Appeared in: Proceedings of the 12th International Conference on Distributed Computing Systems,
Yokohama, Japan, June 1992, pages 452--459.

End-to-End Scheduling to Meet Deadlines in Distributed Systems

R. Bettati and Jane W.-S. Liu

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801

Abstract

In a distributed system or communication network tasks may need to be executed on more than one processor. For time-critical tasks, the timing constraints are typically given as end-to-end release-times and deadlines. This paper describes algorithms to schedule a class of systems where all the tasks execute on different processors in turn in the same order. This end-to-end scheduling problem is known as the flow-shop problem. We present two cases where the problem is tractable and evaluate a heuristic for the $\mathcal{NP}$-hard general case. We generalize the traditional flow-shop model in two directions. First, we present an algorithm for scheduling flow shops where tasks can be serviced more than once by some processors. Second, we describe a heuristic algorithm to schedule flow shops that consist of periodic tasks. Some considerations are made about scheduling systems with more than one flow shop.

1 Introduction

In real-time distributed systems, tasks are often decomposed into chains of subtasks. The subtasks are executed in turn on different processors. We use the terms tasks and subtasks loosely here to mean individual units of work that are allocated resources and then executed. The execution of a subtask requires the exclusive use of some resource, referred to here as a processor. In particular, a task may be a granule of computation or data transmission. Its execution may require a computer or a data link; both are modeled as processors. Each time-critical task has end-to-end timing constraints, typically given by its release time and deadline. The former is the time instant after which its first subtask can begin execution. The latter is the time instant by which its last subtask must be completed. As long as these end-to-end constraints are met, it is not important when its earlier subtasks have completed.

As an example, we consider a distributed control system containing an input computer, an input link, a computation server, an output link, and an output computer. The input computer reads all sensors and preprocesses the sensor data, which is transmitted over the input link to the computation server. The latter computes the control law and generates commands, which are transmitted over the output link to the output computer. Each task models a tracker and controller. Its release time and deadline are derived from its required response. For the purpose of studying how to schedule the task to meet its deadline in the presence of similar tasks, we decompose it into five subtasks: the first subtask is the processing of the sensor data on the input computer; the second subtask is the transmission of sensor data on the input link; and so on. These subtasks must be scheduled on the corresponding processors to meet the overall deadline. Similarly, the transmissions of real-time messages over an n-hop virtual circuit proposed in [8] can be thought of as tasks, each consisting of a chain of n subtasks. Each subtask forwards the message through one hop, modeled as a processor on which this subtask executes. The maximum allowed end-to-end delay gives us the deadline by which the nth subtask must be completed.

This paper is concerned with scheduling tasks that execute in turn on different processors and have end-to-end release time and deadline constraints. It focuses on the case where the system of tasks to be scheduled can be characterized as a *flow shop*. A flow shop models a distributed system or communication network in which tasks execute on different processors, devices, and communication links (all modeled as processors) in turn, following the same order. The systems in the last paragraph are examples of flow shops. The examples of integrated processor and I/O scheduling given by Sha et al. [15] also can be modeled as flow shops or variations of flow shops.

A distributed system may contain many classes of tasks. Tasks in each class execute on different processors in the same order, but tasks in different classes execute in different orders. Rather than scheduling tasks from all classes together, a strategy is to partition the system resources and assign them statically to task classes. The tasks in each class are then scheduled according to a scheduling algorithm suited for the class. As an example, suppose that two classes A and B share a processor. We partition the processor into two virtual processors, on which tasks in A and B execute and which have speeds F and $1 - F$ times the speed of the physical processor. We schedule the tasks in both classes by themselves. A distributed system that contains N classes of tasks and uses such a static resource partition and allocation strategy can be modeled as a system containing N flow shops. In Section 6 we will discuss how to partition resources among different classes of tasks scheduled as flow shops.

We describe here algorithms for finding schedules of tasks in flow shops to meet their end-to-end deadlines. A feasible schedule is one in which all tasks meet their deadlines. An algorithm is optimal if it always finds a feasible

schedule whenever the tasks can be scheduled to meet all deadlines.

The problem of scheduling tasks in a flow shop to meet deadlines is $\mathcal{NP}$-hard, except for a few special cases [5, 10]. In [1] we described two optimal $\mathcal{O}(nlogn)$ algorithms for scheduling tasks that have identical processing times on all processors and tasks that have identical processing times on each of the processors but have different processing times on different processors. These algorithms are used as the basis of a heuristic algorithm for scheduling tasks with arbitrary processing times. We also consider two variations of the traditional flow-shop model, called *flow shop with recurrence* and *periodic flow-shop*. In a flow shop with recurrence each task executes more than once on one or more processors. This model describes systems that do not have a dedicated processor for every function. As an example, suppose that the three computers in the control system mentioned earlier are connected by a bus, not by two dedicated links. We can model the bus as a processor and the system as a flow shop with recurrence. Each task executes first on the input computer, then on the bus, on the computation server, on the bus again, and finally on the output computer. In a periodic flow shop, each task is a periodic sequence of requests for the same computation. Hence, a sequence of requests for a periodic computation in a traditional flow shop is represented as a single task in the periodic flow shop. The periodic flow shop can be viewed as a special case of the flow shop with recurrence, as well as a generalization of the traditional flow shop.

Past efforts in flow-shop scheduling have focused on the minimization of completion time [3, 6, 10]. Johnson [3] showed that tasks in a two-processor flow shop can be scheduled to minimize completion time in $\mathcal{O}(n \log n)$ time. Beyond the two-processor flow shop, however, almost every flow-shop scheduling problem is $\mathcal{NP}$-complete. For example, the general problem of minimizing completion time on three processors is strictly $\mathcal{NP}$-hard [6]. Consequently, many studies of flow-shop problems were concerned with restrictions of the problem that make it tractable, or focused on enumerative methods and heuristic algorithms. Flow-shop scheduling is similar to scheduling in pipelined multiprocessors. Several algorithms for scheduling pipelines to maximize throughput are described in [9, 14]. The general problem of scheduling to meet deadlines on identical multiprocessor systems is also $\mathcal{NP}$-hard [5, 10]. However, polynomial algorithms for optimally scheduling tasks with identical processing times on one or two processors exist [4, 7]. Our algorithms make use of one of them.

We review the traditional flow-shop model in Section 2. The flow shop with recurrence and the periodic flow-shop model are described. Section 3 summarizes our earlier results on optimally scheduling homogeneous tasks with identical processing times on each processor on traditional flow shops [1] and an extension to scheduling such tasks on flow shops with recurrence. In Section 4 we generalize this approach to schedule tasks where the processing times are identical on any processor, but may vary between processors. We describe a heuristic to schedule tasks with arbitrary processing times. Simulation results are given to measure the performance of the heuristic. In Section 5 we consider the periodic flow-shop model. We give a summary in Section 6 and discuss how the algorithms presented here can be used to schedule task systems that consist of subsystems of flow shops.

2 The Model

In a traditional flow shop, there are m different processors $P_1, P_2, \ldots, P_m$ and a set $\mathcal{T}$ of n tasks $T_1, T_2, \ldots, T_n$ that are executed on the processors. Specifically, each task T_i consists of m subtasks $T_{i1}, T_{i2}, \ldots, T_{im}$, that have to be executed in order; first T_{i1} on P_1, then T_{i2} on processor P_2, and so on. Every task passes through the processors in the same order. Let τ_{ij} denote the time required for the subtask T_{ij} to complete its execution on processor P_j. τ_{ij} is referred to as the *processing time* of T_{ij}. Let τ_i denote the sum of the processing times of all the subtasks of the task T_i, called the *total processing time* of T_i. Each task T_i is ready for execution at or after its *release time* r_i and must be completed by its *deadline* d_i. We will occasionally refer to the totality of release times, deadlines, and processing times as the *task parameters*. The task parameters are rational numbers unless it is stated otherwise.

Our algorithms make use of the *effective deadlines* of subtasks. The effective deadline d_{ij} for the subtask T_{ij} is the point in time by which the execution of T_{ij} must be completed to allow the later subtasks, and the task T_i, to complete by the deadline d_i; therefore $d_{ij} = d_i - \sum_{k=j+1}^{m} \tau_{ik}$. Similarly, we define the *effective release time* r_{ij} of T_{ij} to be the earliest point in time at which the subtask can be scheduled. Since T_{ij} cannot be scheduled until earlier subtasks are completed, $r_{ij} = r_i + \sum_{k=1}^{j-1} \tau_{ik}$.

With arbitrary task parameters, the flow-shop problem is $\mathcal{NP}$-hard, even where preemption is allowed [1, 2]. In two special cases of task sets, the scheduling problem becomes tractable. In the first case the processing times τ_{ij} of all subtasks are equal to a unit τ on all processors. We call these task sets *identical-length task sets*. In the second case the processing times τ_{ij} of all subtasks are identical for any one processor, but may vary between different processors. In other words, all the subtasks T_{ip} on a given processor P_p have the same processing time τ_p, but subtasks T_{ip} and T_{iq} on processors P_p and P_q have different processing times, that is, $\tau_p \neq \tau_q$ when $p \neq q$. We call task sets of this sort *homogeneous task sets*.

In the more general flow-shop-with-recurrence model, each task T_i has k subtasks, and $k > m$. Without loss of generality, we let the subtasks be executed in the order $T_{i1}, T_{i2}, \ldots, T_{ik}$ for all tasks T_i, that is, T_{i1} followed by T_{i2}, followed by T_{i3}, and so on. We characterize the order in which the subtasks execute and the processors on which they execute by a sequence $V = (v_1, v_2, \ldots, v_k)$ of integers, where v_j is one of the integers in the set $\{1, 2, \ldots, m\}$. v_j being l means that the subtasks T_{ij} are executed on processor P_l. For example, suppose that we have a set of tasks each of which has 5 subtasks, and they are to be executed on 4 processors. The sequence $V = (1, 2, 3, 2, 4)$ means that all tasks first execute on P_1, then on P_2, P_3, again on P_2, and then P_4, in this order. We call this sequence the *visit sequence* of the tasks. If an integer l

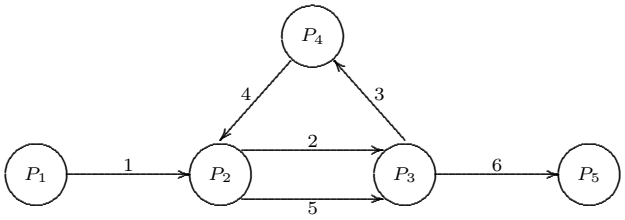


Figure 1: Visit graph for visit sequence $V = (1, 2, 3, 4, 2, 3, 5)$.

appears more than once in the visit sequence, the corresponding processor P_l is a *reused processor*. In this example P_2 is reused, and each task visits it twice. This flow shop with recurrence models the distributed control system of Section 1 with P_2 modeling the bus that is used both as input and as output link. The traditional flow-shop model is therefore a special case of the flow-shop-with-recurrence model with visit sequence $(1, 2, \ldots, m)$. Any visit sequence can be represented by a directed graph $\mathcal{G}$, called a *visit graph*, whose set of nodes P_i represents the processors in the system. There is a directed edge e_{ij} from P_i to P_j with label a if and only if in the visit sequence $V = (v_1, v_2, \ldots, v_a, v_{a+1}, \ldots, v_k)$ $v_a = i$ and $v_{a+1} = j$. A visit sequence can be represented as a path with increasing edge labels in the visit graph. An example of a visit graph is shown in Figure 1. We confine our attention here to a class of visit sequences that contain simple recurrence patterns: some sub-sequence of the visit sequence containing reused processors appears more than once. We call this kind of recurrence pattern a *loop* in the visit sequence. The notion of loops becomes intuitively clear when we look at the visit graph. In Figure 1, the sub-sequence $(2, 3)$ occurs twice and therefore makes the sequence $(4, 2, 3)$ into a loop. P_2 and P_3 are reused processors. The *length* of a loop is the number of nodes in the visit graph that are on the cycle. The loop in Figure 1 has length 3. Loops can be used to model systems where a specific processor (or a sequence of processors) is used before and after a certain service is attained. An example is a database that is queried before and updated after a specific operation.

The periodic flow-shop model is a generalization of both the traditional flow-shop model and the traditional periodic-job model [11, 13]. As in the traditional periodic-job model, the periodic *job system* $\mathcal{J}$ to be scheduled in a flow shop consists of n independent periodic jobs; each job consists of a periodic sequence of requests for the same computation. In our previous terms, each request is a *task*. The *period* p_i of a job J_i in $\mathcal{J}$ is the time interval between the ready times of two consecutive tasks in the job. The deadline of the task in every period is the ready time of the task in the next period. In an m-processor flow shop, each task consists of m subtasks that are to be executed on the m processors in turn following the same order. The processing time of the subtask on processor P_j of each task in the job J_i is τ_{ij}.

3 Scheduling Identical-Length Task Sets

In this section we describe an extension of an optimal algorithm for scheduling identical-length task sets on traditional flow shops [1]. The extension is optimal when used to schedule identical-length tasks on flow-shops with simple recurrence patterns.

In many systems, tasks are designed to have regular structures. In the simplest case, all the processing times τ_{ij} are identical for all i and j, that is, the task set is of identical length. As an example, suppose that we are to schedule a set of identical communication requests over a n-hop virtual circuit which is assigned the same bandwidth on all the links.

When release times and deadlines are multiples of τ, we can simply use the classical *earliest-effective-deadline-first* (EEDF) algorithm to optimally schedule all tasks [3]. Again, an algorithm is optimal if it always produces a feasible schedule whenever such schedule exists. In the EEDF algorithm, the subtasks T_{ij} on each processor P_j are scheduled nonpreemptively in a *priority-driven* manner. An algorithm is priority-driven if it never leaves any processor idle when there are tasks ready to be executed. The EEDF algorithm assigns priorities to subtasks on P_j according to their effective deadlines: the earlier the effective deadline, the higher the priority. The scheduling decision is made on P_1 whenever it is free; the subtask with the highest priority among all ready subtasks is executed until completion. This scheduling decision is propagated on to the subsequent processors; whenever T_{ij} completes on P_j, $T_{i(j+1)}$ starts on P_{j+1}.

The scheduling decision is more complicated if release times and deadlines are arbitrary rational numbers, that is, not multiples of τ. Garey et al. [7] introduce the concept of *forbidden regions* during which tasks are not allowed to start execution. The release times of selected tasks are postponed to insert the necessary idle times to make an EEDF schedule optimal. We call the release times generated from the effective release times by this algorithm the *modified release times*. By release times, we mean modified release times. By the EEDF algorithm, we mean the EEDF algorithm using the modified release times as input parameters rather than the effective release times. We proved in [1] that the EEDF algorithm is optimal for nonpreemptive flow-shop scheduling of identical-length task sets with arbitrary release times and deadlines.

We now extend the EEDF algorithm to optimally schedule simple task sets on flow shops with recurrence. We focus our attention on the simple case where (1) all tasks have identical release times, arbitrary deadlines, and have identical processing times τ on all the processors, and (2) the visit sequence contains one loop. We show that a modified version of the EEDF algorithm, called *Algorithm* $\mathcal{R}$, is optimal for scheduling tasks to meet deadlines.

The key strategy used in Algorithm $\mathcal{R}$ is based on the following observation. If a loop in the visit graph has length q, the second visit of every task to a reused processor in this loop cannot be scheduled before $(q-1)\tau$ time units after the termination of its first visit to the processor. Let P_{v_l} be the first processor in the loop of length q. Let $\{T_{il}\}$ and $\{T_{i(l+q)}\}$ be the sets of subtasks that are executed on P_{v_l}. T_{il} is the subtask at the first and $T_{i(l+q)}$ is the subtask at the second visit of T_i to the processor. While T_{il} is ready for execution after its release

Algorithm $\mathcal{R}$:

Input: Task parameters r_{ij}, d_{ij}, τ, of $\mathcal{T}$ and the visit graph $\mathcal{G}$. P_{v_l} is the first processor in the single loop of length q in $\mathcal{G}$.

Output: A feasible schedule $\mathcal{S}$ or the conclusion that the tasks in $\mathcal{T}$ cannot be feasibly scheduled.

Step 1: Schedule the subtasks in $\{T_{il}\} \cup \{T_{i(l+q)}\}$ on the processor P_{v_l} using the modified EEDF algorithm: whenever P_{v_l} becomes idle and one or more subtasks are ready for execution, start to execute the one with the earliest effective deadline. When a subtask T_{il} (that is, the first visit of T_i to the processor) is scheduled to start at time t_{il}, set the effective release time of $T_{i(l+q)}$ to $t_{il} + q\tau$.

Step 2: Let t_{il} and $t_{i(l+q)}$ to be the start times of T_{il} and $T_{i(l+q)}$ in the partial schedule $\mathcal{S}_R$ produced in Step 1. Propagate the schedule to the rest of the processors according to the following rules:

1. If $j < l$, schedule T_{ij} at time $t_{il} - (l - j)\tau$.
2. If $l < j \leq l + q$, schedule T_{ij} at time $t_{il} + (j - l)\tau$.
3. If $l + q < j \leq k$, schedule T_{ij} at time $t_{i(l+q)} + (j - l - q)\tau$.

Figure 2: Algorithm $\mathcal{R}$.

Tasks	T_1	T_2	T_3	T_4
d_i	8	9	10	12

Table 1: Unit-length task set with identical release times.

time, $T_{i(l+q)}$ is ready after its release time and $(q-1)\tau$ time units after the completion of T_{il}. Algorithm $\mathcal{R}$ is described in Figure 2.

Step 1 differs from the EEDF algorithm. The scheduling decision is made on a reused processor P_{v_l}, the first processor in the loop in the visit graph. More importantly, the effective release times of the second visits are postponed whenever necessary as the first visits are scheduled. Because of this change of the effective release times of later subtasks after earlier subtasks are scheduled, the optimality of Algorithm $\mathcal{R}$ does no longer obviously follow from the optimality of the EEDF Algorithm. In [2] we used a schedule-transformation argument to prove that for nonpreemptive scheduling of tasks in a flow shop with recurrence, Algorithm $\mathcal{R}$ is optimal, for tasks with identical processing times and release times, arbitrary deadlines, and a visit sequence that can be characterized by a visit graph containing a single loop.

The example given in Table 1 and Figure 3 illustrates this algorithm. The visit sequence is shown in Figure 1. The scheduling decision is made on P_2, the first reused processor in the loop.

4 Scheduling Arbitrary Task Sets

The assumption that the task sets must have identical processing times is restrictive. Taking a step toward the removal of this assumption, we considered in [1] a class of flow shops called flow shops with *homogeneous* task sets, where the subtasks T_{ij} have identical processing times τ_j

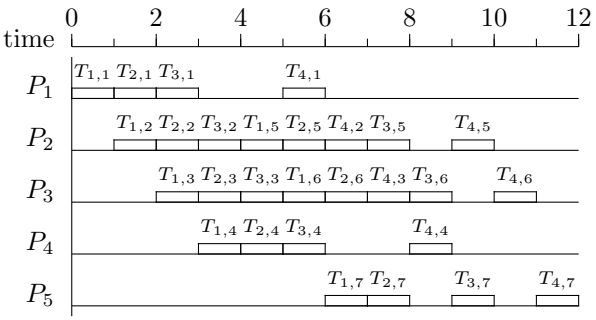


Figure 3: Schedule generated by Algorithm $\mathcal{R}$.

Algorithm $\mathcal{A}$

Input: Task parameters r_{ij}, d_{ij} and τ_j of $\mathcal{T}$.

Output: A feasible schedule $\mathcal{S}$ of $\mathcal{T}$ or the conclusion that feasible schedules of $\mathcal{T}$ do not exist.

Step 1: Determine the processor P_b where $\tau_b \geq \tau_j$ for all $j = 1, 2, \ldots, m$. If there are two or more such processors, choose one arbitrarily. P_b is the *bottleneck processor*.

Step 2: Schedule the subtasks on P_b according to the EEDF algorithm. If the resultant schedule $\mathcal{S}_b$ is not a feasible schedule, stop; no feasible schedule exists. Otherwise, if $\mathcal{S}_b$ is a feasible schedule of $\{T_{ib}\}$, let t_{ib} be the start time of T_{ib} in $\mathcal{S}_b$; do Step 3.

Step 3: Propagate the schedule $\mathcal{S}_b$ onto the remaining processors as follows: Schedule $T_{i(b+1)}$ on P_{b+1} immediately after T_{ib} completes, $T_{i(b+2)}$ on P_{b+2} immediately after $T_{i(b+1)}$ completes, and so on until T_{im} is scheduled. For $r < b$, we schedule T_{ir} on P_r so that its execution starts at time $t_{ib} - \sum_{s=r}^{b-1} \tau_s$, for $r = 1, 2, \ldots, b-1$.

Figure 4: Algorithm $\mathcal{A}$.

on processor P_j, but τ_j and τ_h are different in general. We will use this algorithm as a starting point of a heuristic algorithm for scheduling tasks with arbitrary parameters. An example of flow shops with homogeneous tasks arises in scheduling a set of identical communication requests over a n-hop virtual circuit when the bandwidth is not the same for all links. We can use *Algorithm* $\mathcal{A}$, described in Figure 4, to schedule such a set of tasks.

The optimality of Algorithm $\mathcal{A}$ for nonpreemptive flowshop scheduling of homogeneous task sets that have arbitrary release times and deadlines was proven in [1].

An example illustrating Algorithm $\mathcal{A}$ is shown in Table 2 and Figure 5. The bottleneck processor is the one on which tasks have the longest processing times. In this example, it is P_3, or $b = 3$. The schedule $\mathcal{S}$ generated by Algorithm $\mathcal{A}$ is not an EEDF schedule. It is not priority-driven; processors $P_1, P_2, \ldots, P_{b-1}$ sometimes idle when there are subtasks ready to be executed. These intervals of idle time can easily be eliminated, however.

In real-world applications the processing times of tasks on individual processors are usually not the same. However, it is often possible to distinguish processors with longer processing times from processors with shorter processing times. We can take advantage of this character-

Tasks	r_i	d_i	τ_{i1}	τ_{i2}	τ_{i3}	τ_{i4}
T_1	1	10	3	1	4	2
T_2	1	13	3	1	4	2
T_3	5	30	3	1	4	2
T_4	14	26	3	1	4	2

Table 2: An example of a homogeneous task set.

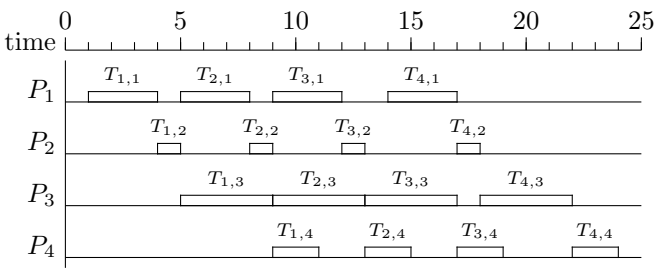


Figure 5: Schedule generated by Algorithm $\mathcal{A}$.

istics in scheduling tasks with arbitrary task parameters. Specifically, the heuristic algorithm described in Figure 6, called *Algorithm* $\mathcal{H}$, transforms an arbitrary task set into a homogeneous task set and uses the latter as a starting point. It first uses Algorithm $\mathcal{A}$ to construct a schedule of the resultant homogeneous task set. Then it tries to improve the schedule produced by Algorithm $\mathcal{A}$ in its attempt to construct a feasible schedule $\mathcal{S}$ for the original task set.

Algorithm $\mathcal{H}$ is relatively simple, with complexity $\mathcal{O}(nlogn+nm)$. By using Algorithm $\mathcal{A}$, Step 4 defines the order in which the tasks are executed on the processors. In the schedules produced by Algorithm $\mathcal{H}$ the subtasks on different processors are scheduled in the same order. Such schedules are called *permutation schedules*.

Algorithm $\mathcal{H}$ is not optimal for 2 reasons. Inflating processing times of the subtasks to generate the homogeneous task set $\mathcal{T}_{inf}$ increases the workload that is to be scheduled on the processors. This increases the number of release-time and deadline constraints that are not met. One way to reduce this bad effect is to add a compaction step that reduces the idle times introduced in Step 3 of Algorithm $\mathcal{H}$. Let t_{ij} be the start time of T_{ij} on P_j. We note again that $\mathcal{S}$ is a permutation schedule. Let the tasks be indexed so that T_{1j} starts before T_{2j}, T_{2j} before T_{3j}, and so on, for all j. The subtask T_{ij} starts execution on P_j at time $t_{ij} = t_{ib} + \sum_{k=b}^{j-1} \tau_{max,k}$ for $j > b$ and $t_{ij} = t_{ib} - \sum_{k=j+1}^{b-1} \tau_{max,k} - \tau_{ij}$ for $j < b$. However, we can start T_{ij} after its effective release time as soon as $T_{(i-1)j}$ terminates and frees P_j. These considerations are taken into account in *Algorithm* $\mathcal{C}$, which is described in Figure 7. This algorithm compacts the schedule $\mathcal{S}$ generated in Step 4 of Algorithm $\mathcal{H}$.

The example in Table 3 and Figure 8 illustrates Algorithm $\mathcal{H}$. T_3 has the longest processing time on P_1, T_1 on P_2, T_4 on P_3, and T_3 on P_4. Therefore, $\tau_{max,1} = \tau_{31}$, $\tau_{max,2} = \tau_{12}$, and so on. Algorithm $\mathcal{A}$ in Step 4 uses P_3 as the bottleneck processor. Figure 8a and Figure 8b show the schedule before and after the compaction in Step 5. T_1 and T_5 miss their deadlines in Figure 8a. Moreover, T_1 is

Algorithm $\mathcal{H}$

Input: Task parameters r_i, d_i and τ_{ij} of $\mathcal{T}$.

Output: A feasible schedule of $\mathcal{T}$, or the conclusion that feasible schedules of $\mathcal{T}$ do not exist.

Step 1: Determine the effective release times r_{ij} and effective deadlines d_{ij} of all subtasks.

Step 2: On each processor P_j, determine the subtask $T_{max,j}$ with the longest processing time $\tau_{max,j}$ among all subtasks T_{ij} on P_j.

Step 3: On each processor P_j, inflate all the subtasks by making their processing times equal to $\tau_{max,j}$. In other words, each inflated subtask T_{ij} consists of a busy segment of length τ_{ij} and an idle segment of length $\tau_{max,j} - \tau_{ij}$. The inflated subtasks form a homogeneous task set $\mathcal{T}_{inf}$.

Step 4: Schedule the inflated task set $\mathcal{T}_{inf}$ using Algorithm $\mathcal{A}$.

Step 5: Use Algorithm $\mathcal{C}$ to compact the schedule. Stop.

Figure 6: Algorithm $\mathcal{H}$.

Algorithm $\mathcal{C}$:

Input: A schedule $\mathcal{S}$ generated in Step 4 of Algorithm $\mathcal{H}$.

Output: A compacted schedule with reduced idle time.

Step 1: Set $\tilde{r}_{ij} = r_{ij}$ for all i and j.

Step 2: Perform the following steps:

$t_{11} = max(t_{11}, \tilde{r}_{11})$
for $j = 2$ **to** m **do**
  $t_{1j} = t_{1(j-1)} + \tau_{1(j-1)}$
endfor
for $i = 2$ **to** n **do**
  for $j = 1$ **to** $m - 1$ **do**
    $t_{ij} = max(t_{(i-1)j} + \tau_{(i-1)j}, \tilde{r}_{ij})$
    $\tilde{r}_{i(j+1)} = t_{ij} + \tau_{ij}$
  endfor
  $t_{im} = max(t_{(i-1)m} + \tau_{(i-1)m}, \tilde{r}_{im})$
endfor

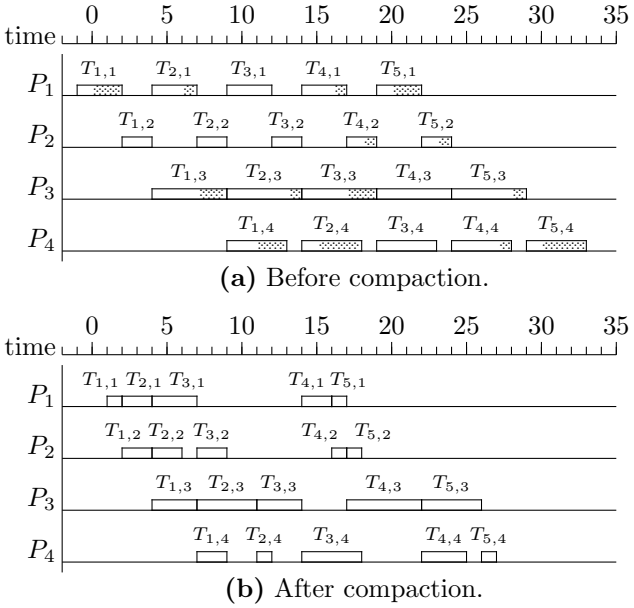
Figure 7: Algorithm $\mathcal{C}$.

forced to start before its release time. All tasks meet their release time and deadline in Figure 8b.

The second reason for Algorithm $\mathcal{H}$ being suboptimal arises because it considers only permutation schedules. In flow shops with more than two processors it is possible that the order of execution of subtasks may vary from processor to processor in all feasible schedules. Algorithm $\mathcal{H}$ fails to find a feasible schedule for such cases. Even when feasible permutation schedules exist, Algorithm $\mathcal{H}$ can fail because Step 4 may generate a wrong execution order of subtasks on the bottleneck processor P_b. This can be caused by the wrong choice of the bottleneck processor in Algorithm $\mathcal{A}$. Even when the choice of the bottleneck processor was correct, Step 2 in Algorithm $\mathcal{A}$ is not optimal for scheduling the original set of subtasks T_{ib} on P_b. Algorithm $\mathcal{H}$ can therefore fail to generate any existing feasible schedule on P_b.

Tasks	r_i	d_i	τ_{i1}	τ_{i2}	τ_{i3}	τ_{i4}
T_1	1	10	1	**2**	3	2
T_2	1	16	2	2	4	1
T_3	1	22	**3**	2	3	**4**
T_4	14	28	2	1	**5**	3
T_5	14	29	1	1	4	1

Table 3: Task set with arbitrary processing times.

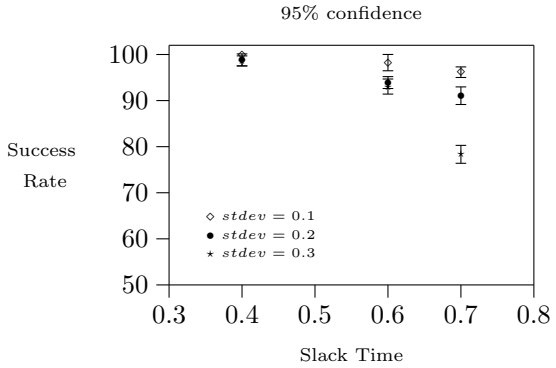


(a) Before compaction.

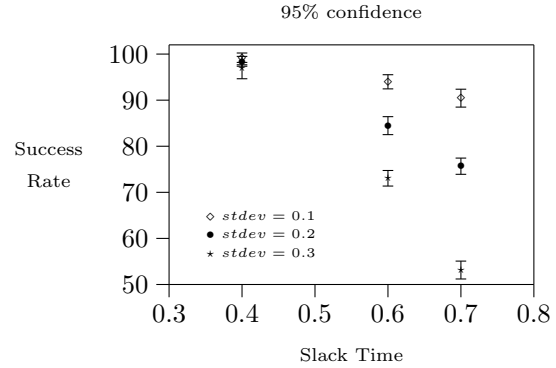
(b) After compaction.

Figure 8: A schedule produced by Algorithm $\mathcal{H}$

We are investigating the conditions under which Algorithm $\mathcal{H}$ fails to generate a feasible schedule when such a schedule exists. Figure 9 and Figure 10 show the results of simulation experiments to determine the probability for Algorithm $\mathcal{H}$ to succeed in generating a feasible schedule, given that one exists. We fed Algorithm $\mathcal{H}$ with task sets that have feasible schedules. The experiments where repeated to simulate different amounts of slack time in the task sets and different variances of processing times on a processor. We see in Figure 9 that the smaller this variance, that is, the more the task sets resemble homogeneous task sets, the better Algorithm $\mathcal{H}$ performs. This is to be expected, since Algorithm $\mathcal{H}$ bases on Algorithm $\mathcal{A}$, which is optimal for homogeneous task sets. Algorithm $\mathcal{H}$ performs worse with decreasing slack time per task. By *slack time* we mean the difference between the length of a task and the amount of time between its release time and deadline. This also was to be expected. It is inherently more difficult to schedule a task set when the slack time is scarce. The amount of slack time is in the order of 1.5 to 0.4 times the processing time of a task. Figure 10 shows the results of experiments with more tasks and larger amounts of slack time (in the order of 4 times the processing time of a task).



(a) 4 Tasks on 4 Processors.



(b) 6 Tasks on 4 Processors.

Figure 9: Performance of Algorithm $\mathcal{H}$ on small task sets

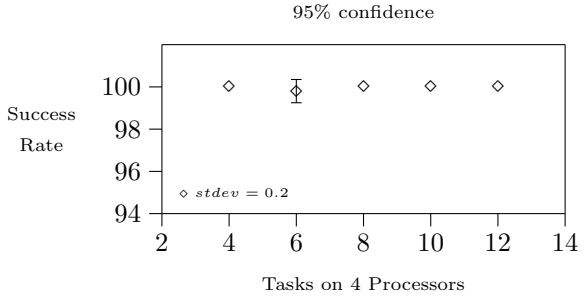


Figure 10: Algorithm $\mathcal{H}$ and larger task sets

5 Periodic Flow Shops

To explain a simple method to schedule periodic jobs in flow shops, we note that each job J_i of the n jobs in a m-processor periodic flow-shop job set can be logically divided into m subjobs J_{ij}. The period of J_{ij} is p_i. The subtasks in all periods of J_{ij} are executed on processor P_j and have processing times τ_{ij}. In other words, for a given j each of the n subjobs can be characterized by the 2-tuple (p_i, τ_{ij}). The set of m periodic subjobs $\mathcal{J}_j = J_{ij}$ is scheduled on processor P_j. The *total utilization factor* of all the subjobs in $\mathcal{J}_j$ is $u_j = \sum_{i=1}^{n} \tau_{ij}/p_i$. When it is necessary to distinguish the individual subtasks, the subtask in the kth period of subjob J_{ij} is called $T_{ij}(k)$. For a given j, the subjobs J_{ij} of different jobs J_i are independent, since the jobs are independent. On the other hand, for a given i, the subjobs J_{ij} of J_i on the different processors are not indepen-

dent since the subtask $T_{ij}(k)$ cannot begin until $T_{i(j-1)}(k)$ is completed. There are no known polynomial-time optimal algorithms that can be used to schedule dependent periodic jobs to meet deadlines, and there is no known schedulability criteria to determine whether the jobs are schedulable. Hence, it is not fruitful to view the subjobs of each job J_i on different processors as dependent subjobs. Instead, we consider all subjobs to be scheduled on all processors as independent periodic subjobs and schedule the subjobs on each processor independently from the subjobs on the other processors. We take into account the actual dependencies between subjobs of each job in the manner described below.

Let us assume that the consecutive tasks in each job are dependent. $T_{i1}(k)$ cannot begin until $T_{im}(k-1)$ is completed. Let b_i denote the *phase* of J_i, the time at which the first task $T_{ij}(1)$ becomes ready. It is also the phase b_{i1} of the subjob J_{i1} of J_i on P_1. Hence the kth period of J_{i1} begins at $b_{i1} + (k-1)p_i$. Suppose that the set $\mathcal{J}_1$ is scheduled on P_1 according to the well-known rate-monotone algorithm. This algorithm assigns priorities statically to jobs on the basis of their periods; the shorter the period of a job, the higher its priority. Suppose that the total utilization factor u_1 on P_1 is such that we can be sure that every subtask $T_{i1}(k)$ is completed by the time $\delta_1 p_i$ units after its ready time $r_{ik} = b_i + (k-1)p_i$ for some $\delta_1 < 1$. Now, we let the phase of every subjob J_{i2} of J_i on P_2 be $b_{i2} = b_{i1} + \delta_1 p_i$. By postponing the ready time of every subtask $T_{i2}(k)$ in every subjob J_{i2} on P_2 until its predecessor subtask $T_{i1}(k)$ is surely completed on P_1, we can ignore the precedence constraints between subjobs on the two processors. Any schedule produced by scheduling the subjobs on P_1 and P_2 independently in this manner is a schedule that satisfies the precedence constraints between the subjobs J_{i1} and J_{i2}. Similarly, the phase b_{i3} of J_{i3} can be postponed, and so on.

Suppose that the total utilization factors u_j for all $j = 1, 2, \ldots, m$ are such that, when the subjobs on each of the m processors are scheduled independently from the subjobs on the other processors according to the rate-monotone algorithm, all subtasks in J_{ij} complete by $\delta_j p_i$ time units after their respectively ready times, for all i and j. Moreover, suppose that $\delta_j > 0$, and $\sum_{j=1}^{m} \delta_j \leq 1$. We can postpone the phase of each subjob J_{ij} on P_j by $\delta_j p_i$ units. This generates a feasible schedule where all precedence constraints and all deadlines are met. Given the parameters of $\mathcal{J}$, we can compute the set $\{u_j\}$ and use the existing schedulability bounds given in [11, 12, 13] to determine whether there is a set of $\{\delta_j\}$ where $\delta_i > 0$ and $\sum_{j=1}^{m} \delta_j \leq 1$. The job system $\mathcal{J}$ can be feasibly scheduled in the manner described above if such a set of δ_j' exists. With a small modification of the required values of δ_j, this method can handle the case where the deadline of each task in a job with period p_i is equal to or less than mp_i units from its ready time. Similarly, this method can be used when the subjobs are scheduled using other algorithms, or even different algorithms on different processors, so long as schedulability criteria of the algorithms are known.

Table 4 shows an example of a set of periodic jobs to be scheduled on a flow shop with 2 processors. We want to know if we can guarantee the jobs to complete by the end of their period. We schedule the jobs on the processors using the rate-monotone algorithm. The total utilization factors on P_1 and P_2 for the job set in Table 4 are $u_1 = 0.4125$ and $u_2 = 0.45$, respectively. Equation (1) [12] gives the least upper bound of the total utilization; a set of jobs whose total utilization is equal to or less than $u_{max}(\delta)$ is surely schedulable by the rate-monotone algorithm to complete within δp_i units after their ready time.

Jobs	τ_{i1}	τ_{i2}	p_i
J_1	2	1	8
J_2	1	2	10
J_3	1	2	16

Table 4: Set of periodic jobs on a 2-processor flow shop.

Jobs	τ_{i1}	τ_{i2}	p_i
J_1	5	5	10
J_2	0.5	0.5	10

Table 5: Set of periodic jobs on a 2-processor flow shop.

$$u_{max}(\delta) = \begin{cases} n((2\delta)^{1/n} - 1) + (1 - \delta), & \frac{1}{2} \leq \delta \leq 1 \\ \delta & 0 \leq \delta \leq \frac{1}{2} \end{cases} \quad (1)$$

We apply this formula to the two processors, and get $\delta_1 = 0.4125$ and $\delta_2 = 0.45$. Therefore we can guarantee $T_{11}(k)$ to terminate by time $\delta_1 p_1 = 3.3$ units, $T_{21}(k)$ by time $\delta_1 p_2 = 4.125$ units, and $T_{31}(k)$ by time $\delta_1 p_3 = 6.6$ units after their release times. We postpone the phase of the jobs on P_2 by 3.3, 4.125, and 6.6 units for J_{12}, J_{22}, and J_{32} respectively. On P_2 we can guarantee $T_{12}(k)$ to terminate by time $\delta_2 p_1 = 3.6$ units,$T_{22}(k)$ by time $\delta_2 p_2 = 3.6$ units, and $T_{32}(k)$ by time $\delta_2 p_3 = 3.6$ units after their respective release times. Every invocation of J_1 is completed at or before time 6.9 units after its release time and therefore before the end of its period. Hence, it meets its deadline. The same holds for J_2 and J_3.

Table 5 shows an example described in [12] of a job set that can not always be scheduled so that both jobs meet their deadlines at the end of their period. When the two jobs have the same phase, J_1 has to be interrupted to let J_2 execute and will miss its deadline. The maximum utilization to schedule the two jobs on a two-processor flow shop dropped to 0.5 from 0.83 in the single-processor case. The maximum utilization drops as low as $1/r$ for r processors. We can achieve higher utilization bounds if we allow the deadlines of the jobs to be postponed beyond the end of the period. The total utilization factors u_1 and u_2 are both 0.55 . By solving equation (1) for δ_1, given u_1, we deduce that we can guarantee the subjobs on P_1 to complete within 0.553 times their period. We therefore postpone the phase of the subjobs J_{i2} by $0.553p_i$. A similar analysis for P_2 shows that we can guarantee all the jobs to complete within $1.106p_i$ time units. By postponing the deadlines of the jobs slightly more than 10% beyond the period, we can guarantee the job set to be schedulable.

6 Summary

We have described ways to schedule task systems in which tasks execute on different processors in a distributed system in the same order. Such a task system, called a flow shop in the deterministic models used here, is modeled as a routing chain in queuing models. Our scheduling objective is that all tasks begin execution after their ready times and complete by their deadlines. We assume that all parameters of the task system to be scheduled are known to a centralized scheduler which produces a schedule for each processor in the flow shop.

Polynomial-time algorithms for flow-shop scheduling, that never fail to produce feasible schedules, whenever feasible schedules exist, are known only for some special cases. We have described such algorithms for two special cases where the tasks have identical processing times. We have also described two heuristic algorithms for scheduling tasks with arbitrary processing times on flow shops and periodic flow shops. The complexity of these algorithms is sufficiently low that they can be used for on-line scheduling.

A typical distributed system contains many flow shops. We assume that, when two or more flow shops share a processor, the processor time is allocated to the task sets on different flow shops on a round robin basis, thus creating a virtual processor for each flow shop. Tasks in each flow shop are scheduled by one of the algorithms described here on its virtual processor independently of the tasks on the virtual processors for the other flow shops.

A problem that remains to be addressed is what fractions of (physical) processor time should be allocated to the different flow shops sharing one processor. For periodic flow shops, it is reasonable to make the fraction of processor time allocated to a task set on a flow shop proportional to the total utilization of its subtasks executed on the processor. Suppose that on processor P_j, the total utilization of the subtasks in the periodic task set $\mathcal{T}$ on a flow shop is u. The total utilization of the subtasks in the periodic task sets on all the flow shops that share P_j is U. The fraction of P_j allocated to the task set $\mathcal{T}$ is u/U. The processing time of the subtasks in $\mathcal{T}$ on this processor is increased by a factor of U/u.

For traditional flow shops, we can define the utilization of a subtask in a analogous way to be the ratio of its processing time to the length of the time interval between its ready time and deadline. This definition allow us to use the strategy described above to determine the fraction of processor time allocated to the task set on each flow shop. The performance of this strategy and other processor time allocation strategies need to be determined. This is a part of our future work on end-to-end scheduling.

Acknowledgements

This work was partially supported by the Navy ONR Contract No.NVY N00014 89-J-1181.

References

[1] R. Bettati and J. W.-S. Liu. Algorithms for end-to-end scheduling to meet deadlines. In *Proceedings of the 2nd IEEE Conference on Parallel and Distributed Systems*, Dallas, Texas, December 1990.

[2] R. Bettati and J. W.-S. Liu. Algorithms for end-to-end scheduling to meet deadlines. Technical Report UIUCDCS-R-1594, Department of Computer Science, University of Illinois, 1990.

[3] M. R. Garey and D. S. Johnson. Scheduling tasks with nonuniform deadlines on two processors. *J. Assoc. Comput. Mach.*, 23:461–467, 1976.

[4] M. R. Garey and D. S. Johnson. Two-processor scheduling with start-times and deadlines. *SIAM J. Comput.*, 6:416–426, 1977.

[5] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, New York, 1979.

[6] M. R. Garey, D. S. Johnson, and R. Sethi. The complexity of flowshop and jobshop scheduling. *Math. Oper. Res.*, 1:117–129, 1976.

[7] M. R. Garey, D. S. Johnson, B. Simons, and R. E. Tarjan. Scheduling unit-time tasks with arbitrary release times and deadlines. *SIAM J. Comput.*, 10-2:256–269, 1981.

[8] D. D. Kandlur, K. G. Shin, and D. Ferrari. Real-time communication in multi-hop networks. In *Proceedings of the 11th International Conference on Distributed Computing Systems*, May 1991.

[9] E. Lawler, J. K. Lenstra, C. Martel, B. Simons, and L. Stockmeyer. Pipeline scheduling: A survey. Technical Report RJ 5738, IBM Research Division, San Jose, CA, 1987.

[10] E. L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan, and D. B. Shmoys. Sequencing and scheduling: Algorithms and complexity. Technical report, Centre for Mathematics and Computer Science, Amsterdam, 1989.

[11] J. P. Lehoczky and L. Sha. Performance of real-time bus scheduling algorithms. *ACM Performance Evaluation Review*, 1986.

[12] J. P. Lehoczky, L. Sha, J. K. Strosnider, and H. Tokuda. Fixed priority scheduling theory for hard real-time systems. In A. M. Tilborg and G. M. Koob, editors, *Foundations of Real-Time Computing, Scheduling and Resource Management*, chapter 1. Kluwer Academic Publishers, 1991.

[13] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *J. Assoc. Comput. Mach.*, 20:46–61, 1973.

[14] K. V. Palem and B. Simons. Scheduling time-critical instructions on risc machines. In *ACM Symposium on Principles of Programming Languages*, pages 270–280, 1990.

[15] L. Sha, J. P. Lehoczky, and R. Raikumar. Solutions for some practical problems in prioritized preemptive scheduling. In *Proceedings of Real-Time Systems Symposium*, December 1986.