E. B. Fernández
T. Lang

Scheduling as a Graph Transformation

Abstract The scheduling of a set of tasks, with precedence constraints and known execution times, into a set of identical processors is considered. Optimal scheduling of these tasks implies utilizing a minimum number of processors to satisfy a deadline, or finishing in minimal time using a fixed number of processors. This process can be seen as a transformation of the original graph into another graph, whose precedences do not violate the optimality constraints and has a unique basic schedule. Analysis of this transformation provides insight into the scheduling process and also into the determination of lower bounds on the number of processors and on time for optimal schedules.

1. Introduction

Consider the scheduling of a set of tasks, with arbitrary precedence constraints and known execution times, into a set of identical processors. Optimal scheduling of these tasks implies utilizing a minimum number of processors to satisfy a deadline, or finishing in minimal time using a fixed number of processors. In this paper we look at this process as a transformation of the original partial order into another partial order, whose precedences do not violate the optimality constraints and which has a unique basic schedule.

A few algorithms for optimal scheduling of arbitrary graphs have been presented [1–3]. One of them, based on an idea of Barskiy [4], implies a transformation of the graph by the addition of arcs between tasks in sets of independent vertices of the graph [3].

To obtain insight into this type of graph transformation, the precedences associated with a schedule are analyzed. This leads to the characterization of the set of graphs associated with a given schedule. In particular, it is found that the set of graphs that corresponds to a given schedule forms a Boolean lattice.

Scheduling, and the related problem of determining lower bounds for optimal schedules, can be visualized as a graph transformation, and the analysis of the changes in the precedences of the corresponding graphs gives insight into these processes. This leads to the suggestion of improvements for the algorithm in [3].

The intention of this paper is to present a unified view of some scheduling problems, which contributes to understanding them and guides one in the search for efficient algorithms.

Section 2 introduces the model and some basic definitions. The precedences associated with schedules, the nature of optimal scheduling, and the determination of lower bounds as graph transformations are discussed in Section 3. The conclusions of the previous sections are applied in Section 4 to suggest improvements for the scheduling algorithm of [3], and in Section 5 to understand why a previous lower bound expression [5] is not exact. Some conclusions are presented in the final section.

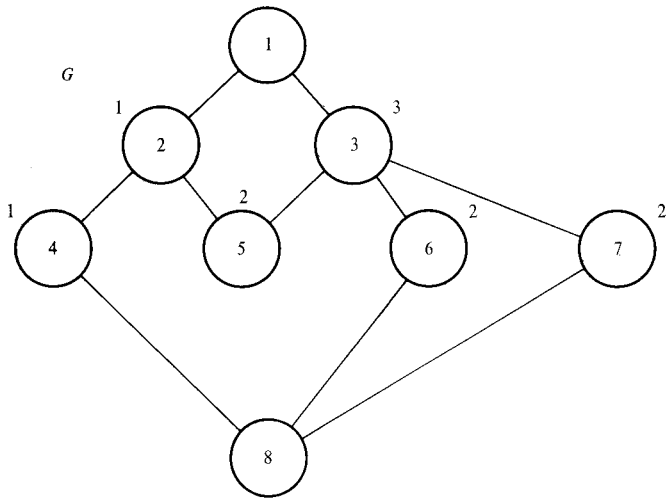
2. Model and basic concepts

A set of *tasks* $T = \{T_1, T_2, \cdots, T_n\}$, is to be executed by a set of identical *processors* P_i $(i = 1, 2, \cdots, m)$. A partial order $<$ is given on T, and a non-negative integer d_j represents the *duration of execution* of task T_j.

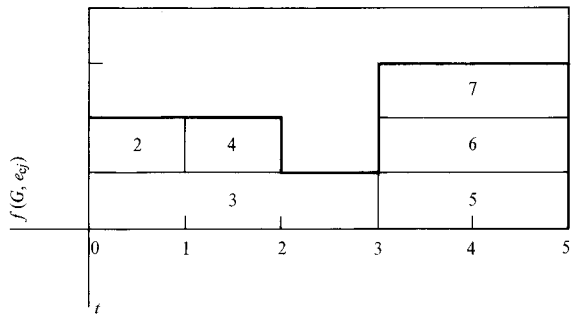
The partially ordered set $(T, <)$ is described by a finite, acyclic digraph $G = (V, A)$, where V is a finite set of vertices of cardinality n, and A is a set of arcs represented as vertex pairs. The tasks T_i correspond to the elements of V, and the terms tasks or vertices are used interchangeably. The arcs in A describe the precedences among the tasks. We assume (without loss of generality) that this graph has only one *entry vertex*, i.e., a vertex with no predecessors, and only one *exit vertex*, i.e., a vertex with no successors; that every vertex is reachable from the entry vertex; and that there exists at least one path from it that reaches the exit vertex.

In the following sections we draw graphs as follows.

1. The arrows of the arcs are assumed to be directed downward, and are not explicitly drawn.
2. The numbers outside the vertices are the task execution times; if no number is written the task time is zero.



T_i	e_{ij}	l_{ij}	e_{cj}	l_{cj}
2	0	2	1	3
3	0	0	3	3
4	1	4	2	5
5	3	3	5	5
6	3	3	5	5
7	3	3	5	5



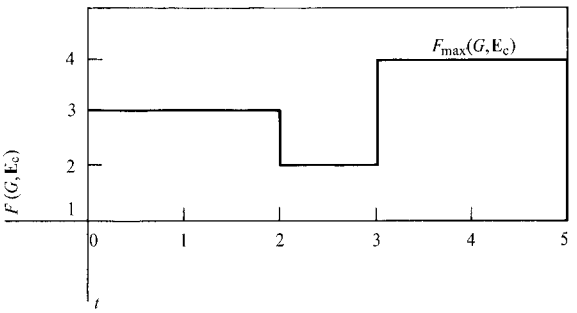


Figure 1 Basic concepts.

Once a processor begins to execute a task it cannot be interrupted until its completion; that is, we have a *non-preemptive* type of scheduling. It is assumed that tasks are scheduled starting only at integer values of time.

The length of the critical path of the graph, t_{cp}, is the minimum time to perform the set of computations. It would be more general and more useful to define a deadline, D, i.e., a time within which the set of computations must be finished. Clearly, $D \geq t_{cp}$. According to some possible schedule, for each task T_j we have a specific *completion time* which we denote as c_j. C is the completion time vector, whose jth component is c_j. Of particular interest are the two extreme task completion times defined below.

The *earliest completion time*, e_{cj}, of a task T_j, is the minimum time in which this task can be finished given the precedence constraints of the graph.

The *latest completion time*, l_{cj}, of a task T_j, indicates how long the completion time of this task can be delayed without exceeding the deadline.

According to a possible schedule, there exists also an *initiation time* i_j for a given task T_j. Analogously, it is then possible to define for a task T_j, an *earliest initiation time*, e_{ij}, and a *latest initiation time*, l_{ij}.

The partially ordered set is represented by a digraph in which no redundant precedences are drawn. In other words, if G contains arc (i, j) and arc (j, k), an arc of the form (i, k) is not accepted as part of the set of arcs. A graph of this type is said to be in its *τ-minimal form* [6]. There exist general methods to convert a graph which is not in this form to its minimal equivalent form, such as the algorithms in [7, 8]. For the particular type of partial orders corresponding to computational graphs, however, a simpler algorithm is possible [3].

Two tasks T_i, $T_j \in T$ are *comparable* if either $T_i < T_j$ or $T_j < T_i$. A subset B of T is *independent* if every two elements of B are not comparable. An independent subset is also called an *antichain*. A subset of T is a *chain* if every two of its elements are comparable.

The *activity* of task T_j in graph G is defined as

$$f(G, c_j) = \begin{cases} 1, \text{ for } t \in [c_j - d_j, c_j] \\ 0, \text{ otherwise.} \end{cases}$$

The *load density function* is defined by

$$F(G, C) = \sum_{j=1}^{n} f(c_j, t).$$

Then, $f(G, c_j)$ indicates the activity of task T_j along time, according to some schedule that does not violate the restrictions imposed by the graph, and $F(G, C)$ indicates the total activity as a function of time. Clearly, $\max_t F(G, C)$ indicates the number of processors required for the schedule defined by $F(G, C)$.

If we call $\mathbf{E}_c$ the vector of the earliest completion times of graph G, then $F(G, \mathbf{E}_c)$ is the *earliest load density function*, that is, it corresponds to a schedule where all tasks are processed as early as possible. Similarly, calling $\mathbf{L}_c$ the vector of the latest completion times l_{cj}, we can define a *latest load density function*. We denote by $F_{\max}(G, C)$ its maximum along time.

Due to the precedences of the graph not all the vertices in an antichain are simultaneously active according to the load density function in consideration. If we use some specific load density function as reference, the sets of simultaneously active vertices will be called *active antichains*. When not said otherwise, it will be understood that we use the earliest load density function as a reference.

For a given schedule, we indicate as ω the *total time to process all the tasks in T*.

A graph G *includes* a graph G', denoted by $G \supseteq G'$, if G possesses at least all the precedences of G'. In analogy with set theory it is possible to define for graphs the operations of: 1) precedence union, $G_1 \cup G_2$, denoting the set of the precedences either in G_1 or G_2; and 2) precedence intersection, $G_1 \cap G_2$, denoting the set of precedences common to G_1 and G_2. Figure 1 illustrates some of these concepts.

3. Scheduling as a graph transformation

• *3.1. Precedences of the original graph*

Given a graph G there exist a set of precedences corresponding to the ordering of the elements of V. Some of these precedences are defined by an arc of G, i.e., an element of the set A, and others are implicit due to the transitivity of the ordering relation. When the partial order describes a set of computations these precedences correspond to logical dependencies of the vertices and, if G is in its τ-minimal form as assumed earlier, no arc can be removed without violating some of the logical constraints of the computations. Therefore, we assume that G has the minimum possible number of precedences consistent with the logical constraints of the set of computations.

The *set of logical precedences* of G, P_L, is the set of all the precedences implied directly by the set of arcs, A,

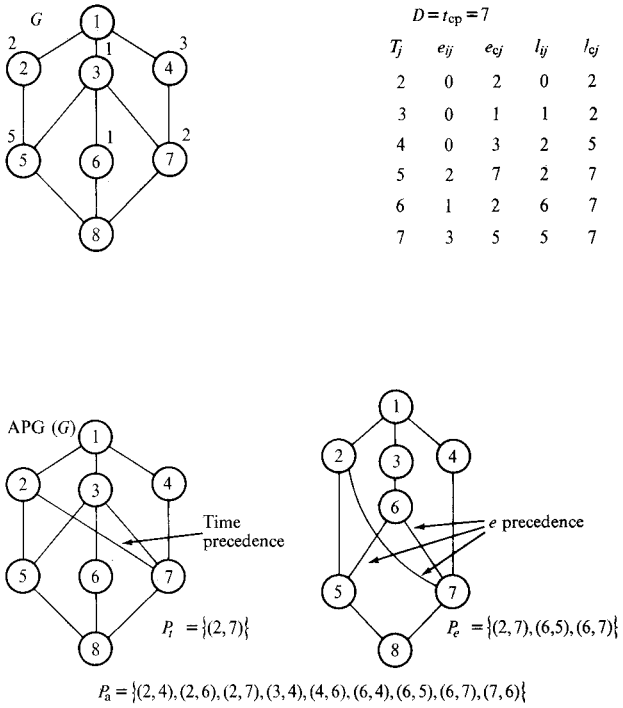


Figure 2 Graph precedences.

and by the transitivity of the ordering relation. Notice that $P_L = G^t$, the transitive closure of G. Due to the earliest and latest completion times of each task in G, there are tasks which always finish before others start, which implies the existence of time precedences.

The *set of time precedences*, P_t, is the set of all those precedences $(\alpha, \beta) \notin P_L$, such that $e_{i\beta} \geq l_{c\alpha}$. An *all precedence graph*, APG, is the graph obtained by adding to G the time precedences P_t, and eliminating the redundant precedences (this is called in [4] a "complete informational graph"). An APG is clearly equivalent to G with respect to the logical relationships of the tasks, and the set P_t only makes explicit precedences that are implicit by the timing relationships of the network.

The *set of e-precedences*, P_e, is the set of precedences $(\alpha, \beta) \notin P_L$, such that $e_{i\beta} \geq e_{c\alpha}$. That is, the e-precedences are the implicit precedences which exist when all the tasks are in their earliest positions. Notice that $P_e \supseteq P_t$. Similarly, if all the tasks are in their latest positions we have a *set of l-precedences*, P_l, which are those precedences $(\alpha, \beta) \notin P_L$, such that $l_{i\beta} \geq l_{c\alpha}$. The set of *admissible precedences*, P_a, is the set of those precedences $(\alpha, \beta) \notin P_L$, such that $l_{i\beta} \geq e_{c\alpha}$. That is, P_a contains precedences which can be added to G without its critical path exceeding D. Clearly, $P_a \supseteq P_e$.

Figure 2 illustrates the concepts above. Notice that while P_t and P_e contain precedences which are mutually compatible, P_a contains precedences which could be

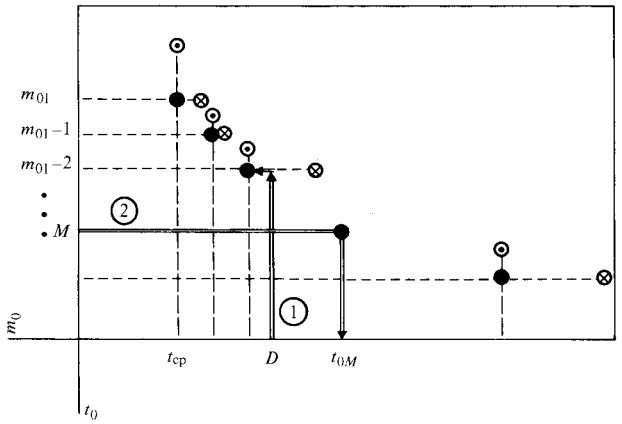


Figure 3 Relationship between optimal m and optimal t.

mutually incompatible, i.e., adding one of them could invalidate any of the others. The reason for this is that adding one of these precedences forces the task at the head of the arc, γ, to be performed at a time which could be later than its earlier starting time, thus preventing the existence of some precedences in P_a which have γ as a tail vertex (and which were defined assuming γ to be in its earliest position).

• *3.2. Basic scheduling and optimal scheduling*

Two of the fundamental problems in scheduling theory are: 1) a deadline for the execution time of a given set of tasks must be satisfied using a minimum number of processors; and 2) a fixed number of processors must be used to execute a set of tasks in a minimum time. There are some dual aspects in these two problems although they are not completely symmetric. In this paper we consider the first of them, but most of the considerations apply to the second one as well.

A *basic scheduling* strategy [9] consists of assigning a task whose predecessors have been completed to the first available processor. In other words, basic schedules contain no artificially idle processors, that is, if a processor is idle it is because no task can be assigned to it. We denote as $S(G)$ the set of all the basic schedules of G. $S(G)$ may contain many schedules because there could be more than one task ready when a processor becomes available.

It is known that $S(G)$ may not contain the optimal schedules of G [2], i.e., allowing some idle time can decrease the number of processors or the execution time with respect to the best corresponding schedules in $S(G)$. However, if G is transformed in the way discussed later, it can be shown [4] that it is then possible to obtain the optimal schedules of G by determining only the basic schedules of the transformed graph.

Figure 3 shows the relationship, for a given graph G, between m_0 (the minimum number of processors required to execute the set T within the deadline D) and t_0, (the minimum time in which the set T can be completed with m_0 processors). The ⊗'s represent the minimum values of time for basic schedules using a given number of processors, and the ⊙'s represent the minimum number of processors required to achieve a given deadline using basic scheduling on G. The two basic optimization problems can then be expressed as follows.

1. For a given deadline D the corresponding value of m_0 (given by the nearest optimal point on the left), must be determined.
2. For a given value of m, say M, the corresponding minimal time, t_{0M}, must be determined. Notice that in general, basic scheduling will not be optimal, in fact for time-optimal schedules it can give times almost twice as long as those obtained with optimal schedules [2, 9].

• *3.3. Graphs associated with a schedule*

Consider an arbitrary schedule s, described by a Gantt chart in Fig. 4(a). The sequencing of the tasks in this schedule determines implicit precedences among them, i.e., if $c_j \preceq i_k$, then $T_j < T_k$. If we study all the graphs that satisfy this schedule, i.e., whose scheduling does not violate the precedences established by s, we obtain a set, which is characterized below.

Associated with a given schedule s, there exists a set of graphs, C_s, where a graph $G_i(s)$ in C_s has the same set of weighted vertices as G, and does not contain any precedences that violate the precedences of s. The set C_s is composed of the following graphs.

1. A *total-precedence graph*, G_{tp}, where all the precedences of the schedule appear as logical precedences.
2. A *set of minimum-precedence graphs*, MPG, where there exist a minimum number of the logical precedences of G_{tp}, necessary to maintain m chains between the entry and exit vertices.
3. All the graphs G_c such that $G_{tp} \supseteq G_c \supseteq G_{mp}$, where G_{mp} is a graph in MPG.
4. All the graphs $G_d(s)$ that can be obtained by deleting precedences from the three sets above.

The graphs in 1), 2), and 3) have exactly m chains if s is a schedule for m processors, while the graphs in 4) have more than m chains. The set in 4) includes I, the graph of independent tasks. Clearly, all the schedules that can be obtained by permuting processors, for example switching the processors for T_6 and T_7 in Fig. 4(a) have the same C_s. Figure 4 shows some of the graphs in the C_s of a given schedule.

As the set of graphs in C_s is obtained by starting from G_{tp} and deleting precedences until the graph of indepen-

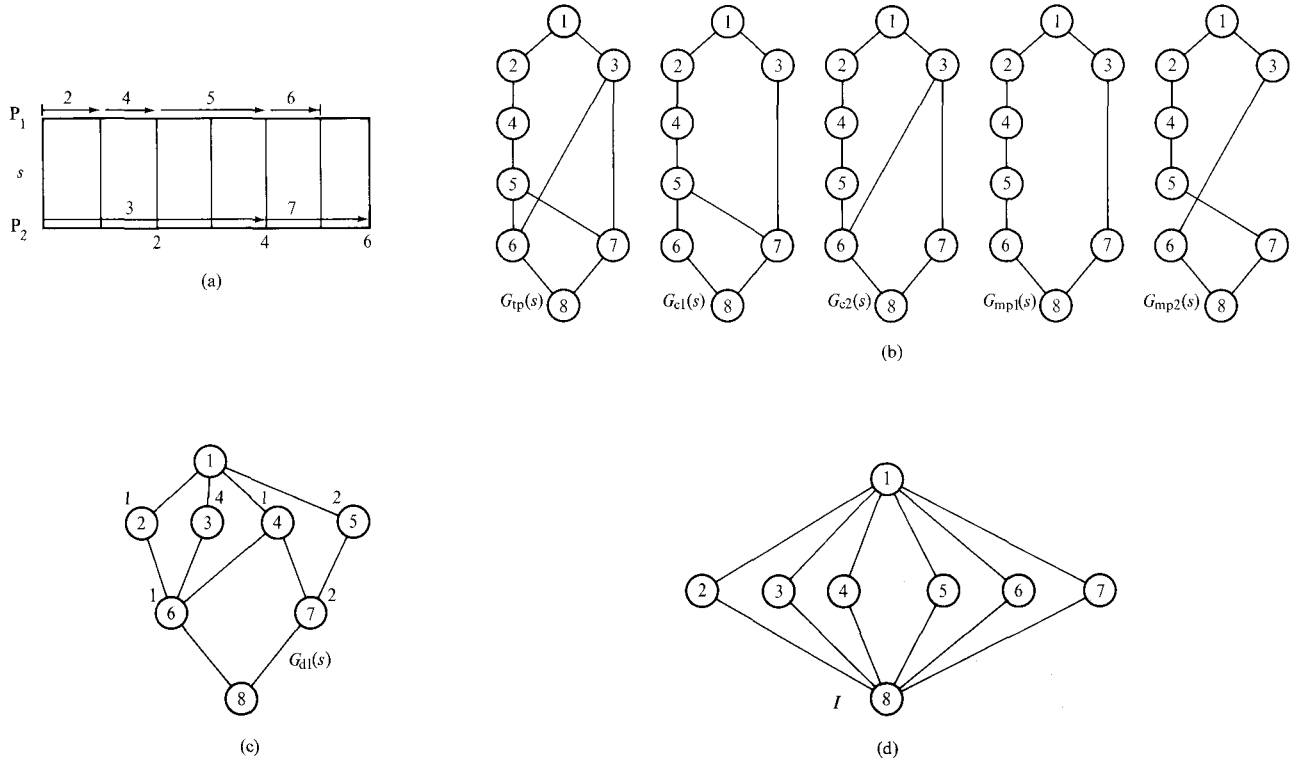


Figure 4 Graphs associated with a schedule.

dent tasks is reached, it constitutes a partial ordering. Furthermore, C_s is isomorphic to the set of subsets of a given set (in this case the original set is the set of precedences in s), therefore C_s together with the operations of precedence-union and precedence-intersection (defined in Section 2), constitutes a Boolean lattice [10]. Similarly, the partial orders that start from any G_i in C_s and go down to I or up to G_{tp}, by deletion or addition or precedences are Boolean lattices. Figure 5 shows the general form of the partial ordering of C_s, indicating the number of chains of its graphs.

• *3.4. Interpretation of the scheduling process*

The optimal scheduling of a graph G can be described as transforming G into a graph G' defined below, and performing basic scheduling on G'.

For processor-optimal schedules we have the mapping $K_p: G \rightarrow G'$ such that $F_{max}(G', \mathbf{E}_c)$ is a minimum and $t_{cp}(G') \leq D$, and for time-optimal schedules the mapping is $K_T: G \rightarrow G'$ such that $F_{max}(G', \mathbf{E}_c) \leq m$ and $t_{cp}(G')$ is a minimum.

The determination of processor-optimal schedules can be visualized in terms of the lattices of the graphs associated with given schedules. The set of all the schedules of G defines a set of C_s lattices, one for each schedule as in Fig. 6 (considering as one schedule all the schedules which can be formed by permuting the processors associated with given tasks). All these lattices have a common part formed by G and all the graphs that can be obtained deleting precedences from G, including I. The sublattices between G and its G_{tp}'s are all different. Some of these schedules are optimal, and their C_s lattices then contain one of the G' as defined by K_p above. To go from G to any of the G' requires addition of arcs, which indicates a way of proceeding to obtain optimal schedules (indicated by a double line in the figure). A wrong choice of arcs at a given stage can lead to a schedule where $t_{cp}(G_{tp}) > D$, and backtracking is then necessary. Scheduling algorithms, such as the one given in [3] and further discussed here (Section 4), are ways to guide the choice of arcs to add in the search for a G'. The use of lower bounds on the number of processors [11], is of great value in this procedure (Section 4).

• *3.5. The number of processors*

We relate now the number of processors m to the structure of the graph and its basic scheduling.

The number of processors for maximum parallelism, m_p, is the number of required processors so as to have a processor available for every executable task in G, when a basic schedule is being constructed.

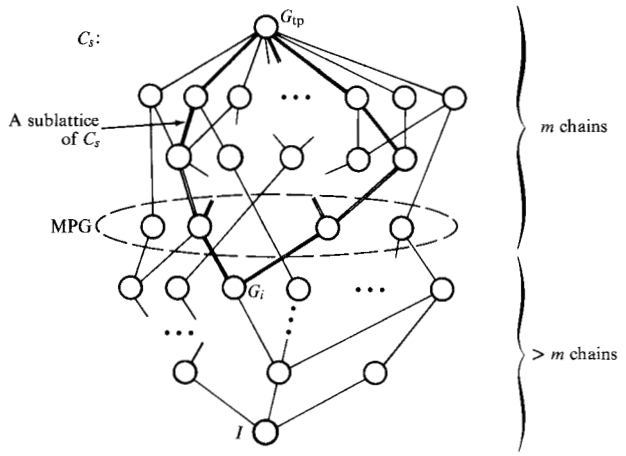


Figure 5 Lattice of the C_s graphs of a schedule s.

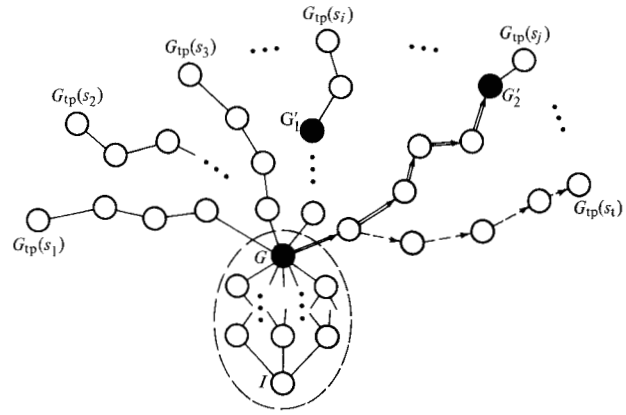


Figure 6 Optimal scheduling and schedule lattices.

Clearly, because of the definition of $F(\mathbf{E}_c, t)$ the following lemma is true.

Lemma $m_P = F_{max}(\mathbf{E}_c, t)$. Notice the difference between m_P, m_0 (the minimum number of required processors so that $\omega = D$), and m, the number of available processors. Also, $m_P \geq m_0$.

Theorem Consider a graph G to be scheduled into m processors. The following cases according to the value of m exist.

1. If $m \geq m_P$, there exists a unique basic schedule $s(G)$, and the C_s are such that
   a. $G_{tp}(s)$ is obtained by adding to G its e-precedences,
   b. $t_{cp}(G_i) \leq t_{cp}(G)$, $\forall G_i \in C_s$.
2. If $m_0 \leq m \leq m_P$, the C_s are such that all graphs G_i, $G_{mp} \subseteq G_i \subseteq G_{tp}$, contain at least $r - m$ additional precedences (including e-precedences) between the vertices corresponding to those active antichains of G that have cardinality $r > m$.
3. If $m < m_0$, all graphs G_i in C_s, such that $G_{mp} \subseteq G_i \subseteq G_{tp}$,
   a. include at least $r - m$ precedences (including e-precedences) added to all those antichains of G with cardinality $r > m$;
   b. $t_{cp}(G_i) > D$.

Proof

1. If $m \geq m_P$ as soon as the predecessors of a task are completed there is a processor ready for this task. Therefore, no precedences are forced upon the tasks to accommodate them into a given number of processors. There exists then a unique basic scheduling for G. In this basic scheduling the tasks are scheduled as early as possible, and the precedences in s are the union of the logical precedences of G and the e-precedences of G. Removing precedences from s, that is from $G_{tp}(s)$, results in graphs that have critical paths at most as long as $t_{cp}[G_{tp}(s)]$, which, since there are always processors available, is equal to $t_{cp}(G)$.
2. When $m < m_P$ some of the tasks have to be serialized to fit into a number of processors smaller than m_P. Arcs have to be added to the graph to make some tasks to share processors. If in every active antichain of cardinality $r > m$, $r - m$ precedences are added either explicitly or as the result of the e-precedences of the transformed graph, then the graph will nowhere have more than m chains.
3. Part a) is the same as 2). Part b) is proved by noting that, since $m < m_0$, there are not sufficient processors to finish the tasks within D, so that $t_{cp}(G_i) > D$.

We present now an example to illustrate these ideas. Consider the graph G of Fig. 4(c). A schedule into $m_P = 4$ processors is shown in Fig. 7(a), which also shows the G_{tp} corresponding to this schedule. It can be seen that the G_{tp} does not have any precedences which are not also in G, and that $t_{cp}(G_{tp}) = t_{cp}(G)$. Figure 7(b) shows a schedule for $m = m_0 = 3$. Here $\omega = t_{cp}(G)$, and arc (4, 5) was added to the antichain {2, 3, 4, 5} of G. Figure 4(a) shows a schedule for $m = 2 < m_0$ processors. Here $\omega = 6 > t_{cp}(G)$. Finally, Fig. 7(c) shows the case of one processor. Here $\omega = 11$. When we only have one processor, we have to convert G into a linear ordering, of which two possible configurations are shown. There are as many possible linear orderings as "topological sortings." The problem of topological sorting consists of embedding a partial order in a linear order such that precedences are not violated [12].

4. Application to a scheduling algorithm

The study of the precedences associated with schedules gives insight into defining new optimal scheduling algorithms, in order to improve existing algorithms or to define heuristic scheduling algorithms. In this section we analyze an algorithm based on the addition of precedences [3], and we study ways of improving its performance. Scheduling of arbitrary graphs seems to be an essentially enumerative problem [9] and it is then hopeless to look for algorithms which are efficient in a general sense. However, improvements on existing algorithms can make a difference from a practical point of view.

A processor-optimal scheduling process is started by assuming some value for the number of processors, m, and then applying a scheduling algorithm. If no adequate schedule is found, the process is repeated after increasing by one the number of processors, and so on until a successful schedule is found. As the initial value for m it is convenient to use a lower bound since no optimal schedule requires a smaller number of processors than this value. A sharp lower bound is valuable, and an expression more accurate than earlier ones is given in [11]. Its efficient evaluation is discussed in [5], and some of its characteristics are discussed in Section 5.

The algorithm under consideration is a generalization of an idea of Barskiy [4]. For the case of processor-optimal scheduling, it consists of adding arcs into the active antichains of the graph that have a cardinality greater than the number of processors initially assumed. In order to obtain a graph whose basic scheduling is the optimal scheduling, the earliest load density function is used as a reference in this process. The graph is transformed so that its load density function never exceeds the estimated number of processors. Then it suffices to perform basic scheduling on the transformed graph to obtain a processor-optimal schedule. When there is no way of adding arcs at some active antichain without exceeding the deadline, it is necessary to backtrack to the previous active antichain, add a new set of arcs there and continue the process. Barskiy proved that if the cardinality of an active antichain is $r(>m)$ it suffices to add $r - m$ arcs in m chains to obtain processor-optimal schedules. This was generalized in [3] where it was proved that to obtain time-optimal schedules this result is still valid.

The process of balancing the load density function as described above corresponds to moving from G to a G' in the C_s lattice of an optimal schedule $s(G)$. It suffices to reach any graph G' between G_{mp} and G_{tp} in the corresponding lattice, to have a graph whose basic scheduling is optimal. The algorithms in [3] and [4] are practical realizations of the abstract process described in Fig. 6.

The advantages of adding only one arc at a time (instead of a set of $r - m$ arcs) have been pointed out in [3]. Proceeding in this way also allows the improvement proposed below.

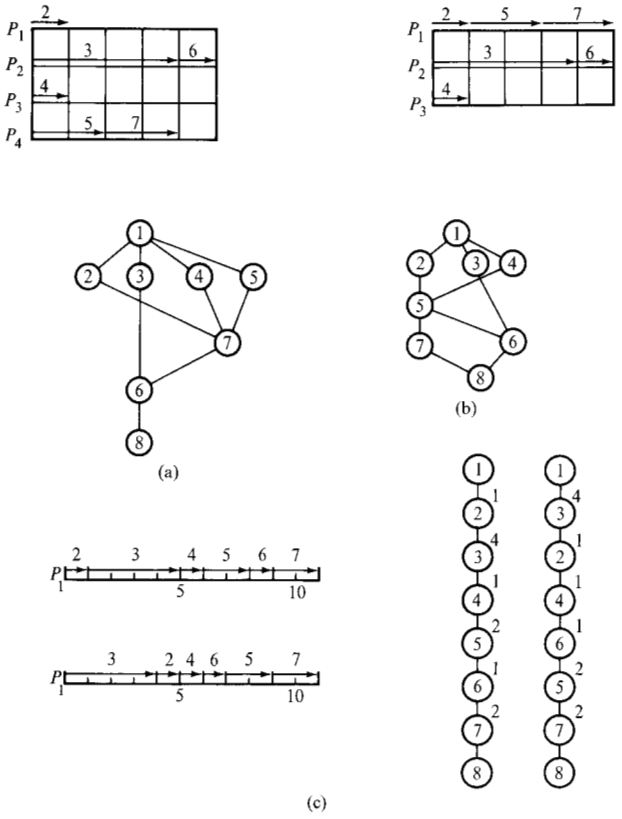


Figure 7 Relation between number of processors, precedences, and schedules.

If we look at the arcs that can be added at a given moment, i.e., at a particular antichain, we can classify them as follows. There exists a set of arcs such that the addition of any one of these to the graph will not make the total completion time exceed the deadline. This set consists of all the admissible precedences P_a. A subset of these, P_b, is the set of arcs such that addition of one of them will not change the value of the lower bound. Notice that P_b varies according to the lower bound expression used. The set of arcs which are part of optimal schedules, P_c, is a subset of P_b. Therefore, as a way of improving this scheduling algorithm we propose to recalculate the lower bound on the number of processors after addition of an arc. If the addition of (α, β) makes the previous lower bound to be exceeded, then (α, β) is not added to the graph and another arc is tried.

If the lower bound of [11] is used, and the calculation is performed incrementally [5], computational speed can be improved in this test by considering the following aspects.

1. If the active antichain where the arc is being added starts at $t = t_i$, only the intervals between t_i and D must be considered.
2. As soon as the previous lower bound is exceeded at any interval, the calculation can be stopped and the corresponding arc discarded.
3. Lower bounds can be calculated in an incremental manner, starting from the previous values and considering only the changes produced by adding a given arc.
4. If arc (α, β) fails this test, it means that β cannot be delayed as given by arc (α, β). This eliminates from consideration any other arcs that would delay β by this amount or more. Also, choosing those arcs that will produce the smallest delay on β will increase the probability of finding an arc that will pass the lower bound test.
5. Consider two consecutive active antichains of cardinality greater than m at t_i and t_j. If the addition of arcs to reduce the width of the antichain at t_i does not delay any tasks past t_j, then it can be proved that the antichain at t_i can be bypassed in the backtracking procedure. Moreover, if only a subset of the tasks in the antichain at t_i delays tasks past t_j, then for the backtracking procedure it suffices to consider only that subset of tasks. If this condition occurs, there is a reduction in the number of arcs to be considered for possible addition.

It is necessary to take into account, however, the following concerns.

- Arcs that satisfy lower bound test might still make scheduling fail. This occurs because the minimum number of processors may be higher than the lower bound and because previous arcs added may not be part of optimal schedules.
- The complexity of using these lower bound tests to select arcs should be less than the extra work performed because an arc that should have been rejected by this test, was chosen. This is difficult to evaluate, although improvements should be reasonably expected since the amount of wasted computation when the wrong arc is chosen, is high.
- The selection criterion of 4) above is in conflict with the heuristics proposed in [3], which selects those tasks with the smallest latest completion times. Both heuristics, however, could be combined as follows.

  a. Choose arc according to previous heuristics and perform lower bound test.
  b. If arc so chosen fails lower bound test eliminate from consideration for addition all arcs that produce the same or greater delay on the end task of this arc. Select another arc using criterion 4) and perform lower bound test and go to b); else go to a).

- When there is no way to add arcs in an antichain without exceeding D, backtracking to the previous antichain is necessary. This implies the need to save all the partial values for all the intervals of previous graphs. An alternative is to apply the incremental lower bound algorithm in reverse $r - m$ times.

When the lower bound of a given graph is known and an arc is added to the graph, the lower bound of the modified graph can be evaluated efficiently starting from the set of values used in the calculation of the previous lower bound. A method to perform this evaluation, together with some improvements to the general calculation of lower bounds is described in another paper [13].

H. O. Levy has studied the performance of the heuristics proposed in [3] as the basis of an approximate algorithm for time-optimal scheduling. In such an algorithm, whenever the load density function exceeds the given number of processors m, arcs are added so that this load density does not exceed m, but no backtracking is done [14]. Empirical results show this to be a promising heuristics, but more conclusive tests are needed. The same improvement suggested here for the exact algorithm, i.e., the use of the lower bound to test arcs to be added, can be applied to the heuristic method.

5. Lower bounds as graph transformations

A lower bound for the minimum number of processors and for the minimal time required for the scheduling problems considered here was presented in [5, 11] and proved to be more precise than previous bounds. The expression for the lower bound on the minimum number of processors required to perform the computations of G in time D is

$$m_L = \left\lceil \max_{[t_1, t_2]} \left[\frac{1}{t_2 - t_1} \sum_{j=1}^{n} \min(e_j(t_1, t_2), l_j(t_1, t_2)) \right] \right\rceil,$$

where $e_j(t_1, t_2)$ is the number of units of task T_j that lie in the interval $[t_1, t_2]$ if all the tasks are in their earliest possible positions, and $l_j(t_1, t_2)$ is a similar concept but with the tasks in their latest positions.

By interpreting this expression as a transformation on the graph to which it is applied, it is possible to find out why the lower bound expression does not provide the exact value of the minimum number of processors. The reasons are (Fig. 8) as follows.

1. The position of a task within a given interval does not affect deadlines of the other tasks. This implies that all tasks are considered as independent tasks with initiation and completion constraints. In other words, the calculation of the bound transforms the original graph G into a graph G_a, where the tasks of G become independent tasks with time constraints. In the

particular case where $D = t_{cp}$ (shown in Fig. 8), if the tasks of the critical path are decomposed into unit length tasks, the critical path can be used as a time marker to define the initiation and completion times of the other tasks.

2. As intervals are considered independent of each other, tasks that have a portion that has to be done in interval $[t_1, t_2]$ and a portion that can be processed out of this interval are considered as if the portion inside the interval could be processed at any time within $[t_1, t_2]$. This effect can be described by splitting tasks as shown in G_c of Fig. 8 for interval [2, 5].
3. As the bound considers the average of the activity that has to be processed in interval $[t_1, t_2]$, if r is the number of time units of task t_i that have to be processed in $[t_1, t_2]$. This part of T_i is assumed to consist of r independent unit-length tasks (G_d in Fig. 8).

In the particular case when the tasks of G are all of unit length, the graph transformation indicated above converts G into a set of unit length tasks with predetermined initiation and completion times. Only for graphs of the class of the transformed graph is it known that the lower bound is exact [15].

6. Conclusions

Characterization of the graphs that satisfy the precedences of a given schedule is of theoretical interest although it does not lead directly to better scheduling algorithms. On the other hand, analysis of the precedences (explicit or implicit), that are present in a given computational graph, is of great value to understand the process of scheduling the tasks of this graph. It gives insight into means of improving existing scheduling algorithms and formulating new heuristic methods for scheduling. Based on this study, a few ways of improving a previous scheduling algorithm have been proposed. As their effect depends largely on the structure of the graph being scheduled, a theoretical analysis of the improvement is very difficult. The enumerative nature of the scheduling process suggests this approach as promising, but an empirical evaluation would be needed to ascertain the computational effect of the proposed ideas.

Analysis of the process of determining lower bounds as a graph transformation also helps to improve calculation of these lower bounds. An algorithm for this purpose is given in another paper [13].

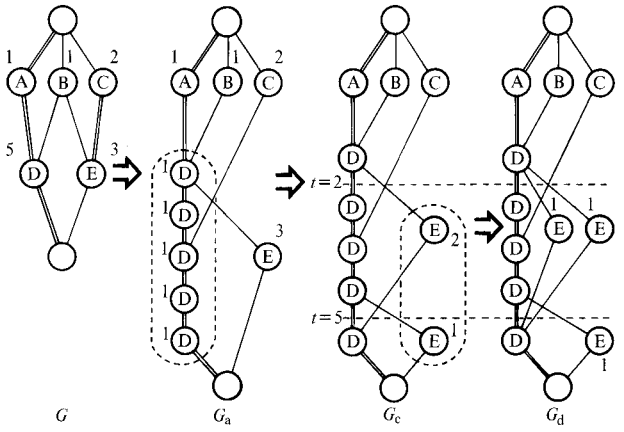


Figure 8 Interpretation of the lower bound.

References

1. V. S. Linskiy and M. D. Kornev, "Construction of Optimum Schedules for Parallel Processors," *Eng. Cybernetics* (USSR) **10,** 506 (1972).
2. C. V. Ramamoorthy, K. M. Chandy, and M. J. González, "Optimal Scheduling Strategies in a Multiprocessor System," *IEEE Trans. Comput.* **C-21,** 137 (1972).
3. B. Bussell, E. B. Fernández, and H. O. Levy, "Optimal Scheduling for Homogeneous Multiprocessors," *Information Processing 74,* 286, North Holland Publishing Co., Amsterdam, 1974.
4. A. B. Barskiy, "Minimizing the Number of Computing Devices Needed to Realize a Computational Process Within a Specified Time," *Eng. Cybernetics* (USSR) **6,** 59 (1968).
5. E. B. Fernández and T. Lang, "Computation of Lower Bounds for Multiprocessor Schedules," *IBM J. Res. Develop.* **19,** 435 (1975).
6. B. Roy, *Algèbre Moderne et Théorie des Graphes*, Vol. **I**, Dunod, Paris, 1969.
7. V. V. Martynyuk, "Transitively Equivalent Directed Graphs," *Cybernetics* (USSR) **9,** 45 (1973).
8. H. T. Hsu, "An Algorithm for Finding a Minimal Equivalent Graph of a Digraph," *J. Assoc. Comput. Mach.* **22,** 11 (1975).
9. R. L. Graham, "Bounds on Multiprocessing Anomalies and Related Packing Algorithms," *AFIPS Conf. Proc.* **40,** 205, AFIPS Press, Montvale, NJ 1972.
10. R. R. Stoll, *Sets, Logic, and Axiomatic Theories*, W. H. Freeman & Co., San Francisco, 1961.
11. E. B. Fernández and B. Bussell, "Bounds on the Number of Processors and Time for Multiprocessor Optimal Schedules," *IEEE Trans. Comput.* **C-22,** 745 (1973).
12. D. E. Knuth, *The Art of Computer Programming*, Vol. I, Addison-Wesley Publishing Co., Inc., Reading, MA 1969.
13. E. Fernández and T. Lang, "Improving the Computation of Lower Bounds for Optimal Schedules," submitted to *IBM J. Res. Develop.*
14. H. O. Levy, *Application of Graph Transformations to Scheduling*, M. S. Thesis, Univ. of California, Los Angeles, 1973.
15. T. Lang and E. Fernández, "Scheduling of Unit-Length Independent Tasks with Execution Constraints," *Information Processing Letters* **4,** 95 (1976).

Received January 20, 1976

E. B. Fernández is located at the IBM Scientific Center, Data Processing Division, 1930 Century Park West, Los Angeles, CA 90067. T. Lang is with the Computer Science Department, University of California, Los Angeles, CA 90024.