JOURNAL OF PARALLEL AND DISTRIBUTED COMPUTING **3**, 553–576 (1986)

The Parallel Complexity of Scheduling with Precedence Constraints*

DANNY DOLEV†

Institute of Mathematics and Computer Science, Hebrew University, Jerusalem, Israel

ELI UPFAL‡

Computer Science Department, Stanford University, Stanford, California 94305

AND

MANFRED K. WARMUTH§

Computer Science Department, University of California, Santa Cruz, California 95064

Received September 13, 1985

We study the problem of parallel computation of a schedule for a system of n unit-length tasks on m identical machines, when the tasks are related by a set of precedence constraints. We present NC algorithms for computing an optimal schedule in the case where m, the number of available machines, does not vary with time and the precedence constraints are represented by a collection of outtrees. The algorithms run on an exclusive read, exclusive write (EREW) PRAM. Their complexities are $O(\log n)$ and $O((\log n)^2)$ parallel time using $O(n^2)$ and $O(n)$ processors, respectively. The schedule computed by our algorithms is a *height–priority schedule*. As a complementary result we show that it is very unlikely that computing such a schedule is in NC when any of the above conditions is significantly relaxed. We prove that the problem is P-complete under logspace reductions when the precedence constraints are a collection of intrees and outtrees, or for a collection of outtrees when

*A preliminary version of this paper appeared in the International Workshop on Parallel Computation and VLSI, Amalfi, Italy, May 1984 [DUW84]. The research of the first and third authors was partially supported by the United States–Israel Binational Science Foundation under Grant 2439/82. The research of the second author was supported by a Weizmann Post-Doctoral fellowship and in part by DARPA under Grant N00039-83-C-1036.

†Part of this work was done while the author was visitng IBM Research, San Jose, Calif.

‡Part of this work was done while the author was visiting the University of California at Berkeley.

§Part of this work was done while the author was visiting the Hebrew University, Jerusalem.

0743-7315/86 $3.00
Copyright © 1986 by Academic Press, Inc.
All rights of reproduction in any form reserved.

the number of available machines is allowed to increase with time. The time span of a height–priority schedule for an arbitrary precedence constraints graph is at most $2 - 1/(m - 1)$ times longer than the optimal (N. F. Chen and C. L. Liu, *Proc. 1974 Sagamore Computer Conference on Parallel Processing,* T. Fend (Ed.), Springer-Verlag, Berlin, 1975, pp. 1–16). Whereas it is P-complete to produce the classical height–priority schedules even for very restricted precedence constraints graphs, we present a simple NC parallel algorithm which produces a different schedule that is only $2 - 1/m$ times the optimal. © 1986 Academic Press, Inc.

1. INTRODUCTION

One of the main issues in the theory of parallel computation is to classify problems with respect to the class NC, the class of problems that are solvable in polylog time using polynomial number of processors. While an NC algorithm is sufficient in order to prove that a problem is in NC, it is much harder to show directly that a problem does not lie in this class. Instead, one usually proves that the problem is P-complete under logspace reductions. The class NC is closed w.r.t. logspace reductions. If a P-complete problem was in NC, then all problems in P would have an NC solution, which is very unlikely [C83]. Thus, by proving that a problem is P-complete under logspace reductions, one essentially shows that this problem is outside of the class NC.

Our aim is to explore the border line between the class NC and the class P-complete. We concentrate on the fundamental problem of scheduling a set of n unit-length tasks subjected to some precedence constraints which are presented by a directed acyclic graph. When every task in the precedence graph has at most one incoming (resp. outcoming) edge, we say that the graph is an *outforest* (resp. *inforest*). The tasks are scheduled on a system of identical parallel machines. A profile indicates the number of machines available at any time slot. If the number of machines is the same for every time slot we say that the profile is straight (precise definitions are given in the next section).

The classical schedule considered in the literature is the *height–priority schedule,* in which tasks are chosen according to their height in the precedence graph. Height–priority schedules are optimal for straight profiles in the case where the graph is an inforest [H61] or an outforest [B81; DW85a]. It is easy to find a height–priority schedule in $O(n \log n)$ sequential time: Fill the slots in increasing order; keep track of the set of tasks of depth zero, i.e., the tasks that are candidates to be scheduled in the next slot; pick tasks according to highest height using a priority queue. Surprisingly, there are even linear time algorithms for finding a height–priority schedule for inforests [BG77]. In the case of outforests one can easily design a linear algorithm using the notion of Elite and Median introduced in [DW85a]. This leads to the interesting question of computing optimal schedules in polylog parallel time. Our first result is an NC algorithm to obtain an optimal height–priority

schedule for a given outforest and a straight profile. To obtain an optimal schedule for an inforest and a straight profile we reverse the precedence graph to an outforest and apply our algorithm.

It seems that to be able to construct a height–priority schedule in parallel we need to predict the remaining graph at various times. This might be hard to do in polylog time because of the sequential nature of the precedence constraints. In the case of outforests and straight profiles we circumvent this difficulty (Section 3); we assign to every task an integer release time and deadline and then drop the precedence constraints and find a certain schedule in which the new release times and deadlines are not violated. The release times and deadlines are chosen such that the obtained schedule does not violate the precedence constraints given by the outforest. The release times and deadlines are related to the depths and heights of the tasks, respectively, and these can be computed using an Eulerian path of the outforest [TV85; V85] in $O(\log n)$ parallel time using $O(n)$ processors on an EREW PRAM.

Finding a schedule in which the release times and deadlines are not violated is equivalent to finding a perfect matching for a convex bipartite graph and vice versa. These problems can be solved in $O((\log n)^2)$ parallel time using n processors [DS84] on an EREW PRAM. In Section 5 another algorithm is presented for solving the same problems; it runs in $O(\log n)$ parallel time using $O(n^2)$ processors on an EREW PRAM.[1] Our algorithm finds a perfect bipartite matching (resp. release-time deadline schedule) if one exists. Note that the algorithm of [DS84] also works if there is no perfect matching and finds a maximum cardinality matching in that case. We reduce optimal scheduling of an outforest to finding a certain release-time deadline schedule. The reduction will guarantee that such a schedule (perfect matching) always exists. Our $O(\log^2 n)$ algorithm improves the time bounds of several other scheduling problems presented in [DS84] by a factor of $\log n$.

This far we have assumed the profiles to be straight. In [DW81] it was shown that height–priority schedules are also optimal for outforests and nonincreasing profiles (i.e., the number of machines does not increase with time). The reduction of Section 3 and the algorithm of Section 5 can be adapted to the case of optimal scheduling of outforests on nonincreasing profiles. The complexity of the problem is sharply changed when we consider the complementary case, that is, outforests and nondecreasing profiles. In that case height–priority schedules are no longer optimal (see Fig. 1 and Table I) and even finding a schedule which is no longer than the shortest height–priority schedule is NP-hard [Wa81; M81]. Instead of finding a height–priority schedule of minimum length one can ask for finding some

[1] Less efficient algorithms have recently been presented in [HM86]: $O(\log n)$ time and n^3 (resp. n^4) processors for inforests (resp. outforests).

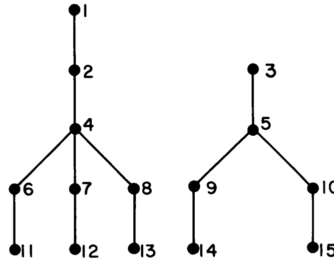

FIG. 1. An outforest precedence graph. (The edges are directed downwardly.)

height–priority schedule. In Section 6 we prove that even this problem is logspace complete for P.

Our P-completeness result implies that finding the lexicographically first schedule or a greedy schedule in which tasks are always chosen according to maximum weight are also P-complete. Note that these new requirements of the schedule are much more restrictive than scheduling according to height. They lead to P-completeness even if the profile is straight and the precedence graph is an outforest. Another path that leads to P-completeness results is to allow more general precedence graphs. For example, to find a height–priority schedule for straight profiles and opposing forests (unions of inforests and outforests) or level orders is P-complete (Section 6). Finding a schedule which is no longer than the shortest height–priority schedule is again NP-hard for these cases [Wa81; M81]. Note that if the number of processors is constant then finding an optimal schedule for the same cases is polynomial [GJ83; DW85b].

In the case of arbitrary precedence graphs and straight profiles with m machines it is not known how to find an optimal schedule in polynomial time even if m is constant. The case where m is a variable of the problem instance is NP-hard as mentioned above. Only the case $m = 2$ is known to have a

TABLE I
A HEIGHT–PRIORITY SCHEDULE (NONOPTIMAL) FOR THE OUTFOREST OF FIG. 1 AND THE NONDECREASING PROFILE

Slot	*0*	*1*	*2*	*3*	*4*	*5*	*6*
P_1	1	2	4	6	9	12	15
P_2		3	5	7	10	13	
P_3				8	11	14	
μ(slot)	1	2	3	3	3	3	3

Note. Starting with task 3 in the slot zero gives a schedule of length 6, which is optimal.

polynomial (in fact linear) sequential solution [Ga82], and recently has been shown to be in NC [HM85].

Height–priority schedules are used to approximate optimal schedules of arbitrary precedence graphs. The lengths of the height–priority schedules are no longer than $4/3$ times the optimal if $m = 2$, and $2 - 1/(m - 1)$ times if $m \geq 3$ [CL75]. Unfortunately producing a height–priority schedule is P-complete even for very restricted precedence graphs (Section 6). Height–priority schedules are a special case of greedy schedules which are by a factor of $2 - 1/m$ from optimal [Gr69].

For large m the greedy heuristic has essentially the same performance as the height–priority heuristic. Greedy schedules seem to be inherently sequential. But in Section 4 we present a simple NC algorithm which approximates the optimal schedule by the same factor as the greedy heuristic. The algorithm produces a nongreedy schedule: first all tasks of depth 0 are scheduled, then starting with a new slot all tasks of depth 1, and so forth.

It remains open whether there are approximation algorithms in NC which perform as well as the height–priority heuristic or better.

2. Preliminaries

A scheduling[2] problem is defined by a directed acyclic graph G and a profile μ. The graph G specifies the precedence constraints among the n tasks, which are the vertices of the graph. A directed path from task x to task y in G implies that the execution of task y cannot begin before task x is completed. Processing each task requires one unit of time.

The *profile* μ is a function from $\mathbf{N}_0$ into $\mathbf{N}$, where $\mu(i)$ is the number of machines available at the ith *time slot* (the interval $[i, i + 1)$). If a profile has only one value, m, then it is called *straight* and denoted by the letter m.

A schedule s for a graph G and a profile μ is a function from the vertices of G onto an initial segment $\{0, \ldots, l - 1\}$ of $\mathbf{N}_0$, such that:

(1) $|s^{-1}(r)| \leq \mu(r)$, for all r in $\{0, \ldots, l - 1\}$;
(2) if y is a successor of x in G, then $s(x) < s(y)$.

A task x starts to be executed at time $s(x)$ and finishes one time unit later. We say that task x is scheduled in slot $s(x)$ and that slot r has $\mu(r) - |s^{-1}(r)|$ *idle periods;* l denotes the *length* of the schedule s.

A minimum length schedule is called *optimal*. A schedule is *greedy* if the maximum number of tasks is scheduled at every slot, i.e.,

(3) $|s^{-1}(k)| < \mu(k)$ implies that every y s.t. $s(y) > k$ is a successor of some vertex z in $s^{-1}(k)$.

[2] Our definitions are similar to the ones given in [M81].

A *priority* q is a function from the set of tasks into $\mathbf{N}_0$. A schedule s is a *q-priority schedule* if vertices of higher priority are preferred over vertices of lower priority. Among the vertices of the same priority ties are broken arbitrarily. A q-priority schedule has the following additional property (note that (4) implies (3)):

(4) $s(x) > s(y)$ and $q(x) > q(y)$ imply that x is a successor of some vertex z with $s(z) = s(y)$.

For example, we can use the height or the depth of the vertices as a priority function. The *height* $h(x)$ (resp. *depth* $p(x)$) is the length of the longest path starting (resp. ending) with x. Note that according to this definition vertices which do not have any successors (resp. predecessors) have height (resp. depth) zero. Note that if the priority is an injective function then the corresponding schedule is unique. Clearly, there is usually more than one height– and depth–priority schedule.

For any graph G or any set of tasks T, $h(G)$ and $h(T)$ denote maximum heights of the vertices of G and T, respectively. The height of the empty graph and set is defined to be zero. The definitions for depth are generalized in a similar fashion.

A graph G is an *outforest* (resp. *inforest*) if every vertex has at most one immediate predecessor (resp. immediate successor). Note that our definition assumes no transitive edges in the representations for inforests or outforests.

The model of parallel computation we use is the *weakest* shared memory model—the Exclusive Read Exclusive Write (EREW) PRAM. In this model we have a collection of processors (RAMs) with access to a shared memory. In a single EREW–PRAM step, a processor may perform some internal computation or access (read or write) one memory cell. Each memory cell is accessed by at most one processor at each step.

3. Scheduling Outforests and the Release-Time Deadline Scheduling Problem

In this section we describe the reduction that leads to the polylog parallel time algorithms discussed later in the paper. A unit-length scheduling problem with release times and deadlines is given by a set T of n unit-length tasks, s.t. every task x has a nonnegative integer release time $r(x)$ and a positive integer deadline $d(x)$. The tasks have to be scheduled on m machines, s.t. every task is executed during its release time deadline interval. A schedule s for T is a function that maps T into $\mathbf{N}_0$, s.t.:

(1) $r(x) \leq s(x) \leq d(x) - 1$, for all x in T, and
(2) $|s^{-1}(k)| \leq m$, for k in $\mathbf{N}_0$.

A schedule for T can be sequentially obtained as follows [J55]: Scan the

slots in increasing order and among all the unscheduled tasks that were released at the current slot or before, schedule the task with the earliest deadline. Always put as many tasks as possible at each slot. Ties among tasks with the same deadline are broken arbitrarily. The produced schedule is called *ED-schedule* (Earliest-Deadline-schedule).

The produced schedule has the *Earliest-Deadline-Property* given below.

DEFINITION. An ED-schedule is a schedule s in which for any two tasks x and y and $k \geq 1$:

(1) if $s(x) = k$ and $|s^{-1}(k-1)| < m$, then $r(x) = k$;
(2) $s(x) < s(y)$ and $d(y) < d(x)$ imply that $r(y) > s(x)$.

A release time and a distinct deadline are assigned to every task such that the corresponding unique ED-schedule is a height–priority schedule for the outforest and in this schedule the precedence constraints are not violated.

Let π be an Eulerian path of the outforest (see Fig. 2). Number the tasks according to their order on the Eulerian path. Let L be a list of the vertices of G sorted according to nonincreasing height where vertices of the same height are ordered according to their Eulerian path numbering. For any task x define

$$r(x) := p(x) \qquad \text{(its depth)};$$
$$d(x) := n + \text{ the index of } x \text{ in list } L.$$

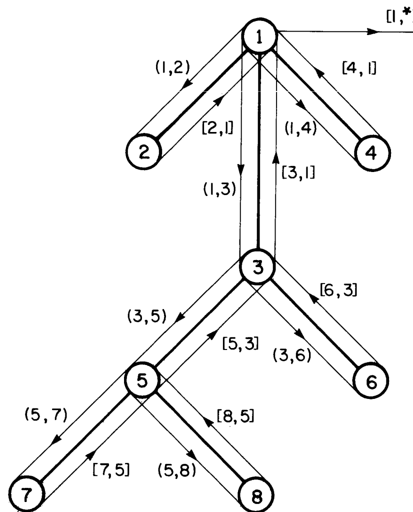

FIG. 2. Example of an Eulerian Path. Straight lines denote edges (directed downwardly). The lines with arrows denote the Eulerian Path.

Note that if $h(x) > h(y)$, for some vertices x and y, then $d(x) < d(y)$. The distinct deadlines have the following additional property, which is essential for the reduction:

For any tasks x, x', y, y', s.t. x' is an immediate successor of x of height $h(x) - 1$ and y' is a successor of y,

$$d(x) < d(y) \Rightarrow d(x') < d(y').$$

Note that the deadlines are large enough that there always exists a schedule that does not violate the release times and deadlines. In the next section we show how to compute $r(x)$ and $d(x)$.

THEOREM 3.1. *For any outforest G, the ED-schedule of the derived release-time deadline problem does not violate the precedence constraints of the outforest G.*

Proof. Let s be an ED-schedule of the derived release-time deadline problem. Let k be the smallest slot, i.e., interval $[k, k+1]$, that contains tasks y and y' s.t. y' is a successor of y. Clearly, $r(y') \leq k$ and thus $r(y) \leq k - 1$. It follows that y was available to be scheduled at slot $k - 1$. Since y was not scheduled at slot $k - 1$, we know by the definition of ED-schedule that $s^{-1}(k-1)$ contains m tasks, all of which have deadlines smaller than $d(y)$. This implies that their heights are at least $h(y)$, where $h(y) > 0$. Each task x in slot $k - 1$ has an immediate successor x' of height one less than x for which according to the above property $d(x') < d(y')$. Clearly, x' is released at slot k, since x was scheduled in slot $k - 1$. There are m such tasks with a deadline smaller than $d(y')$, because slot $k - 1$ contains m tasks. We conclude that y' should not be scheduled in slot k, which is a contradiction. ∎

We complete this section by showing that the ED-schedule is a height–priority schedule of the outforest; this implies the optimality of the ED-schedule [B81; DW85a].

THEOREM 3.2. *The derived ED-schedule is a height–priority schedule for the given outforest G.*

Proof. Let λ be the last nonempty slot of the ED-schedule s. For any slot k $(0 \leq k \leq \lambda)$ let Z_k be the set of all tasks of depth zero in the outforest induced by the vertices of $s^{-1}(k, \ldots, \lambda)$. Similarly, let R_k be the set of all tasks of $s^{-1}(k, \ldots, \lambda)$ that are released at time k. The fact the $r(\cdot) = p(\cdot)$ and the previous theorem assure that $Z_k \subseteq R_k$. If $|R_k| \leq m$ then $s^{-1}(k) = R_k$. Otherwise $s^{-1}(k)$ consists of m tasks with the m smallest deadlines. The definition of deadlines implies that $s^{-1}(k)$ is a set of m highest tasks of R_k. The previous theorem guarantees that $s^{-1}(k) \subseteq Z_k$, which completes the proof of the theorem. ∎

4. Computing the Release Times and Deadlines

The release times (depth) and deadlines (height) are computed on an EREW PRAM using an Eulerian path of the given outforest (a generalization of an Eulerian path on an outtree). The Eulerian path for outtrees was first used in [TV85; V85] for computing various tree functions. The algorithms require $O(\log n)$ time and use $O(n)$ processors. The computation of the depth was given in [V85]. Computing the height is very much related to computing High in [TV85; V85]. We present a slightly simpler procedure for computing the height.

The input to the scheduling problem consists of m, the number of machines available at any time slot, and E_0, the list of nontransitive edges that define the outforest (see Fig. 2 and Table II). Denote an edge from task x to task y by (x, y) and assume that E_0 is sorted according to the first entry of each edge, i.e., all outgoing edges of each vertex are grouped together. Note that if E_0 does not have this form, then it can be rearranged using a parallel bucket sort [GW84].

The algorithm is composed of several stages. We will present each stage and its complexity. Recursive Doubling [Wy79] is one of the common techniques in parallel algorithms. The aim of the algorithm is to compute an associative function for all prefixes of the linked list. For example, Recursive Doubling can be used to sum up the values along each prefix of the linked list. We make use of the Recursive Doubling idea at several places in our algorithm. Therefore, we first describe it as a general module. For this algorithm it is not necessary that the elements of the list appear consecutively in the storage. If the place of each element is known in advance then there are algorithms that compute all prefixes $O(\log n)$ time using $O(n)$ operations [Sc80; BK82; Sn86] as opposed to $O(n \log n)$ for the algorithm below.

The Recursive Doubling Algorithm

Assume that the input to the Recursive Doubling Algorithm is given by a linked list of n elements, and to each one we dedicate a processor. For convenience, let i be the processor dedicated to the ith element of the linked list. At the beginning every processor knows its successor in the linked list. Let NEXT be the vector describing the order of the linked list, i.e., NEXT(p) is p's successor. The value of NEXT(p) equals NIL when p is the last processor of the linked list. Let $I(p)$ be p's initial value at the beginning of the algorithm. Again for convenience, our algorithm computes an associative function F of all suffixes of the list instead of all prefixes. Examples for the function F are the sum or the maximum. The vector $V[1, \ldots, n]$ will contain the final values of the processors at the end of the algorithm.

TABLE II

THE COMPUTATION OF THE EULERIAN PATH FOR THE OUTTREE OF FIG. 2

Index i	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*
$E_0(i)$	(1, 2)	(1, 3)	(1, 4)	(3, 5)	(3, 6)	(5, 7)	(5, 8)								
$E(i)$	(1, 2)	(1, 3)	(1, 4)	[1, *]	[2, 1]	(3, 5)	(3, 6)	[3, 1]	[4, 1]	(5, 7)	(5, 8)	[5, 3]	[6, 3]	[7, 5]	[8, 5]
$F(i)$	1	5	6	9	10	13	14	15							
$L(i)$	4	5	8	9	12	13	14	15							
NEXT $(E(i))$	[2, 1]	(3, 5)	[4, 1]		(1, 3)	(5, 7)	[6, 3]	(1, 4)	[1, *]	[7, 5]	[8, 5]	(3, 6)	[3, 1]	(5, 8)	[5, 3]
$Q(i)$	1	3	13	15	2	4	10	12	14	5	7	9	11	6	8
$E'(i)$	(1, 2)	[2, 1]	(1, 3)	(3, 5)	(5, 7)	[7, 5]	(5, 8)	[8, 5]	[5, 3]	(3, 6)	[6, 3]	[3, 1]	(1, 4)	[4, 1]	[1, *]
$F'(i)$	1	2	4	14	5	11	6	8							
$L'(i)$	15	2	12	14	9	11	6	8							

Note. Back edges are denoted with [ ] and original edges with ( ).

```
DOUBLING (F, I, NEXT; V)
  V(p) := I(p)
  NEXT(p) := LINK(p)
    (*Each processor p performs the following loop:*)
  WHILE LINK (p) <> NIL DO
    V(p) := F(V(p), V(LINK(p)))
   LINK(p) := LINK(LINK(p))
ENDWHILE
```

THEOREM 4.1 [Wy79]. *The Doubling Algorithm terminates after* $\lceil \log_2 (n-1) \rceil$ *iterations of the loop. There is neither read conflict nor write conflict in the algorithm. Let* $V^k(p)$ *be the value of* $V(p)$ *after the kth loop and denote* $r_k = p - 1 + \min(n, 2^k)$. *If* F *is an associative function then* $V^k(p) = F(I(p), I(NEXT(p)), \ldots, I(NEXT^{r_k}(p)))$.

The Eulerian Path

The two complex tasks we face are how to find the depth and the height of the vertices in the outforest. For the sake of simplicity we assume that the outforest consists of only one tree. To get this we can add a dummy root, say r, such that the outforest on $(n-1)$ vertices becomes an outtree of n vertices. This can be done by first identifying the roots of the outforest, then using the Doubling Algorithm to order and index them [Wy79], and finally adding the extra edges to the list. Altogether creating the dummy root will cost $O(\log n)$ time using $O(n)$ processors. For the rest of the paper we assume that the precedence graph is an outtree.

The Eulerian Path of the outtree consists of $2n - 1$ edges (see Fig. 2). Every edge (i, j) is traversed forwardly, i.e., from i to j, and backwardly, i.e., from j to i. For every edge (i, j) we create a back edge $[j, i]$. We also add a dummy back edge $[1, *]$ which will be the last edge of the Eulerian Path. Every edge is tagged to remember whether it is an original edge or a back edge. We merge E_0 with the list of back edges, s.t. the resulting list E is sorted according to the first entry of each edge except for the fact that the back edges $[i, j]$ appear at the end of all original edges $(i, \cdot)$ (see Table II). This can be done in $O(\log n)$ time using $O(n)$ processors, using a merging algorithm that simulates a merging network [K72]. Assume that we are given one processor for every edge in E.

To establish the Eulerian Path we need to determine the next edge NEXT$[e]$ for every edge e of E. We make use of two vectors $F(1, \ldots, n)$ and $L(1, \ldots, n)$, where $E(F(x))$ (resp. $E(L(x))$) is the first (resp. last) edge starting with x in E (see Table II). The vectors F and L can be constructed in constant time. Note that $E(L(i))$ is always the back edge of the form $[i, \cdot]$; moreover, if i is a leaf in the outtree, then $F(i) = L(i)$. The following procedure uses F and L to create the vector NEXT (see the example of Fig. 2 and Table II).

```
EULERIAN PATH (E, F, L; NEXT)
  each processor p performs:
  IF E(p) = (x, y), i.e., it is not a back edge THEN
    NEXT(E(p)) := E(F(y))
    NEXT(E(L(y))) := E(p + 1)
ENDIF
```

Observe that since processors of back edges do not read and write, there are no read and write conflicts in the above procedure.

LEMMA 4.1. *The vector NEXT constructed by the above algorithm is an Eulerian Path, on the outtree described by the array E.*

To determine the deadlines of the tasks we need to know the index $Q(e)$ for every edge e of the Eulerian Path, i.e., $Q(e) = k$ if e is the kth edge of the path. We first index the edges backward using the Doubling Algorithm with the parameters

$F(x, y) = x + y$
$I(E(p)) = 1$ (for all edges of E)
NEXT is given by the Eulerian Path.

Let V be the vector obtained by the Doubling Algorithm using the above F and I. Then $Q(i) = 2n - V(F(i))$. Observe that if y is a successor of x then

$$Q(F(x)) \leq Q(F(y)) \leq Q(L(y)) \leq Q(L(x)).$$

LEMMA 4.2. *The vectors V and Q can be computed in $O(\log n)$ time using $O(n)$ processors.*

Evaluating the Depth

Our next task is to compute the depth of every vertex in the outtree. To get this we again make use of the Doubling Algorithm with the parameters

$$F(x, y) = x + y$$

$$I(E(p)) = \begin{cases} -1 & \text{if } E(p) \textit{ is an original edge} \\ 0 & \text{if } E[p] = [1, *] \\ +1 & \text{if } E(p) \textit{ is a ``back'' edge other than } [1, *] \end{cases}$$

NEXT—from the Eulerian Path.

Let V be the vector obtained by the Recursive Doubling Algorithm. Rename the vector V with P.

LEMMA 4.3. *For every v, $P(L(v))$ is the depth of vertex v in the outtree described by E. The vector P can be computed in time $O(\log n)$ using $O(n)$ processors.*

Note that if $E(p)$ is an edge that starts at x then $P(p) = p(x)$.