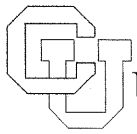
An Almost-Linear Algorithm for Two-Processor Scheduling *

Harold N. Gabow

CU-CS-169-80



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

* This research was supported in part by National Foundation Grant MCS 78-18909..

Abstract

n unit-length jobs subject to precedence constraints are to be scheduled on two processors to minimize finish time. Previous algorithms for this well-known problem begin by finding the transitive closure, and so use time $O(\min(mn, n^{2.61}))$. An $O(m + n\alpha(n))$ algorithm is presented. The algorithm constructs a "lexicographic maximum schedule," which is shown to be optimal.

1. Introduction

Consider the following well-known model for multiprocessor scheduling [C]: A set of n unit-length jobs is to be executed by p identical processors; a dag specifies precedence relations among the jobs. We seek a schedule minimizing the finish time ω.

For arbitrary p this problem is NP-complete [U]. For fixed $p \geq 3$ no polynomial-time algorithm is currently known. Here we investigate the tractable case p=2.

For this case several polynomial-time algorithms have been given. They all begin by finding the transitive closure of the dag. This requires time $O(\min(mn, n^{2.61}))$, where m is the number of edges of the dag. (The first bound follows from doing n depth-first searches; the second follows from reducing transitive closure to matrix multiplication [AHU,P].)

The algorithm of Fujii, Kasami and Ninomiya [FKN] is based on matching techniques. Excluding transitive closure time, it requires the time to find a maximum matching, $O(n^{2.5})$ [K]. Coffman and Graham [CG] give an algorithm based on a lexicographic numbering scheme. Sethi [S] shows the numbering can be done in time $O(m + n\alpha(n))$. (This algorithm can begin by finding the transitive closure or transitive reduction. However, both operations require the same time [AGU]). Garey and Johnson [GJ] give an algorithm for scheduling with precedence constraints and deadlines, which also solves our problem. It uses time $O(n^2)$ to compute modified deadlines.

Time bounds for these algorithms often assume the transitive closure (or reduction) of the dag is given. In practice this is unlikely. On general dags the transitive closure step dominates, and the algorithms

all use time $O(\min(mn,n^{2.61}))$. (Actually the algorithm of [FKN] is $O(n^{2.61})$).

Here we present an algorithm that does not use the transitive closure. The time on an arbitrary dag is almost linear, $O(m+n\alpha(n))$. The algorithm uses the idea of a "lexicographic maximum schedule." Such a schedule always executes nodes on the longest paths of the dag. (This is a refinement of Hu's rule for scheduling trees [H].)

Section 2 proves that a lexicographic maximum schedule is always optimal. Section 3 gives the algorithm.

2. Lexicographic Maximum Schedules

This section provides the theoretical basis of the algorithm. Lexicographic maximum schedules are defined and shown to be optimal.

First we review some well-known definitions. A scheduling problem is defined by a dag (directed acyclic graph), having n nodes and m edges. If there is a directed path of one or more edges from node x to node y, then x is a predecessor of y, and y is a successor of x. A dag can be partitioned into levels i, i=u, ..., 1: level i contains all nodes x that start paths of length i-1 but not i. In this case, level(x) = i.

A schedule is a sequence of sets S_i, $i = 1, \ldots, \omega$, that partition the nodes, such that $|S_i| \leq 2$, and if $x \in S_i$, $y \in S_j$ and x precedes y, then $i < j$. A schedule executes the nodes of S_i in the i^{th} time slot. ω is the finish time. We seek an optimal schedule, i.e., one with minimum finish time.

It is convenient to introduce dummy nodes for idle time in the schedule. Thus, we denote set S_i of a schedule as the pair (x_i,y_i), where either $S_i = \{x_i,y_i\}$, or $S_i = \{x_i\}$ and y_i is a dummy node. By convention, level 0 contains the dummy nodes. So level(d) = 0 for a dummy node d.

A level-by-level schedule "executes levels" in the order u,...,1. Suppose levels u,...,i+1 have been executed. If level i contains s nodes, it is executed in $\lceil \frac{s}{2} \rceil$ time slots: The first $\lfloor \frac{s}{2} \rfloor$ slots each execute two nodes of level i. If s is odd, the $\lceil \frac{s}{2} \rceil^{nd}$ slot executes a node f of level i and a node t of a lower level. (t may be a dummy node of level 0).

The $\lceil \frac{s}{2} \rceil$ time slots form the execution of level i. The ordered pair (f,t) is a jump from f to t. Note a node in level i can be executed (i.e., jumped) before the execution of level i. Thus s can be less than the number of nodes originally in level i.

Suppose a level-by-level schedule S makes jumps (f_i,t_i), $1 \le i \le k$, where $level(f_i)$ is decreasing. Then $(level(t_1)..., level(t_k))$ is the jump sequence of S. Note the jump sequence determines ω, since the number of time slots with an idle processor is the number of zeros in the sequence. We compare jump sequences using lexicographic order. A lexicographic maximum schedule is a level-by-level schedule with the largest possible jump sequence. A lexicographic maximum schedule always jumps to the highest level; if on some level several jumps go to the highest level, the one that enables subsequent jumps to be highest is chosen.

We show any lexicographic maximum schedule is optimal. (Clearly it suffices to show just one such schedule is optimal). It is interesting to note that if we extend the definition in the obvious way to an arbitrary number of processors, then any lexicographic maximum schedule is optimal, for dags that are in-forests or out-forests [H].

We begin by studying "covers." Two nodes are compatible if neither precedes the other. If S is a set of nodes, a cover of S is a partition of S into sets of one or two compatible nodes. Extending

our convention for schedules, we denote a set of a cover as a pair (x,y), with y possibly a dummy node. A cover is a cover of all the nodes. Clearly a schedule is a cover (if we ignore the order of the schedule).

Fujii, Kasami and Ninomiya [FKN] show how a cover gives a schedule. For this result, define a node to be ready at a given point in a schedule if all its predecessors have been executed, but it has not; a set of nodes is ready if its individual nodes are all ready.

Lemma 2.1: If C is a cover, there is a schedule with $\omega = |C|$. Further, any sequence of ready pairs can be chosen to begin the schedule.

Proof: See [FKN]; the Appendix contains a slightly different proof. Note a "sequence of ready pairs" means a sequence (x_i,y_i), $i=1,\ldots,k$, where (x_i,y_i) is ready after the preceding pairs have been executed. □

Our main method for modifying schedules is the following.

Lemma 2.2: Let (x_i,y_i), $i = 1,2$, be in a schedule S*, such that

(i) (x_1,y_1) is executed before (x_2,y_2)

(ii) $\text{level}(x_2) > \text{level}(y_1)$.

Then there is a cover C of all nodes but x_1, x_2, such that $|C| = |S|-1$.

Proof: We define pairs $(x_i,y_i) \in S$, $3 \le i \le \ell$, $\ell \ge 2$, so as in Figure 1,

(1) x_i precedes x_{i+1}, for $2 \le i \le \ell - 1$.

(2) y_i precedes y_{i+1}, for $1 \le i \le \ell - 2$; $y_{\ell-1}$ and y_ℓ are compatible.

(3) $\text{level}(x_i) = \text{level}(y_{i-2})$, for $3 \le i \le \ell$.

(Note a dummy node is compatible with any other node). This suffices for

---

*Note the subscripts do not mean these pairs are the first two executed by S. Also, recall y_i may be a dummy node.

the Lemma. For then $C = S - \{(x_i,y_i) | 1 \le i \le \ell\} + \{(x_i,y_{i-2}) | 3 \le i \le \ell\} + (y_{\ell-1},y_\ell)$ is the desired cover, by (2) - (3). (Note all nodes x_i, y_i are distinct, by (1) - (2).)

If y_1 and y_2 are compatible, take $\ell = 2$. Otherwise node y_1 precedes y_2 by (i).

Now assume $(x_1,y_1),\ldots(x_i,y_i)$ have been defined for some $i \ge 2$, so (1) - (3) are satisfied (for $i < \ell$). Choose x_{i+1} as any successor of x_i on level (y_{i-1}). Note x_{i+1} exists since level (x_i) > level (y_{i-1}). (For $i = 2$ this is (ii); for $i \ge 3$, level (x_i) = level (y_{i-2}) > level (y_{i-1}), by (3) and (2).) Choose y_{i+1} so $(x_{i+1},y_{i+1}) \in S$.

Clearly (1) and (3) hold for i+1. For (2), take $\ell = i+1$ if y_i and y_{i+1} are compatible. Otherwise y_i precedes y_{i+1}, since x_i precedes x_{i+1}. So (2) holds for i+1. □

Lemma 2.3: An optimal, level-by-level schedule always exists.

Proof: By induction, suppose the Lemma is true for dags with < n nodes. We show it is true for dags with n nodes by considering two cases. Let S be an optimal schedule. Recall level u is the highest level of the dag.

Case 1: Level u has at least two nodes.

If S contains a pair (x,y) with both nodes in level u, then we can assume S begins by executing (x,y); the Lemma follows by induction. If S contains no such pair we modify it so it does, as follows.

Let $(x_i,y_i) \in S$ with x_i, but not y_i, in level u, i = 1, 2. Construct cover C of Lemma 2.2. (Note level (x_i) > level (y_j), so the hypotheses hold). $C' = C + (x_1,x_2)$ is a cover with $|C'| = |S|$. Applying Lemma 2.1 to C' gives an optimal schedule that begins by executing (x_1,x_2), as desired.

Case 2: Level u has exactly one node.

Let x_2 be on level u, with $(x_2,y_2) \in S$. If y_2 has no predecessors (or is a dummy) we can assume S begins by executing (x_2,y_2), so the Lemma follows by induction. Otherwise, let x_1 be a predecessor of y_2 with no predecessors; let $(x_1,y_1) \in S$. Construct cover C of Lemma 2.2. Now $C' = C + (x_1,x_2)$ is a cover with $|C'| = |S|$. As above Lemma 2.1 gives an optimal schedule that begins by executing (x_1,x_2). □

Lemmas 2.4-5 are useful in transforming schedules to become lexicographic maximum. For these Lemmas take O and M as level-by-level schedules; take j, $1 \le j \le u$, so the jump sequence of O is lexicographic maximum up to, but not at, a jump at level j, while M is lexicographic maximum up to and including j.

We first refine the notion of "readiness": Let S be a level-by-level schedule, and let nodes x, y satisfy level(x) > level(y). y is x-ready in S if x does not precede y and further, all predecessors of y below level(x) are jumped from above level(x). Clearly if y is x-ready we can make (x,y) a jump, assuming x itself is not jumped.

Lemma 2.4: (a) If (x,y) is a jump in M with level(x) ≥ j, then y is x-ready in O.

(b) If (x,y) is a jump in O with level(x) > j, then y is x-ready in M. (We do not allow level(x)=j).

Proof: (a) Let (x,y) be a jump that violates the Lemma. Choose level(y) as high as possible. We derive a contradiction as follows.

Node y has a predecessor w below level(x) that is not jumped from above level(x) in O. M has a jump (v,w). Since level(w) > level(y), the Lemma holds for (v,w), so w is v-ready in O. Hence, w is ready in O when level(x) is reached, and O can be modified to jump

from level(x) to w. The modified schedule has a larger jump sequence than M (on or before level j). This is the desired contradiction.

(b) The argument is similar. It is necessary to assume level(x) > j, so that modifying M to jump from level(x) to w increases the jump sequence. □

We use "chains" to modify schedules, as illustrated in Figure 2. A chain consists of nodes x_i, $1 \le i \le c+1$, and y_i, $0 \le i \le c$, for $c \ge 1$, where

(i) (x_i, y_i) is a jump in 0, for $1 \le i \le c$;

(ii) (y_i, x_{i+2}) is a jump in M, for $0 \le i \le c-1$;

(iii) $\text{level}(x_i) = \text{level}(y_{i-1})$, for $1 \le i \le c+1$.

These conditions imply all nodes in a chain are distinct, except possibly $x_1 = y_0$ and $x_{c+1} = y_c$. (It is also possible that x_{c+1} and y_c are dummies.)

Lemma 2.5: (a) Let (s,t) be a jump in M with $\text{level}(x_i) > \text{level}(s) > \max(\text{level}(y_i), j-1)$ for some i, $1 \le i \le c$. Additionally for $i = c$, y_c is not jumped from above level(s) in M. Then $\text{level}(t) \ge \text{level}(y_i)$.

(b) Let (s,t) be a jump in 0 with $\text{level}(x_i) > \text{level}(s) > \max(\text{level}(y_i), j)$ for some i, $1 \le i \le c$. Additionally for $i = c$, x_{c+1} is not jumped from above level(s) in 0. Then $\text{level}(t) \ge \text{level}(y_i)$.

Proof: We prove (a); (b) is similar. We show that in M,

(i) y_i is s-ready

(ii) y_i is not jumped from above level(s).

Together these imply (s, y_i) is a valid jump for M. Since M is lexicographic maximum on or above j, $\text{level}(t) \ge \text{level}(y_i)$, as desired.

For (i), note y_i is x_i-ready in M, by Lemma 2.4. This implies y_i is s-ready, since level(s) < level(x_i) and s is not jumped in M.

For (ii), if $i < c$, M jumps from y_i, so y_i is not jumped. If $i = c$, (ii) holds by hypothesis. □

The next argument refers to a k-schedule. This is a level-by-level schedule for levels u,...,k (and nodes in lower levels jumped from these levels). A k-schedule is just a prefix of a complete schedule. We also refer to a lexicographic maximum k-schedule, i.e., one whose jump sequence is maximum. Such a jump sequence is a prefix of the jump sequence of a lexicographic maximum schedule.

Theorem 2.1: A lexicographic maximum schedule is optimal.

Proof: Suppose the Theorem is false. Let j be the lowest level such that some optimal level-by-level schedule has a lexicographic maximum jump sequence up to but not including a jump at level j. Let O be such a schedule. Let M be a lexicographic maximum j-schedule. (Thus M's jump sequence is maximum up to and including j. So Lemmas 2.4-5 apply to O and M. It does not matter that M is a j-schedule rather than a schedule). Take O and M to have as many jumps in common as possible. We derive a contradiction, by showing O can be modified to decrease j, or one of O and M can be modified to get more common jumps. In short, a "better" schedule than O or M can be found. We consider two cases:

Case A: O and M agree on all jumps except the one from level j.

Case B: O and M do not agree on all jumps above level j.

Case A: Let J consist of all nodes on or below level j, not jumped from above level j. O restricts to an optimal schedule O_j on J.

We obtain the desired contradiction by modifying O_j so it starts with M's jump from level j.

Let the jumps from level j be (x_1,y_1) in O, (y_0,x_2) in M, so $level(x_1) = level(y_0) = j$, $level(y_1) < level(x_2)$. Assume $x_1 \neq y_0$; otherwise the argument is similar but simpler. Let (x_0,y_0), $(x_2,y_2) \in O$. So $level(x_0) = level(y_0)$. We obtain the desired schedule from a cover.

First get a cover C of $J - x_1 - x_2$, by applying Lemma 2.2 to O_j and (x_1,y_1), (x_2,y_2). (The hypotheses of Lemma 2.2 obviously hold.) It is easy to see from the proof of Lemma 2.2 that C and O_j agree on level j, except at x_1. So $(x_0,y_0) \in C$. Set $C' = C - (x_0,y_0) + (x_0,x_1) + (y_0,x_2)$.

C' is a cover of J, and $|C'| = |O_j|$. On level j, C' has exactly one jump (y_0,x_2). From Lemma 2.1, C' gives an optimal schedule that begins by executing level j, with jump (y_0,x_2). From Lemma 2.3, levels below j can be executed level-by-level. This gives the desired schedule.

Case B: Take jumps (x_1,y_1) in O and (y_0,x_2) in M, with $level(x_1) = level(y_0)$, $level(y_1) = level(x_2)$, $(x_1,y_1) \neq (y_0,x_2)$, and $level(x_1)$ as high as possible. Let these jumps be the start of a chain (defined as in (i)-(iii) above), with c maximal. The chain ends in one of several ways:

Case 1: There is no jump from y_c in M.

Case 2: There is no jump from x_{c+1} in O, and $level(x_{c+1}) \geq j$.

Case 3: There are jumps (x_{c+1},y_{c+1}) in O, (y_c,x_{c+2}) in M, with $level(x_{c+1}) = j$, $level(y_{c+1}) < level(x_{c+2})$.

These cases exhaust all possibilities. For if Case 1 does not apply,

level(y_c) = level(x_{c+1}) $\geq$ j, since M is a j-schedule. If Cases 1 and 2 do not apply, the jumps in O and M are to different levels; this only occurs at level j. (Note Case 1 includes the possibility that y_c is a dummy.)

Now we derive a contradiction, by showing that in each case we can get more common jumps.

Cases 1-2: The arguments are similar. We give the argument for Case 2. Modify O to O', as follows:

(i) Replace jump (x_i,y_i) by (y_{i-1},x_{i+1}), for $1 \leq i \leq c$.

(ii) If $y_0 \neq x_1$ and $(u,y_0) \in O$, replace it by (u,x_1).

(iii) If $y_c \neq x_{c+1}$ and $(v,x_{c+1}) \in O$, replace it by (v,y_c).

Clearly $|O'| = |O|$, O' has the same jump sequence as O and more jumps in common with M. We will show O' is a valid schedule, i.e., it respects the precedence constraints. Then these properties show O' is a better choice than O, the desired contradiction.

We show O' is valid in two steps. First we show the new jumps in O' (those of (i)-(iii)) are valid. Then we show the old ones (those in $O \cap O'$) remain valid.

Consider the new jumps of (i). Note x_{i+1} is y_{i-1}-ready in O, by Lemma 2.4. Thus the new jump (y_{i-1},x_{i+1}) is valid. (In Case 1 this argument uses the fact that level(x_c) > j.)

(ii) does not introduce jumps, since it is easy to see level(u) = level(y_0): (u,y_0) is not a jump from u, by the choice of level(x_1). It is not a jump to u, since O has jump (x_1,y_1).

For (iii), level(v) $\geq$ level(x_{c+1}) (by Case 2). It suffices to show level(v) < level(x_c). For then (v,y_c) is valid because (x_c,y_c) was.

Assume on the contrary, level(v) > level(x_c). The choice of level(x_1) implies level(v) < level(x_1). Now Lemma 2.5 applied to jump (v,x_{c+1}) implies

level(x_{c+1}) ≥ level(y_i) for some i < c, a contradiction.

Now consider an old jump (s,t) of O. Clearly (s,t) remains valid if level(s) > level(x_1) or level(s) ≤ level(x_{c+1}). So assume level(x_i) > level(s) > level(y_i) for some i, $1 \le i \le c$. If i = c and x_{c+1} is jumped from above level(s) in O, then again it is clear (s,t) is valid. Otherwise, all hypotheses of Lemma 2.5 hold. (Note level(s) > j, by Case 2. In Case 1 level(s) ≥ j.) Thus level(t) ≥ level(y_i), and (s,t) remains valid.

Thus all jumps of O' are valid, as desired.

Case 3: We construct a schedule P that is better than O, in two steps.

First modify O to a j-schedule O', in a way similar to Cases 1-2:

(i) Replace jump (x_i,y_i) by (y_{i-1},x_{i+1}), for $1 \le i \le c+1$.

(ii) If $y_0 \neq x_1$, and $(u,y_0) \in O$, replace it by (u,x_1).

O' is a valid lexicographic maximum j-schedule. The proof is the same as in Cases 1-2. Note to show the new jump (y_c,x_{c+2}) is valid, we use Lemma 2.4 (a) with level(x) = j. Also, O does not jump x_{c+2} from above j, as in the argument for v in Cases 1-2.

Now construct P: Let O'' be an optimal level-by-level schedule on the nodes below j not jumped in O'. Take P to be O' above and on level j, and O'' below it.

P is a valid level-by-level schedule. It has the same jump sequence as M up to and including level j, by (i). Finally, P is optimal. For let J consist of all nodes on or below level j not jumped from above j in O. Now follow the argument of Case A: O restricts to an optimal schedule O_j on J. O_j can be modified to a schedule of the same cardinality on $J - x_{c+1} + y_c$. The schedule begins by executing level j with jump (y_c,x_{c+2}). Below level j the schedule is O''. Thus, since |O'| is the size of O up to and including level j, and |O''| is the size

of O below level j, $|P| = |O|$, and P is optimal.

The above properties of P show it is a better choice than O, a contradiction. □

3. Scheduling Algorithm

This section shows how to compute a lexicographic maximum schedule in time $O(m + n\,\alpha(n))$.

We start by sketching a simple two-pass procedure that is $O(n^2)$. Pass I computes the jump from level f (if it exists), for f = u, ..., 1: It finds the highest level t that f can jump to. If level t contains several nodes that can be jumped, it guesses one arbitrarily.

This guess is practically irrelevant. It clearly does not effect a subsequent jump to level t or above. Suppose some level jumps below t. At that point level t contains no ready nodes. Any lexicographic maximum schedule must have jumped all the nodes in level t that Pass I jumped. So the highest level with a ready node in Pass I is the lexicographic maximum level.

The guess does matter in computing the jump from level t. Pass I may inadvertantly jump a node that gives the best jump from t. To remedy this, Pass I keeps track of the nodes that must be jumped (as explained above.) The remaining nodes are called "free" nodes. Pass I always computes the best jump from a free node in level f. Pass II fixes up bad guesses, i.e., free nodes that were inadvertantly jumped.

This approach leads to an $O(n^2)$ algorithm. The bottleneck is finding the highest level t to jump to. Because of the arbitrary

behavior of t, it is not apparent how to do better than a sequential search for t. Doing O(n) such searches uses time $O(n^2)$.

For greater efficiency we restructure the computation. Pass I computes the jumps to level t, for t = u, ..., 1: For each node y in level t, it finds the highest level f that can jump to y. It guesses that f jumps to y. As above, this guess is irrelevant to computing subsequent jumps, except for the jump from level t. Pass I keeps track of the free nodes in each level, and always computes jumps from free nodes. Pass II fixes bad guesses.

This second approach has the advantage of a simpler "highest level" computation. The first approach computes the highest level t to jump to; a given t may be highest at various, arbitrary times. The second approach computes the highest level f to jump from; a given f is highest only once. (After its jump has been found, f is no longer a candidate.) This fact permits the use of set merging techniques to replace an O(n) search by an $\alpha(n)$ FIND.

Now we state the algorithm in pseudo-Algol. The schedule is specified in arrays FROM and TO. For $u \geq f \geq 1$, (FROM(f), TO(f)) is the jump from level f. If TO(f) = -1, there is no jump from f i.e., level f is even when it is executed. If TO(f) = 0, node FROM(f) is scheduled with an idle period. Clearly these arrays give enough information to deduce the entire schedule, if desired.

Pass I uses two main data structures. First, it uses set merging techniques to find when nodes become ready and to assign jumps. When level t is being scanned, a level $f \geq t$ is called <u>unassigned</u> if either f = t, or f > t and the jumps to level f make

f odd (so there is a jump from f) but TO(f) = 0 i.e., no non-trivial jump has been found. Each unassigned level f has a set,

$U(f) = \{g \mid u \geq g \geq f$ and f is the highest unassigned level $\leq g\}$.

These sets are manipulated by operations FIND(g) (which returns f with $g \in U(f)$) and UNION(f,g) (which does a destructive merge of U(f) into U(g)) [AHU].

The second data structure is used to assign jumps and find free nodes. When Pass I scans level t, it computes for each node y in level t the value

R(y) = the highest unassigned level that can jump to y,

and for each unassigned level $f > t$ the list

$LIST(f) = \{y \mid R(y) = f\}$.

LIST is used to assign jumps. It is also used to compute a substitute jump node in level t, SUB(t). Call a node y in level t free if

$y = TO(f)$ implies $f \leq R(SUB(t))$.

If a free node y is TO(f), y need not be jumped: SUB(t) can be jumped instead. Pass II uses SUB(t) to eliminate situations where a free node y is both a FROM and a TO value.

```
procedure LMS; comment An arbitrary dag is given as input. LMS
      returns the jumps of a lexicographic maximum schedule in
      arrays FROM and TO;
begin

Initialization:

0. do a breadth-first search of the dag to define levels u,...,1;
     set SUB(t) = 0, TO(t) = 0, U(t) = {t}, LIST(t) = φ, for u ≥ t ≥ 1;
     set U(0) = {0}, R(0) = 0;
```

```
Pass I:

1.  for t ← u to 1 do begin
2.     for each node y in level t do begin
3.        if y has no predecessors then R(y) ← FIND(u)
             else begin
4.           ℓ ← min {e|a predecessor p of y is executed at level e,
                i.e., p = TO(e), or p is in level e and is not jumped};
5.           if TO(ℓ) = 0 and some free node in level ℓ does not precede y
                comment the test for "free" is described in the text above;
                then R(y) ← ℓ
                else R(y) ← FIND(ℓ-1);
             end;
6.        if R(y) > t then add y to LIST (R(y));
          end;
7.     while some LIST(f) ≠ φ  do begin
8.        remove the first node y from LIST(f);
9.        TO(f) ← y; g ← FIND(f-1); UNION(f,g);
10.       if g > t then add LIST(f) to the end of LIST(g)
             else if LIST(f) ≠ φ then SUB(t) ← the last node in LIST(f);
          end;
11.    if level t has an even number of nodes that are not jumped (i.e., not
          TO values) then
          begin TO(t) ← -1; UNION(t,t-1) end;
   end Pass I;

   Pass II:

12.  for f ← 1 to u do begin
13.    if TO(f) ≥ 0 then begin
14.       let FROM(f) be a free node in level f, that does not precede TO(f)
             if TO(f) > 0;
15.       if FROM(f) = TO(g) for some g then TO(g) ← SUB(f);
   end  end  end LMS;
```

Figure 3 illustrates this algorithm. Now we show it is correct. Lemmas 3.1-2 show LMS finds a valid level-by-level schedule. Lemmas 3.3-6 show the schedule is lexicographic maximum.

First consider Pass I. It only specifies the TO part of a jump. So say there is a valid semi-jump from (level) f to (node) y if

(i) level f contains a free node that does not precede y;

(ii) any predecessor of y that is below level f is TO(g) for some g > f.

Lemma 3.1: Immediately after Pass I,

(a) if TO(f) > 0, there is a semi-jump from f to TO(f);

(b) if TO(f) is a free node in level t, there is a semi-jump from f to SUB(t).

Proof: We show that during Pass I, after level t is processed (i.e., at line 11), (a)-(b) hold for level(TO(f)) ≥ t. The proof is by induction on t ($u \geq t \geq 1$). Assuming (a)-(b) hold above t, we prove them for t as follows.

First note from lines 9 and 11, the sets U(f) are maintained as in their definition above.

Now we show lines 3-5 set R(y) to the highest unassigned level with a semi-jump to y. (For the purposes of the proof itself, it is unnecessary to show R(y) is highest.) If no semi-jump exists, R(y) = t.

If y has no predecessors, R(y) is the highest unassigned level, as desired. Otherwise, consider ℓ of line 4, and let p be a predecessor executed at level ℓ. Clearly there is no semi-jump from above ℓ to y. Since p is the last predecessor executed, there is a semi-jump to y from any level below ℓ. Level ℓ itself can semi-jump y exactly when

the test of line 5 is true (by the definition of semi-jump). These remarks show line 5 sets R(y) as desired.

Line 6 clearly sets LIST(f) as in the definition above. So the first time line 7 is reached, an unassigned level f has a semi-jump to each node y in LIST(f). This property is maintained every time through the loop (lines 7-10). Thus line 9 assigns valid semi-jumps, and (a) holds.

For (b), note if TO(f) is free in level t, then by definition $f \leq R(SUB(t))$. The above characterization of R shows there is a semi-jump from f to SUB(t). □

At the end of Pass II, let J be the level-by-level execution of the dag defined by jumps (FROM(f), TO(f)), $u \geq f \geq 1$.

Lemma 3.2: J is a valid schedule.

Proof: We show first that J is well-defined, and then that J respects precedence constraints.

"Well-defined" means that node FROM(f) exists, and further, it is not jumped. These two facts imply there is a level-by-level execution with jumps (FROM(f), TO(f)).

Consider a given level f. Pass II may change TO(f) in line 15. However this occurs at most once, and when lines 13-15 are executed for the given f, either

(i) TO(f) = y, the value computed in Pass I,

or (ii) TO(f) = SUB(level(y)).

So in line 14, FROM(f) exists (by Lemma 3.1 (a)-(b)). Line 15 insures FROM(f) is not jumped. So J is well-defined.

To show J respects precedence, we show for any jump (FROM(f), TO(f)), a predecessor p of TO(f) that is below f is jumped from above f. Lemma 3.1 (a)-(b) shows that after Pass I, some level g > f has TO(g) = p. It suffices to show p is not free. For then line 15 does not change TO(g), and level g jumps p, as desired.

Let p be on level t. After the loop of lines 7-10, R(SUB(t)) $\in$ U(t). But f $\notin$ U(t), since f is unassigned at level t. Hence f > R(SUB(t)). So g > R(SUB(t)) and by definition, p is not free. □

Lemma 3.6 shows J is a lexicographic maximum schedule. Consider Pass I after level t is processed (more precisely, right before line 11 is reached). Define a family of sets V(f) as follows. If f is an unassigned level, f > t, then V(f) = U(f). If f = t, V(t) = U(t) $\cap$ {f|f > R(SUB(t))}. Otherwise V(f) does not exist. (In Figure 3 for t = 2, V(2) = {10}.)

In level t, the sets TO(V(f)) partition the non-free nodes. Note this implies Pass II does not change the values TO(V(f)). Thus "TO(V(f))" has a unique meaning.

Let M be a lexicographic maximum schedule. Lemma 3.6 proves that for all levels t and all sets V(f), the following is true:

(a) M jumps exactly the same nodes of level t from V(f) as J does, i.e., those nodes of level t in TO(V(f)).

(b) M jumps below level t from U(f) (sic) exactly when J does, i.e., from level f for f > t.

In (b), the sets U(f) are those sets immediately after level t is processed. So these sets partition the levels t and above. It is easy to see that if (b) holds for all levels t' $\geq$ t, then for such t', M jumps from f to t' exactly when J does. In particular when t = 1, J is lexicographic maximum.

Lemma 3.6 is proved by double induction: assuming (a)-(b) for levels $t' > t$, and on level t for sets $V(f')$, $f' > f$, it shows (a)-(b) for V(f). Lemmas 3.3-5 are used in the induction. So in these Lemmas assume (a)-(b) for t', f' as above.

Lemma 3.3 extends Lemma 3.2 to certain schedules used to compare J and M. In U(f) let g be the lowest level (if any) that jumps to t in J. Construct a g-schedule G as follows:

1. Define values TO(k) for $k \geq g$: If level k does not jump below t in J, then TO(k) is the value set by Pass I. Otherwise TO(k) is the node jumped by M (if any).

2. Modify these values to a g-schedule G: Run Pass II on these values, initializing f (in line 12) to g rather than 1. (The free nodes are as in Pass II for J).

Lemma 3.3: G is a valid g-schedule. It has the same jump sequence as M, except possibly for jumps from U(f) to t in J.

Proof: The characterization of G's jump sequence follows easily from the construction. Note in step 1 above, if TO(k) is the node jumped by M then it is below level t. For in this case k jumps below t in J. So inductive assertion (b) applied to set U(k) shows M jumps below t.

The argument for validity follows Lemma 3.2. First we show G is well-defined. Cases (i)-(ii) are as before. In addition there is the possibility

(iii) TO(f) = y, the node jumped by M.

In this case M has a jump (x,y). So node x is not jumped in M. This implies x is free, since M jumps all non-free nodes (by inductive assertion (a)). Hence FROM(f) exists in line 14.

Next we show G respects precedence. For jumps to level t or above, the argument is as before. So consider a jump for TO(k) = y, where y is the node jumped by M. We show that if p is a predecessor of y below k, G jumps p from above k. Clearly M jumps p from above k. If p is below t, G copies M, by definition. Otherwise suppose p is on or above t. We can apply the inductive assertion (a) to level(p) and the set V(h) that jumps p (in M). (Even if level(p) = t, $h \geq k > f$, so (a) holds for V(h)). This implies p is not free. Hence G jumps p from above k, just as J does. □

Lemma 3.4 characterizes the nodes jumped by J.

Lemma 3.4: For any node y in level t and any set V(h), $y \in TO(V(h))$ if and only if $R(y) \in V(h)$.

Proof: First suppose $R(y) \in V(h)$. Consider the loop of lines 7-10. As long as y is on some LIST, it is on LIST(k), where U(k) is the set that currently contains R(y).

Suppose $h > t$. Since h is unassigned after the loop, LIST(h) is always empty. So y is assigned as a TO value, i.e., $y \in TO(V(h))$, as desired.

Suppose $h = t$. It suffices to show that when U(k), the set currently containing R(y), gets merged into U(t), $y \notin LIST(k)$. For then as above, $y \in TO(V(t))$. (Note showing $y \notin LIST(k)$, for all y, shows LIST(k) is actually empty.)

Suppose on the contrary, $y \in LIST(k)$. Then SUB(t) gets assigned a value so that $R(SUB(t)) \geq R(y)$. Although SUB(t) may subsequently change, R(SUB(t)) only increases. So the inequality holds at the end of the loop, and $R(y) \notin V(t)$, a contradiction.

Now we show the opposite implication of the Lemma. Suppose

$y \in TO(V(h))$. It is clear from the loop that $R(y)$ is in $V(h)$ or above it. The latter possibility is impossible, by the argument above. So $R(y) \in V(h)$, as desired. □

Lemma 3.5 is used to show J jumps at least as high as M.

Lemma 3.5: If M jumps from level k to a node y in level t, then $k \leq R(y)$.

Proof: Let p be a predecessor of y. M and J may execute p at two different levels. However, after the iteration for level(p) in Pass I, these two levels are in the same set $U(h)$. This follows from the inductive assertions applied to level(p).

Now consider Pass I immediately before the iteration for t. By the above remark, the level ℓ of line 4 is in the same set as the lowest level where M executes a predecessor p of y. Call this set $U(h)$. Clearly $h \geq k$. So if $R(y) = h$, the Lemma holds.

Otherwise if $R(y) \neq h$, it is easy to see the test of line 5 is false, and $R(y) = FIND(\ell-1)$. Since $FIND(\ell) \neq FIND(\ell-1)$, $\ell = h$. So for the test to be false, all free nodes of level ℓ precede y (since $TO(\ell) = 0$). The inductive assertion implies M jumps all non-free nodes in level ℓ. Hence level ℓ cannot jump y in M. Thus $k \leq FIND(\ell-1) = R(y)$, as desired. □

Now we can show the desired result.

Lemma 3.6: J is a lexicographic maximum schedule.

Proof: As noted above, it suffices to prove (a) - (b), by double induction. So take set $V(f)$, where (a) - (b) hold for levels $t' > t$, and for $t' = t$ and sets $V(f')$, $f' > f$.

Consider the g-schedule G of Lemma 3.3. Its jump sequence is no less than M's (by Lemma 3.3 and inductive assertion (b)). But M is lexicographic maximum. We conclude M jumps from U(f) to t whenever J does.

This argument shows (b) if f = t. It also starts the proof of (a) (for any f). Now we complete the proof of (a), and then prove (b) for f >t.

The nodes y of level t jumped from V(f) in J are those with $R(y) \in V(f)$, by Lemma 3.4. Let g be the lowest level in V(f) that jumps to t. By Lemma 3.5, if M jumps y from g or above, then $R(y) \geq g$. By Lemma 3.4 and induction, J and M jump the nodes with R(y) above V(f) from above V(f). So the only nodes M can jump from V(f) are those with $R(y) \in V(f)$. Since M does as many jumps as J, it jumps all nodes with $R(y) \in V(f)$, as desired.

Now (b), for f > t, follows easily: M cannot jump from f to t, since this would imply an extra level $R(y) \in V(f)$. Thus f jumps below t. □

Now we consider the timing. We give some details needed for efficiency. For line 4, each node y has a list of predecessors. Further, each node p indicates the level e (if any) that has p = TO(e). These data structures make the total time in line 4 O(m+n).

For the test of line 5, each level ℓ has a count of its free nodes. The count is set after level ℓ is processed. It is easy to see this requires linear time. Line 5 compares this count with the number of free predecessors of y in level ℓ. So the total time in the test of line 5 is O(m+n).

For line 7, there is a list of levels f with $LIST(f) \neq \phi$. For

line 10, the LISTs have end-pointers. It is easy to see the loop of lines 7-10, excluding set merging, uses a total of $O(n)$ time.

Lemma 3.7: LMS uses time $O(m+n\alpha(n))$ and space $O(m+n)$.

Proof: There are at most n FINDs in line 5, and also in line 9. Lines 9 and 11 do at most n UNIONs. So the time for set merging is $O(n\alpha(n))$. The remaining processing is $O(m+n)$, from the above discussion. □

Now we combine Theorem 2.1 and Lemmas 3.6-7.

Theorem 3.1: Procedure LMS finds an optimal schedule for two processors and an arbitrary dag, in time $O(m+n\alpha(n))$ and space $O(m+n)$. □

Appendix

This Appendix proves a fundamental result due to Fujii, Kasami, and Ninomiya [FKN]. It also shows their scheduling algorithm is actually $O(n^{2.61})$.

Lemma 2.1: If C is a cover, there is a schedule with $\omega = |C|$. Further, any sequence of ready sets of C can be chosen to begin the schedule.

Proof: Construct the schedule as follows. First number the nodes topologically, i.e., any node has a larger number than any of its predecessors [Kn]. Then repeatedly schedule nodes using these two rules: (Recall that a ready node is unscheduled, by definition.)

1. If possible, choose a ready set of C, and schedule it in the next time slot.

2. Otherwise, choose $\{u,v\} \in C$ where u is ready and v has the smallest topological number possible. Let w be a ready predecessor of v, with $\{w,x\} \in C$. In C, replace $\{u,v\}$ and $\{w,x\}$ with $\{u,w\}$ and $\{v,x\}$; schedule $\{u,w\}$ in the next time slot.

In rule 2, note C has no ready singleton sets (by rule 1). Hence $\{u,v\}$ exists. Since v is not ready, it has a ready predecessor w. (Note $w \neq u$, since u and v are compatible). Clearly there is a set $\{w,x\} \in C$. $\{u,w\}$ is ready and so can be scheduled. Further, v and x are compatible: v does not precede x, since w and x are compatible; x does not precede v, else $\{w,x\}$ would be a better choice than $\{u,v\}$. Thus C remains a cover after rule 2. Further, $|C|$ never changes.

Continuing, we eventually get a cover with $|C|$ time slots. The second part of the Lemma follows from the choice allowed in rule 1. □

This construction easily leads to an $O(n^2)$ algorithm, improving the $O(n^3)$ procedure of [FKN]. This shows the algorithm of [FKN] is actually $O(n^{2.61})$.

References

[AGU] Aho, A. V., Garey, M. R. and Ullman, J. D., "The transitive reduction of a directed graph," SIAM J. Comput. 1, 1972, pp. 131-137.

[AHU] Aho, A. V., Hopcroft, J. E. and Ullman, J. D. The Design and Analysis of Computer Algorithms, Addison-Wesley, Reading, Mass., 1974.

[C] Coffman, E. G. Jr., Ed., Computer and Job-Shop Scheduling Theory, Wiley & Sons, New York, 1976.

[CG] Coffman, E. G. Jr. and Graham, R. L., "Optimal scheduling for two-processor systems," Acta Informatica 1, 3, 1972, pp. 200-213.

[FKN] Fujii, M., Kasami, T., and Ninomiya, K., "Optimal sequencing of two equivalent processors," SIAM J. Appl. Math. 17, 4, 1969, pp. 784-789. Erratum, SIAM J. Appl. Math. 20, 1971, p. 141.

[GJ] Garey, M. R. and Johnson, D. S., "Scheduling tasks with nonuniform deadlines on two processors," J.ACM 23, 3, 1976, pp. 461-467.

[H] Hu, T. C., "Parallel sequencing and assembly line problems," Op. Res. 9, 6, 1961, pp. 841-848.

[K] Kariv, O. "An $O(n^{2.5})$ algorithm for finding a maximum matching in a general graph," Ph.D. Diss., Weizmann Inst. Science, Rehovot, Israel, 1976.

[Kn] Knuth, D. E., The Art of Computer Programming, Vol. 1: Fundamental Algorithms, Addison-Wesley, Reading, Mass., 1973.

[P] Pan, V. Y., "Field extension and trilinear aggregating, uniting and canceling for the acceleration of matrix multiplications," Proc. 20th Annual Symp. of Found. of Comp. Sci., 1979, pp. 28-38.

[S] Sethi, R., "Scheduling graphs on two processors," SIAM J. Comput. 5, 1, 1976, pp. 73-82.

[U] Ullman, J. D., "NP-complete scheduling problems," J. Comput. System Sci. 10, 1975, pp. 384-393.

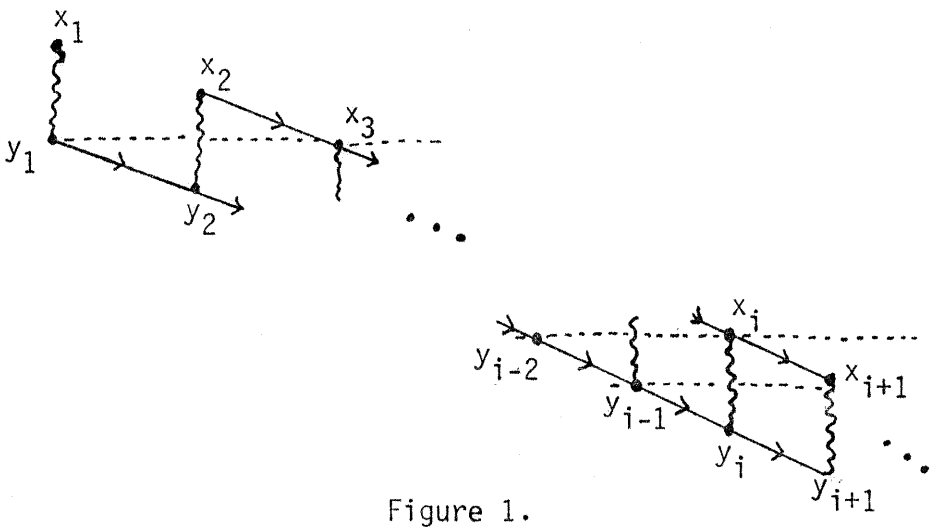


Figure 1.

Construction of Lemma 2.2

(Wavy lines indicate a pair of S)

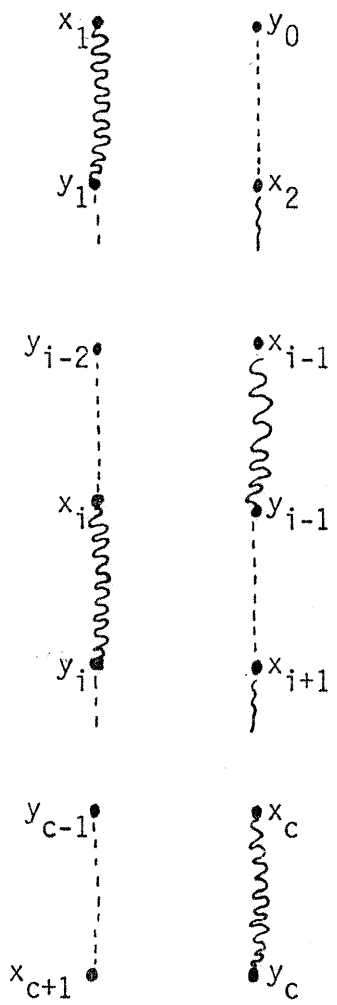
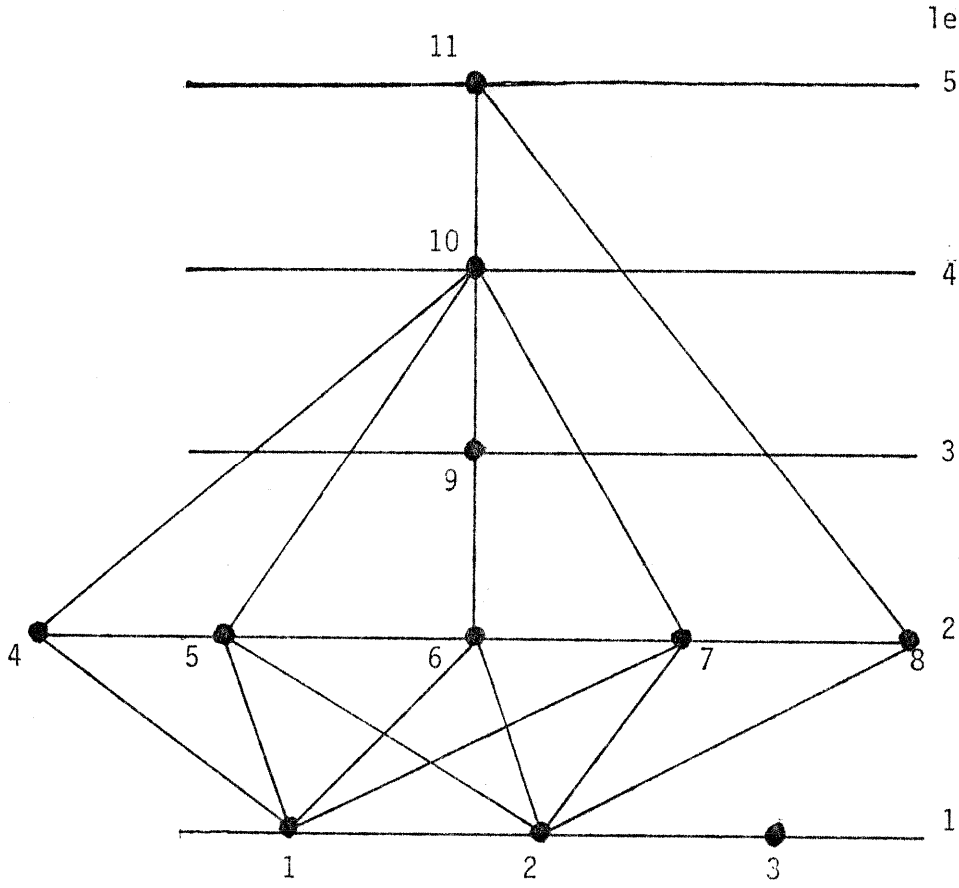


Figure 2. A chain

(Wavy lines indicate a pair of O, and dotted lines a pair of M)



(a)

y	4	5	6	7*	8
R(y)	3	3	2	3	4

(b)

f	5	4	3	2	1
TO(f)	3	8	4	2	0

(c)

f	5	4	3	2	1
FROM(f)	11	10	9	4	1
TO(f)	3	8	7	2	0

(d)

Figure 3.

(a) Dag - all edges are directed downward.

(b) Values for level 2 nodes during Pass I. * means 7 = SUB(2).

(c) Values after Pass I.

(d) Values after Pass II.