Open access • Journal Article • DOI:10.1145/356698.356700

Deterministic Processor Scheduling — Source link

Mario J. Gonzalez

Institutions: University of Texas at San Antonio

Published on: 01 Sep 1977 - ACM Computing Surveys (ACM)

Topics: Multiprocessor scheduling, Job shop scheduling, Rate-monotonic scheduling, Flow shop scheduling and Fair-share scheduling

Related papers:

- Computer and job-shop scheduling theory
- Parallel Sequencing and Assembly Line Problems
- Computers and Intractability: A Guide to the Theory of NP-Completeness
- A comparison of list schedules for parallel processing systems
- Optimization and Approximation in Deterministic Sequencing and Scheduling: a Survey

Share this paper:

View more about this paper here: https://typeset.io/papers/deterministic-processor-scheduling-1j7ecmzgb1

Deterministic Processor Scheduling*

MARIO J. GONZALEZ, JR.

Division of Mathematics, Computer Sciences and System Design, The University of Texas at San Antonio, San Antonio, Texas 78285

This paper surveys the deterministic scheduling of jobs in uniprocessor, multiprocessor, and job-shop environments. The survey begins with a brief introduction to the representation of task or job sets, followed by a discussion of classification categories. These categories include number of processors, task interruptibility, job periodicity, deadlines, and number of resources. Results are given for single-processor schedules in job-shop and multiprogramming environments, flow-shop schedules, and multiprocessor schedules. They are stated in terms of optimal constructive algorithms and suboptimal heuristics. In most cases the latter are stated in terms of performance bounds related to optimal results. Annotations for most of the references are provided in the form of a table classifying the referenced studies in terms of various parameters.

Keywords and Phrases: Deterministic scheduling, optimal schedules, multiprocessors, job-shop, flow-shop, graph structures, deadlines, resources, preemption, periodic jobs.

CR Categories: 4.32, 4.35, 6.20

INTRODUCTION

Although processor scheduling has been studied for more than ten years, most of the effort in this area has taken place during the last five years. Many of the processor scheduling techniques in use today have been adapted from older, well-established results developed in management science and operations research studies. These studies have been concerned with the utilization of people, equipment, and raw materials. If raw materials are equated with computer programs, and if people and equipment in their role of processors of these raw materials are equated with processors in computer systems, then the rationale for the adaptation of management science and operations research techniques is apparent.

In this discussion terminology will be based on computer system components, and reference to the assembly-line counterparts of these components will be made only sparingly. However, it is emphasized here that the scheduling contributions made by non-computer-oriented investigators play a very prominent part in the totality of results in the area of processor scheduling. As evidence of this, many of the results related here have been discussed by Conway, Maxwell, and Miller [11] in their book on the theory of scheduling. For the most part, this book is based on the study of job-shop scheduling problems, i.e., those problems that use the terminology of manufacturing: job, machine, operation, routing, and processing time. The recent book edited by E. G. Coffman, Jr. [42] is oriented toward computer systems and is designed to present a complete update of recent results in computer and job-shop scheduling theory. Coffman's book covers all of the subjects discussed in this survey in a much more comprehensive

* This work was performed while at the Computer Sciences Department, Northwestern University, Evanston, Illinois 60201.

Copyright © 1977, Association for Computing Machinery, Inc. General permission to republish, but not for profit, all or part of this material is granted provided that ACM's copyright notice is given and that reference is made to the publication, to its date of issue, and to the fact that reprinting privileges were granted by permission of the Association for Computing Machinery.

CONTENTS

INTRODUCTION
1 GENERAL CONCEPTS
Background
Classification of Categories
Number of Processors
Task Duration
Precedence Graph Structure
Task Interruptibility
Processor Idleness
Job Periodicity
Presence or Absence of Deadlines
Resource-Limited Schedules
Homogeneous vs Heterogeneous Processors
Measures of Performance
Efficiency of Algorithms
2 SINGLE-PROCESSOR SCHEDULES
Job-Shop Results
Multiprogramming with Hard Deadlines
3 FLOW-SHOP SCHEDULES
4 MULTIPROCESSOR SCHEDULES
Common Scheduling Environments
Schedules to Minimize Maximum Completion Time and Number of Processors
Schedules to Minimize Mean Flow Time
Special Scheduling Environments
Systems with Limited Resources
Periodic Job Schedules
Deadline-Driven Schedules
CONCLUSION
ACKNOWLEDGMENTS
CLASSIFICATION OF REFERENCES
REFERENCES

manner; in addition, many results and topics not addressed in this survey or elsewhere are fully explored.

Throughout this survey the scheduling problems to be examined are expressed in terms of *deterministic* models. By this we mean that all the information required to express the characteristics of the problem is known before a solution to the problem (i.e., a schedule) is attempted. The objective of the resultant schedules is to optimize one or more of the evaluation criteria. The motivation for this objective is that in many situations a poor schedule can lead to an unacceptable response to timing requirements or to an unacceptable utilization of resources. This discussion shows that it is often impossible or prohibitively expensive to obtain the best possible solution. In such situations heuristic solutions must be used. Many of these approximate solutions are examined in this survey.

1. GENERAL CONCEPTS

Background

Processor scheduling implies that *jobs* or *tasks* (i.e., code segments) are to be assigned to a particular processor for execution at a particular time. Because many tasks or jobs (these two terms will be used interchangeably) can be candidates for execution, it is necessary to represent the collection of jobs in a manner which conveniently represents the relationships among the jobs. A *directed graph* or *precedence graph* representation is probably the most popular representation in the scheduling literature. (For other representations see [2]). Figure 1 shows one of several possible equivalent representations for a set of jobs or tasks. The nodes in these graphs can represent independent operations or parts of a single program which are related to each other in time.

By inspecting Fig. 1, several pertinent observations can be made. First, the collection of nodes represents a set of tasks $T = \{T_1, \cdots, T_r\}$. The directed paths between nodes imply that a *partial ordering* or *precedence relation* $\lessdot$ exists between the tasks. Thus if $T_i \lessdot T_j$, task T_i must be completed before T_j can be initiated.[1] In Fig. 1, for example, $T_1 \lessdot T_2$, $T_1 \lessdot T_3$, $T_4 \lessdot T_7$, and $T_5 \lessdot T_7$. Associated with each node is a second number which refers to the time required by a hypothetical processor to execute the code represented by the node. We thus speak of a function μ: $T \rightarrow (0, \infty)$. The program graph can then be represented by the triplet $(T, \mu, \lessdot)$. If the processors are identical, then any task can be run on any processor provided that its precedence requirements are satisfied. Figure 1 contains no information regarding the number of processors available for

[1] A number of ways of indicating that "T_i precedes T_j" are given in the literature. Included among these are $T_i < T_j$, $T_i \lessdot T_j$, $T_i > T_j$, and $T_i \langle T_j$. Most authors are careful to indicate the meaning of a particular symbol. However, if T_i indeed precedes T_j, then T_i must be executed earlier than T_j. Thus the symbol "$<$" to indicate an earlier or lesser time would appear to be the most appropriate symbol, and probably the most commonly used symbol. To distinguish this symbol from the usual "less than" symbol, a dot inside the symbol may be added to minimize possible confusion.

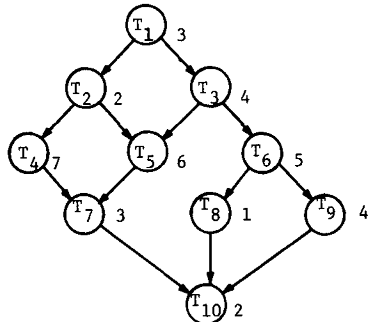


FIGURE 1 Representation of a set of tasks.

the execution of the task-set T. The number of processors, of course, directly determines the amount of time required to execute the tasks in T although, as will be discussed later, it is not necessarily true that execution time is inversely proportional to the number of processors. Among the many classification criteria, however, the number of processors represents the single most important factor in developing optimal or suboptimal schedules.

Further inspection of Fig. 1 leads to additional observations. Notice that the graph as shown is *acyclic*, i.e., there are no loops or cycles in it. A cycle in the graph would prevent the static scheduling of the graph (i.e., a scheduling performed prior to execution time) since the conditional which controls the number of iterations cannot be resolved until execution time. Most published work on processor scheduling either explicitly or implicitly ignores the difficulties presented by loops through the assumption that the entire loop can be contained within a single node in the graph.

Notice also that the graphs of Fig. 1.1 contain no *conditional* or *decision nodes*. A decision node is a node whose execution-time outcome can affect the flow of control in a program (e.g., a data-dependent branch). This assumption is common to most of the literature on this subject. If the outdegree of a node (the number of edges emanating from a node) is n, $n > 1$, then the n nodes which are immediate successors of the node cannot be initiated until the computations represented by the node are completed. Similarly, a node with an indegree (the number of edges incident to the node) greater than one must wait for the completion of all its immediate predecessors before it can be initiated. The scheduling techniques addressed in this paper will be based on the two conditions cited above: the absence of loops and the absence of decision nodes. (A great deal of effort has been invested in the modeling of computational sequences which do not rely on these assumptions; for a detailed discussion of that subject refer to [2].)

A graph of the form shown in Fig. 1 is referred to as a single-entry-node single-exit-node connected graph, or SEC. In many references, however, the graph under investigation is of the form shown in Fig. 2.

Classification Categories

The discussion which follows is based on whether a program graph is to be processed by one processor or a system containing more than one processor. The decision to categorize schedules in this manner is not obvious, in view of the large number of factors that can be used for classification. The following discussion will identify these factors and show how the present system of classification evolved.

Number of Processors

Traditionally, single-processor systems have overwhelmingly dominated computer system installations. However, the search for higher computational bandwidths through the use of several processors has been in progress since before the so-called single-instruction-stream single-data-stream (SISD) organization reached maturity. The realization of large-scale integration and a desire for more reliable computation have given further impetus

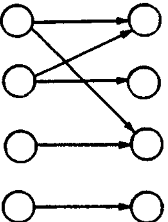


FIGURE 2. A task set with multiple initial tasks.

to the utilization of multiprocessor organizations. As this discussion will show, however, nonenumerative optimal schedules[2] have been generated for only a limited number of cases.

Task Duration

In the previous section, we indicated that the nodes of a program graph can represent tasks of equal or unequal duration. In the case of equal task duration, all the tasks can be said to have a duration or execution time requirement of one unit. (The term "unit" is used here to represent the time required to execute a given number of instructions.)

In the treatment of tasks of unequal duration, a common practice is to assume that all tasks can be subdivided into integer multiples of the smallest of the original tasks. In this discussion, however, we consider as a separate category program graphs for which the subdivision of tasks is not allowed.

Precedence Graph Structure

The individual nodes within a graph can be related to each other in a number of different ways. For example, it is possible for all tasks to be independent of each other. In this situation we say that there is no precedence or partial ordering between tasks. In other situations it is necessary to structure the graph of a program in such a way that every node in the graph has at most one predecessor or at most one successor. Another possibility allows the existence of a general precedence structure for which the previous restrictions do not apply. Each of these conditions is examined in the following discussion.

Task Interruptibility

If the interruption (and subsequent resumption) of a task before its completion is permitted, we speak of a *preemptive* schedule. If interruption before task completion is not permitted, we speak of a *nonpreemptive* or *basic* schedule. In general, preemptive disciplines generate schedules that are better than those generated by nonpreemptive disciplines. It is also true, however, that a certain penalty exists for preemptive schedules that does not exist in the nonpreemptive case. This penalty lies in the task-switching overhead, which consists of system interrupt processing and the additional memory required to preserve the state of the interrupted task. This overhead may be acceptable if it occurs infrequently; in an environment in which preemption occurs frequently, however, unacceptable performance degradation may result.

Processor Idleness

As subsequent discussion will show, a given measure of performance can often be improved by deliberately idling a processor. Determining when this should be done, however, can lead to significant increases in the complexity of a scheduling algorithm. In a "greedy" processor environment, no idle time is inserted into a schedule, and a pending task is begun as soon as a processor is available.

Job Periodicity

The overwhelming majority of investigations reported in the literature and examined here deal with only a single execution of a set of jobs or tasks that is expected to be repeated at irregular intervals over a long period of time. The analysis required to generate optimal or near-optimal schedules can be significant, but it is justified by the time saved during each of these many executions. During this time the code that represents these tasks is unmodified, although modifications to the data processed by the code are permitted and perhaps represent the rule rather than the exception. However, the measure used to evaluate the performance of the set is considered only for a single execution of each element of the set.

Recently, however, consideration has been given to the use of one or more processors in a *control environment*. An envi-

[2] An enumerative schedule is one in which all possible solutions are obtained and the best one is selected.

ronment of this type can be characterized by a set of tasks each of which has a known execution frequency and processing time. The scheduling problem in this environment is especially difficult for two reasons: time and frequency requirements can be different for each task in the set of periodic tasks, and in some cases little or no deviation is permitted in the scheduled initiation time (and consequently the completion time) of each iteration of each task.

Presence or Absence of Deadlines

A number of performance measures have been developed to evaluate the behavior of schedules. In most cases, only the behavior of the entire schedule or the entire job-set is considered. In other cases, however, *deadlines* or *scheduled completion times* are established for individual members of the task-set. If there is some slack or spare time associated with the completion of time of individual tasks and this slack time is bounded, we speak of a *hard deadline* or a *hard real-time schedule*. If the slack time is based on a statistical distribution of terminations we speak of a *soft deadline* or a *soft real-time schedule*. Schedules with deadlines appear most frequently in connection with periodic job schedules.

Resource-Limited Schedules

Most of the effort to date on processor schedules has assumed the unlimited availability of whatever additional resources are necessary to support multiple processors in execution. Although it is not usually mentioned, the processors themselves have been implicitly assumed to be members of a (not usually limited) resource class.

Recently, however, consideration has been given to the generation of schedules in which individual tasks explicitly indicate requirements for elements of one or more resource classes. Aside from the processor itself, the resource class which most readily comes to mind is memory. Most references assume that sufficient memory is available to contain the code and the data required by each of the tasks assigned to a particular processor. In systems in which a single main memory is shared, this implies that the total memory requirement of the set of tasks does not exceed the size of the main memory. With the emergence of distributed systems in which a processor can access both a local memory and a shared memory, resource considerations become particularly significant. Of course, memory is not the only system resource which can be available in limited amounts. The theory resulting from resource-limited models can be expanded to include a multiplicity of resources.

Homogeneous versus Heterogeneous Processors

Along with investigations in resource-limited schedules, consideration of nonidentical or *heterogeneous multiprocessor systems* represents the latest effort in what is now considered a mature field of investigation. Considerations of processor nonhomogeneity will become particularly significant as multiple-processor systems assume a bigger share of the total data-processing load. The ability to have different processors in a set of processors implies that system upgrades can be accomplished using state-of-the-art components, i.e., that a processor (failed or otherwise) can be replaced, or a processor can be added to the system, without having to restrict the replacement or addition to a replica of the original equipment. This is particularly significant if, as is usually the case, cheaper, smaller, and more capable replacements for the original equipment are available.

Measures of Performance

As suggested by the preceding discussion, a number of measures have been developed to evaluate the effectiveness of processor schedules. The five measures most often cited in the literature are listed below in approximate order of popularity; this survey will concentrate on the first three (a number of other measures are discussed in [11, Chapter 2]):

1) minimize finishing or completion time;

2) minimize the number of processors required;
3) minimize mean flow time;
4) maximize processor utilization;
5) minimize processor idle time.

In Fig. 3 we display schedules with timing diagrams known as *Gantt charts*. In this schedule three processors are required. The tasks assigned to each processor and their order of execution and execution time requirements are represented by the horizontal lines and task identifications adjacent to each processor. The *completion* or *finishing time* (denoted by ω in this survey) for the schedule illustrated is 7. The *flow time of a task* is equal to the time at which its execution is completed. The *flow time of a schedule* is defined as the sum of the flow times of all tasks in the schedule. For example, the flow times of tasks T_1 and T_4 in Fig. 3 are 7 and 4, respectively, while the flow time of the schedule is 25.5. The *mean flow time* is obtained by dividing the flow time by the number of tasks in the schedule. The *utilization* (or utilization factor) of P_1, P_2, and P_3 is 0.93, 1.00, and 0.86, respectively. These utilization values are obtained by dividing the time during which the processor was actively engaged in execution by the total time during which it was available for execution. The *idle time* of P_1, P_2, and P_3 is 0.5, 0.0, and 1.0, respectively.

The rationale behind the minimization of finishing or completion time is that system throughput can be maximized if the total computation time of each set of tasks is minimized. *Throughput* is defined as the number of task sets processed per unit of time (e.g., per hour) and is therefore inversely proportional to the sum of the computation times of individual task-sets.

Minimizing the number of processors required can be justified for at least two reasons. The first and most obvious is cost. The second and not so obvious reason is this: if the number of processors required to process a set of tasks in a given time is less than the total number of processors available, then the remaining processors can be used as backup processors for increased reliability and as background processors for noncritical computations.

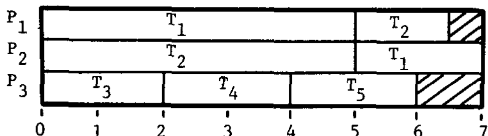


FIGURE 3. Task schedule in Gantt chart form.

Minimizing the mean flow time is related to the extent to which tasks occupy system resources other than processors, memory in particular. The shorter the time during which a set of tasks occupies memory, the greater the amount of time that is available for other tasks to occupy that same memory. (The analogy in a job-shop environment is the amount of warehouse space occupied by raw materials that are to be converted into finished products, i.e., the inventory [3].) Thus flow time is an indirect measure of system throughput.

Efficiency of Algorithms

A key issue in the study of processor scheduling is the amount of computation time needed to locate a suitable schedule. For our purposes we shall say that an *efficient algorithm* is one which requires an amount of time that is bounded in the length of its input by some polynomial. An *inefficient algorithm* is one which essentially requires an enumeration of all possible solutions before the best solution can be selected. Solutions of this type can be characterized by algorithms whose running times are exponential in the number of jobs to be scheduled. For most of the problems of interest in processor scheduling, no efficient algorithm is known; in fact, it is known that if an efficient algorithm for these problems could be constructed, then an efficient algorithm could be constructed for a large family of seemingly intractable problems [53, 54]. That is, it is known that these problems are *NP-hard*.

By saying that a problem is NP-hard we mean that it is at least as difficult to compute as the hardest problem in the family NP, which is the family of problems capable of being solved by *nondeterministic* algorithms in polynomial time. It includes

such problems as whether or not a propositional formula is satisfiable, whether or not a graph possesses a clique of a given size, and a version of the well-known traveling salesman problem. After years of effort, research has failed to find a (deterministic) algorithm that solves any one of these problems in polynomial time.

2. SINGLE-PROCESSOR SCHEDULES

In the single-processor schedules considered here, all candidate tasks are simultaneously available for execution, the exact characteristics of each of the tasks are known and remain constant throughout the lifetime of the task, and a particular performance measure is specified, e.g., minimization of the maximum completion time. Thus schedules considered in this section do not include the type of problem addressed by multiprogrammed or time-shared computer systems, since the exact characteristics of the tasks processed by these systems are not known in advance. The results related here have their origins in one of two environments: a job-shop or a process-control environment. Tasks in both environments are usually considered to be independent; the results in the process-control environment are for periodic jobs only.

Job-Shop Results

An environment consisting of n simultaneously available jobs or tasks of known characteristics and one machine (i.e., one processor)[3] is considered by Conway, Maxwell, and Miller [11] as the simplest scheduling problem. In their text the authors relate several important results for schedules of this type.

1) In scheduling n independent tasks on a single processor, it is not necessary to consider schedules which involve either preemption or inserted idle time. Thus a *regular measure of performance* cannot be improved by preempting (and subsequently resuming) an active task or by idling the processor at any time prior to the completion of the n tasks. (A regular measure is a value to be minimized that can be expressed as a function of the task completion times and that increases only if at least one of the completion times increases.)

2) The maximum flow time for this type of schedule is simply the sum of the n completion times and is the same for each of the $n!$ possible sequences.

3) The mean flow time of a schedule of this type is minimized by sequencing the jobs in order of nondecreasing processing time. Scheduling in this manner is referred to as *shortest-processing-time sequencing* (SPT), and the authors refer to this type of sequencing as the most important concept in the entire subject of sequencing.

To illustrate the concept, consider the scheduling on a single processor of the six independent tasks shown in Fig. 4. This figure also shows a Gantt chart representation of one of the 6! possible schedules for this set of tasks. It will be seen that the maximum flow time equals 25 units, the

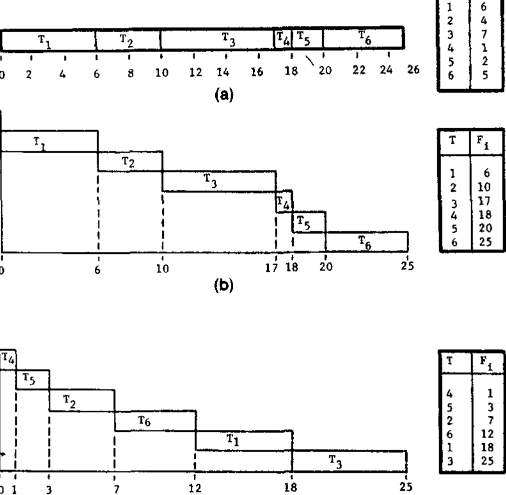


FIGURE 4. Determination of flow time. (a) Gantt chart representation; (b) Alternate representation; (c) Shortest-processing-time (STP) schedule. Tabulations to the right of the figures give the individual tasks' utilization factors for the Gantt chart representation and flow times for the other two illustrations.

[3] In job-shop literature the term "machine" is usually used for a piece of equipment that performs a particular operation. In keeping with the theme of this survey, the term "processor" will be used here.

same as the maximum completion time. In order to determine the mean flow time ($\bar{F}$), it is more convenient to represent the schedule in the manner shown in Fig. 4b. The total area of this graph, including the labeled blocks and the area under the blocks, represents the sum of the task flow times. F_i, the flow time of the ith task in the sequence, is defined as

$$F_i = \sum_{j=1}^{i} p_{[j]},$$

where $p_{[j]}$ represents the processing time of the task occupying the jth position in the schedule. (In words, the flow time of a task is simply its finishing or completion time.) The *mean flow time* is defined as

$$\bar{F} = \left(\sum_{j=1}^{n} F_j\right) \Big/ n.$$

For the example of Fig. 4b, $\bar{F} = 16$. If the tasks of this illustration are rearranged to form an SPT schedule as shown in Fig. 4c, then $\bar{F} = 11$.

A related observation is that *longest-processing-time sequencing* (LPT) maximizes whatever SPT minimizes. Scheduling procedures which produce opposite sequences in an n-task, single-processor problem are called *antithetical.*

In related discussions, Maxwell, Conway, and Miller also provide results for situations in which 1) only average, expected, or estimated processing times are given; 2) priorities are assigned to individual tasks; and 3) all tasks are not simultaneously available but instead arrive intermittently.

Multiprogramming with Hard Deadlines

The subject matter of this section is the so-called time-critical process in a process-control environment. A *time-critical process* is a periodic task of known frequency and execution time. Each activation of a task must be completed within the interval defined by the frequency, and activation of a task can be considered to be signalled by the presence of an external interrupt. A collection of these processes can represent the totality of computations that must be performed in order to satisfy the control demands of a particular real-time environment. (An example of this type of environment is the monitoring and setting of temperatures, pressures, and fuel consumption rates in a refinery.) From the point of view of the processing load, the big difference between this and a conventional multiprogramming system is that the exact nature of the total set of required computations is known beforehand. In addition, response within a set of fixed limits must be guaranteed; it is not permissible to say, as one can with the typical system, that "most responses will occur within x seconds."

Two independent research efforts in this area (Serlin [36] and Liu and Layland [25]) have yielded very similar results. The starting point of the discussion is the time-critical process (TCP) illustrated in Fig. 5 [36]. In this model, E represents the required execution time of one iteration of the process. The time τ, sometimes called the frame time, is the repetition period, i.e., the period between successive occurrences of the interrupt signal associated with the process, and d is the deadline of the computation. In practice, d usually equals τ. An *overflow* is said to occur when the arrival of an interrupt signals the initiation of an iteration of a TCP before the previous iteration of that same TCP has terminated. The problem addressed here is that of scheduling a number of TCPs on a single processor in a manner which guarantees that no overflow will occur. A schedule is said to be *feasible* if the tasks are arranged so that an overflow never occurs [25].

In this model it is assumed that all tasks are independent, and the tasks perform no I/O. Since some tasks will have a higher execution frequency than others, it will sometimes be necessary to interrupt and

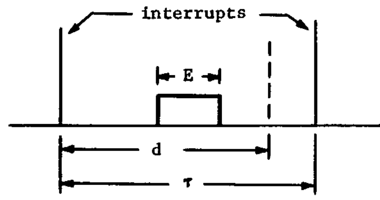


FIGURE 5 Model of a time-critical process (TCP)

subsequently resume the execution of a lower-frequency task (i.e., the lower-frequency task is preemptible) in order to guarantee the deadline of a higher-frequency task. For example, Fig. 6 shows two tasks $T_1(\tau_1 = 2, E_1 = 1)$ and $T_2(\tau_2 = 0.5, E_2 = 1)$, with T_1 having the higher priority. Figure 6a shows a feasible assignment, and Fig. 6b shows that E_2 can be increased to at most 2. If T_2 is given the higher priority, then neither E_1 nor E_2 can be increased beyond 1, as shown in Fig. 6c.

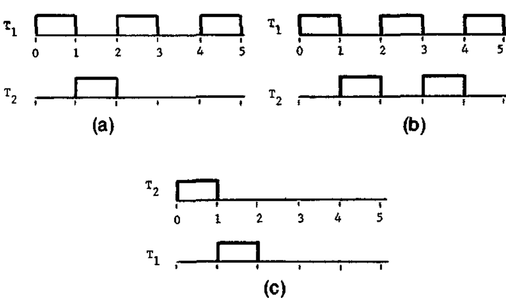


FIGURE 6. Schedules for two time-critical processes. (a) A feasible assignment when T_1 has higher priority than T_2; (b) Another feasible assignment when T_1 has higher priority than T_2; (c) Only feasible assignment if T_2 has higher priority than T_1.

According to Serlin an efficient CPU allocation algorithm is one that awards sufficient processor time to a TCP for it to meet its deadline while minimizing forced idle time. (*Forced idle time* is time during which the CPU must be idle in order to accommodate the occasional worst-case condition.) Liu and Layland seek to find the largest possible *utilization factor* while guaranteeing that all tasks meet their deadline. The utilization factor U (referred to as the "load factor" by Serlin) for a set of n TCPs is defined as

$$U = \sum_{i=1}^{n} E_i/\tau_i.$$

Serlin [36] and Liu and Layland [25] obtained the same optimal result for a fixed-priority scheme in which a task of frequency f_i has a higher priority than a task of frequency f_j if $f_i > f_j$. Liu and Layland refer to a fixed-priority scheme of this type as a *rate monotonic priority* (*RMP*) *assignment*, while Serlin calls it the *intelligent fixed priority* (*IFP*) *algorithm*. Both sets of authors have shown that for this scheme the least upper bound to the utilization factor is $U = n(2^{1/n} - 1)$, where n is the number of TCPs. This result means that the permissible sum of the individual load factors must be considerably less than 1 in order to guarantee that each TCP meets its deadline. For large task-sets, more than 30% of the CPU must remain idle. This scheme is optimum in the sense that no other fixed-priority assignment rule can schedule a task-set that cannot be scheduled by the IFP or RMP algorithms.

The assignment rule discussed above is a fixed or *static rule* in that the relative priority of the tasks is based on the task frequencies and does not change during execution. Both of the studies discussed here have developed *dynamic algorithms* in which priorities are permitted to change and which permit 100% processor utilization. Liu and Layland's rule is called the *deadline-driven scheduling algorithm*. Serlin discusses a similar algorithm developed by M. S. Fineberg [13]. In both of these algorithms, priorities are reevaluated every time that a task-initiating interrupt arrives at the system. Highest priority is given to the task whose deadline is the nearest, and lowest priority is given to the task whose deadline is the farthest away from the current time. This applies only to those tasks whose computation for the current frame has not yet been completed. Coffman [42] refers to a *relative urgency* algorithm in which priorities are reevaluated at each instant of time.

Serlin also speaks of the *minimal time slicing* (*MTS*) *algorithm* based on what he calls scheduling intervals. A *scheduling interval* is the time between the occurrence of an interrupt and the occurrence of the first deadline beyond the interrupt. During this interval each task with an incomplete computation is given a CPU burst whose duration is directly proportional to the percentage of the overall utilization factor contributed by the load factor of the task. This approach guarantees that all tasks will meet their deadlines, but its success depends on a small context-switching time.

Liu and Layland discuss a *mixed sched-*

uling algorithm which is a combination of the fixed- and dynamic-priority algorithms. For a set of n tasks, the k tasks having the shortest periods are scheduled according to the fixed-priority-rate monotonic scheduling algorithm, and the remaining tasks are scheduled according to the deadline-driven scheduling algorithm when the processor is not occupied by the first k tasks. According to the authors this algorithm does not always achieve 100% utilization but appears to provide most of the benefits of the deadline-driven scheduling algorithm. At the same time it may be more readily implementable, since the static scheduling of the k tasks is compatible with interrupt hardware that acts as a fixed-priority scheduler.

3. FLOW-SHOP SCHEDULES

After considering single-processor schedules, it would seem natural to consider multiprocessor schedules. However, there is a class of schedules—the so-called flow-shop schedules—in which more than one processor is involved in the cooperative execution of a set of tasks, and in which a sequential relationship exists between the processors. (This is not the case with multiprocessor schedules.) Thus, a task to be executed must be processed first by one of the processors and then by the other(s). This ordering must be observed for all the tasks to be executed, and there is no requirement that the processors be identical.

The origin of this kind of schedule is once again the job-shop environment, in which a job must be sequenced through a set of machines that perform unique operations. The analogous situation in a computer environment is a task requiring a series of CPU and I/O bursts. The ordering of these bursts corresponds to a sequencing through a set of machines in which the number of different machines is small.

As indicated by Conway, Maxwell, and Miller, probably the most frequently cited paper in the field of scheduling is Johnson's solution to the two-machine flow-shop problem [21]. Johnson's algorithm sequences n jobs, all simultaneously available, in a two-machine flow-shop so as to minimize the maximum flow time. Using the Conway, Maxwell, and Miller terminology and notation adopted from Johnson, we say that each task consists of a pair (A_i, B_i), where A_i is the work to be performed on the first machine of the shop and B_i is the work to be performed on the second machine. The tasks will be executed on the two machines in this order, although it is permissible for some of the A_i and B_i to be zero since some of the tasks will have only one operation performed upon them. It is assumed that each machine can work on only one task at a time, and that operation A_i must be completed before operation B_i can be initiated. Given n pairs of the form (A_i, B_i), the problem is to order the n jobs so that the maximum flow time (i.e., schedule or completion time) is minimized. Johnson showed that job J_j should precede job J_{j+1} if

$$\min(A_j, B_{j+1}) < \min(A_{j+1}, B_j).$$

Consider an example that was presented in [11, p. 89]. Table I defines the characteristics of the tasks to be scheduled. Table II shows that $T_2 \lessdot T_3$ and $T_4 \lessdot T_5$*,[4] since $\min(A_2, B_3) = \min(0, 4) = 0 < \min(A_3, B_2) = \min(5, 2) = 2$, and $\min(A_4, B_5) = \min(8, 1) = 1 < \min(A_5, B_4) = \min(2, 6) = 2$.

TABLE I. CHARACTERISTICS OF A SET OF TASKS TO BE SCHEDULED

Task Number, i	A_i	B_i
1	6	3
2	0	2
3	5	4
4	8	6
5	2	1

TABLE II. MINIMUM FLOW TIMES FOR DIFFERENT PAIRINGS IN THE SET OF TASKS OF TABLE I

j	(A_j, B_{j+1})	*Min.*	(A_{j+1}, B_j)	*Min.*
1	$(A_1, B_2) = (6, 2)$	2	$(A_2, B_1) = (0, 3)$	0
2	$(A_2, B_3) = (0, 4)$	0	$(A_3, B_2) = (5, 2)$	2
3	$(A_3, B_4) = (5, 6)$	5	$(A_4, B_3) = (8, 4)$	4
4	$(A_4, B_5) = (8, 1)$	1	$(A_5, B_4) = (2, 6)$	2

[4] Notice that use of the symbol $\lessdot$ here means that T_i should precede T_j in the optimal sequence. It does not mean that T_i must precede T_j in all orderings, since it is given that the tasks are independent and simultaneously available.

Further examination of Table II shows that $T_2 < T_4$ and $T_2 < T_5$. Furthermore, $T_4 < T_3$ and $T_3 < T_5$. Summarizing these results, we find that the ordering should be T_2, T_4, T_3, T_5. The only task not yet scheduled is T_1. From Table II and Johnson's result, $T_1 \not< T_2$, $T_1 < T_3$, $T_1 \not< T_4$, and $T_1 < T_5$. Thus the only position for T_1 that satisfies these procedence relations and the ordering of T_2, T_4, T_3, T_5 is one in which $T_3 < T_1$ and $T_1 < T_5$, and the final ordering is T_2, T_4, T_3, T_1, T_5. The schedule that results from this ordering is shown in Figure 7a; it yields a minimum flow time of 23. An SPT schedule on the basis of the A_i is shown in Fig. 7b, and a schedule based on the order in which the tasks appears in the initial table is given in Fig. 7c.

Conway, Maxwell, and Miller indicate that, aside from mathematical programming solutions, no efficient algorithm similar to Johnson's exists for the minimization of the mean flow time for the two-machine flow-shop problem (i.e., the problem is NP-hard). Branch-and-bound techniques have been used to minimize $\bar{F}$, and the results of these efforts have shown that the amount of computation doubles every time a job is added to a set of jobs to be sequenced. The authors note that this 2^n rate of increase is still better than the $n!$ rate of computation that would be required for exhaustive enumeration. Mathematical programming techniques have been used with varying degrees of success in an

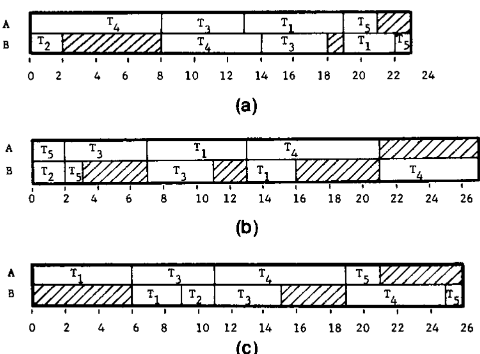


FIGURE 7 Illustration of Johnson's algorithm for the tasks of Table I. (a) Optimal sequence: $F_{max} = 23$; (b) SPT sequencing on A_i: $F_{max} = 27$; (c) Sequencing in the order (T_1, T_2, T_3, T_4, T_5): $F_{max} = 26$.

attempt to minimize the maximum flow time for the three-machine flow-shop.

The results of the preceding paragraphs have been extended for the situation in which more than one processor exists in each of two classes, Class A and Class B. In the "more-and-earlier" (ME) scheduling strategy, V. Y. Shen and Y. E. Chen [37] consider a system with m processors in Class A and n processors in Class B, with the objective of minimizing the maximum completion time. The authors show that although the ME strategy is not optimal, it is simple and works quite well. In developing the ME strategy, a partial ordering is defined such that task T_i precedes task T_j if

$$A_i + B_i \geq A_j + B_j \quad \text{and} \quad A_i \leq A_j$$

where A_i and B_i, respectively, represent the requirements of T_i for a processor in Class A and a processor in Class B.

In a subsequent related work, Buten and Shen [5] drop the restriction that task T_i must precede task T_j if $A_i + B_i \geq A_j + B_j$ and $A_i \leq A_j$. Instead they consider what they call a modified Johnson ordering (MJO). (The Johnson ordering (JO) is based on the results of Johnson's algorithm cited earlier.) In a flow-shop environment with m processors of type A and n processors of type B, $T_i < T_j$ according to MJO, if

$$\min (A_i/m, B_j/n) < \min (A_j/m, B_i/n).$$

The authors develop two theorems which describe upper and lower bounds for the behavior of the MJO approach.

In flow-shop problems, it is assumed that when a job must wait for a machine because the machine is busy, sufficient storage is available to contain the partially completed products. In a computer system environment this may not be a valid assumption, since the intermediate storage may consist of buffers as a task progresses from main memory to the CPU to an I/O unit. Reddi and Ramamoorthy [33, 43, 44] have investigated flow-shop schedules which do not assume that the available intermediate storage is infinite. Such an environment is considered to have a finite amount of intermediate storage

(FSFIS), in contrast to an environment with infinite intermediate storage (FSIIS). As a first step toward the solution of the FSFIS problem, Reddi and Ramamoorthy generated a solution for an environment with no intermediate storage (FSNIS).

The effect of intermediate storage can be seen by examining the job-set of Fig. 8a. In an FSIIS environment, total completion time is minimized by scheduling the jobs in the order (J_1, J_3, J_2), as shown in Fig. 8b. Suppose, however, that no intermediate storage is available. Then the ordering (J_1, J_3, J_2) yields a schedule requiring 52 units, as shown in Fig. 8c. The optimal FSNIS schedule is for the ordering (J_1, J_2, J_3) and requires 45 units, as illustrated in Fig. 8d.

The problem environment defined here in effect relaxes several of the constraints utilized in the development of Johnson's algorithm. First, more than two machines are allowed, and second, the amount of intermediate storage available is assumed to be zero. As indicated earlier, Conway, Maxwell, and Miller have indicated that no efficient algorithms exist for the solution of the flow-shop problem in environments which relax the constraints assumed by Johnson. The FSNIS problem investigated by Reddi and Ramamoorthy is no exception. However, Gilmore and Conroy have developed a polynomial-time solution for this "no-wait" environment when the number of machines is limited to two [45].

In a related investigation, Reddi and Feustel [34] consider additional problems in a flow-shop environment. Basically, they consider the overhead required to generate an optimal schedule in a computer system environment in which the two machines are the CPU and I/O processor. They conclude that, since the computational overhead is nonzero, it is best in some circumstances to optimally schedule a subset of the total set of tasks and randomly schedule the remaining tasks.

It should be pointed out here that some authors refer to "multiprocessor" schedules when considering flow-shop schedules because more than one processor is involved. In this paper the requirement that a task must flow through one machine or processor and then through the other is used to distinguish flow-shop schedules from the multiprocessor schedules of the following sections. Most of the major results contained in Section 4 have been rigorously examined from a more formal, mathematical point of view by Coffman and Denning [9] and by Coffman [42, 49].

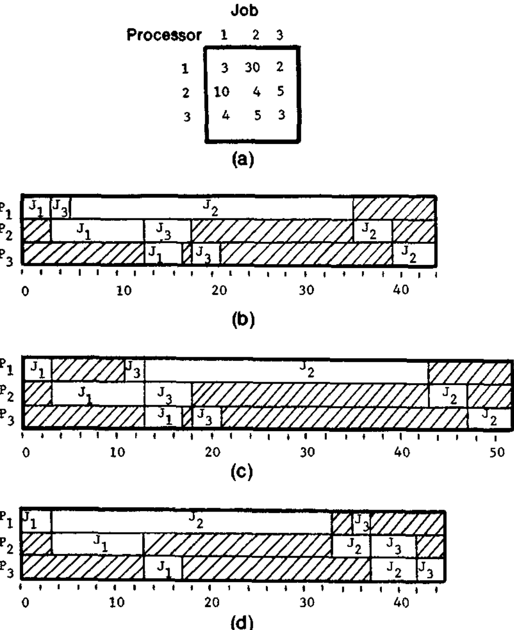


FIGURE 8. Comparison of schedules for environments with infinite intermediate storage (FSIIS) and environments with no intermediate storage (FSNIS). (a) Processing times of jobs to be scheduled; (b) Optimal schedule for FSIIS and the order (J_1, J_3, J_2); (c) Optimal schedule for FSNIS and the order (J_1, J_3, J_2); (d) Optimal schedule for FSNIS and the order (J_1, J_2, J_3).

4. MULTIPROCESSOR SCHEDULES

In this section we examine schedules in which more than one processor can be used to optimize measures of performance. This section is divided into two major parts. In the first part—Common Scheduling Environments—the parameters identified in most of the scheduling literature and discussed earlier prevail unless stated otherwise. That is, we assume a number of identical processors, a set of tasks with equal or unequal execution times, and a (possi-

bly empty) precedence order. Both preemptive and nonpreemptive disciplines are examined. In the second part – Special Scheduling Environments – additional constraints are introduced. These constraints include systems with a finite number of resources in each member of a set of resource classes, periodic jobs with specified initiation and completion times, and the presence of intermediate deadlines within a schedule.

Common Scheduling Environments

This portion of the survey is divided into sections according to the measure of performance that is to be optimized. The first part of the discussion takes up the minimization of the maximum completion time and the minimization of the number of processors; the second part of the discussion has as its objective the minimization of the mean flow time.

Schedules to Minimize Maximum Completion Time and Number of Processors

Schedules considered here are discussed separately according to whether or not preemption is allowed.

Preemptive schedules——The most significant contributions in the area of preemptive schedules (PS) have been made by Muntz and Coffman [29, 30]. We first consider only their optimal result for the case where any graph with mutually commensurable node weights is executed by exactly two processors [29]. (A set of nodes is said to be *mutually commensurable* if there exists a w such that each node weight is an integer multiple of w.) In subsequent discussion we consider their optimal results for rooted trees with mutually commensurable node weights and any number of processors.

Preemptive schedules can be contrasted with nonpreemptive or basic schedules (BS). In the latter type of schedule, a task that is awarded a processor retains that processor until the task is complete. In a preemptive schedule, a processor may be preempted from an executing task if such an action results in an improved measure of performance.

In obtaining their results, Muntz and Coffman rely on a result generated by McNaughton [28] which places a lower bound on the optimal PS for a set of n independent tasks with weights (task duration or execution times) of $\{w_1, w_2, \cdots, w_n\}$, and k processors. This optimal length is given as

$$\max\left\{\max_{1\le i\le n}\{w_i\},\ \left(\sum_{i=1}^{n} w_i\right)\Big/k\right\}.$$

In words: the optimal PS length cannot be less than the larger of the longest task or the sum of the execution times divided by the number of processors.

For their optimal algorithm, Muntz and Coffman partition the set of nodes in a graph having nodes of unit weight into a sequence of disjoint subsets such that all nodes in a subset are independent. All nodes in the same subset or at the same *level* are candidates for simultaneous execution. In a graph of N levels, the terminal node occupies the first level exclusively. Those nodes which may be executed during the unit time period preceding the execution of the terminal node occupy the second level, etc., up until the initial or entrance node in the graph which occupies the Nth level. (The assignment of levels in this manner corresponds to the methods of precedence partitions discussed by Ramamoorthy and Gonzalez in [31]. In particular, the assignment procedure outlined above corresponds to the latest precedence partitions. That is, the assignment of nodes to levels is done in a manner which defers task initiation to the latest possible time without increasing the minimum completion time, assuming that the number of processors available is greater than or equal to the maximum number of tasks at any level.) For an arbitrary graph G, a precedence relation will exist between the subsets due to the precedence which exists between nodes in the original graph. A PS schedule can be constructed for G by first scheduling the highest-numbered subset, then the subset at the next-lower level, etc. (When a subset consists of only one node, a node from the next-lower subset is moved

up if it does not violate precedence constraints.) If each of the subsets is scheduled optimally, a *subset schedule* results. Muntz and Coffman show that, for two processors and equally weighted nodes, an optimal subset schedule for G is an optimal PS schedule for G.

This result is extended to the case of graphs having mutually commensurable node weights. In order to generate the optimal result it is necessary to convert graph G into another graph G_w in which all nodes have equal weights. This is done by taking a node of weight w_i and creating a sequence of n nodes such that $w_i = nw$, as illustrated in Fig. 9 [29]. Note that the integrity of the original graph must be maintained, in that edges into or out of a node in G must enter or leave an entrance or exit node in the sequence representing the original node. The authors then show that an optimal subset schedule for G_w is an optimal PS for G with $k = 2$.

In this approach, one must note whether the number of tasks at any level is even or odd. If it is even, then all tasks at that level can be executed in the minimum amount of time with no idle time for either of the two processors. If the number of tasks is not a multiple of two, then the last three tasks to be scheduled at that level (or all the tasks to be scheduled if there were only three originally) can be executed in no less than $1\frac{1}{2}$ units, since all tasks in G_w

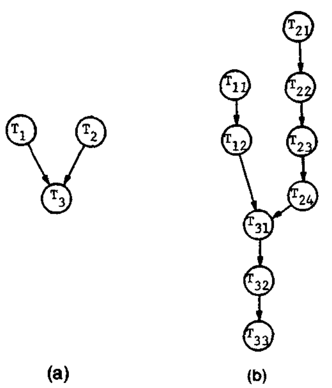


FIGURE 9. Comparison of a graph with mutually commensurable node weights with the corresponding graph having nodes of equal weight. (a) Graph G node weights $w_1 = 7$, $w_2 = 14$, $w_3 = 10^1/_2$; (b) Graph G_w, $w = 3^1/_2$.

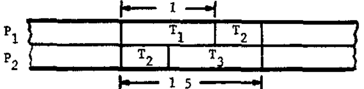


FIGURE 10. Minimum-time execution format for three unit tasks with two processors.

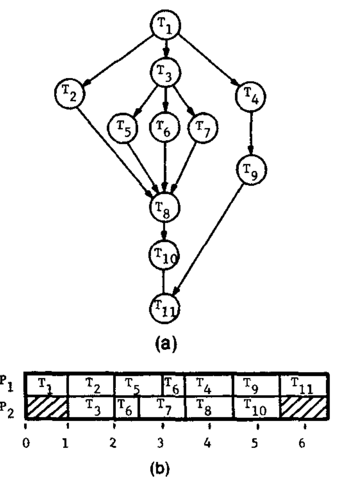


FIGURE 11. Illustration of subset sequence algorithm. (a) Graph G for a set of tasks, with all nodes having unit weight; (b) Optimal preemptive schedule

are of unit duration. By using the form shown in Fig. 10, three tasks in a given level can be executed in minimum time without processor idle time. Since scheduling in this manner ensures that no processor is idle, the subset sequence can be seen to generate a minimal-length PS. An example of the optimal PS algorithm is shown in Fig. 11 [29]. For this example, the optimal subset sequence for G is $\{T_1\}$, $\{T_2, T_3\}$, $\{T_5, T_6, T_7\}$, $\{T_4, T_8\}$, $\{T_9, T_{10}\}$, $\{T_{11}\}$.

Muntz and Coffman have extended their optimal results to the case in which any number of processors are allowed when the computation graph is a rooted tree (i.e., a tree in which each node has at most one successor, with the exception of the root or terminal node which has no successors), and the node weights w_i are mutually commensurable [30]. On the way the authors consider general schedules (GS) and the concept of processor sharing.

Normally one thinks of k processors in a system as constituting k discrete units of

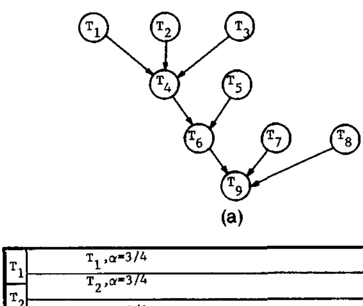
computation. A task may be assigned to a processor on a preemptive or nonpreemptive basis, and during the time that a task is assigned to a processor the total capacity of the processor is dedicated to that task. We may assume, however, that the capacity of a processor can be assigned to tasks in fractional parts α which vary between 0 and 1. Thus, for example, a task requiring t units of time when assigned a complete processor would require $2t$ units when assigned one-half of a processor (i.e., $\alpha = \frac{1}{2}$). If the amount of processor capability is allowed to vary before the task is completed, we may speak of general schedules made possible by the technique of *processor sharing*.

The first result related by Muntz and Coffman is that for a given graph, a given number of processors, and a performance measure of minimum completion time, schedules generated by a GS discipline are equivalent to schedules generated by a PS discipline. This theorem implies that processor sharing is not necessary to generate an optimal schedule if preemption is permitted.

The second major result uses the concept of levels and develops an algorithm which generates optimal preemptive schedules for tree-structured computations, an arbitrary number of processors k, and mutually commensurable node weights.

The algorithm begins by assigning one processor (i.e., $\alpha = 1$) to each of the k tasks farthest from the root of the tree. Two tasks T_i and T_j are *equidistant* from the terminal task (i.e., at the same level) if the sum of the weights of the tasks from T_i to the terminal task (including T_i) is the same as the sum of the weights of the tasks from T_j to the terminal task. If at any time the number of tasks n competing for processors is greater than k, then each of the tasks at the same level is assigned a fractional part α of a processor such that $\alpha = k/n$. Tasks are executed by their assigned processors with $0 < \alpha \leq 1$ until either 1) a task in the tree is completed; or, 2) if the current processor assignment is continued, some task(s) at the same distance from the terminal node is (are) bypassed unless a reassignment is made. When either of these two events occurs, processors are reassigned according to the initial assignment procedure. A schedule formulated according to these rules is termed an M-schedule; it is illustrated in Fig. 12. The authors show that the M-schedule generated in this manner is an optimal schedule. Since the M-schedule is a GS, and since a PS is equivalent to a GS, it follows that the algorithm yields an optimal preemptive schedule. The preemptive schedule of Fig. 12c is obtained by observing that all tasks executed within each of the completion time intervals of Fig. 12b are independent. Thus within each interval the tasks can be scheduled optimally by using preemptive techniques [42]. This is done by assigning a task to a processor until the task is completed or the execution interval is exceeded. In the first case a new task is initiated at the point of completion; in the second case the task is allocated to the next processor in sequence.

Coffman and Graham [10] informally refer to the algorithm described above as a generalized "critical-path" algorithm since



T1, α=3/4; T2, α=3/4; T3, α=3/4; T5, α=3/4; T4; T6; T9; T7; T8; T5; T7; T8

0 1 2 3 4 5 6 7 8 9 10 11 12 13

(b)

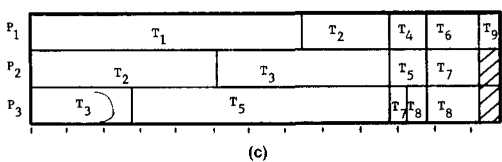


FIGURE 12. Illustration of Muntz and Coffman algorithm with $k = 3$. (a) Rooted tree with mutually commensurable node weights. Individual node weights are: $w_1 = 7^1/_2$, $w_2 = 7^1/_2$, $w_3 = 7^1/_2$, $w_4 = 1$, $w_5 = 8$, $w_6 = 1^1/_2$, $w_7 = 2$, $w_8 = 2$, $w_9 = {}^1/_2$; (b) M-schedule; (c) Optimal preemptive schedule.

tasks are given priority based on their distance from terminal tasks.

Lam and Sethi [42, 48] have adapted the level algorithms used by Muntz and Coffman to study this type of algorithm on a system in which the processors are not identical. They show that the level algorithm produces the shortest preemptive schedules on two processors for arbitrary task systems. When the number of processors is increased to three or more, the level algorithm does not produce an optimal schedule even if the precedence structure is a tree. Instead, the level algorithm is used to provide bounds on the execution time, for both identical and nonidentical processors, when compared to the optimal schedule. These bounds—expressed in terms of m, the number of processors—are shown to be worse when the processors are of different speeds ($\sqrt{1.5m}$) than when the processors are alike ($2-2/m$).

In their study, Liu and Yang [26] schedule independent tasks in a heterogeneous system in which the capacity of a processor is stated in terms of what they call a standard processor. A processor is said to have speed b if it is b times as fast as a standard processor. Lin and Yang consider a multiprocessor system which contains n_1 processors with speed b_1, n_2 processors with speed $b_2, \cdots$, and n_k processors with speed b_k. For this configuration they develop an expression for the minimum completion time using an optimal preemptive scheduling algorithm.

Nonpreemptive schedules——In nonpreemptive or basic schedules, a processor assigned to a task is dedicated to that task until it is completed. The initial investigations discussed here develop optimal nonpreemptive two-processor schedules for arbitrary task orderings in which all tasks are of unit duration.

The key to the solution of this problem by Fujii, Kasami, and Ninomiya [14, 15] is the division of the total task set into compatible and incompatible task pairs. A pair of tasks T_i and T_j is said to be *compatible* if $T_i \nless T_j$ and $T_j \nless T_i$. For a set of n tasks, let m represent the maximum number of disjoint compatible task pairs. Then $n - m$ is a lower bound on the time necessary to execute all the tasks. The approach reduces to finding the maximum number of compatible task pairs and then finding an optimal sequencing from the tasks in this set to the remaining tasks.

In their study, Coffman and Graham [10] develop an algorithm for generating a job list and show that the schedule generated by this list is at least as good as any schedule generated by any other list. A *list schedule* (or list, or task list) L for a graph G of n tasks—denoted by $L = (T_1, T_2, \cdots, T_n)$—represents some permutation of the n tasks. A task is said to be *ready* if all of its predecessors have been completed; in using a list to generate a schedule, an idle processor is assigned to the first ready task found in the list. It follows, therefore, that if a list is to be used to generate an optimal schedule the ordering of the tasks in the list is of primary importance. Thus the key to the Coffman and Graham approach is finding the list from which the optimal schedule can be produced.

The algorithm used for generating the optimal list is a recursive procedure which begins by assigning subscripts in ascending order to the task or tasks which is (are) executed last owing to precedence constraints in the task graph. Notice that the set of successors of these tasks is empty. Assignment proceeds "up the graph" in a manner that considers as candidates for the assignment of the next subscript all tasks whose successors have already been assigned a subscript. Consideration of tasks in this manner amounts to examining tasks in a given latest-precedence partition, although tasks are not executed at a time that corresponds to this partition. In effect, the tasks in a graph can be initially assigned subscripts in an arbitrary manner. The Coffman and Graham algorithm then reassigns subscripts in the manner outlined above, and the list is formed by listing the tasks in decreasing subscript order, beginning with the last subscript assigned. The optimal schedule is formed by assigning ready tasks in this list to idle processors. The algorithm is illustrated in

Fig. 13 by means of a task graph with reassigned subscripts, the resultant list L^*, and the optimal schedule [10].

Through the use of counterexamples Coffman and Graham show that their algorithm does not always yield optimal results when the number of processors is increased to three or more, or when the number of processors is two and tasks are allowed to have arbitrary durations. This is true even when task durations are allowed to be one or two units. Fujii, *et al.*, indicate that in the two-processor case tasks of nonunit length can be split into a series of tasks of length one, and their algorithm yields a lower bound on the processing time of the original problem.

As indicated by Sethi [42], Muraoka [46] developed an optimal algorithm for this environment "by first considering · · · task systems in which for all tasks T, the sum of the maximal path length from an initial node to T and the level of T is a constant. The algorithm is then extended to general task systems."

Optimal results for nonpreemptive schedules have also been addressed by T.

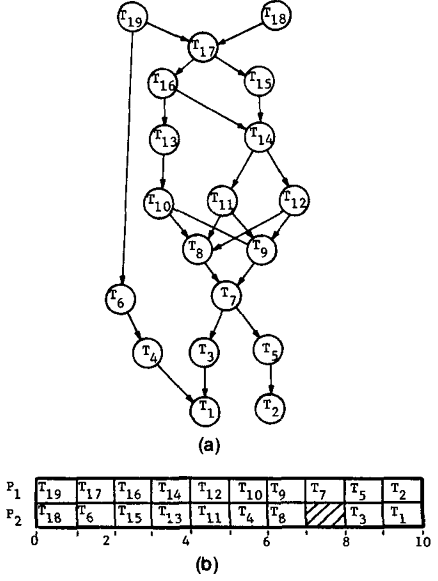


FIGURE 13. Illustration of Coffman and Graham algorithm. (a) Task graph with reassigned subscripts $L^* = (T_{19}, T_{18}, \cdots, T_1)$, (b) Optimal schedule.

S. Hu [20] in what, next to Johnson's results for two-machine flow-shop schedules ([21]; see also Section 3 above), is probably the most frequently cited reference in multiprocessor scheduling. Hu addressed two problems for tasks of unit duration: 1) Given a fixed number of processors, determine the minimum time required to process a task graph; and 2) Determine the number of processors required to process a graph in a given time.

The first step in arriving at a solution to these problems involves the labeling of the nodes of an arbitrary graph. A node N_i is given the label $\alpha_i = X_i + 1$, where X_i is the length of the longest path from N_i to the final node in the graph. Labeling begins with the final node, which is given the label $\alpha_1 = 1$. Nodes that are one unit removed from the final node are given the label 2, and so on. This labeling scheme makes it clear that the minimum time T_{min} required to process the graph is related to α_{max}, the node(s) with the highest numbered label, by

$$T_{min} \geq \alpha_{max}.$$

Hu's optimal solutions to the questions posed earlier are limited to rooted trees. Using the labeling procedure described above, one can obtain an optimal schedule for m processors by processing a tree of unit-length tasks in the following manner:

1) Schedule first the m (or fewer) nodes with the highest numbered label, i.e., the starting nodes. If the number of starting nodes is greater than m, choose m nodes whose α_i is greater than or equal to the α_i of those not chosen. In case of a tie the choice is arbitrary;
2) Remove the m processed nodes from the graph. Let the term "starting node" now refer to a node with no predecessors;
3) Repeat steps 1 and 2 for the remainder of the graph.

Schedules generated in this manner are optimal under the stated constraints. The labeling and scheduling procedures are quite simple to implement and are illustrated in Fig. 14.

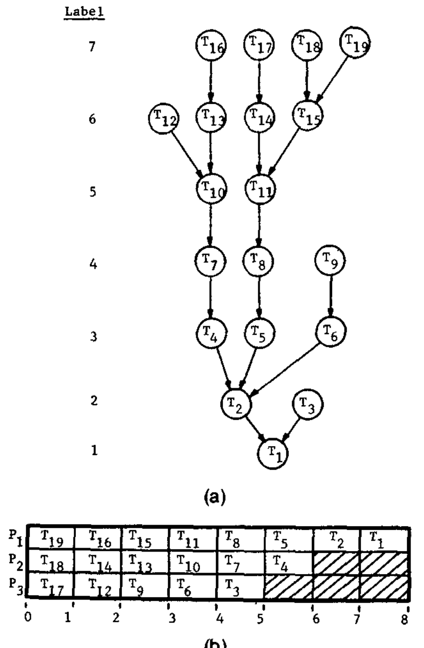


FIGURE 14 Illustration of Hu's optimal algorithm. (a) Rooted tree labeled according to Hu's procedure; (b) Optimal schedule for three processors.

As indicated earlier, the minimum time required to process a graph labeled according to Hu's procedure is α_{max}. Suppose one wishes to process a graph within a prescribed time t, where $t = \alpha_{max} + C$ and C is a nonnegative integer. The minimum number of processors m required to process the graph in time t is given by

$$m - 1 < [1/(\gamma^* + C)] \sum_{j=1}^{\gamma^*} p(\alpha_{max} + 1 - j) < m,$$

where $p(i)$ denotes the number of nodes in the graph with label α_i, and γ^* is the value of the constant γ which maximizes the given expression. To illustrate this result, consider Fig. 4.6. For $C = 0$, for example, value γ^* occurs when $\gamma = 1$ or $\gamma = 2$. This indicates that in order to process the graph in minimum time four processors are needed. For $C = 1$, $t = 8$ and γ^* occurs when $\gamma = 2$ or $\gamma = 5$, and three processors are required. Varying C further we find that three processors are required when the tasks must be processed within nine units, but only two processors are needed for a maximum processing time of 10 units.

The environment just described deals with a *level algorithm* as defined by Chen and Liu [7]. The algorithm developed by Coffman and Graham [10] is also a level algorithm. Given a set of tasks, a partial order, unit task times, and a nonpreemptive discipline, Chen and Liu define a *level* of a job as follows:

1) The level of a task that has no successor is equal to 1.
2) The level of a task that has one or more successors is equal to one plus the maximal level value of the levels of the successors of the job.

A *simple level algorithm* (SLA) is a level algorithm in which the scheduling of jobs within the same level is completely arbitrary. If ω_{SLA} is the total execution time produced by a SLA, and ω_O is the total execution time with an optimal schedule, Chen and Liu show that for a two-processor system

$$\omega_{SLA}/\omega_O \leq 4/3.$$

For a three-processor system this ratio is expressed as

$$\omega_{SLA}/\omega_O \leq 3/2.$$

This notion of comparing the performance of an algorithm to a theoretical optimum is considered in depth in subsequent discussion.

In contrast to the previously cited optimal results, most of the results described below are expressed in terms of bounds. That is, the schedule which results from a heuristic or approximate approach is expressed as a ratio that compares the suboptimal schedule to the optimal schedule. This ratio is greater than or equal to one, and provides an indication of how the performance of a simplified approach compares to an optimal solution. Two observations should be made here. First, it often happens that a heuristic schedule yields a solution which is as good as the optimal solution. Thus it is tempting to attribute undeserved merit to an approach before its worst-case performance bounds are determined. Second, when comparing a heuristic schedule to an optimal schedule, it should be remembered that the latter is not necessarily determined since, with the

exception of the special cases already discussed, nonenumerative solutions to determine the optimal result are not available. That is, short of enumerating all possible solutions and then selecting the best one, the optimal solution cannot be found. For problems with a small number of possibilities, enumeration may not be especially difficult. However, when the possibilities are very large the exponentially increasing computational time required to enumerate all possible solutions enhances the attractiveness of heuristic approaches.

The results below are discussed in terms of list schedules. A *list scheduler* executes a task only if all of its predecessors have been completed and no task preceding it in the (priority) list is ready to run.

Probably the most significant and the earliest contributions on generating bounds for suboptimal multiprocessor schedules have been made by R. L. Graham [17-19]. It was in connection with the study of so-called *multiprocessor anomalies* that the bounds discussed here were developed. These anomalies, cited in earlier investigations (see [18]), arise from the counterintuitive observation that the existence of one of the following conditions can lead to an *increase* in execution time:

1) Replace a given task list L by another list L' leaving the set of task times μ, the precedence order $<$, and the number of processors n unchanged;
2) Relax some of the restrictions of the partial ordering;
3) Decrease some of the execution times;
4) Increase the number of processors.

Graham has developed a general bound by executing a set of tasks twice. During the first execution the tasks are characterized by the parameters μ, $<$, L, n, and ω (the length of the schedule) and during the second execution by μ', $<'$, L', n', and ω' such that $\mu' \leq \mu$, every constraint of $<'$ is also in $<$, i.e., $<'$ is contained in $<$. The result of this general bound is that

$$\omega'/\omega \leq 1 + [(n - 1)/n'].$$

Graham has shown that this bound is the best possible, and for $n = n'$ the ratio $2 - 1/n$ can be achieved by the variation of any one of L, μ, or $<$.

The anomaly that results when task execution times are reduced is referred to by Manacher [27] as a "Richard's anomaly" since anomalies of this type were apparently first discussed by Richards [35]. Results of simulations reported by Manacher showed that approximately 80% of all test cases displayed Richard's anomaly. Manacher developed an algorithm to provide "stability in a strong sense" such that the completion time of all tasks in a task list is not increased by reducing the execution time of any of the tasks. This is accomplished by adding "a modest number" of precedence constraints to the original partial ordering. Manacher also considered the stability problem for the case in which multiple initial tasks have different starting times.

In the preceding discussion it was assumed that once a task list is created it remains fixed or static until all tasks in the list are executed. A variation of this, the dynamically formed list [18], seeks to redefine the list every time that a processor becomes free. When this occurs, the task that heads the "longest chain of unexecuted tasks" (in the sense that the sum of the task times in the chain is maximal) is executed first. Let ω_L be the finishing time for the set of tasks executed in this manner, and let ω_0 be the minimum finishing time. Using the general bound cited earlier, we find $\omega_L/\omega_0 \leq 2 - 1/n$, since the dynamically formed list amounts to replacing L by L'. Graham [18] has developed a slightly better best-possible bound given by

$$\omega_L/\omega_0 \leq 2 - [2/(n + 1)].$$

An alternative to this approach is to assign to a ready processor the task whose execution time plus the execution time of all of its successors is maximal. If a set of tasks executed in this manner has a finishing time of ω_M, then ω_M/ω_0 is also bounded by the preceding expression.

A special case for the "dynamic longest chain" approach occurs when $<$ is empty, i.e., when the tasks are independent. For this case, Graham produced a best possible bound given by

$$\omega_L/\omega_0 \leq 4/3 - 1/3n.$$

As stated earlier, the primary reason for the development of these bounded expressions is to provide good suboptimal schedules while investing only a fraction of the computational effort required to generate an optimal result. Suppose that upon examination of a set of r tasks to be scheduled it is determined that the set is too large for an enumerative approach. Then the following alternative appears promising (again for the case $< \, = \emptyset$): optimally schedule the k longest tasks ($k \geq 0$), and schedule the remaining $r - k$ tasks in an arbitrary manner. The bound developed by Graham for this approach is given by:

$$\frac{\omega(k)}{\omega_0} \leq 1 + \frac{1 - 1/n}{1 + \lceil k/n \rceil},$$

where n is the number of processors being used. Two special cases exist for this result. When $k = 0$,

$$\omega(0)/\omega_0 \leq 2 - 1/n.$$

This was the bound developed for the initial general bound for $n = n'$. If $k = 2n$,

$$\omega(2n)/\omega_0 \leq 4/3 - 1/3n.$$

Thus the two previous results were simple special cases of a more general result.

In a later reference [19], Graham addresses the reverse question: Given a fixed deadline, what is the minimum number of processors required? (Recall that T. C. Hu [20] addressed this question for the special case of a rooted tree.) If it is assumed that $<$ specifies no relation (that is, tasks are independent), then the problem reduces to the one-dimensional cutting-stock problem as well as a special case of the assembly-line balancing problem. In effect, the problem can be viewed in the following manner. Assume that a set of objects are to be placed in a set of identical boxes. Assume that the objects all have the same length and width (but not height) and these two dimensions exactly match the corresponding dimensions of the boxes. The problem then becomes one of minimizing the number of boxes to contain the objects. If we equate tasks to objects and boxes to processors, then the heuristics developed by Graham can be used to find upper bounds for the minimum number of processors. An in-depth treatment of this subject is given by Graham in [42].

Thus we see that, for the special case in which there is no precedence, the results from another discipline can be used to obtain good suboptimal results with significant decreases in computational requirements. Heuristic approaches to the problem discussed here and the flow-shop problem discussed earlier are considered in [24].

Longest-path algorithms have also been investigated by Kaufman [23] for tree-structured graphs. The environment allows unequal task times but does not allow preemption. In a manner similar to that of Muntz and Coffman [30], tasks with weights greater than one are represented by a string of unit-weight tasks whose sum equals the weight of the original task. Representing the graph in this manner allows one to determine the optimal nonpreemptive solution using Hu's algorithm, since the graph is a tree. Kaufman's longest-path or G algorithm, however, does not allow a processor to be preempted from a task upon completion if that task is a member of the string of tasks representing a non-unit-weight task.

If ω_P represents the optimal preemptive schedule, ω_N represents the optimal nonpreemptive schedule, and ω_G represents the schedule determined by the G algorithm, then the bounds obtained by Kaufman are

$$\omega_P \leq \omega_N \leq \omega_G \leq \omega_P + k - \lceil k/n \rceil,$$

where k is the weight of the largest task in the original graph and n is the number of processors available to any of the algorithms.

Adam, Chandy, and Dickson [1] have compared through extensive simulation the performance of several list schedules made in an unrestricted environment. The environment of Adam, et al., allows for general graph structures, two or more processors, unequal task durations, and no preemption of tasks.[5] The five heuristics studied are:

1) HLFET (Highest Levels First with

[5] These heuristics were also compared in a nondeterministic environment.

Estimated Times). The term "level" as used here refers to the sum of the weights of all vertices in the longest path from a task to the terminal node. (Since we are not assuming independent tasks, predecessor tasks must be completed before a task can be initiated);

2) HLFNET (Highest Levels First with No Estimated Times). In effect, all tasks are assumed to have equal task times;
3) RANDOM. Tasks are assigned priorities randomly;
4) SCFET (Smallest Colevels First with Estimated Times). A colevel of a task is measured in the same manner as its level, except that the length of the path is computed from the entry node rather than from the terminal node. Priority is assigned according to colevel (i.e., the smaller the colevel, the higher the priority);
5) SCFNET (Smallest Colevels First with No Estimated Times). SCFNET is the same as SCFET except that all tasks are assumed to have equal duration. This amounts to an earliest precedence partition if execution times are ignored.

Extensive simulations based on real and on randomly generated graphs show that the order of accuracy among the graphs is: HLFET, HLFNET, SCFNET, RANDOM, and SCFET. The near-optimal performance of HLFET again confirms the usefulness of longest-path schedules when the measure of performance is minimum completion time. The level of performance achieved by Adam, et al. (within 4.4% of optimal) for longest-path scheduling is comparable to that reported by Manacher in [27] (15%). The near-optimality of longest-path or critical-path scheduling has also been confirmed by Kohler [47], who demonstrated that the performance of this heuristic increases as the number of processors increases.

Recall that in an earlier discussion the objective was to minimize mean flow time for a single processor. For a set of m processors ($m \geq 2$) and a set of independent tasks, if all tasks are scheduled according to the shortest-processing-time (SPT) discipline, then the resultant schedule is guaranteed to display a minimum mean finishing time [11]. The SPT discipline does not necessarily minimize the maximum finishing time, however. (The problem of minimizing the maximum finishing time of a set of independent tasks on two processors—and, therefore, on more than one processor—is known to be NP-hard, i.e., not likely to be solvable by a nonenumerative procedure.) Evidence that two SPT schedules with the same mean finishing time do not yield the same finishing time is shown in Fig. 15—where SPT* denotes the minimum-finishing-time SPT schedule for the same three processors and set of tasks. Bruno, Coffman, and Sethi [3] have compared the finishing time characteristics of the optimal SPT schedule to those of the optimal schedule and developed the following bound:

$$\omega_{SPT}^*/\omega_{OPT} \leq 2 - 1/m.$$

Coffman and Sethi [42, 50] have shown that the longest SPT schedule is at most 50% longer than the SPT* one. If the list of nondecreasing tasks is assigned in rotation to the m processors, this figure can be reduced to 33% if the longest m tasks are assigned largest-first on the m processors. Assigning all sets of m tasks largest-first drops the bound to at most 25% worse.

By forming longest-processing-time (LPT) schedules (which tend to maximize

TABLE III. TASK PROCESSING TIMES FOR A SET OF TASKS

T	P_i
1	1
2	2
3	4
4	4
5	5
6	8

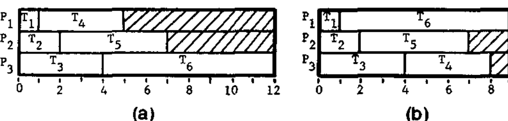


FIGURE 15. Comparison of STP and STP* schedules for three processors and the tasks listed in Table III. (a) STP schedule; (b) STP* schedule.

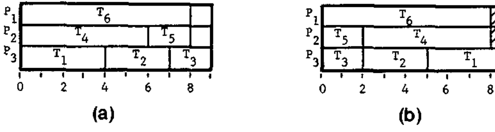


FIGURE 16. Comparison of LPT and RPT schedules for three processors and the tasks listed in Table III. (a) LPT schedule; (b) RPT schedule.

mean finishing time but minimize the maximum finishing time)[6] and then arranging the tasks assigned to a processor in SPT fashion, the so-called RPT discipline produces schedules with good maximum-finishing-time and near-optimal mean-finishing-time properties, as shown in Fig. 16 [3].

Ramamoorthy, Chandy, and Gonzalez [32] use the concept of precedence partitions to generate bounds on processing time and the number of processors for graph structures whose nodes require unit execution time. As indicated earlier, precedence partitions group tasks into subsets to indicate the earliest and latest times during which tasks can be started and still guarantee minimum execution time for the graph. This time is given by the number of partitions and is a measure of the longest path in the graph. For a graph of l levels, the minimum execution time is l units. In order to execute a graph in this minimum time, the absolute minimum number of processors required is given by

$$\max \{|L_j \cap E_j|\}, 1 \leq j \leq l,$$

where L_j and E_j refer to the jth latest and earliest precedence partitions, respectively, and $|x|$ represents the cardinality of the set x. Ramamoorthy, et al., refer to the tasks contained in $L_j \cap E_j$ as *essential tasks*. Those tasks contained in the jth subset given by $L_j \cap E_j$ must be initiated $j - 1$ units after the start of the initial task in the graph to guarantee minimum execution time. In a manner similar to that of T. C. Hu [20], the authors develop a lower bound for the minimum number of processors when the execution time is allowed to exceed l and for the minimum execution time when the number of processors is fixed. A rooted tree structure for the graph is not required. The L-partition is also used by the authors to develop lower and upper bounds for the minimum number of processors required to process a graph in the least possible time.

Ramamoorthy, et al., developed algorithms to determine the minimum number of processors required to process a graph in the smallest possible time and to determine the minimum time to process a task-graph given k processors. The second of these algorithms is modified to allow for the scheduling of graphs with unequal task durations. A complication in this case is that it is often desirable to keep a processor idle even when there is something to do. Figure 17b, for example, is an optimal schedule for the graph of Fig. 17a and requires 15 units. If Processor 2 is assigned to Task 6 upon completion of Task 3, however, the time required is 17 units, as shown in Fig. 17c.

The computational time required to obtain the optimal solution by means of these algorithms is considerable. This

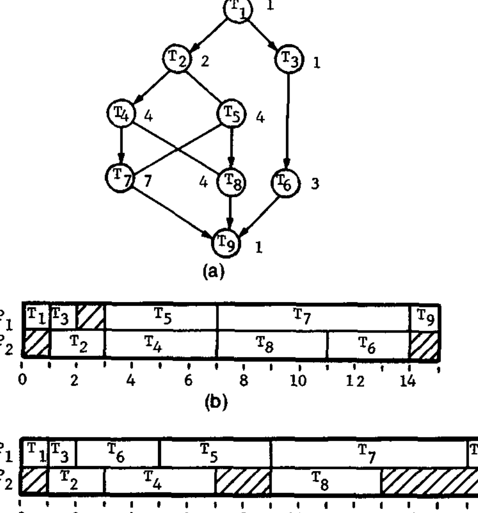


FIGURE 17. Illustration of the effects of deliberately idling a processor. (a) Task graph for a set of tasks; (b) Optimal schedule; (c) Schedule when processors are activated as soon as possible.

[6] Denning and G. S. Graham [52] have shown, however, that it is possible for an LPT schedule to display the worst-case finishing time characteristics of the bounds developed by R. L. Graham [18] for an arbitrary schedule compared to the optimal one.

time was significantly reduced by means of two heuristics that yielded the optimal result most of the time. In Heuristic A, no processor is deliberately idled, and tasks are chosen according to their position in the L partitions. In Heuristic B, essential tasks are chosen first. Of the two, Heuristic A was the faster; in all the cases tested the heuristics also yielded the optimal solution.

The bounds discussed in the preceding paragraphs have been improved by Fernandez and Bussell [12] using the critical-path approach. For a given graph there exists a path called the *critical path*, from the entry node to the exit node, which defines a minimum execution time for the graph. (This concept does not require equal task times.) Given the critical path execution time t_{cp}, there exists a "completion interval" (based on earliest and latest start times) for each task in the graph, during which that task must be completed in order that the completion time not exceed t_{cp}.

In arriving at a lower bound on the number of processors, Fernandez and Bussell consider integer intervals between 0 and t_{cp}. Within each of these intervals, tasks are shifted to give minimum overlap within the interval. The average number of processors required within an interval represents the minimum number of processors required for that interval. If all such intervals are examined, the maximum average value rounded up to the nearest integer represents the minimum number of processors required to process the graph in minimum time.

Similar concepts are used to determine the minimum execution time when the number of processors is fixed. During each interval a certain amount of processing must take place to ensure that total execution times does not exceed t_{cp}. If the number of processors used is less than a certain minimum, then each interval will contribute an amount of time in excess of what it would contribute if t_{cp} is to be satisfied. The maximum deficit contributed by all intervals represents the minimum amount of time over and above t_{cp} to process the graph.

In a later study, Bussell, Fernandez, and Levy [4] address the problem of designing algorithms for minimizing the number of processors required to execute a schedule in a given time and for minimizing the execution time given a fixed number of processors. Schedules that result from these objectives are referred to as *processor-optimal schedules* and *time-optimal schedules*, respectively. An environment consisting of a set of tasks, a partial ordering, unequal task times, and no preemption of active tasks is assumed.

Reduced to the simplest terms, their algorithm consists of adding precedence conditions (i.e. additional arcs) to the original graph at those points in the graph where the number of candidate tasks exceeds the number of processors. The effect is to distribute processor demands throughout the length of the graph, without adversely affecting the overall execution time. As is the case for the Ramamoorthy, Chandy, and Gonzalez algorithms cited earlier [32], the actual description and implementation of the algorithm and its component parts is considerably more difficult than the basic premise on which the algorithm is based would suggest. This is borne out in both cases by appreciable computational time on large computer systems. The alternative—enumeration—is, of course, much less desirable. Ramamoorthy, et al. [32] provide suboptimal heuristic alternatives to the optimal solution, and Bussell, et al. [4] provide suboptimal alternatives which can be implemented interactively. Both of these efforts emphasize once again the difficulty of obtaining optimal solutions in the general scheduling environment.

Liu and Yang [26] have developed bounds for the minimum completion time of an arbitrary set of tasks when the tasks are not all independent and the processors are not necessarily identical.

Comparison of Preemptive and Nonpreemptive Schedules——The minimization of completion time for some special cases has been discussed by Coffman and Graham in the previously cited work concerning optimal nonpreemptive two-processor schedules [10]. The pivotal point in

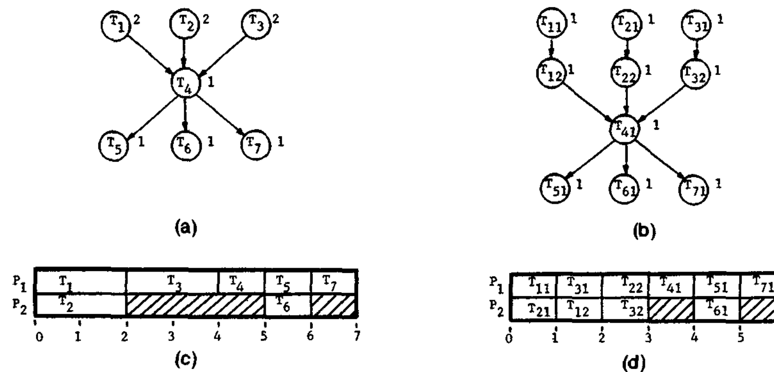


FIGURE 18. Comparison of optimal preemptive and nonpreemptive schedules. (a) Task graph G; (b) Task graph G_w ($W = 1$); (c) Optimal nonpreemptive schedule corresponding to Graph G; (d) Optimal schedule corresponding to Graph G_w.

the comparison is the ability to transform a graph G with arbitrary mutually commensurable task weights into a Graph G_w having execution times of w time units. As before, w is the largest task weight in G which evenly divides all the task weights in G. If preemptions are allowed only at times that are multiples of w, then an optimal nonpreemptive schedule for G_w can be viewed as an optimal preemptive schedule for G. For example, Fig. 18 shows the performance improvement achieved by allowing preemption at the end of each unit interval (i.e., $w = 1$) [10]. Allowing preemptions at the end of each unit interval results in a performance improvement of 7/6.

Coffman and Graham observe that no benefit is achieved by allowing preemption more frequently than every $w/2$ units and show that, for an arbitrary number of processors m ($m \geq 1$), the length ω_N of a nonpreemptive schedule is related to the length ω_P of a preemptive schedule by:

$$\omega_N/\omega_P \leq 2 - (1/m).$$

Schedules to Minimize Mean Flow Time

This section considers the generation of schedules when the objective is to minimize mean flow time for a set of independent tasks. Earlier it was indicated that this measure of performance can be minimized for a set of identical processors by scheduling the tasks according to the SPT discipline. In this section, however, the processors are allowed to be nonidentical or heterogeneous. The results related here provide evidence that scheduling considerations are becoming more sophisticated. These considerations reflect the growing acceptance of multiple and distributed processor systems and the practicality of increasing system capacity or replacing failed or obsolete components by adding nonidentical replacements.

In their study Bruno, Coffman, and Sethi [3] develop an efficient algorithm for scheduling independent tasks to reduce mean finishing time (i.e., mean flow time). Because the processors are not identical, it is no longer valid to use a single value to represent a task's execution time. Instead it is necessary to consider task execution time on each of the processors. A convenient way to do this for m processors and n tasks is by means of an m times n matrix $[\tau_{ij}]$ such that the nonnegative integer τ_{ij} denotes the execution time of task T_j on processor P_i. An example of this matrix for five tasks and three processors is:

$$[\tau_{ij}] = \begin{bmatrix} 2 & 3 & 1 & 4 & 6 \\ 1 & 4 & 1 & 5 & 5 \\ 3 & 2 & 3 & 2 & 3 \end{bmatrix}$$

The corresponding optimal schedule is shown in Fig. 19.

From the matrix $[\tau_{ij}]$, the $nm \times n$ matrix Q is formed, where

$$Q = \begin{bmatrix} [\tau_{ij}] \\ 2[\tau_{ij}] \\ \vdots \\ n[\tau_{ij}] \end{bmatrix}.$$

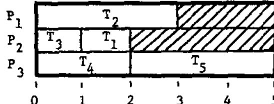


FIGURE 19. Illustration of optimal schedule using Bruno, et al., algorithm.

A set of n elements in this matrix is called a feasible set if no two elements are in the same row. The cost of this set is the sum of the weights of the n elements. The objective, then, is to find a feasible set with the smallest possible cost. The authors formulate the problem as a transportation problem and arrive at a nonenumerative optimal solution.

If a priority or urgency measure w is assigned to each task and the processors are assumed to be identical, then Bruno, et al., show that the problem is NP-hard. For m processors and n tasks, however, Eastman, Even, and Isaacs [51] show that the following is a lower bound for the weighted mean flow-time $\bar{F}_w(m)$:

$$\bar{F}_w(m) \geq [(m + n)/m(n + 1)]\,\bar{F}_w(1),$$

where $\bar{F}_w(1)$ is the weighted mean flow time with one processor.

The optimal solution of Bruno, et al., is used by Clark [8] to generate simple heuristics which provide near-optimal mean-finishing-time schedules. Clark assumes n independent tasks on m nonidentical processors and, like Bruno, et al., uses a matrix P to describe P_{ij}, the processing time of task i on processor j.

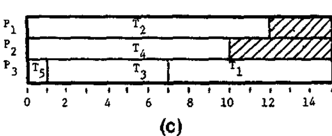
In his first result Clark observes that P_{ij} can be expressed as "the product of a time associated with job i and an efficiency factor associated with processor j, that is, $P_{ij} = p_i w_j$." In the resulting algorithm, a matrix of processing times and processor coefficients is formed as shown in Fig. 20a [8]. Note that the processing times are arranged so that $p_1 \geq p_2 \geq \cdots \geq p_5$. Starting with the top row of the matrix, the smallest coefficient is circled. This means that the job found in the first row of the matrix (Job 1 in this example) is assigned to the processor corresponding to the circled coefficient (Processor 3 in this example). The second row of the matrix is

i	Pi	potential coefficients 1	2	3	machine chosen
1	8	2	2 5	1	3
2	6	2	2 5	2	1
3	6	4	2 5	2	3
4	4	4	2 5	3	2
5	1	4	5	3	3

(a)

i	1	2	3	h_1	h_2	h_3
1	16	20	8	1	1	1
2	12	15	6	1	1	2
3	12	15	6	2	1	2
4	8	10	4	2	1	3
5	2	2 5	1	2	2	3

(b)



(c)

FIGURE 20. Illustration of Clark's algorithm. (a) Matrix of processing times and job efficiencies, and selection of machines; (b) Alternate method of selecting machines; (c) Optimal schedule.

formed by copying the uncircled coefficients from the previous row and increasing the value of a circled coefficient by the corresponding w_j. Repeating the above process $n - 1$ times (with arbitrary selection in case of ties) results in a schedule with optimal mean finishing time. The optimal schedule for this example and the calculation of the flow time are shown in Fig. 20b. In a variation of this approach, the entire matrix is filled initially with explicit processing times, and a matrix of coefficients is formed in a manner similar to that outlined above. Let the coefficients in a particular row i be $h_1, h_2, \cdots, h_m$ such that the coefficients in each row represent the possible sequence positions on the three processors, counting from the *end* of the schedule, for the corresponding job. Then the minimum $h_i\, p_{ij}$ is chosen, p_{ij} is circled, and h_j is increased by one in the next row. (See Fig. 20c.) The value of F for this example is:

$$\begin{aligned} F &= w_1 p_2 + w_2 p_4 + 3w_3 p_5 + 2w_3 p_3 + 1w_3 p_1 \\ &= 2\cdot 6 + 2.5\cdot 4 + 3\cdot 1\cdot 1 + 2\cdot 1\cdot 6 + 1\cdot 1\cdot 8 \\ &= 45. \end{aligned}$$

Since machine factors are not used, this latter approach has more general applica-

bility; however, it does not guarantee optimality. Because of its simplicity, Clark uses this approach throughout his subsequent work and refers to it as the quick-and-dirty (QAD) algorithm. (A variation of this algorithm, QAD*, sorts jobs on each machine in SPT order since this guarantees that flow time on each machine is minimized.) Clark then proves that there exists a renumbering of the jobs (a permutation of the rows in the processing-time array) such that QAD yields a schedule with minimal flow time (and, equivalently, mean flow time).

Liu and Yang [26] have developed an algorithm for minimizing mean flow time for a set of independent tasks and the special case consisting of one processor of speed b plus n standard processors. The authors also provide a bound which compares the performance with respect to mean flow time of a homogeneous system of $n + 1$ standard processors to that of a nonhomogeneous system containing n standard processors plus one processor which is b times more powerful than a standard processor.

Special Scheduling Environments

In this section new constraints are added to the more common constraints assumed in the preceding section. These new constraints are in the form of resource classes, periodic jobs with hard deadlines, and intermediate deadlines.

Systems with Limited Resources

The results surveyed in this paper thus far have been concerned primarily with the allocation of processors. Computational requirements have been expressed in terms of processing time and precedence conditions. It has therefore been assumed that a processor is the only resource that a job or task requires. Recognition of the fact that a task may require resources other than a processor has recently led to investigations of "systems with limited resources" in which it is assumed that requirements exist for multiple resources that are limited in number. The primary references in this area are by Garey and Graham [16, 42] and Yao [40].

The Garey and Graham model augments a standard model—consisting of a set of r tasks of unequal duration related by a precedence order and executed on a nonpreemptive basis—by a set of n identical processors. In addition it is assumed that a set $R = \{R_1, \cdots, R_s\}$ of resources is available. If task T_i requires resource R_j, we assume that the requirement exists throughout the execution of the task. The need of task T_j for resource R_i is denoted by ρ_{ij}, where $0 \leq \rho_{ij} \leq 1$. Let $r_i\ (t)$ denote the total amount of resource R_i which is being used at time t. Then $r_i\ (t) = \sum \rho_{ij}$ for all T_j being executed at time t and $r_i\ (t) \leq 1$. The basic problem considered is to what extent the use of different list schedules for this model can affect the finishing time ω.

Suppose that for two arbitrary lists L and L' the augmented system of n processors executes the set of r tasks with the resulting finishing times ω and ω' respectively. For this environment Garey and Graham provide the following results:

1) For $R = \{R_1\}$ (i.e., when there is only one resource other than processors in the system), $\omega/\omega' \leq n$;
2) For $R = \{R_1\}$ and all tasks independent, $\omega/\omega' \leq 3 - 1/n$;
3) For $R = \{R_1, R_2, \cdots, R_s\}$, all tasks independent, and $n \geq r$, $\omega/\omega' \leq s + 1$.

The net effect of these results is to indicate that addition of resource considerations to the standard model causes an increase in the worst-case behavior bounds.

Yao [40] uses essentially the same model as Garey and Graham except that all tasks require one unit of time to complete. Using this model, Yao provides bounds for a large number of cases, based on the number of tasks, the number of processors, and the rules used to form list schedules. Like Garey and Graham, Yao observes that his algorithms behave less well when resource constraints are eliminated. A related observation is that investing some effort in the preparation of a list can lead to better results.

In a somewhat less abstract sense, Kafura and Shen [22] assume that individual

tasks require a minimum amount of memory in addition to a certain amount of processing time. A system of m identical processors and n independent tasks is assumed such that each processor is associated with a private storage device of a given capacity. When a processor completes a task it examines the list of tasks and selects the first task whose memory requirement is less than or equal to its own memory capacity. Assuming a nonpreemptive environment in which no processor is allowed to remain idle if there is a task in the task list that it could execute, the authors develop bounds and heuristic strategies for selecting tasks on the basis of time and memory requirements.

Periodic Job Schedules

Periodic jobs were considered earlier in this survey, in the section on single-processor schedules. At that time, in addition to limiting attention to a multiprogrammed uniprocessor environment, preemption of the periodic jobs in order to meet the deadline of a higher-priority job was permitted. Multiprocessor schedules for a set of independent periodic jobs on a nonpreemptive basis [38, 41] are considered here.

In this section we assume that all tasks are simultaneously available. The objective is to minimize the number of processors required to execute a job set while guaranteeing that the periodic iterations of the individual jobs begin and end exactly on time.

If we let E_i represent the maximum execution time of one iteration of job J_i and if we represent the execution frequency by f_i, then J_i can be expressed by these two parameters as $J_i: (f_i, E_i)$, $1 \leq i \leq n$, where n is the number of jobs to be scheduled. The repetition period is represented by T_i, the

TABLE IV. JOB CHARACTERISTICS FOR A SET OF JOBS WITH BINARY FREQUENCY DISTRIBUTION

Jobs	*Frequency*	*Period*	*Execution time*
J_1	1/4	4	1
J_2	1/8	8	2
J_3	1/16	16	$1^1/_2$
J_4	1/32	32	5
J_5	1/64	64	3

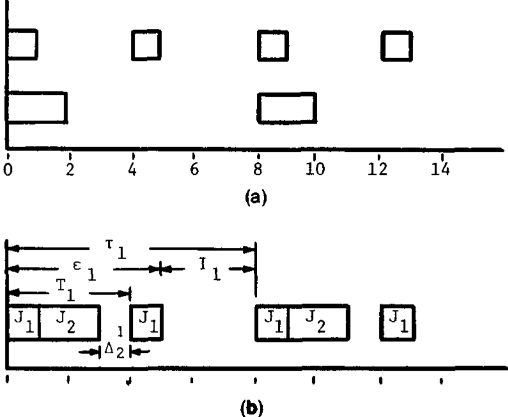


Figure 21. Scheduling of periodic jobs with a binary frequency distribution (a) Timing diagram for the first two jobs of Table IV; (b) Reduction in processor requirements through the merging of jobs.

inverse of f_i. In the following discussion two classes of jobs are considered: 1) If the n jobs, J_1 to J_n are arranged such that $f_i > f_{i+1}$, assume that $f_i = 2f_{i+1}$—i.e., all jobs have a frequency which is related to the frequency of the highest-frequency job by some power of two; 2) Jobs of any frequency are allowed.

Periodic jobs with a binary frequency distribution——A set of jobs satisfying the constraints of this section is shown in Table IV. Figure 21a shows J_1 and J_2 scheduled on separate processors, and Fig. 21b shows a schedule that reduces the number of processors from two to one. The problem, of course, is to determine the minimum number of processors required for the entire set without having to consider all possible alternatives.

Notice that the merged form of two jobs shown in Fig. 21b creates a new periodic composite job with a period of τ_1 (equal to $2T_1$) and an execution time of ϵ_1 (equal to $T_1 + E_1$). In addition two idle times are created: I_1, the *periodic idle time* with duration $\tau_1 - \epsilon_1$, and Δ_2^1 the *forced idle time* of length $I_1 - E_2$. (The notation Δ_j^i indicates that the forced idle time is formed when J_j is merged with J_i.) In attempting to merge further jobs into the schedule it is not necessary to consider the placement of jobs in the interval repre-

sented by the forced idle time. Instead, for this environment a schedule requiring the minimum number of processors is generated by the following algorithm:

1) Let $J_1^*, J_2^*, \cdots$ represent the subset of jobs assigned to processor $P_1, P_2, \cdots$. Initially $J_1^* = J_2^* = \cdots = \phi$ and $I_1 = I_2 = \cdots = \infty$. Whenever a job J_j is assigned to an empty J_l^*, $I_l = T_j - E_j$;
2) To assign J_i, find the least l such that $E_i \leq I_l$, and assign J_i to J_l^*.

The optimal schedule for the set of jobs shown in Table IV is shown in Fig. 22. This result has been extended to the case where $f_i = k(f_{i+1})$ and k is a positive integer greater than 1.

Periodic jobs with an unconstrained frequency distribution——In this section the frequency relationship between jobs assumed in the previous section is eliminated. As might be expected, the problem is now much more difficult, and no optimal solution has been found. Instead, heuristic approaches were developed and compared to each other through extensive simulation. These heuristics fall into three groups:

1) Frequency Decreasing Order. Jobs are arranged in frequency-decreasing order and are to be assigned in this order.
2) Load Factor Decreasing Order. The *job load factor* of J_i, denoted by L_i, is defined as follows: $L_i = E_i/T_i$.
3) Preserving Minimum Length of the Critical Interval. The critical interval between two jobs is defined as the minimum interval between the completion time of the first job and the initiation time of the second job at some point in the schedule. (The determination of this interval does not include the first iteration of the two jobs, where by definition the initiation of the second job immediately follows the completion of the first job.)

In testing these heuristics, job sets were classified into two types. In Type I job frequencies are multiples of more than two base frequencies, and in Type II job frequencies are multiples of two or fewer base frequencies. As might be expected, outstandingly better performance of one algorithm over the others in all cases was not found. In general, however, Heuristic 2 performed exceptionally well for Type I problems. Heuristic 3 performed best for certain Type II problem sets, and both Heuristics 1 and 2 performed well on job sets in which Heuristic 3 performed poorly. Not surprisingly, the number of processors required for Type II problem sets was considerably smaller than the number required for Type I sets.

Some very interesting anomalies were discovered as well. In many cases it was found that decreasing the job frequencies or the job execution times can result in an increase in the number of processors required. Conversely, processor requirements can be reduced by increasing the job frequencies or execution times, i.e., by increasing processor load.

Deadline-Driven Schedules

We have already noted the use of the term "deadline-driven scheduling" for an environment consisting of a single processor and a set of periodic tasks of known frequency and period. In this case we consider a multiprocessor environment in which tasks of unequal execution times are related by a given precedence structure and are to be executed in a nonpreemptive manner.

In particular, Manacher [27] considers the case in which terminal and nonterminal tasks require different completion

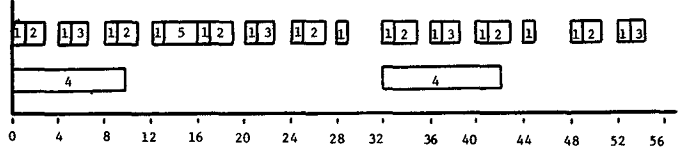


FIGURE 22. Optimal schedule for the jobs of Table IV.

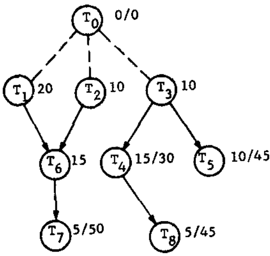


FIGURE 23 A graph with multiple deadlines.

times. An example of a graph model satisfying these requirements is shown in Fig. 23. In this figure a pair of numbers of the form A/B next to a node represents a task whose execution time is A units and which must be completed B units after the start of execution.

Manacher's heuristic solution to this problem is a variation of the longest-path schedules considered earlier. In this case, however, multiple longest paths or critical paths can be defined for the tasks contained in paths that contain tasks with deadlines. Manacher's procedure is, in effect, a variation of the latest-precedence partition for unequal task times and multiple deadlines.

In early work on scheduling with deadlines, McNaughton [28] dealt with the case in which there is a deadline associated with each task and a loss for failure to meet the deadline. In the case of an *absolute deadline*, a task has no value at all if it is not completed by the deadline. In a relative-deadline situation, the loss is zero up to a certain point, and a monotonic increasing function of the time of completion beyond that point. His principal result with respect to multiple processors assumes that a set of independent tasks all have a deadline at time zero, i.e., that all tasks are simultaneously available and have equal priority. For this case, McNaughton shows that no preemptions are necessary in order to minimize the loss function.

CONCLUSION

This discussion has attempted to survey some of the more prominent results in the scheduling of deterministic job sets. It was assumed that task graphs are acyclic with no branching and that task execution times are exactly known. It should be remembered, though, that in many computer-system environments these assumptions are not valid. Baer [2] discusses some of the implications of cycles and branches in graphs in addition to discussing some of the results considered here. Chandy [1, 6] among others considers schedules in which task execution times are not exactly known. A sizeable portion of Conway, Maxwell, and Miller's book [11] is devoted to what they call the general n-job m-machine job-shop problem, and indicates the complexity of this problem.

This survey has shown that efficient optimal algorithms exist in only a few special cases and suggests that perhaps future efforts should concentrate on the study of heuristics. The original objective of identifying deterministic schedules for use in practical computer-scheduling environments is likely to remain unfulfilled. Although these results may be of interest in operations research where the assumptions of deterministic schedules often apply, the only approach to practical job-scheduling problems may lie with either well-tested heuristics or statistical methods (the latter representing a new approach to the problem).

ACKNOWLEDGMENTS

The author would like to thank the referees for their many valuable suggestions regarding the organization and contents of this paper. Many thanks also to Peter J. Denning for additional suggestions, including the use of the term NP-hard, to I. H. Sudborough of the Computer Sciences Department at Northwestern University for his assistance in the preparation of the section on the efficiency of algorithms, and to Mark Kerstetter, Computer Sciences Department, Northwestern University, for the many contributions made by him throughout this effort.

CLASSIFICATION OF REFERENCES

Table V provides a concise annotation to most of the references cited in this survey. References are categorized according to whether the study reported in the reference considers a system with exactly one, exactly two, or two-or-more processors, or a flowshop environment. Task times can be equal (i.e., all tasks of unit duration) or unequal. Precedence con-

TABLE V. CHARACTERISTICS OF CITED REFERENCES

REF. #	# OF PROCESSORS				TASK TIMES		PRECEDENCE			TASK PREEMPTION		LIMITED RESOURCES	DEADLINES	IDENTICAL PROCESSORS		MINIMIZE			OPTIMAL RESULTS	BOUNDS
	1	2	≥2	FLOW SHOP	1	≥1	∅	TREE	GEN	PS	BS			YES	NO	MAX F	$\bar{F}$	N_p		
[1]			X			X			X		X			X		X				
[3]			X			X	X				X				X		X			X
[4]			X			X			X		X			X		X		X		X
[5]				X		X	X				X				X	X				X
[7]			X		X				X		X			X		X				X
[8]			X			X	X				X				X		X			X
[10] [14] [46]		X			X				X		X			X		X			X	
[12]			X			X			X		X			X		X		X		X
[16]			X			X			X		X	X		X		X				X
[17] [18] [19]			X			X	X		X		X			X		X		X		X
[20]			X		X			X			X			X		X		X	X	
[21]				X		X	X				X			X		X			X	
[22]			X			X	X				X	X		X		X				X
[23]			X			X		X			X			X		X				X
[25]	X					X	X			X			X							
[26]			X			X	X			X	X				X	X	X			X
[27]			X			X			X		X		X	X						
[28]			X			X	X				X		X	X	X					
[29]		X				X			X	X				X		X			X	
[30]			X					X		X				X		X			X	
[32]			X		X	X			X		X			X		X		X		
[33] [43] [44]				X		X					X				X	X				
[36]	X					X	X			X			X							
[37]				X		X	X			X	X				X	X				
[38]			X			X	X				X		X	X				X	X	
[40]			X			X	X		X		X	X		X		X				X
[41]			X			X	X				X		X	X				X	X	
[47]			X			X			X		X			X		X				
[48]		X	X			X		X	X	X				X	X	X				X
[50]			X			X	X				X			X		X				X
[51]			X			X	X				X			X			X			X
[52]			X			X			X		X			X		X		X		X

straints can be such that tasks are independent (∅), represent a tree structure (TREE), or permit a general precedence relationship (GEN). Further categorization is based on whether or not task preemption is permitted (PS and BS, respectively), whether or not system resources are limited, whether or not deadlines must be observed, and whether or not the processors are identical. The measures of performance used to categorize the references are the minimization of the maximum finishing time (MAX F), the minimization of the mean flow time ($\bar{F}$), and the minimization of the number of processors (N_p). The last two columns indicate whether the cited reference discusses optimal or approximate (in terms of bounds) solutions In general, the presence of an X in a column means that the reference in the corresponding row bases its discussion on that categorization feature. References that are too general or that do not refer to specific scheduling results are omitted.

REFERENCES

[1] ADAM, T. L., CHANDY, K M., AND DICKSON, J. R. "A comparison of list schedules for parallel processing systems," *Comm. ACM* 17, 12

(Dec. 1974), 685-690.

[2] BAER, J. L. "A survey of some theoretical aspects of multiprocessing," *Computing Surveys* 5, 1 (March 1973), 31-80.

[3] BRUNO, J.; COFFMAN, E. G., JR.; AND SETHI, R. "Scheduling independent tasks to reduce mean finishing time," *Comm. ACM* 17, 7 (July 1974), 382-387.

[4] BUSSELL, B.; FERNANDEZ, E.; AND LEVY, O. "Optimal scheduling for homogeneous multiprocessors," in *Proc. IFIP Congress 74*, North-Holland Publ. Co., Amsterdam, American Elsevier, N.Y., 1974, 286-290.

[5] BUTEN, R. E.; AND SHEN, V. Y "A scheduling model for computer systems with two classes of processors," in *Proc. 1973 Sagamore Computer Conf, on Parallel Processing*, Springer-Verlag, N.Y., 1973, 130-138.

[6] CHANDY, K. M.; AND DICKSON, J. R. "Scheduling unidentical processors in a stochastic environment," in *Proc. IEEE COMPCON 1972*, IEEE, N Y , 1972, 171-174.

[7] CHEN, N F.; AND LIU, C. L "On a class of scheduling algorithms for multiprocessor computing systems," in *Proc. 1974 Sagamore Computer Conf. on Parallel Processing*, Springer-Verlag, N.Y., 1974, 1-16

[8] CLARK, D. *Scheduling independent tasks on non-identical parallel machines to minimize mean flow-time*, Dept. of Computer Science, Carnegie-Mellon Univ., Pittsburgh, Pa , June 1974.

[9] COFFMAN, E. G., JR.; AND DENNING, P. J. *Operating systems theory*, Prentice-Hall Inc., Englewood Cliffs, N.J., 1973.

[10] COFFMAN, E G., JR.; AND GRAHAM, R. L "Optimal scheduling for two processor systems," *Acta Informatica* 1 (1972), 200-213

[11] CONWAY, R. W.; MAXWELL, W. L.; AND MILLER, L. W. *Theory of scheduling*, Addison-Wesley Publ. Co. Inc , Reading, Mass., 1967.

[12] FERNANDEZ, E B.; AND BUSSEL, B. "Bounds on the number of processors and time for multiprocessor optimal schedule," *IEEE Trans. Comp* C22, 8 (August 1973), 745-751

[13] FINEBERG, M S ; AND SERLIN, O "Multiprogramming for hybrid computation," in *Proc AFIPS 1967 Fall Jt Computer Conf*, Thompson Book Co., Washington, D C., 1967, 1-13.

[14] FUJII, M.; KASAMI, T.; AND NINOMIYA, K. "Optimal sequencing of two equivalent processors," *SIAM J Appl Math.* 17, 4 (July 1969), 784-789.

[15] —— Erratum *SIAM J. Appl Math.* 20, 1 (Jan. 1971), 141.

[16] GAREY, M. R.; AND GRAHAM, R. L "Bounds on scheduling with limited resources," *Fourth Symp Operating System Principles*, 1973 104-111. (Published as *Operating Systems Rev.* 7, 4, ACM, N.Y.).

[17] GRAHAM, R. L. "Bounds for certain multiprocessing anomalies," *Bell Syst. Tech. J.* 45 (1966), 1563-1581.

[18] GRAHAM, R. L "Bounds on certain multiprocessing anomalies," *SIAM J Appl. Math* 17, 2 (March 1969), 416-429.

[19] GRAHAM, R L "Bounds on multiprocessing anomalies and packing algorithms," in *Proc AFIPS 1972 Spring Jt. Computer Conf.*, AFIPS Press, Montvale, N.J., 1972, 205-217.

[20] HU, T. C. "Parallel sequencing and assembly line problems," *Operations Research* 9, 6 (1961), 841-848.

[21] JOHNSON, S. M. "Optimal two- and three-stage production schedules with setup times included," *Nav. Res. Log. Quart.* 1, 1 (March 1954).

[22] KAFURA, D. G ; AND SHEN, V. Y. "Scheduling independent processors with different storage capacities," in *Proc ACM National Conf*, 1974, Vol. 1, ACM, N.Y., 1974, 161-166.

[23] KAUFMAN, M. T. "An almost-optimal algorithm for the assembly line scheduling line scheduling problem," *IEEE Trans. Comp.*C-23, 11 (Nov. 1974), 1169-1174.

[24] KRONE, M. "Heuristic programming applied to scheduling models," in *Proc Fifth Annual Princeton Conf. Mathematical Programming*, Princeton Univ. Press, Princeton, N.J., 1971, 193-195

[25] LIU, C L.; AND LAYLAND, J. W. "Scheduling algorithms for multiprogramming in a hard-real-time environment," *J. ACM* 20, 1 (Jan 1973), 46-61

[26] LIU, J. W. S.; AND YANG, A. T. "Optimal scheduling of independent tasks on heterogeneous computing systems," in *Proc. ACM National Conf.* 1974, Vol. 1, ACM, N Y., 1974, 38-45.

[27] MANACHER, G. K. "Production and stabilization of real-time task schedules," *J ACM* 14, 3 (July 1967), 439-465.

[28] MCNAUGHTON, R. "Scheduling with deadlines and loss functions," *Management Science* 6, 1 (Oct. 1969), 1-12.

[29] MUNTZ, R. R.; AND COFFMAN, E. G. JR. "Optimal preemptive scheduling on two-processor systems," *IEEE Trans Comp* C-18, 11 (Nov. 1969), 1014-1020.

[30] MUNTZ, R. R ; AND COFFMAN, E. G. JR , "Preemptive scheduling of real-time tasks on multiprocessor systems," *J. ACM* 17, 2 (April 1970), 324-338.

[31] RAMAMOORTHY, C V.; AND GONZALEZ, M J. "A survey of techniques for recognizing parallel processable streams in computer programs," in *Proc AFIPS 1969 Fall Jt. Computer Conf.*, AFIPS Press, Montvale, N.J., 1969, 1-15.

[32] RAMAMOORTHY, C. V ; CHANDY, K M.; AND GONZALEZ, M. J. "Optimal scheduling strategies in a multiprocessor system," *IEEE Trans. Comp.* C-21, 2 (Feb. 1972), 137-146.

[33] REDDI, S. S.; AND RAMAMOORTHY, C. V. "Some aspects of flow-shop sequencing problem," in *Proc. Sixth Annual Princeton Conf Mathematical Programming*, Princeton Univ. Press, Princeton, N.J., 1972, 650-654.

[34] REDDI, S S., AND FEUSTEL, E. "Analytic and implementation considerations of two-facility sequencing in computer systems," in *Proc. 1974 Sagamore Computer Conf on Parallel Processing*, Springer-Verlag, N.Y., 1974, 205-206.

[35] RICHARDS, P. *Timing properties of multiprocessor systems*, Tech. Paper, Rep. No. TD-B60-

27, Technical Operations, Inc., Burlington, Mass. August 1960.

[36] SERLIN, O. "Scheduling of time critical processes," in *Proc. AFIPS 1972 Spring Jt Computer Conf.*, AFIPS Press, Montvale, N.J., 1972, 925-932.

[37] SHEN, V. Y.; AND CHEN, Y. E. "A scheduling strategy for the flow-shop problem in a system with two classes of processors," in *Proc Sixth Annual Princeton Conf. Mathematical Programming*, Princeton Univ. Press, Princeton, N.J., 1972, 645-649.

[38] SOH J. W. "Scheduling strategies for periodic jobs in a multiprocessor environment," PhD Dissertation, Computer Sciences Dept., Northwestern Univ., Evanston, Ill. August 1974.

[39] ULLMAN, J. D. "Polynomial complete scheduling problem," in *Fourth Symp. Operating System Principles*, 1973, 96-101. (Published as *Operating Systems Rev.* 7, 4, ACM, N.Y.)

[40] YAO, A. C. "Scheduling unit-time tasks with limited resources," in *Proc. 1974 Sagamore Computer Conf. on Parallel Processing*, Springer-Verlag, N Y., 1974, 17-36.

[41] GONZALEZ, M. J.; AND SOH, J W. "Periodic job scheduling in a distributed processor system," *IEEE Trans. Aerospace and Electronic Systems* AES-12, 5 (Sept. 1976), 530-536.

[42] COFFMAN, E G., JR. (Ed.), *Computer and job-shop scheduling theory*, John Wiley & Sons, N.Y., 1976

[43] REDDI, S. S.; AND RAMAMOORTHY, C. V. "On the flow-shop sequencing problem with no wait in process," *Operational Research Quarterly* 23, 3 (Sept 1972), 323-331.

[44] REDDI, S. S.; AND RAMAMOORTHY, C. V. "A scheduling problem," *Operational Research Quarterly* 24, 3 (Sept. 1973), 441-446.

[45] GILMORE, P. C.; AND GOMORY, R. E. "Sequencing a one state-variable machine: A solvable case of the traveling salesman problem," *Operations Research* (Sept-Oct 1964), 655-679.

[46] MURAOKA, Y. "Parallelism, exposure and exploitation in programs," PhD Thesis, Computer Science Dept., Univ. of Illinois, 1971.

[47] KOHLER, WALTER H. "A preliminary evaluation of the critical path method for scheduling tasks on multiprocessor systems," *IEEE Trans Comp.* (Dec. 1975), 1235-1238.

[48] LAM, SHUI; AND SETHI, R. "Analysis of a level algorithm for preemptive scheduling," in *Proc. Fifth Symp. Operating System Principles*, 1975, 178-186. (Published as *Operating Systems Rev.* 9, 5, ACM, N.Y.)

[49] COFFMAN, E. G., JR. "A survey of mathematical results in flow-time scheduling for computer systems," in *Proceedings, GI 73*, Hamburg, Springer-Verlag, N Y., 1973, 25-46.

[50] COFFMAN, E. G., JR.; AND SETHI, R. "Algorithms minimizing mean flow time: scheduling length properties," *Acta Informatica* 1 (1976), 1-14.

[51] EASTMAN, W L.; EVEN, S; AND ISAACS, I. H. "Bounds for the optimal scheduling of n jobs on m processors," *Management Science*, (Nov. 1964), 268-279.

[52] DENNING, PETER J; AND GRAHAM, G. SCOTT. "A note on sub-expression ordering in the execution of arithmetic expressions," *Comm ACM* 16, 11 (Nov. 1973), 700-702.

[53] COOK, S. A. "The complexity of theorem-proving procedures," in *Proc. 3rd ACM Symp. Theory of Computing*, 1971, ACM, N.Y., 1971, 151-158.

[54] KARP, R. M. "Reducibility among combinatorial problems," in *Complexity of computer computation*, R. E. Miller and J. W. Thatcher (Eds.), Plenum Press, N.Y., 1972, 85-104.