs of S. We define the *inverse* of SUC, SUC^{-1}, by $SUC^{-1}(A_j) = \{A_i \in S \mid SUC(A_i) = A_j\}$. Obviously, we can have $|SUC^{-1}(A_j)| > 1$. The computation of SUC and SUC^{-1} is given in Section 3.

The outline of our algorithm is now given. The details of implementing the steps below are described in Section 3 and Section 4.

Algorithm Min-Cover(S, flag)

Input: A set $S = \{A_1, A_2, \ldots, A_n\}$ of n circular-arcs on circle C, each of which is specified by its two endpoints. The integer flag is 1 if the endpoints of the arcs in S are given sorted, and 0 otherwise.

Output: A subset of S whose arcs form a minimum cover for C.

The main steps:

(1) If flag is 0, then sort the endpoints of the arcs in S.

(2) Eliminate all contained arcs in S (Section 3).

(3) Compute $SUC(A_j)$ for each $A_j \in S$ (Section 3). If for some A_j, $SUC(A_j) = A_j$, then report "no cover exists in S" and stop the algorithm.

(4) Compute $SUC^{-1}(A_j)$ for every $A_j \in S$ (Section 3).

(5) Execute the main procedure to find and report a minimum cover (Section 4).

end.

Theorem 2.3 *Given a set S of n circular-arcs on a circle, algorithm* **Min-Cover** *solves the minimum circle-cover problem in $O(\log n)$ time using $O(n/\log n)$ processors in the EREW PRAM model if the set of endpoints of the arcs in S is given sorted, and using $O(n)$ processors otherwise. The algorithm is optimal to within a constant factor.*

Proof: The optimality follows from the lower bounds shown in [6]. The correctness and the time and processor complexities follow from the descriptions to be presented in Section 3 and Section 4. □

3 Preprocessing

In this section, we show how to perform three preprocessing tasks: (i) the elimination of contained arcs; (ii) computing the *SUC* function; (iii) computing SUC^{-1}. We assume that we are given, in addition to the set of arcs S, the sorted array E containing the $2n$ endpoints of the arcs in S. These appear in E in the same order they are encountered by a clockwise scan of the circle, beginning at x_1. We show in what follows that tasks (i)–(iii) can each be done in $O(\log n)$ time using $O(n/\log n)$ processors, in the EREW PRAM model.

3.1 Eliminating Contained Arcs

This subsection describes the procedure for eliminating contained arcs. Let the *rank* of an endpoint in the array E be i iff it occupies position i in E. Let W denote the set of arcs in $S - \{A_1\}$ that contain x_1. Let y' denote the largest-ranked (in E) y_j for which $A_j \in W$. For the example of Figure 1a, y' is y_8, whose rank in E is 5. Remove from S every arc A_i that is contained in the portion of the circle going from x_1 (clockwise) to y' (such an arc is surely contained in some arc in W), then update E by deleting from it the endpoints of the arcs so removed. The rest of this subsection assumes that this has already been done (and S and E modified accordingly). For the example of Figure 1a, arc A_2 is removed because it is contained in the portion of the circle going clockwise from x_1 to y_8, resulting in Figure 1b.

Let E_1 be obtained from E by removing from it every y_j for which $A_j \in W$. Let E_2 be obtained from E by only keeping in it the y_j's for which $A_j \in W$. Let E_1E_2 denote the concatenation of E_1 with E_2. For the situation of Figure 1b, we would get

$$E_1E_2 = x_1x_3y_1x_4y_3x_5x_6y_4y_6x_7y_5x_8y_7x_9y_9y_8.$$

Intuitively, what we are doing by constructing E_1E_2 is "linearizing" the circular problem by "unrolling" it on a line. If W were empty then clearly we would achieve this by simply "opening"

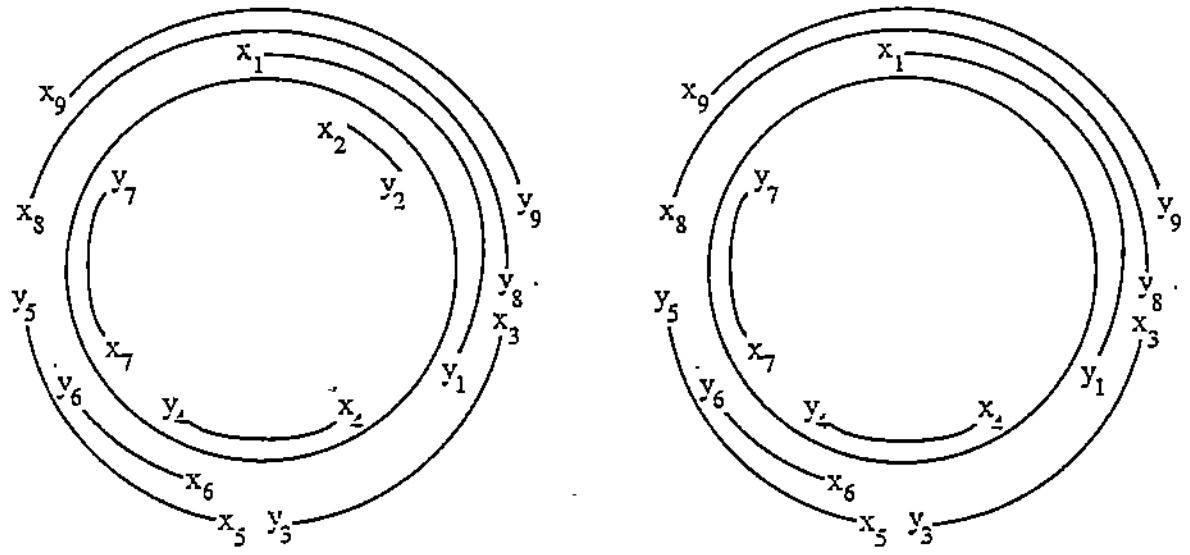

Figure 1: Illustrating the step before linearizing the problem.

the circle (i.e., cutting it open) at x_1. In other words, if W is empty then the array E already constitutes a linearization of the circular problem (in fact, E_1E_2 equals E in that case). However, if W is not empty then the way to linearize the problem is to make *two* clockwise trips around the circle, starting from position x_1: during the first trip we only look at the endpoints of the arcs in $S - W$ *and* at the first endpoints of the arcs in W, while in the second trip we only look at the second endpoints of the arcs in W. Thus the first trip sees the sequence E_1, while the second trip sees E_2.

The sequence E_1E_2 has the property that an arc A_i is contained if and only if it is contained in the linear (i.e., noncircular) sequence E_1E_2. In other words, A_i is contained if and only if there exists a j such that, in E_1E_2, x_j occurs before x_i and y_j occurs after y_i. We now turn our attention to the detailed implementation of these ideas.

Clearly, we can obtain each of W, E_1, E_2, and E_1E_2 in $O(\log n)$ time and $O(n/\log n)$ EREW PRAM processors, by using parallel prefix.

Once we have E_1E_2, eliminating contained arcs is done as follows.

(1) Using parallel prefix, compute for each element of E_1E_2 its rank in array E_1E_2. For the example we are using, the rank of y_5 is 11.

(2) Assign to every x_i in E_1E_2 a *weight* equal to the rank of y_i in E_1E_2. For the example we are using, the weight of x_5 is 11.

(3) Assign to every y_i in E_1E_2 a weight of zero.

(4) Do a parallel prefix on the weighted array E_1E_2 to find, for every x_i, the largest weight that occurs before it. If that weight is larger than x_i's own weight then A_i is contained; otherwise it is not.

(5) Remove all contained arcs from S, and delete their endpoints from E.

(6) Relabel the surviving arcs in S so that their indices are in consecutive order (i.e., $A_1, A_2, \ldots$).

Correctness of the above procedure follows from the discussion in the paragraph preceding it. Its time and processor complexities are those of parallel prefix: $O(\log n)$ time and $O(n/\log n)$ EREW PRAM processors.

From this point on we assume that we have already performed the above procedure for eliminating the contained arcs from S and relabeling the surviving arcs in S so that their indices are in consecutive order. For simplicity of notation, we still use n to denote the number of arcs in S.

3.2 Computing the *SUC* Function

We compute the *SUC* function as follows.

(1) In the array E, let the weight of each x_i be i, and let the weight of every y_i be zero.

(2) Do a parallel prefix on the weighted E to find, for every y_i, the largest weight that occurs before it in E. If that weight is (say) j for a particular y_i, then $SUC(A_i) = A_j$.

Correctness of the above procedure follows from the fact that the largest weight that occurs before y_i in E is the subscript j of x_j such that $x_j \in A_i$ and the clockwise sweep from x_j to y_i encounters no other x_k (possibly $j = i$). It can clearly be implemented in $O(\log n)$ time using $O(n/\log n)$ EREW PRAM processors.

After the *SUC* function has been computed, we check whether for some i we have $SUC(A_i) = A_i$. If so then we immediately stop and decide that there is no circle cover (because in that case the portion of the circle immediately after y_i in the clockwise direction is not covered by any arc). Otherwise, we proceed to find a minimum cover of C as explained in the rest of this paper. (This test is easily done in $O(\log n)$ time using $O(n/\log n)$ EREW PRAM processors.)

3.3 Computing SUC^{-1}

Recall that for every A_j, $SUC^{-1}(A_j) = \{A_i \in S \mid SUC(A_i) = A_j\}$. The next lemma is useful in computing SUC^{-1}efficiently.

Lemma 3.1 *Let A_i and A_j be such that x_j occurs on A_i, and $SUC(A_i) = SUC(A_j)$. Let A_t be such that x_t occurs on $A_i - A_j$ (i.e., on the portion of A_i that is not covered by A_j). Then $SUC(A_t) = SUC(A_i)$.*

Proof: Since x_t occurs on $A_i - A_j$, by Lemma 2.2, y_t must occur on $A_j - A_i$. This and the fact that $SUC(A_i) = SUC(A_j)$ imply that $SUC(A_t) = SUC(A_i)$. □

The above lemma implies that for every A_j, the arcs in $SUC^{-1}(A_j)$ occur around C consecutively. Therefore we can compute SUC^{-1} in $O(\log n)$ time using $O(n/\log n)$ EREW PRAM processors simply by "marking" on S the indices i at which $SUC(A_i) \neq SUC(A_{i+1})$. More specifically,

we compute for each j the pair (l_j, r_j) such that $SUC^{-1}(A_j) = \{A_{l_j}, A_{l_j+1}, \ldots, A_{r_j}\}$, with the notational convention that $A_{n+i} = A_i$ for every $i \in \{1, 2, \ldots, n\}$. This is done as follows.

(1) For every A_i, initialize $SUC^{-1}(A_i) = \emptyset$.

(2) For every i, compare $SUC(A_i)$ and $SUC(A_{i+1})$: if $SUC(A_i) = A_j \neq SUC(A_{i+1}) = A_k$, then we set $r_j = i$ and $l_k = i + 1$.

Note also that, given the pair (l_i, r_i), one processor can easily obtain $|SUC^{-1}(A_i)|$ in constant time.

4 The Main Procedure

After the preprocessing of Section 3, the noncontainment property of S is ensured and SUC and SUC^{-1} are available. Using SUC and SUC^{-1}, we compute the minimum cover of C from S in the main procedure, which is described in this section.

A greedy algorithm for finding a cover of C is given in [6], and when started at A_j, it produces a (not necessarily optimal) cover $S(A_j)$ of C as follows:

```
let     S(A_j) = {A_j},
        B = A_j;
while   S(A_j) does not form a cover do
        let B = SUC(B);
        add B to S(A_j);
        endwhile.
```

In what follows we shall use $S(A_j)$ to denote the cover produced when the above greedy procedure terminates. Again, we let W be the set of arcs in $S - \{A_1\}$ that contain x_1.

Lemma 4.1 *For some $A_i \in W$, $S(A_i)$ is a minimum cover.*

Proof: Easy, and omitted. □

Thus our problem is that of simulating, in parallel for all $A_i \in W$, the sequential greedy algorithm. Once we have $S(A_i)$ for all $A_i \in W$, we choose a minimum-cardinality $S(A_i)$ as our minimum cover. This is done as follows.

Create a new copy of each $A_i \in W$, denoted by $New(A_i)$ (without discarding the old copy A_i). Let $New(W)$ denote the set of new copies of the elements in W. Then we modify the SUC function by replacing every $SUC(A_j) = A_i$, where $A_i \in W$, with $SUC(A_j) = New(A_i)$, and for every $New(A_i)$ setting $SUC(New(A_i))$ equal to $\emptyset$. Figure 2b depicts the modified SUC function corresponding to Figure 2a. This new SUC function defines an in-forest F in which $SUC(A_k)$ is the parent of A_k. There are $|W|$ trees in F, whose roots are the elements of $New(W)$,

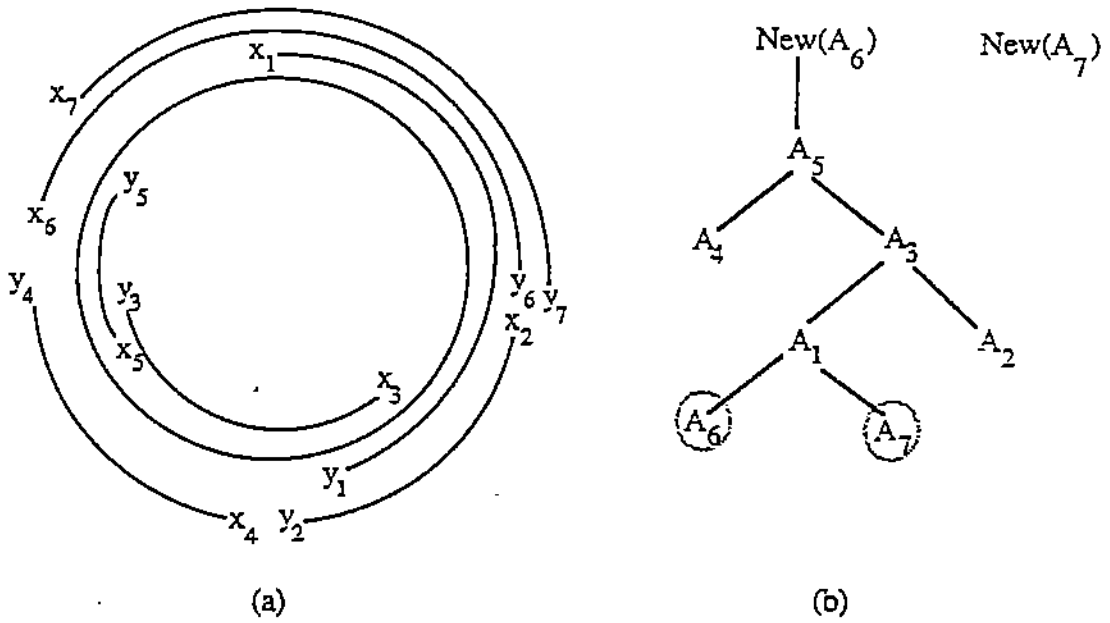

Figure 2: Illustrating the modified *SUC* function.

and whose leaves include all the elements of W, as well as other leaves. For every node A_k of F, let $Root(A_k)$ be the element of $New(W)$ that is at the root of the tree containing A_k, let $ChildRoot(A_k)$ be the child of $Root(A_k)$ that is ancestor of A_k, and let $Depth(A_k)$ be the depth of A_k in its tree (i.e., the number of nodes on the A_k to $Root(A_k)$ path in F). For example, in Figure 2, $Root(A_6) = Root(A_7) = New(A_6)$, and $ChildRoot(A_6) = ChildRoot(A_7) = A_5$. Before proceeding with the exposition, let us observe that the arrays *Root*, *ChildRoot* and *Depth* can easily be computed in $O(\log n)$ time and with $O(n/\log n)$ EREW PRAM processors using the Euler Tour technique [7] in conjunction with optimal parallel list-ranking [3] (we can use the Euler Tour technique because we have SUC^{-1}, which provides the list of children for every node in F). Assume that this has already been done.

Now, for each node (say, A_k) of F, tag that node as being "good" or "bad" according to the following rule: if $A_k \in W$, then A_k is good; otherwise it is bad. Thus A_k is bad if it is not a leaf, or a leaf that is not in W. The good nodes in Figure 2b are circled. Every good leaf A_k can tell the value of $|S(A_k)|$ by testing whether the second endpoint of $ChildRoot(A_k)$ occurs on A_k or not: if the second endpoint of $ChildRoot(A_k)$ does not occur on A_k, then $|S(A_k)| = Depth(A_k)$ and $S(A_k)$ is the A_k to $Root(A_k)$ path in F; otherwise $|S(A_k)| = Depth(A_k) - 1$ and $S(A_k)$ is the A_k to $ChildRoot(A_k)$ path in F. In Figure 2, A_7 is an example of the first case ($|S(A_7)| = Depth(A_7) = 5$), while A_6 is an example of the second case ($|S(A_6)| = Depth(A_6) - 1 = 4$). It follows from these remarks that obtaining an $A_k \in W$ that has a minimum $|S(A_k)|$ is easily done in $O(\log n)$ time and with $O(n/\log n)$ EREW PRAM processors. Once we have such an A_k, its $S(A_k)$ (which is a minimum cover) can easily be retrieved within the same time and processor bounds, since it is defined by a path in F and can thus be traced using (again) the Euler Tour technique [7] in conjunction with optimal parallel list-ranking [3].

5 Conclusion

We gave a parallel algorithm in the EREW PRAM computational model for the minimum circle-cover problem. Our algorithm runs in $O(\log n)$ time using $O(n)$ processors if the input arcs are not sorted and using $O(n/\log n)$ processors otherwise. Since the time and processor products of our algorithm are within a constant factor of the sequential lower bounds ($\Omega(n \log n)$ for unsorted arcs and $\Omega(n)$ for sorted arcs), this algorithm is optimal. The previous best known parallel algorithm runs in $O(\log n)$ time using in the worst case $O(n^2)$ processors in the (stronger) CREW PRAM computational model. Hence, our solution improves the processor complexity by a factor of n, and by using a less powerful computational model.

Acknowledgement. The authors are grateful to an anonymous referee for his useful comments.

References

[1] A.A. Bertossi, Parallel circle-cover algorithms, *Inform. Process. Lett.* 27 (1988) 133-139.

[2] R. Cole, Parallel merge sort, *SIAM J. Comput.* 17 (4) (1988) 770-785.

[3] R. Cole and U. Vishkin, Approximate parallel scheduling. Part I: the basic technique with applications to optimal parallel list ranking in logarithmic time, *SIAM J. Comput.* 17 (1) (1988) 128-142.

[4] C.P. Kruskal, L. Rudolph and M. Snir, The power of parallel prefix, *IEEE Trans. on Computers* C-34 (1985) 965-968.

[5] R.E. Ladner and M.J. Fischer, Parallel prefix computation, *J. of ACM 27 (4)* (1980) 831-838.

[6] C.C. Lee and D.T. Lee, On a circle-cover minimization problem, *Inform. Process. Lett.* 18 (1984) 109-115.

[7] R.E. Tarjan and U. Vishkin, An efficient parallel biconnectivity algorithm, *SIAM J. Comput.* 14 (4) (1985) 862-874.