Part I

COMPUTER SCIENCE

BIT 24 (1984), 134–150

A PARALLEL SEARCH ALGORITHM FOR DIRECTED ACYCLIC GRAPHS

RATAN K. GHOSH and G. P. BHATTACHARJEE

Mathematics Department, Indian Institute of Technology, Kharagpur 721302, INDIA

Abstract.

A parallel algorithm for depth-first searching of a directed acyclic graph (*DAG*) on a shared memory model of a SIMD computer is proposed. The algorithm uses two parallel tree traversal algorithms, one for the preorder traversal and the other for the *r*postorder traversal of an ordered tree. Each of these traversal algorithms has a time complexity of $O(\log n)$ when $O(n)$ processors are used, n being the number of vertices in the tree. The parallel depth-first search algorithm for a directed acyclic graph G with n vertices has a time complexity of $O((\log n)^2)$ when $O(n^{2.81}/\log n)$ processors are used.

Keywords: Parallel algorithm, depth-first, antilexicographic, graph, search, acyclic, spanning tree, traversal, preorder, *r*postorder.

1. Introduction.

There are two principal ways to traverse a general tree, viz. the preorder traversal and the postorder traversal. In the preorder traversal a vertex of the tree is visited before visiting any of its sons and in the postorder traversal a vertex is visited after visiting all of its sons. There are optimal sequential algorithms to tackle these traversal problems (see e.g. [6]). Obviously every vertex is to be visited at least once in a sequential algorithm for tree traversal, and this is the reason that optimal sequential traversal algorithms for trees have time complexity $O(n)$ where n is the number of vertices in the tree. As far as parallel algorithms are concerned, Wyllie [17] suggested $O(\log n)$ algorithms for traversals of binary trees having n vertices when $O(n)$ processors are used.

In this paper parallel algorithms for traversals of a general ordered tree are proposed. In section 3 of the paper it is shown how the binary tree representation of a general tree with n vertices can be obtained in $O(k \log n)$ time using $O(n^{1+1/k})$ processors, for integer $k \geqq 2$. Then section 4 includes a straightforward parallel algorithm for the preorder traversal of a general ordered tree assuming its binary tree representation is available already. In section 5 another straightforward parallel algorithm for *r*postorder traversal of a

Received May 3, 1983. Revised October 3, 1983.

general ordered tree is presented. By *r*postorder traversal is meant a traversal of the corresponding binary tree in the following way, starting with the root:

> *r*postorder traverse the right subtree,
>
> visit the root vertex,
>
> *r*postorder traverse the left subtree.

Each of these traversal algorithms has a time complexity of $O(\log n)$ when $O(n)$ processors are available. The reason for presenting an *r*postorder traversal algorithm instead of a postorder traversal algorithm is that the parallel algorithm for depth-first search of a *DAG* presented in section 6 presupposes the *r*postorder traversal algorithm.

Depth-first search of a *DAG* is a more general problem than the traversal problems for trees and has wide applications [4, 11, 15, 16]. The best sequential algorithm for this problem is given by Tarjan [15]. The time complexity of Tarjan's algorithm is $O(n+e)$, where n and e are respectively the number of vertices and the number of edges in the *DAG*. In fact Tarjan's algorithm covers a much broader spectrum since it is designed to tackle both undirected graphs and directed graphs with cycles. The first attempt for parallel graph search algorithms was done by Reghbati and Corneil [11], but they remarked that depth-first search is not a suitable search technique for graphs, both sparse and dense, in the context of parallel processing. Eckstein and Alton [4] later proposed an $O(n)$ parallel algorithm for depth-first search of graphs with n vertices using $O(e)$ processors. The gain in time complexity by using so many processors is clearly not appreciable in comparison with that of Tarjan's algorithm. This fact seems to indicate that at least the remark of Reghbati and Corneil is correct to the extent that for sparse graphs depth-first search is not a good graph search technique as far as the parallel processing environment is concerned. But in section 6 of this paper we propose an $O((\log n)^2)$ parallel algorithm for depth-first search of a *DAG* with n vertices using $O(n^{2.81}/\log n)$ processors which has much better asymptotic time bound compared to Eckstein and Alton's parallel algorithm and also compares favourably with Tarjan's sequential algorithm.

2. The model of computation.

A shared memory model of Single Instruction-stream, Multiple Data-stream (SIMD) computer is used as the computational model. This model has been used by many other authors for a wide variety of problems such as evaluation of polynomials [9], solution of recurrence relations [8], sorting and merging [5, 10] and graph theoretic problems [4, 7, 11, 12].

In this model a *master* processor is capable of broadcasting instructions to each one of a number of other processors known as *slave* processors. The slave processors have substantial local memory and have access to an unbounded global memory. This structure of a shared memory model of a SIMD computer includes the idea of synchronous computation of ILLIAC IV [2] and also captures the elegent memory structure of *Cm** [14]. The master processor broadcasts an instruction which all the slave processors in enabled mode execute simultaneously and synchronously over different data. While measuring the time complexity of execution of a certain algorithm designed for this model we assume that the cost of one such simultaneous and synchronous execution of an instruction described above is one unit of time irrespective of the number of slave processors that may be active.

3. The parallel binary tree representation algorithm.

The parallel tree traversal algorithms described in the later sections of this paper accept the binary tree representation of the given input tree. However, a tree T for which these traversals are sought is usually available in the form of its adjacency lists or more frequently, since every vertex in a tree has a unique predecessor, just the predecessor or the *parent* of every vertex in T is specified. Hence as a preprocessing step the parallel algorithm given in this section obtains the binary tree representation for a given tree specified by the vertices and their corresponding parents. For presenting the algorithm we use Pidgin ALGOL [1] with a few additional constructs adopted for presentation of parallel algorithms. A parallel algorithm is indicated by the following statement:

```
    for all processor $p = 1$ to $n$ do
    prl-begin................ prl-end;
```

which means that the processors with indices 1, 2,...,n work in parallel on the algorithm between **prl-begin** and **prl-end**.

Algorithm: structure-binary.

Purpose: To obtain the binary tree representation of a tree T.
Input: parent (v) for each vertex v in T.
Output: leftson (v), rightson (v) and father (v) for each vertex v in T where

$$\text{father } (v) = \begin{cases} \text{parent } (v), \text{ if rightson } (v) = 0 \\ 0, \text{ otherwise} \end{cases}$$

```
procedure structure-binary (parent, leftson, rightson, father);
  begin comment The vertices of tree $T$ are assumed to be numbered
        arbitrarily from 1 through $n$ and the root vertex $r$ being identified by
        parent $(r) = 0$;
```

```
    for all processor $p = 1$ to $n$ do
      prl-begin comment The vertices of $T$ are first sorted so that those having
                the same parent get consecutive ranks in descending order of
                their respective numbers. A key is computed for each vertex,
                which realizes the desired ordering when vertices are sorted after
                ascending values of associated keys;
        if parent $(p) \neq 0$ then key $(p) = n \times$ parent $(p)+n-p$
        else key $(p) := n \times$ parent $(p)$
      prl-end;
    comment Actual sorting is accomplished through a parallel sorting
    algorithm (see, e.g. [10]). It is assumed that procedure parallel-sort based
    on such a parallel sorting algorithm produces an array vertex $(1:n)$
    representing the array of vertices of $T$ sorted after ascending values of their
    respective keys;
    invoke parallel-sort (key, vertex);
    for all processor $p = 1$ to $n$ do
      prl-begin
        leftson (vertex $(p)$) := rightson (vertex $(p)$) := father (vertex $(p)$) := 0;
        if $p = 1$ and parent (vertex $(p)$) $\neq 0$ then
          begin comment The vertex of $T$ represented by vertex(1) is taken as
                the leftson of its parent if vertex(1) itself is not the root
                vertex;
            leftson (parent (vertex $(p)$)) := vertex $(p)$
          end;
        if $p < n$ and parent (vertex $(p)$) = parent (vertex $(p+1)$) then
          begin comment The younger brother is the rightson of its immediate
                elder brother;
            rightson (vertex $(p)$) := vertex $(p+1)$
          end
        else if $p < n$ and parent (vertex $(p+1)$) $\neq 0$ then
              begin comment Two neighbouring vertices having different
                    parents implies that the vertex occurring later in the array,
                    i.e. vertex$(p+1)$, is the leftson of its parent;
                leftson (parent (vertex $(p+1)$)) := vertex $(p+1)$
              end;
        if rightson (vertex $(p)$) = 0 then
              begin comment The father relations are set up;
                    father (vertex $(p)$) := parent (vertex $(p)$)
              end
      prl-end
end structure-binary;
```

The algorithm is a straightforward parallel implementation of the sequential method for obtaining the binary tree representation of an ordered tree and finds leftson, rightson relations for every vertex v of an ordered tree T. The only exception from the corresponding sequential method is in setting of father relations. The vertices which have a rightson have their respective father relations showing null. In fact algorithm *structure-binary* obtains the binary tree-like representation of the ordered tree in fig. 1 as shown in fig. 2. The null relations in fig. 2 are not shown. Considering time complexity of *structure-*

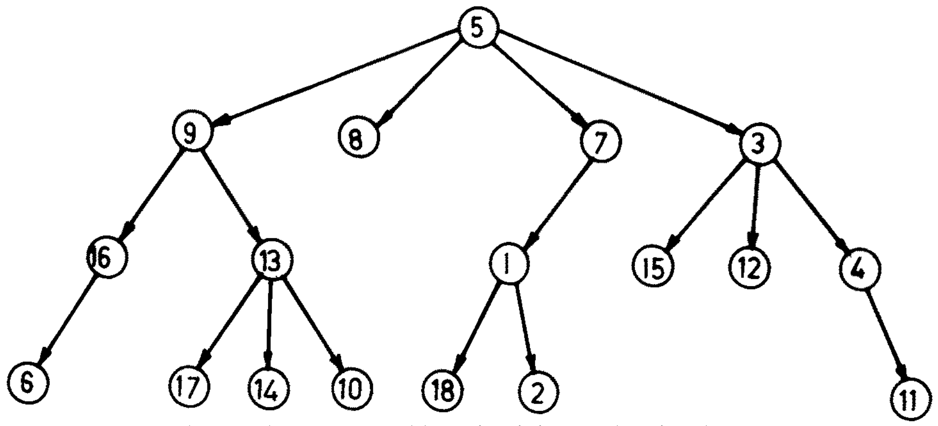

Fig. 1. The input tree with vertices being numbered arbitrarily.

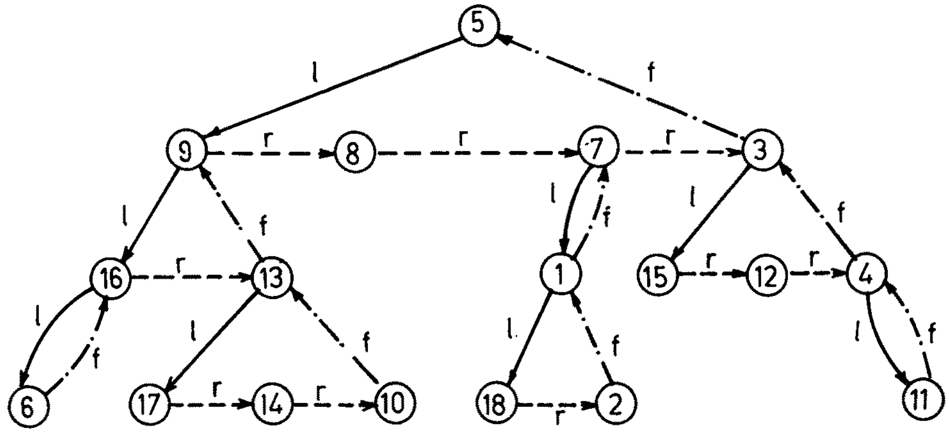

Fig. 2. The binary tree-like representation of the tree of fig. 1. The leftson, the rightson and the father relations are shown by labeled arrows $\xrightarrow{\;l\;}$, $\overset{r}{\dashrightarrow}$ and $\overset{f}{\dashrightarrow}$ respectively.

binary we see that except for the steps where *parallel-sort* is used all other steps contribute only terms of $O(1)$ when $O(n)$ processors are available. The parallel sorting algorithm due to Preparata [10] has a time complexity of $O(k \log n)$ when $O(n^{1+1/k})$, for integer $k \geqq 2$, processors are available. Thus the time complexity of *structure-binary* is $O(k \log n)$ when $O(n^{1+1/k})$ processors are employed.

4. The parallel preorder traversal algorithm.

Once the given ordered tree T is obtained in its binary tree-like representation the task of obtaining the preorder traversal list of the vertices of T is made fairly simple. The idea is that the leftson of a vertex v, if it exists, should appear as the immediate next vertex to v in the preorder list. Else, the rightson of v, if it exists, appears as the immediate next vertex to v. In case v has neither a leftson nor a rightson the vertex to appear as the immediate next vertex to v is located by first locating the youngest ancestor of v in T having a rightson. Then this rightson of the ancestor vertex of v is made the immediate next vertex of v. So far as a vertex has either a leftson or a rightson the task is quite trivial. But when it has neither, the locating of the youngest ancestor with a non-null rightson is not simple. This task is achieved by applying folding on father relations. The folding updates non-null father relations of the vertices to their penultimate depths. Here the depth k of a relation R on an argument x implies that $R^k(x) = 0$ but $R^{k-1}(x) \neq 0$. The detailed preorder traversal algorithm described below is a straightforward parallel implementation of the idea above.

Algorithm: preorder-traversal.

Purpose:	To obtain the preorder traversal list and the rank of each vertex in the list.
Input:	The binary tree-like representation of the given general ordered tree T.
Output:	The preorder traversal list and the ranks as desired.

```
procedure preorder-traversal (vertex, leftson, rightson, father, pre);
  begin comment Each vertex vertex($v$) in $T$ contains another field viz. next
        (vertex($v$)) which is initially set to 0. But at the termination of the
        algorithm the vertices of $T$ are linked around by "next" fields to produce
        the singly linked list defining the preorder traversal list of $T$;
   for all processor $p = 1$ to $n$ do
     prl-begin
       next (vertex($p$)) := 0;
       for $\lceil \log n \rceil$ iterations do
         begin comment The folding on father relations of every vertex is
                 accomplished by this loop;
           if father (vertex($p$)) $\neq 0$
             and father (father (vertex($p$))) $\neq 0$ then
             father (vertex($p$)) := father (father (vertex($p$)))
         end;
       comment The vertices of $T$ are now linked around by setting next
       (vertex($v$)) for each vertex vertex($v$);
       if leftson (vertex($p$)) $\neq 0$ then
```

```
            next (vertex (p)) := leftson (vertex (p))
        else if rightson (vertex (p)) ≠ 0 then
                next (vertex (p)) := rightson (vertex (p))
        else if father (vertex (p)) ≠ 0 then
                  next (vertex (p)) := rightson (father (vertex (p)));
        comment To obtain the preorder ranks a technique similar to that for
        computing the positions of elements of a singly linked list as given in
        [17] is used;
        pre (vertex (p)) := 1; far (vertex (p)) := 0;
        if next (vertex (p)) ≠ 0 then
            far (next (vertex (p))) := vertex (p);
        for ⌈log n⌉ iterations do
          if far (vertex (p)) ≠ 0 then
            begin comment pre (vertex (v)) is updated to give the preorder rank
                  of vertex (v) in T;
              pre (vertex (p)) := pre (vertex (p)) + pre (far (vertex (p)));
              far (vertex (p)) := far (far (vertex (p)))
            end
      prl-end
end   preorder-traversal;
```

Except for the **for** loop of the *preorder-traversal* procedure all steps contribute terms of $O(1)$ to time complexity when $O(n)$ processors are used. Hence the time complexity of the parallel preorder traversal algorithm is $O(\log n)$ when $O(n)$ processors are used. Figs. 3 and 4 illustrate the working of *preorder-traversal* procedure assuming that input tree is given in fig. 1.

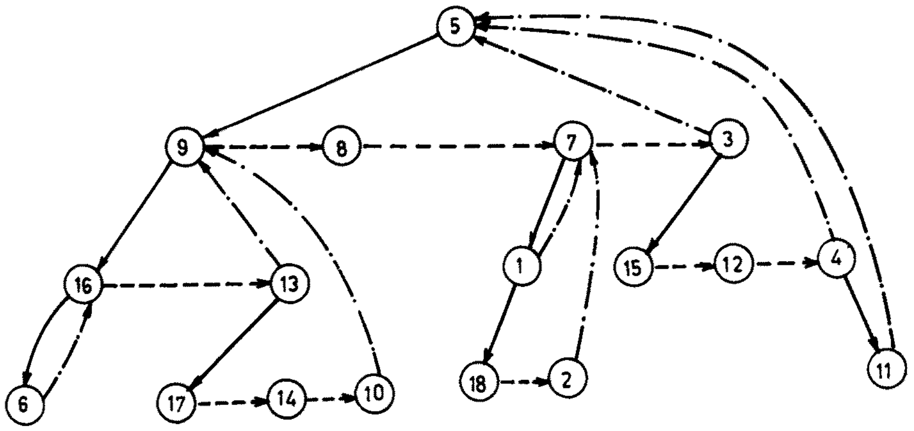

Fig. 3. The result of the folding on the father relations of the binary tree-like representation given in fig. 2.

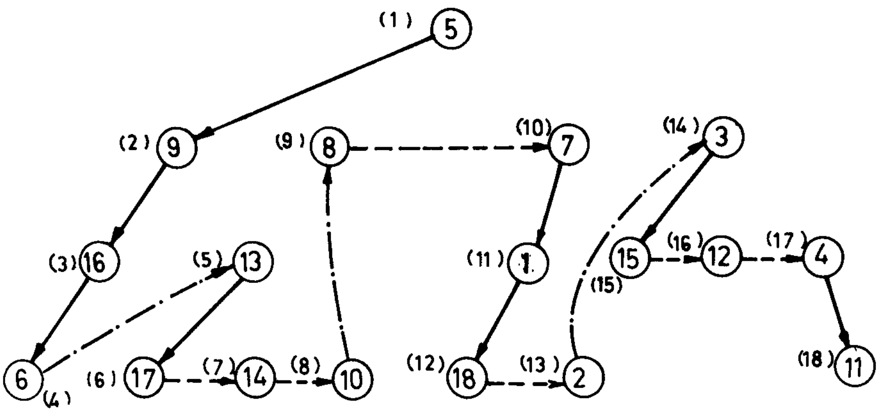

Fig. 4. The preorder traversal list of the tree of fig. 1 with preorder ranks of vertices shown alongside within parentheses.

5. The parallel *r*postorder traversal algorithm.

The parallel *r*postorder traversal algorithm described in this section also assumes that the binary tree-like representation of a given order tree is available as input. However, the folding in this case is applied on non-null leftson relations of the vertices of T. The detailed algorithm is given below in Pidgin ALGOL.

Algorithm: rpostorder-traversal.

Purpose:	To obtain the *r*postorder traversal list of vertices of a general ordered tree T and the rank of each vertex in this list.
Input:	The binary tree-like representation T.
Output:	The singly linked list of vertices of T defining the *r*postorder traversal and ranks of vertices as desired.

```
procedure rpostorder-traversal (vertex, leftson, rightson, father, rp);
 begin
   for all processor $p = 1$ to $n$ do
     prl-begin
       next (vertex($p$)) := 0;
       for $\lceil \log n \rceil$ iterations do
         begin comment Folding is applied on leftsons;
           if leftson (vertex($p$)) $\neq 0$
               and leftson (leftson (vertex($p$))) $\neq 0$ then
             leftson (vertex($p$)) := leftson (leftson (vertex($p$)))
         end;
       comment The vertices of $T$ are now linked by setting next (vertex($v$))
       for each vertex vertex($v$) to produce the rpostorder traversal list;
```

```
        if father (vertex (p)) ≠ 0 then
            next (father (vertex (p))) := vertex (p)
        else if rightson (vertex (p)) ≠ 0 and
                      leftson (rightson (vertex (p))) ≠ 0 then
                next (leftson (rightson (vertex (p)))) := vertex (p)
        else if rightson (vertex (p)) ≠ 0 then
                          next (rightson (vertex (p))) := vertex (p);
        comment The rpostorder ranks of a vertex is computed by finding the
        position of this vertex in the rpostorder traversal list of T;
        rp (vertex (p)) := 1; far (vertex (p)) := 0;
        if next (vertex (p)) ≠ 0 then far (next (vertex (p))) := vertex (p);
        for ⌈log n⌉ iterations do
           if far (vertex (p)) ≠ 0 then
              begin comment rp (vertex (v)) is updated to give rpostorder rank
                    of vertex (v) in T;
                 rp (vertex (p)) := rp (vertex (p)) + rp (far (vertex (p)));
                 far (vertex (p)) := far (far (vertex (p)))
              end
       prl-end
end    rpostorder-traversal;
```

For *rpostorder-traversal* folding is applied on leftson relations of the vertices of T. So if we consider a mirror image of the process involved in this algorithm it becomes identical to *preorder-traversal* with leftson and father relations having their roles interchanged. Hence the correctness of this algorithm follows directly from *preorder-traversal*. From the presentation of the algorithm it is evident that it has a time complexity of $O(\log n)$ when $O(n)$ processors are used. Time complexity of the algorithm is attributed to the folding carried by the **for** loop. The folding can require at most $O(\log n)$ time because the depth of the leftson relation can be at most $n-1$ as there are n vertices in T.

6. The parallel depth-first search algorithm for a *DAG*.

In this section a parallel algorithm for finding a depth-first search (*dfs*) spanning tree of a *DAG* with a root vertex is presented. The algorithm also obtains the corresponding *dfs* ordering of the vertices.

It is assumed that the vertices of the input *DAG* are arbitrarily numbered from 1 through n. The *DAG* is available in the form of its adjacency matrix $A(1:n, 1:n)$. The underlying idea is that a spanning tree of the given *DAG* is obtained by specifying for every vertex v of the *DAG* another vertex w, belonging to the set of predecessors of v, as its parent in the spanning tree. It may be noted that the root vertex r is identified by the fact that the parent of r

is null. After initially obtaining any spanning tree, T_0, of the *DAG* it is converted to a *dfs* spanning tree T of the *DAG*. Transformation of T_0 to T requires that the vertices be first ordered in some sort of a left to right order with respect to T_0. Then for each vertex v considering all its predecessors in the *DAG* a unique predecessor is chosen as the parent of v in T so that T admits no edge in the *DAG* directed from left to right with respect to itself. The selection of the unique predecessor for each vertex v is achieved with the help of the Reaching Relations (*RR*) of the vertices in the *DAG* and the *r*postordering with respect to T_0. Incidentally the *r*postordering of the vertices with respect to T_0 also enforces a left to right ordering of the vertices. The process of selection of a unique predecessor for each vertex v in order to obtain T can be explained briefly as follows. Assuming that the vertices are now referred to by their *r*postorder numbers with respect to T_0 the *RR* in the *DAG* for every vertex v is obtained in the form of an array of 0's and 1's. The presence of a 1 in position w of the array indicates that the vertex w can reach v in the *DAG*. Then for every v considering all its predecessors we select the predecessor w whose *RR* is *antilexicographically* largest compared to those of other predecessors, where the antilexicographic order of the *RR* is equivalent to the lexcicographic order of the *RR* taken in the reverse order.

Next if the *RR* of w is also antilexicographically greater compared to an array of n elements consisting of 0's in all positions except the position corresponding to v itself, which is here referred to as the Local Reaching Relation (*LRR*), w is set as parent (v) in T. Otherwise we let parent(v) = parent$_0$(v), parent$_0$(v) being the parent of v in T_0. This selection process ensures that the tree path in T from the root to v is to the left of the tree path in T from the root to any of the other predecessors of v in that *DAG* when the successors of every vertex in T are arranged in their descending orders. It may be observed that for deciding which of the *RR* is antilexicographically largest, we can compute the binary integers represented by the reverse of each *RR* and find the maximum among them.

The basic steps of the algorithm are given below with appropriate explanations. However, an elaborate Pidgin ALGOL representation, as was done for algorithms of the previous sections, is omitted.

Algorithm: dfs-DAG.

Purpose:	To obtain a *dfs* spanning tree of the input *DAG* and find the corresponding *dfs* ordering of the vertices.
Input:	The adjacency matrix $A(1:n, 1:n)$ of *DAG*.
Output:	A *dfs* spanning tree T and *dfs*(v) for each vertex $v \in DAG$ giving its *dfs* ordering corresponding to T.

Step 1. Extract an arbitrary spanning tree T_0 of the input *DAG*.

The algorithm outlined below may be used to obtain T_0:

(i) Obtain the transpose $A^T(1:n, 1:n)$ of $A(1:n, 1:n)$.

(ii) Compute the cumulative sums $CS(u, 1:n)$ of each row $A^T(v, 1:n)$ for $v = 1, 2, \ldots, n$.

(iii) Set $CS(v, u) = 0$ if $A^T(v, u) = 0$, for $v = 1, 2, \ldots, n$ and for $u = 1, 2, \ldots, n$.

(iv) For $v = 1, 2, \ldots, n$ form arrays $IP(v, 1:CS(v, n))$ such that $IP(v, CS(v, u)) = u$ if $CS(v, u) \neq 0$.

(v) Set $\text{parent}_0(v) = IP(v, 1)$.

It is easy to see that except for step (ii) all other steps take $O(1)$ time and require at most $O(n^2)$ processors. A technique similar to the one given in [17] for computing the positions of the elements in a singly linked list can easily be adopted to compute the cumulative sums of an array of n numbers in time of $O(\log n)$ using $O(n)$ processors. Hence step (ii) which simultaneously computes the cumulative sums of n arrays with each array consisting of n numbers, has a time complexity of $O(\log n)$ when $O(n^2)$ processors are used. Thus the complexity of Step 1 is $O(\log n)$ when $O(n^2)$ processors are available.

Step 2. Find the *r*postorder rank $rp_0(v)$ for each vertex v with respect to T_0.

First the binary tree-like representation of T_0 is obtained using procedure *structure-binary* of section 3. Then using the binary tree-like representation of T_0 as input the *r*postorder rank of each vertex is obtained through the procedure *rpostorder-traversal* of section 5. Thus this step has time complexity $O(k \log n)$ when $O(n^{1+1/k})$ processors are used, k being an integer $\geqq 2$.

Step 3. Find a new adjacency matrix of the *DAG* in such a way that the vertices could be referred to by their rp_0 labels from here onwards.

For an implementation of this step a temporary matrix $M(1:n, 1:n)$ is built from $A(1:n, 1:n)$ such that $M(rp_0(v), rp_0(w)) = A(v, w)$ for each pair v, w, where $1 \leqq v, w \leqq n$. Then later $A(1:n, 1:n)$ is changed with the help of $M(1:n, 1:n)$ so that $A(v, w) = M(v, w)$ for every pair v, w, where $1 \leqq v, w \leqq n$. A straightforward parallel method for accomplishing the task of this step has only $O(1)$ time complexity when $O(n^2)$ processors are available.

Step 4. Obtain REACHING$(v, 1:n)$ for each vertex v as a set of 1's and 0's such that REACHING$(v, w) = 1$ implies that vertex w can reach v in the *DAG*.

For computing REACHING$(v, 1:n)$ with the property as indicated we can use a parallel matrix multiplication algorithm due to Chandra [3] and compute the transitive closure of the adjacency matrix $A(1:n, 1:n)$. Then the row v of the transpose of the transitive closure matrix yields REACHING(v, 1:n). Chandra's parallel algorithm has time complexity $O(\log n)$ when $O(n^{2.81}/\log n)$ processors are used. The computation of the transpose of the transitive closure matrix of the adjacency matrix includes $\lceil \log n \rceil$ matrix multiplications. Therefore this step has time complexity $O((\log n)^2)$ when $O(n^{2.81}/\log n)$ processors are employed. It may be noted here that Chandra utilized Strassen's [13] sequential matrix multiplication algorithm which has time complexity $O(n^{2.81})$.

Step 5. For each vertex v compute $\text{bin}(v) = \sum_{w=1}^{n} \text{REACHING}(v, w) \cdot 2^{w-1}$.

The computation of bin(v) for each v in this step is done by using a parallel algorithm for the evaluation of polynomials of degree n due to Maruyama [9]. The time complexity of Maruyama's algorithm is $O(\log n)$ when $O(n)$ processors are used. Since for each of the n vertices we need to evaluate a polynomial of degree n, using $O(n^2)$ processors bin(v) for each v is computed in $O(\log n)$ time.

Step 6. For each vertex v select the immediate predecessor w of v in DAG such that bin(w) is the maximum. If $\text{bin}(w) > 2^{v-1}$ then let parent(v) = w, else let parent(v) = $\text{parent}_0(v)$.

To obtain a w among the set of immediate predecessors, IPRED(v), of each vertex v in DAG such that

$$\text{bin}(w) = \max\{\text{bin}(u) | u \in IPRED(v)\},$$

we proceed as follows. First an array $N(v, 1{:}n)$ is set up for each v, where

$$N(v,u) = \begin{cases} \text{bin}(u), & \text{if } A(u,v) = 1 \\ 0, & \text{otherwise.} \end{cases}$$

Then for each v obtain the vertex w such that

$$N(v,w) = \max\{N(v,u) | 1 \leqq u \leqq n\}. \tag{1}$$

It may be observed that the array $N(v, 1{:}n)$ for each v can be set up simultaneously in time $O(1)$ when $O(n^2)$ processors are available. Since it takes $O(\log n)$ time to find the maximum of an array of n numbers using $O(n)$ processors, an immediate predecessor w for each vertex v which satisfies (1) can be found simultaneously in time $O(\log n)$ when $O(n^2)$ processors are used. The rest of the job in this step viz. deciding whether w is to be made parent (v) is trivial and takes $O(1)$ time when $O(n)$ processors are available. Thus the overall time complexity of this step is $O(\log n)$ when $O(n^2)$ processors are used.

Step 7. Find the preorder rank $pre(v)$ for each vertex v with respect to T and let $dfs(v) = pre(v)$.

This step is similar to Step 2; instead of procedure *rpostorder-traversal* procedure *preorder-traversal* is to be used to compute the preorder rank $pre(v)$ for each vertex v. Thus the time and processor complexities of this step and those of Step 2 are the same.

The correctness of the algorithm is shown below by Theorem 1. The complexity analysis, however, has already been done separately for each step and is therefore just summed up in a statement given in Theorem 2. The entire algorithm is illustrated with the help of an example in figs. 5, 6, 7 and Table 1.

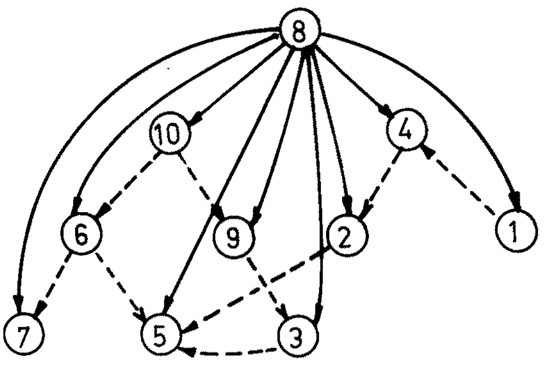

Fig. 5. The DAG containing T_0. The edges of T_0 are shown by solid lines.

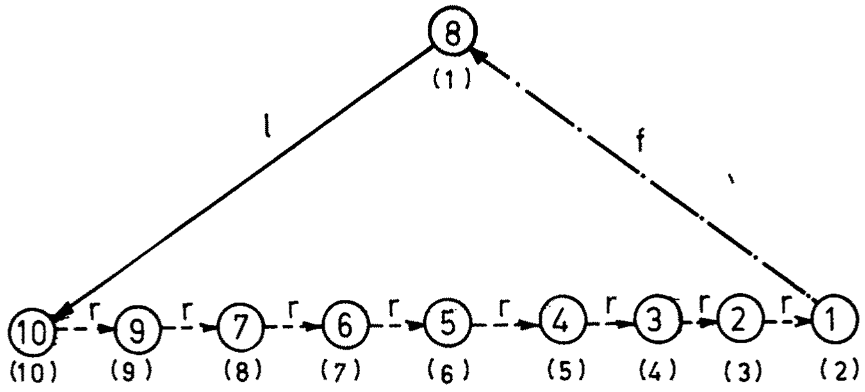

Fig. 6. The binary tree-like representation of T_0 and rp_0 labels of the vertices shown alongside, within parentheses.

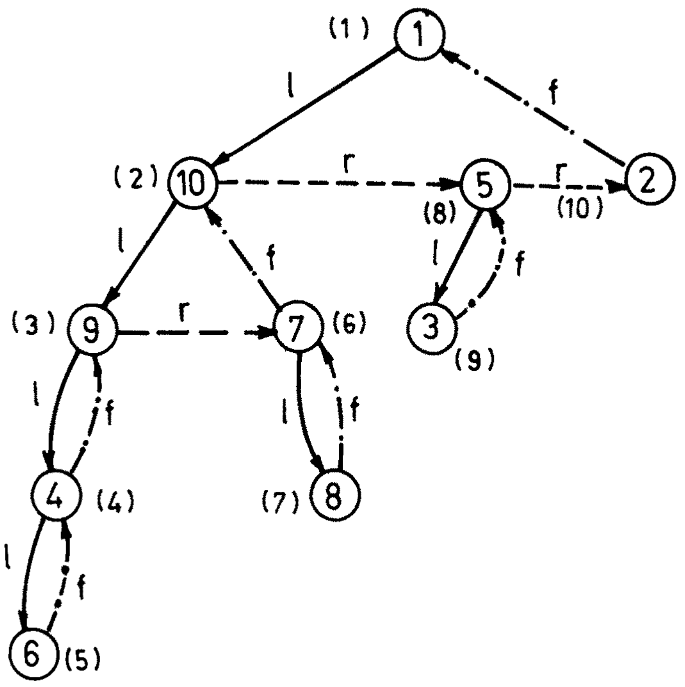

Fig. 7. The binary tree-like representation of *dfs* tree *T* of *DAG* obtained at Step 6 of procedure *dfs-DAG*. The numbers shown alongside the vertices within parentheses are their *dfs* numbers.

Table 1. *The results of computations involved in various steps of procedure dfs-DAG*

Vertex v	Immediate predecessors of v in *DAG*	$rpo(v)$	REACHING in hexadecimal	$\langle \text{bin}(rpo(v)) \rangle$	$2^{rpo(v)-1}$	Parent of v in T	*dfs*
8	ϕ	1	200	1	1	ϕ	1
10	8	10	201	513	512	8	2
9	8, 10	9	203	769	256	10	3
7	6, 8	8	20D	705	128	6	7
6	8, 10	7	209	577	64	10	6
5	2, 3, 6, 8	6	3FB	895	32	3	5
4	1, 8	5	320	19	16	8	8
3	8, 9	4	243	777	8	9	4
2	4, 8	3	3A0	23	4	4	9
1	8	2	300	3	2	8	10

THEOREM 1: *Algorithm dfs-DAG is correct.*

PROOF. The correctness of algorithm *dfs-DAG* depends on the choice of the parent for each vertex in *T*. Hence Step 6 is crucial for proving the correctness. After Step 2 the vertices are referred to by their *r*postorder ranks with respect to T_0. For the convenience of the presentation of proof, the vertices are referred to by their *r*postorder ranks with respect to T_0 and the result of a comparison among the *RR*s or the *LRR*s is simply referred to as greater or less while actually it is antilexicographically greater or less.

Let u and v be any two sons of w in T with $u > v$. From Step 6, which determines the parent of each vertex in T, it follows that either

(i) the *RR* of w is greater than both the *LRR* of v and the *RR* of any other predecessor of v, or

(ii) $w = \text{parent}_0(v)$ and no predecessor of v has the *RR* greater than the *LRR* of v.

Since $w \in \text{REACHING}(u, 1{:}n)$ the *RR* of u is greater than the *RR* of w. Therefore the condition (i) implies $u \notin \text{REACHING}(v, 1{:}n)$. As $u > v$ the *LRR* of u is greater then the *LRR* of v. So under condition (i) the *RR* of u is greater than that of v. Next consider the condition (ii). As the *LRR* of u is greater than the *LRR* of v, $u \notin \text{REACHING}(v, 1{:}n)$ and the *RR* of u is greater than the *RR* of v.

Let s be a son of v in T. Once again because of Step 6 in *dfs-DAG* either

a) the *RR* of v is greater than both the *LRR* of s and the *RR* of any other predecessor of s, or

b) $v = \text{parent}_0(s)$ and no predecessor of s has the *RR* greater than the *LRR* of s.

It has already been shown that the *RR* of u is greater than the *RR* of v. So condition (a) implies $u \notin \text{REACHING}(s, 1{:}n)$ and *RR* of u is greater than the *RR* of s. As $u \notin \text{REACHING}(v, 1{:}n)$ vertices u and v are in the different tree paths in T_0. So from the properties of *r*postorder ranks (see [16]) $u > v$ and $\text{parent}_0(s) = v$ of condition (b) imply $u > s$. Hence the *LRR* of u is greater than the *LRR* of s under condition (b). Also no predecessor of s has the *RR* greater than the *LRR* of s in (b). Therefore as the *LRR* of u is greater than the *LRR* of s, under condition (b) once again $u \notin \text{REACHING}(s, 1{:}n)$ and the *RR* of u is greater than the *RR* of s.

Extending the arguments given above, if x is a descendant of v in T, we have $u \notin \text{REACHING}(x, 1{:}n)$ and the *RR* of u is greater than the *RR* of x. If y is a descendant of u in T, the *RR* of y is greater than the *RR* of u. Therefore $y \notin \text{REACHING}(x, 1{:}n)$, where y is a descendant of u and x is a descendant of v.

Now let $T(x)$ and REACHABLE(x) denote respectively the subtree of T rooted at x and the set of vertices reachable through x in the *DAG*. So a restatement of the fact proved in the preceding paragraph using these notations is

$$T(v) \subseteq \{y \in \text{REACHABLE}(v) \cap T(w) - T(u)\}. \tag{2}$$

v are just any two arbitrary sons of w in T with $u > v$. Therefore if a vertex x has k sons x_i, $1 \leqq i \leqq k$, in T the relation (2) tells that

$$T(x_i) = \{y \in \mathrm{REACHABLE}(x_i) \bigcap T(x) - \bigcup_{x_j > x_i} T(x_j)\}. \tag{3}$$

At Step 7 of *dfs-DAG* while obtaining the binary tree-like representation by algorithm *structure-binary* adjacency relations in T are sorted so that the vertices with the same parent appear in descending order (cf. Section 3).

Therefore from relation (3) we conclude that T admits no edge of *DAG* directed from left to right with respect to T. This proves the fact that T is a depth-first search spanning tree of the *DAG*. So the preorder ranks of the vertices in T define their corresponding *dfs* numbers. ■

THEOREM 2: *Algorithm dfs-DAG has time complexity $O((\log n)^2)$ when $O(n^{2.81}/\log n)$ processors are used.*

PROOF. The complexity analysis for each step of algorithm *dfs-DAG* proves the theorem. ■

Acknowledgement.

The authors are thankful to the referee for his critical comments and helpful suggestions. They also acknowledge the valuable comments of Professor S. Kundu of University of Florida on an earlier version of this paper. The work of the first author (R. K. Ghosh) was supported by Department of Atomic Energy under grant 22/101/82-G dated 8.1.83.

REFERENCES

1. A. V. Aho, J. E. Hopcroft and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading MA, (1974).
2. G. H. Barnes, M. Richard, M. Kato, D. Kuck, D. Slotnik and R. Stokes, *The ILLIAC IV Computer*, IEEE Trans. Comp., vol. C-17; 8, (1968), pp. 746–757.
3. A. K. Chandra, *Maximal parellelism in matrix multiplication*, RC-6193, IBM TJW Res. Centre, (1976).
4. D. M. Eckstein and D. Alton, *Parallel graph processing using depth-first search*, Proc. Symp. Theo. Comp. Sci., (1977).
5. F. Gavril, *Merging with parallel processors*, CACM, vol. 18, (1975), pp. 588–591.
6. E. Horowitz and S. Sahni, *Fundamentals of Data Structures*, Comp. Sci. Press Inc., Potomac, MD, (1977).
7. J. Ja' Ja' and J. Simon, *Parallel algorithms in graph theory: Planarity testing*, SIAM J. Comput., vol. 11; 2, (1982), pp. 314–328.
8. P. Kogge, *Parallel solution of recurrence problems*, IBM J. Res. and Dev., vol. 18, (1974), pp. 138–148.
9. K. Maruyama, *On the parallel evaluation of polynomials*, IEEE Trans. Comp., vol. C-22; 1, (1973), pp. 2–5.

10. F. P. Preparata, *New parallel sorting schemes*, IEEE Trans, Comp., vol. C-27; 7, (1978), pp. 669–673.
11. E. Reghbati and D. G. Corneil, *Parallel computations in graph theory*, SIAM J. Comput., vol. 7; 2, (1978), pp. 230–237.
12. C. Savage and J. Ja' Ja', *Fast, efficient parallel algorithms for some graph problems*, SIAM J. Comput., vol. 10; 4, (1981), pp. 682–691.
13. V. Strassen, *Gaussian elimination is not optimal*, Num. Math., vol. 13; 4, (1969), pp. 354–356.
14. R. J. Swan, S. Fuller and D. Siewiorek, *Cm* a modular multi-microprocessor*, Proc. AFIPS National, Comp. Conf., vol. 46, (1977), pp. 637–644.
15. R. E. Tarjan, *Depth-first search and linear graph algorithms*, SIAM J. Comput., vol. 1; 2, (1972), pp. 146–166.
16. R. E. Tarjan, *Finding dominators in a directed graph*, SIAM J. Comput., vol. 3; 1, (1974), pp. 62–89.
17. J. Wyllie, *The complexity of parallel computations*, Ph.D. Thesis, Tech. Rep. 79–387, Cornell University, Ithaca, NY, (1979).