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.

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.

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).

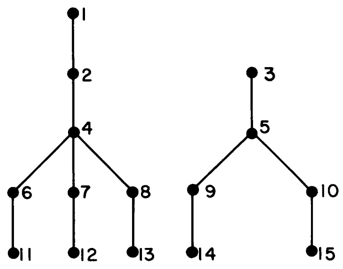

FIG. 1. An outforest precedence graph. (The edges are directed downwardly.)

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

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

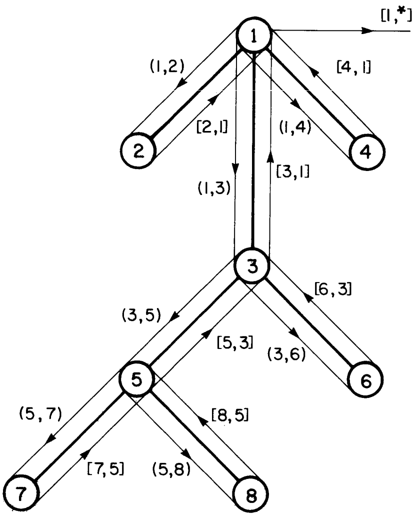

FIG. 2. Example of an Eulerian Path. Straight lines denote edges (directed downwardly). The lines with arrows denote the Eulerian Path.

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
    $V(p) := F(V(p), V(\text{LINK}(p)))$
   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 $P(p) = p(x)$.

Evaluating the Height

The evaluation of the heights is the most complex part of our algorithm. The basic observation that enables us to compute the heights is the following lemma.

LEMMA 4.4. *For every vertex v*

$$h(v) = \max_{x \in \operatorname{succ}(v)} (p(x)),$$

where succ(v) *is the set containing v and all its successors* (*not only immediate ones*).

Define $R(x) = [Q(F(x)), Q(F(x)) + 1, \ldots, Q(L(x))]$. Let $H(1, \ldots, n)$ be the vector for the heights, i.e., the vector we want to compute. Lemma 4.4 can be restated as:

For every vertex v,

$$H(v) = \max_{Q(p) \in R(v)} (P(p)).$$

To find a maximum among some linked list it is enough to use the Doubling Algorithm with the function max. In our case it is not so simple, because we have to compute the maximum over all ranges $R(x)$ in parallel. In the rest of the section we describe a way to do this.

First we need to rearrange the array E and the vector P in an increasing order according to the vector Q. Let E' and P' be the resulting vectors, i.e.,

$$E'(x) = E(Q(x)) \qquad \text{and} \qquad P'(x) = P(Q(X)).$$

Similarly define

$$F'(x) = Q(F(x)),\ L'(x) = Q(L(x)) \qquad \text{and} \qquad \rho(x) = F'(x) - L'(x) + 1.$$

Note that the components of $R(x)$ appear consecutively in E'.

Define the set of NODE entries in E' to be

$$\text{NODE} = \{p \mid F'(x) = p, \text{ for some task } x\}.$$

All the above vectors and sets can be constructed from the original ones in constant time. In the following algorithm all processors behave as in the regular Doubling Algorithm. The processors of the set NODE are making extra calculations and at the right moment they read another value not according to the Recursive Doubling process. This additional value will enable $p \in$ NODE to find $h(x)$ for which $F'(x) = p$.

Computing the height is a special case of computing values High in [TV85] and [V85], and vice versa. In both cases we compute maxima over certain intervals of an array. In [TV85] and [V85] each interval is decomposed into log n subintervals. The High value is the maximum over the log n maxima of the subintervals. We decompose into only two subintervals.

Algorithm Height

Comment. The current value of p will be VALUE(p). To avoid read conflicts we keep a copy of that value in DUPVAL(p).

```
processor $2n - 1$:         LINK($2n - 1$) = NIL
processor $p \neq 2n - 1$:  LINK($p$) = $p + 1$
every processor:            VALUE($p$) = $P'(p)$
                            DUPVAL($p$) = VALUE($p$)
                            ITERATION($p$) = 0
$p \in$ NODE:               CRITIC($p$) = $[\log (\rho(x) - 1)]$, where
                            $E'(p) = (x, \cdot)$
                            $\Delta(p) = \rho(x) - 2^{\mathrm{CRITIC}(p)} - 1$
every $p$:                  WHILE LINK($p$) $\neq$ NIL DO
$p \in$ NODE:                   IF ITERATION($p$) = CRITIC($p$) THEN
                                    $H(x)$ = MAX(VALUE($p$), DUPVAL
                                        ($p + \Delta(p)$)), where $E'(p) = (x, \cdot)$
                                ENDIF
every $p$:                      ITERATION($p$) = ITERATION($p$) + 1
                                VALUE($p$) = MAX(VALUE($p$), VALUE
                                    (LINK($p$)))
                                DUPVAL($p$) = VALUE($p$)
                                LINK($p$) = LINK(LINK($p$))
                            ENDDO
ENDHEIGHT
```

LEMMA 4.5. *There are no read or write conflicts in Algorithm Height and the Algorithm Height computes the correct heights in time $O(\log n)$ using $O(n)$ processors.*

The proof is based on the fact that the ranges of the NODE processors are nested, therefore no two of them will need to access the same cell concurrently. ■

Evaluating the Deadlines

To get the list L (defined in the previous section) we sort the vertices according to the tuple $(-H(x), Q(x))$. This is done using a parallel bucket sort [GW84] in $O(\log n)$ time using $O(n)$ processors. Given L, it is easy to compute the deadlines, i.e., $d(x) = n +$ the index of x in L. Note that the release time of a task equals its depth.

5. The Release-Time Deadline Scheduling Algorithm

We present an algorithm that runs in $O(\log n)$ parallel time and uses n^2 processors. The algorithm produces a schedule that does not violate the release times and deadlines if one exists. The reduction of Section 3 guarantees the existence of such a schedule. Our algorithm produces a particular schedule, the ED-schedule, since this schedule corresponds to the height–priority schedule of the original outforest.

Finding a valid schedule for the release-time deadline scheduling problem is equivalent to finding a perfect matching in a convex bipartite graph. The latter problem was solved in [DS84] in $O(\log^2 n)$ parallel time using $O(n)$ processors. If no perfect matching exists then the algorithm of [DS84] produces a matching of maximum cardinality.

Assume that the tasks are given in an array L which is sorted lexicographically on the tuples $(r(x), d(x))$. The release-time deadline scheduling problem we need to solve in this paper is derived from scheduling outtrees; in this case all the values of the deadlines and release times fall into a small range and therefore a parallel bucket sort can be used to obtain L in $O(\log n)$ time using $O(n)$ processors [GW84].

We now describe how to find a schedule in $O(\log n)$ parallel time using $O(n^2)$ processors given the sorted list L. Outline of the algorithm:

Step 1. Make n copies of the array L using a binary tree schema. At each phase the number of copies is doubled. A single copy of L can be copied in constant time using n processors. Thus, the whole process can be done in $\log n$ steps using n^2 processors.

Step 2. Dedicate n processors and a copy of L to every task x. Determine $s(x)$, where s denotes the ED-schedule. This step is more involved. Several parallel prefix computations are used. We first develop some notation and theory.

We need to determine $s(x)$ for every task x. Let us concentrate on a particular task x. Denote by B_x the set $\{y \mid d(x) < d(y)$ or $d(x) = d(y)$ and y appears before x in $L\}$. The following theorem shows how to determine $s(x)$.

THEOREM 5.1. *For a given task x, let S_x be an ED-schedule for B_x and let k be the minimal slot with an idle period in S_x, s.t. $k \geq r(x)$. Then, $s(x) = k$.*

Proof. An ED-schedule for the tasks of L can be produced by the following sequential algorithm [J55]: scan the slots in an increasing order and fill each slot with the leftmost task of L which is released. Let k be the slot $s(x)$. The proof follows from the fact that $s^{-1}(r(x), \ldots, k-1)$ contains only tasks of B_x. ■

To find the minimal slot k which has an idle period and $k \geq r(x)$, it is sufficient to know the number of tasks in each slot of S_x. These numbers can be found from the following simpler scheduling problem: Let $\bar{B}_x$ be the set of tasks in B_x with the same release times and the deadline $\bar{d}(x) = \max(\{d(x) \mid x \in B_x\})$.

THEOREM 5.2. *Let $S_x(\bar{S}_x)$ be an ED-schedule for $B_x(\bar{B}_x)$. Then for every i, $|S_x^{-1}(i)| = |\bar{S}_x^{-1}(i)|$.*

Proof. All ED-schedules of $\bar{B}_x$ have the same number of tasks in each slot. It is easy to see that S_x is an ED-schedule for $\bar{B}_x$. ■

In Step 2 of the algorithm we dedicate n processors and a copy of L to each task x. For every task y we can determine in a constant number of parallel steps its inclusion in B_x. By a simple prefix computation we can find the number of tasks before y in L which are in B_x. Therefore we can eliminate all tasks in L that are not in B_x in $O(\log n)$ time and obtain the sublist L_x of L containing the tasks in B_x.

Let k be the number of distinct release times appearing in L, and r_i, $1 \le i \le k$, be their values, where $r_i < r_{i+1}$. Define $r_{k+1} = r_k + 1$. The tasks of B_x which have a release time in the interval $[r_p, r_q)$, for $1 \le p < q \le k + 1$, are partitioned into two sets according to where they appear in S_x w.r.t. the interval $[r_p, r_q)$:

$$\mathrm{IN}_{p,q} = \{y \mid r_p \le r(y) < r_q \text{ and } S_x(y) < r_q\}$$

and

$$\mathrm{OUT}_{p,q} = \{y \mid r_p \le r(y) < r_q \text{ and } S_x(y) \ge r_q\}.$$

The value $i_{p,q}$ denotes the number of periods in the interval $[r_p, r_q)$ that are not occupied by a task of $\mathrm{IN}_{p,q}$: $i_{p,q} = m(r_q - r_p) - |\mathrm{IN}_{p,q}|$. Given the values of $i_{1,p}$ it is easy to find $s(x)$.

Step 2 (computing $s(x)$).

(2.1) Find B_x.
(2.2) Compute $i_{1,p}$ for $1 \le p \le k + 1$.
(2.3) $i_p^* = i_{1,p+1} - i_{1,p}$ for $1 \le p \le k$.
(2.4) Find the minimal p for which $r_p \ge r(x)$ and $i_p^* > 0$.
(2.5) $s(x) = r_{p+1} - \lceil i_p^*/m \rceil$.

The more complex part of the above algorithm consists of computing the $i_{1,p}$. Let $j_{p,q} = |\mathrm{OUT}_{p,q}|$. The following recurrences hold.

THEOREM 5.3. *For $1 \le p < q < t \le k + 1$, given $i_{p,q}$, $i_{q,t}$, $j_{p,q}$, and $j_{q,t}$, then*

$$i_{p,t} = i_{p,q} + \max(i_{q,t} - j_{p,q}, 0)$$

and

$$j_{p,t} = j_{q,t} + \max(j_{p,q} - i_{q,t}, 0).$$

Proof. The equalities follow from the fact that $j_{p,q}$ tasks with release times in the interval $[r_p, r_q)$ are candidates to be scheduled in the $i_{q,t}$ idle periods of the interval $[r_q, r_t)$. ■

Theorem 5.3 implies a way to compute $i_{1,p}$ via a parallel prefix computation:

$$\mathrm{NEXT}(p) = p - 1, \qquad 1 < p \le k + 1;$$

$$I(p) = (i_{p,p+1}, j_{p,p+1}), \qquad 1 \le p \le k;$$

$$F((i, j), (i', j')) = (i + \max(i' - j, 0), j' + \max(j - i', 0)).$$

After running the Recursive Doubling algorithm $V(p)$ will be $(i_{1,p}, j_{1,p})$.

It remains to be shown that we can find $i_{p,p+1}$ and $j_{p,p+1}$. Let n_p be the number of tasks in B_x with release time r_p. All these tasks appear consecutively in the sublist of L that contains the tasks of B_x. Therefore n_p can be computed easily. Now, when $n_p \le m(r_{p+1} - r_p)$, then $j_{p,p+1} = 0$ and $i_{p,p+1} = m(r_{p+1} - r_p) - n_p$. Otherwise, $i_{p,p+1} = 0$ and $j_{p,p+1} = n_p - m(r_{p+1} - r_p)$. In summary we have proved the following theorem:

THEOREM 5.4. *When L is sorted, then the release-time deadline problem can be implemented in $O(\log n)$ parallel time using $O(n^2)$ processors.* ■

6. THE P-COMPLETENESS RESULTS FOR HEIGHT–PRIORITY SCHEDULES

Our P-completeness result is a reduction from a version of the boolean circuit value problem [L75]. The circuit has only one input α_0 which is zero. All remaining inputs are computed from α_0. We use nand- and nor-gates which come in pairs. Every nand-gate (nor-gate) is succeeded (preceded) by a nor-gate (nand-gate) which has the same two inputs. We assume that the circuit is given in some topological order; i.e., it is a sequence $(\alpha_0, \alpha_1, \ldots, \alpha_{2n})$, s.t. $\alpha_0 = 0$ and $\alpha_{2i-1} = \overline{\alpha_j \wedge \alpha_k}$, $\alpha_{2i} = \overline{\alpha_j \vee \alpha_k}$, for $1 \le i \le n$ and some j, k s.t. $0 \le j \le k \le 2i - 2$. Let $v(\alpha_r)$, $0 \le r \le 2n$, denote the boolean value of α_r. It is easy to see that computing $v(\alpha_{2n})$ is P-complete for this version of the boolean circuit value problem.

The P-completeness proof uses the fact that the profile is not straight. A profile is called *nondecreasing* if $\mu(k) \le \mu(k + 1)$ for every slot k.

THEOREM 6.1. *Given an outforest G and a nondecreasing profile μ, then to find a height–priority schedule is P-complete.*

Proof. We reduce a boolean circuit problem of the above form to producing a height–priority schedule for a scheduling problem. Let $(\alpha_0, \alpha_1, \ldots, \alpha_{2n})$ be such a circuit. From this circuit the outforest G and the profile μ of the corresponding scheduling problem are constructed. The construction can be done by an algorithm that uses only $O(\log n)$ work space. A task of G will have two subscripts i, j meaning that this task corresponds to α_i and will be scheduled in slot $j - v(\alpha_i)$ of any height–priority schedule. The graph G consists of a number of chains. A *chain* is a sequence of tasks s.t. the jth task of the chain precedes the $(j + 1)$st task. Chains are combined to outtrees by making the first tasks of the chains successors of other chain tasks.

The input $\alpha_0 = 0$ corresponds to the chain T_0 which consists of tasks $\{t_{0,j} \mid 0 \leq j \leq 5n\}$ with the precedence constraints $t_{0,j-1} \Rightarrow t_{0,j}$ $(1 \leq j \leq 5n)$. Let α_{2k-1} and α_{2k}, $1 \leq k \leq n$, be a pair of nand- and nor-gates using the same inputs α_p and α_q. The gates α_{2k-1} and α_{2k} correspond to chains T_{2k-1} and T_{2k}, respectively, where

$$T_{2k-1} := \{t_{2k-1,j} \mid k \leq j \leq 5n - 4k + 2\}$$

and

$$T_{2k} := \{t_{2k,j} \mid k \leq j \leq 5n - 4k\}.$$

Note that the first vertices in T_{2k-1} and T_{2k} have the same second subscript and that $h(T_{2k-1}) = h(T_{2k}) + 2 = h(T_{2k+1}) + 4$.

The gates α_{2k-1} and α_{2k} are also responsible for two chains B_p^k and B_q^k. The chain B_p^k consists of the set of tasks, $\{b_{p,j}^{2k-1} \mid k \leq j \leq 5n - 2p\}$. An additional edge $t_{p,k-1} \Rightarrow b_{p,k}^{2k-1}$ connects B_p^k and T_p to an outtree. The definitions for B_q^{2k-1} are identical except q is substituted for p. This completes the definition of G.

The profile μ has the form

$$\mu(j) = \begin{cases} 4j + 3 & \text{if } 0 \leq i \leq n - 1 \\ 4n + 1 & \text{if } i \geq n. \end{cases}$$

The following claim shows that there exists exactly one height–priority schedule s. It is easy to determine $v(\alpha_r)$, $1 \leq r \leq 2n$, from this schedule.

CLAIM. *Slot j of any height–priority schedule for G and μ contains exactly all tasks of G having i and $j - v(\alpha_i)$ as their subscripts, for $0 \leq i \leq 2n$, and $0 \leq j \leq 5n$,*

Proof of the Claim. The claim implies that depending on whether $v(\alpha_i)$, for $0 \leq i \leq 2n$, is zero or one, all tasks of T_i (and the B-chains connected to it) will be scheduled in the slot of their second subscript or one earlier, respectively. Call T_i *late* if its tasks are scheduled in the slot of their second subscript and *early* if they are scheduled one earlier. In slot f $(0 \leq f \leq n - 1)$, it is decided whether T_{2f+1} and T_{2f+2} are early or late (see Fig. 3 and Table III). The proof is by an induction using the following relationships between the heights of the tasks and their subscripts.

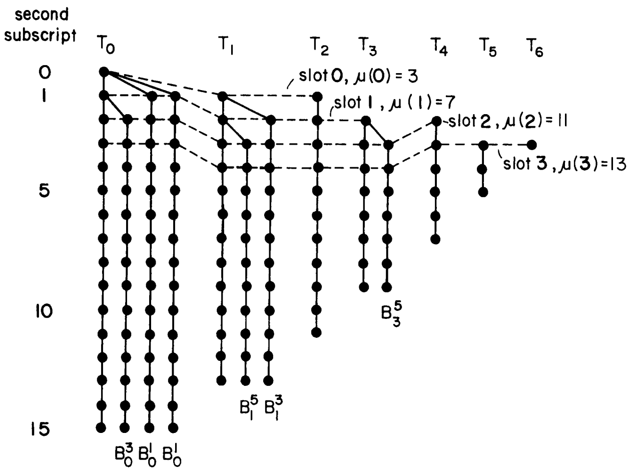

FIG. 3. The outforest G corresponding to the circuit of Table IV.

Let $x_{i,j}$ and $y_{i+1,j}$ be tasks of G with indices i, j and $i + 1, j$, respectively. That is, $x_{i,j}$ equals $t_{i,j}$ or $b^p_{i,j}$, for some p, and similarly for $y_{i+1,j}$. Then, $h(x_{i,j}) = h(y_{i+1,j}) + 2$. Furthermore, for any $x_{i,j}$ and $y_{p,q}$ in G, if $i < p$ and $|j - q| \leq 1$, then $h(x_{i,j}) > h(x_{p,q})$.

Thus in any height–priority schedule, tasks with lower first subscript are preferred if their second subscripts differ at most by one.

We prove the claim by an induction on the slot j. The slot zero of a height–priority schedule contains the three tasks $t_{0,0}$, $t_{1,1}$, $t_{2,1}$ of G of depth zero. Note that $v(\alpha_0) = 0$, $v(\alpha_1) = 1$, and $v(\alpha_2) = 1$ in all circuits, because $v(\alpha_0) = 0$. The tasks $t_{0,0}$, $t_{1,1}$, and $t_{2,1}$ are late, early and early, respectively. We showed that the claim holds for the slot zero.

Assume the claim holds for the slots $0, 1, 2, \ldots, j - 1$ and we want to prove it for slot j. If $j \geq n$ then $\mu(j) = 4n + 1$. Since there are $4n + 1$ chains in G, all available tasks are scheduled in slot j. Thus all the tasks are scheduled in slot j whose predecessors are scheduled in the slots $0, 1, \ldots, j - 1$. By induction we know that these predecessors have sub-

TABLE III
A BOOLEAN CIRCUIT WITH PAIRED NAND- AND NOR-GATES.[a]

Index k	*0*	*1*	2	3	4	5	6
α_k	0	$\overline{\alpha_0 \wedge \alpha_0}$	$\overline{\alpha_0 \vee \alpha_0}$	$\overline{\alpha_0 \wedge \alpha_1}$	$\overline{\alpha_0 \vee \alpha_1}$	$\overline{\alpha_1 \wedge \alpha_3}$	$\overline{\alpha_1 \vee \alpha_3}$
$v(\alpha_k)$	0	1	1	1	0	0	0

[a] $n = 3$.

scripts i, p s.t. $p \leq j - v(\alpha_i) - 1$. We conclude that all tasks with subscripts $i, j - v(\alpha_i)$ are scheduled in slot j, i.e., the claim holds if $j \geq n$.

Assume $1 \leq j \leq n$. In slot j it is determined whether the tasks of T_{2j+1} and T_{2j+2} are scheduled at the slot of their second subscript or one earlier, i.e., $v(\alpha_{2j+1})$ and $v(\alpha_{2j+2})$ are computed in slot j. Slot j has size $4j + 3$. Let us first determine which tasks of the trees corresponding to α_i $(0 \leq i \leq 2j)$ are available for slot j. All these tasks have a height greater than $t_{2j+1,j}$ and $t_{2j+2,j}$, the roots of the trees T_{2j+1} and T_{2j+2}, respectively.

$$t_{i,j-v(\alpha_i)}, \qquad \text{for } 0 \leq i \leq 2j : 2j + 1 \text{ tasks}$$
$$b^k_{i,j-v(\alpha_i)}, \qquad \text{for } 1 \leq i \leq 2j \text{ and appropriate } k : 2j \text{ tasks.}$$

Altogether this gives us $4j + 1$ tasks, and thus there are two spaces left for slot j. These two spaces must be filled with two tasks of maximum height. Assume the gate α_{2j+1} and α_{2j+2} use α_q and α_r as inputs.

Then $b^{2j+1}_{q,j}(b^{2j+1}_{r,j})$ is available to be scheduled in slot j iff $v(\alpha_{2j+1}) = 1$ $(v(\alpha_{2j+2}) = 1)$. Note that $h(b^{2j+1}_{q,j}) \geq h(b^{2j+1}_{r,j}) > h(t_{2j+1,j}) > h(t_{2j+2,j})$. Table IV shows what tasks fill the two spaces. $t_{2j+1,j}$ is scheduled in slot j iff $\overline{v(\alpha_q) \wedge v(\alpha_r)}$ holds, and $t_{2j+2,j}$ iff $\overline{v(\alpha_q) \vee v(\alpha_r)}$ is true.

This claim completes the proof of the theorem. ■

The reduction of Theorem 6.1 leads to various related P-completeness results for straight profiles. For example, we can pad the nondecreasing profile with an intree to get the following:

COROLLARY 6.1. *To find a height–priority schedule for an opposing forest and a straight profile is P-complete.*

Proof. The number of machines m in the straight profile is $\mu(n) + 1$, where μ is defined as in the proof of Theorem 6.1. The precedence constraints consist of the outforest G of the proof of Theorem 6.1 plus one intree I which will contribute $m - \mu(j)$ tasks to slot j:

$$I := \{e_{i,j} \mid 1 \leq i \leq m - \mu(j),\ 0 \leq j \leq 5n + 1\}$$
$$e_{i,j} \Rightarrow e_{1,j+1} \qquad \text{for } 1 \leq i \leq m - \mu(j),\ 0 \leq j \leq 5n.$$

TABLE IV
THE TASKS SCHEDULED IN THE TWO ADDITIONAL SPACES OF SLOT j

$v(\alpha_q)$	$v(\alpha_r)$	*In slot j*
0	0	$t_{2j+1,j}$, $t_{2j+2,j}$
0	1	$t_{2j+1,j}$, $b^{2j+1}_{r,j}$
1	0	$b^{2j+1}_{q,j}$, $t_{2j+1,j}$
1	1	$b^{2j+1}_{q,j}$, $b^{2j+1}_{r,j}$

Note that $h(I) > h(G)$. Let s be the height–priority schedule for G and μ. It is easy to see that there is exactly one height–priority schedule s' for $G \cup I$ and m:

$$\begin{aligned} s'(x) &= s(x), && \text{for } x \in G \\ s'(e_{i,j}) &= j, && \text{for } 1 \leq i \leq m - \mu(j),\ 0 \leq j \leq 5n + 1. \end{aligned}$$

Thus the theorem follows. ■

Scheduling according to height is also hard for various other classes of precedence graphs. For example, a *level order* is a precedence graph in which the vertices of every component are partitioned into levels (according to the height of the vertices) and the vertices of a level precede all vertices of the same component of the levels below it.

COROLLARY 6.2. *To find a height–priority schedule for a level order and a straight profile is P-complete.*

Proof. We just observe that in the height–priority schedule for $G \cup I$ of the previous proof all tasks belonging to the same level of a component appear in the same slot. Thus we can add edges to $G \cup I$ such that each component becomes a level order component. ■

We also get P-completeness results for straight profiles if we do not change the class of precedence graphs but the priority function. Assume every task x is given a weight $w(x) \in 1, 2, 3$. Note that w is a priority function.

COROLLARY 6.3. *Let G' be a weighted outforest using only three different weights. To find a weight–priority schedule for G' and a straight profile is P-complete.*

Proof. Again we use the reduction of Theorem 6.1 but this time we pad the straight profile with an outforest O. (In Corollary 6.1 we used an inforest.) Let $G' = G \cup O$, $m = \mu(n) + 1$ and

$$\begin{aligned} O &:= \{o_{i,j} \mid 1 \leq i \leq m - \mu(j),\ 0 \leq j \leq 5n + 1\} \\ o_{1,j} &\Rightarrow o_{i,j+1}, \qquad \text{for } 1 \leq i \leq m - \mu(j),\ 0 \leq j \leq 5n. \end{aligned}$$

We will define the weights s.t. there will be only one weight–priority schedule s' for G' and m:

$$\begin{aligned} s'(x) &= s(x), && \text{for } x \in G \\ s'(o_{i,j}) &= j && \text{for } 1 \leq i \leq m - \mu(j),\ 0 \leq j \leq 5n + 1. \end{aligned}$$

This holds if

$$\begin{aligned} w(o) &:= 3, && \text{for } o \in O \\ w(b) &:= 3, && \text{for all } b\text{-type tasks of } G \end{aligned}$$

$$w(\text{first task of } T_i) = \begin{cases} 3 & \text{if } i = 0 \\ 2 & \text{if } \alpha_i \text{ is a nand-gate} \\ 1 & \text{if } \alpha_i \text{ is a nor-gate} \end{cases}$$

all other t-type tasks have weight 3. ∎

Let the list L be the tasks of G' sorted according to nondecreasing weight. If we define G' and m as in the proof of Corollary 6.3 then the weight–priority schedule for G' and m is the lexicographically first schedule for G' and m and thus:

COROLLARY 6.4. *To find a lexicographically first schedule for an out-forest and a straight profile is P-complete.*

7. APPROXIMATING $2 - 1/m$ FROM OPTIMAL

Let G be an arbitrary precedence graph and let m be the number of machines. Any greedy m processor schedule is $2 - 1/m$ from optimal [Gr69]. We define a nongreedy schedule which approximates the optimal schedule by the same factor. This schedule can be found in $O(\log^2 n)$ parallel time on an EREW PRAM using an $O(n^3)$ processor.

A *depth schedule* for G is defined by the following procedure. Let d_k denote the number of tasks in G of depth k.

```
**for** $k := 0$ to $p(G)$ **do**
    schedule the $d_k$ tasks of $G$ of depth $k$ in the next $\lceil d_k/m \rceil$ time slots.
```

Let $D_m(G)$ (resp. $O_m(G)$) denote the length of the m processor depth schedules (resp. optimal schedules) of G.

THEOREM 7.1. $D_m(G)/O_m(G) \leq 2 - 1/m$.

Proof. Let G be a precedence graph for which the theorem is false. Let i be the number of idle periods in an optimal schedule for G. Add i independent tasks to G. Call the new graph G'. Clearly, $D_m(G') \geq D_m(G)$ and $O_m(G) = O_m(G') = n/m \geq p(G')$, where n is the number of tasks in G'. It follows that G' also contradicts the theorem since $D_m(G')/O_m(G') \geq D_m(G)/O_m(G)$.

Let s be a depth schedule for G' in which f slots contain m tasks and e slots contain at least one but less than m tasks. Observe that $e \le p(G')$ and $f \le (n - p(G'))/m$. Now the following inequalities lead to a contradiction.

$$\frac{D_m(G')}{O_m(G')} = \frac{e+f}{n/m} \le \frac{p(G') + (n - p(G'))/m}{n/m} = \frac{p(G')(m-1) + n}{n}$$
$$\le \frac{(n/m)(m-1) + n}{n} = 2 - \frac{1}{m}. \quad \blacksquare$$

Once the depth of every vertex is found then it is trivial to find a depth schedule.

THEOREM 7.2 [DS81]. *The depth of every vertex in a directed acyclic graph can be determined in $O(\log^2 n)$ time on an EREW PRAM using $O(n^3)$ processors.*

REFERENCES

[BG77] Brucker, P., Garey, M. R., and Johnson, D. S. Scheduling equal-length tasks under treelike precedence constraints to minimize maximum lateness. *Math. Oper. Res.* **2,** (1977), 275–284.

[BK82] Brent, R. P., and Kung, H. T. A regular layout for parallel adders. *Trans. Comput. IEEE* **TC-31** (1982), 260–264.

[B81] Bruno, J. Deterministic and stochastic scheduling problems with treelike precedence constraints. *Proc. NATO Conference,* Durham England, 1981.

[C83] Cook, S. A. An overview of computational complexity, *Comm. ACM* **26** (1983), 400–408.

[CL75] Chen, N. F., and Liu, C. L. On a class of scheduling algorithms for multiprocessor computing systems. In Fend, T. (Ed.). *Proc. 1974 Sagamore Computer Conference on Parallel Processing.* Springer-Verlag, Berlin, 1975, pp. 1–16.

[DS81] Dekel, E., Nassimi, D., and Sahni, S. Parallel matrix and graph algorithms, *SIAM J. Comput.* **10,** 4 (1981), 657–675.

[DS84] Dekel, E., and Sahni, S. A parallel matching algorithm for convex bipartite graphs and applications to scheduling. *J. Parallel Distributed Computation* **1,** 2 (1984), 152–184.

[DUW84] Dolev, D., Upfal, E., and Warmuth, M. K. Scheduling trees in parallel, *Proc. International Workshop on Parallel Computing and VLSI,* Amalfi, Italy, May 23–25, 1984; published as Bertolazzi, P., and Luccio, F. (Eds.). *VLSI: Algorithms and Architectures.* North-Holland, Amsterdam, 1985, pp. 91–102.

[DW85a] Dolev, D., and Warmuth, M. K. Scheduling flat graphs. *SIAM J. Comput.* **14,** 3 (1985), 638–657.

[DW85b] Dolev, D., and Warmuth, M. K. Profile scheduling of opposing forests and level orders. *SIAM J. Algebraic Discrete Methods* **6,** 4 (1985), 665–687.

[Ga82] Gabow, H. N. An almost linear algorithm for two processor scheduling. *J. Assoc. Comput. Mach.* **29** (1982), 766–780.

[Gr69] Graham, R. L. Bounds on multiprocessing timing anomalies. *SIAM J. Appl. Math.* **17,** (1969), 416–429.

[GJ83] Garey, M. R., Johnson, D. S., Tarjan, R. E., and Yannakakis, M. Scheduling opposing forests. *SIAM J. Algebraic Discrete Methods* **4,** 1 (1983), 72–93.

[GW84] Galil, Z., and Warmuth, M. K. Bucket sorting in parallel. Private communication.

[H61] Hu, T. C. Parallel sequencing and assembly line problems. *Oper. Res.* **9,** 6 (1961), 841–848.

[HM85] Helmbold, D., and Mayr, E. Two processor scheduling is in NC, Tech. Rep. STAN-CS-85-1079, Department of Computer Science, Stanford University, 1985.

[HM86] Helmbold, D., and Mayr, E. Fast scheduling algorithms on parallel computers. *Adv. in Comput. Res.*, in press.

[J55] Jackson, R. Scheduling a production line to minimize maximum tardiness. Res. Rep. 43, Management Science Research Project, University of California, Los Angeles, 1955.

[K72} Knuth, D. E. *The Art of Computer Programming,* Vol. 3. Addison–Wesley, Reading, Mass., 1972.

[L75] Lander, R. E. The circuit value problem is LOG SPACE complete for P, *SIGACT News* **7,** 1 (1975), 18–20.

[M81] Mayr, E. W. Well structured parallel programs are not easier to schedule. Tech. Rep. STAN-CS-81-880, Department of Computer Science, Stanford University, Sept. 1981.

[Sc80] Schwartz, J. T. Ultracomputers. *ACM TOPLAS* **2** (1980), 484–521.

[Sn86] Snir, M. Depth-size tradeoffs for parallel prefix computation. *J. Assoc. Comput. Mach.,* in press.

[TV85] Tarjan, R. E., and Vishkin, U. An efficient parallel biconnectivity algorithm. *SIAM J. Comput.* **14,** 4 (1985), 862–874.

[V85] Vishkin, U. An efficient parallel strong orientation. *Inform. Process. Lett.* **20,** 5 (1985), 235–240.

[Wa81] Warmuth, M. K. Scheduling on profiles of constant breadth. Ph.D. thesis, Department of Computer Science, University of Colorado, Boulder, 1981.

[Wy79] Wyllie, J. C. The complexity of parallel computations. Ph.D. thesis, Tech. Rep. 79-387, Department of Computer Science, Cornell University, Ithaca, N.Y., 1979.