Annals of Operations Research 90(1999)221–246

Scheduling parallel tasks with sequential heads and tails*

Maciej Drozdowski☆ and Wiesław Kubiak
Faculty of Business Administration, Memorial University of Newfoundland, St. John's, Newfoundland, Canada
E-mail: maciej_d@sol.put.poznan.pl

This paper considers scheduling of parallel tasks in a multiprogrammed, multiprocessor system. The problem of preemptive scheduling of n tasks on m processors to minimize makespan is studied. Task j starts and finishes with sequential parts $head_j$ and $tail_j$, respectively. Between these two, j runs its parallel part $parallel_j$. The sequential parts have to be executed by one processor at a time. The parallel part can be executed by more than one processor at a time. It is shown that this problem is NP-hard in the strong sense even if there are fewer tasks than processors. A linear program is presented to find an optimal schedule for a given sequence of completion times of heads and start times of tails. If the optimal schedule for tasks longer than the mth longest task is given, an efficient, polynomial-time merging algorithm is proposed to obtain an optimal schedule for all n tasks. The algorithm builds an optimal schedule with at most $m-1$ tasks running their parallel parts on more than one processor at a time, the remaining tasks run their parallel parts as if they were *sequential*. Therefore, there always exist optimal schedules with only a few tasks exploiting the parallel processing capability of a parallel system. Finally, polynomially solvable cases are discussed, and the worst-case performance of three heuristics for the problem is analyzed.

Keywords: parallel processing, multiprocessor tasks, preemptive scheduling, complexity analysis

AMS subject classification: 68Q25, 90B35

1. Introduction

A constantly increasing demand for fast computer systems has attracted attention for years. Due to physical limitations of sequential computers, parallel computers seem to be the only known way of further increasing speed of computations. However, the advantage of parallel execution of tasks can only be obtained if it is properly scheduled. This creates a need for good scheduling algorithms in parallel computer systems. Such

* This research has been supported by the Natural Sciences and Engineering Research Council of Canada under Grant OGP0105675.
☆ On leave from Instytut Informatyki, Politechnika Poznańska, Poznań, Poland. This research has been partially supported by KBN Grant 8T11C 04012.

algorithms often use a *directed acyclic graph* (DAG) representation of tasks [5]. The DAG representation allows any two independent nodes of the graph to be scheduled in parallel. However, no node is allowed to use more than one processor at a time. Recently, a new model of scheduling in parallel computer systems, called *multiprocessor* task systems [3], has been proposed. This model allows tasks to use more than one processor at a time. Two branches have emerged from this assumption. One is where a task can be processed simultaneously on *any set* of processors [3,7,10, 14,17], the other where a *fixed set* of simultaneously required processors is specified for a task [1,2,13]. Both, however, assume that the number of processors used by a task does not change during its execution.

In this paper, we relax this assumption and allow that the number of processors executing a task change over time. We call such tasks *parallel*. These parallel tasks start and finish with sequential parts, heads and tails, respectively. The completion of a head naturally corresponds to the execution of a *fork* operation and the completion of a parallel task naturally corresponds to the execution of a *join* operation. This model is also motivated by a very common situation where unfolding application code on parallel processors and/or distribution of input data yield sequential heads. The bulk of computation is then done in parallel. Finally, the collection and consolidation of results give the sequential tail. Also in the master–slave model of parallel computations (e.g. [12]), master operations are sequential and thus constitute heads and tails, and slave operations constitute parallel parts of tasks.

We consider set $\mathcal{T}$ of n tasks scheduled on set $\mathcal{P}$ of m parallel, identical processors. Task j starts and finishes with sequential parts, $head_j$ and $tail_j$, respectively. Between these two, j runs its parallel part, $parallel_j$. The sequential parts have to be executed by one processor at a time. The parallel part can be executed by more than one processor at a time. The execution times of $head_j$ and $tail_j$ are h_j and t_j, respectively. The execution time of $parallel_j$ on a single processor is p_j, $p_j > 0$. The sequential as well as parallel parts are preemptable. More precisely, any feasible schedule is a sequence of l intervals. In any interval, the number of processors assigned to a task does not change. Thus, interval k ($k = 1,\ldots,l$) uniquely determines set Q_k of tasks executed in it, the number of processors $proc_k(j)$ occupied by task j in it, and its length L_k. In any feasible schedule, $\sum_{k=1}^{l} proc_k(j)L_k = h_j + p_j + t_j$, for $j \in \mathcal{T}$.

To illustrate the difference between sequential, multiprocessor and parallel tasks, let us consider an example: $m = 3$, $n = 2$, $h_1 = h_2 = 1$, $p_1 = 3$, $p_2 = 6$, $t_1 = 2$, and $t_2 = 0.5$. The two tasks executed sequentially require 7.5 units of time, see figure 1(a). When executed as multiprocessor tasks, their parallel parts cannot change the number of processors used. If we assume that $parallel_1$ and $parallel_2$ use all available processors, then this will result in a schedule of length 6, see figure 1(b). Finally, if $parallel_1$ and $parallel_2$ can change the number of processors used during their executions, which is the case for parallel tasks, a shorter schedule of length 5 is possible, see figure 1(c).

We focus on the problem of minimizing the makespan. Our problem will be referred to as $P|1any1, pmtn|C_{max}$. Moreover, we use notation $P|1any, pmtn|C_{max}$,

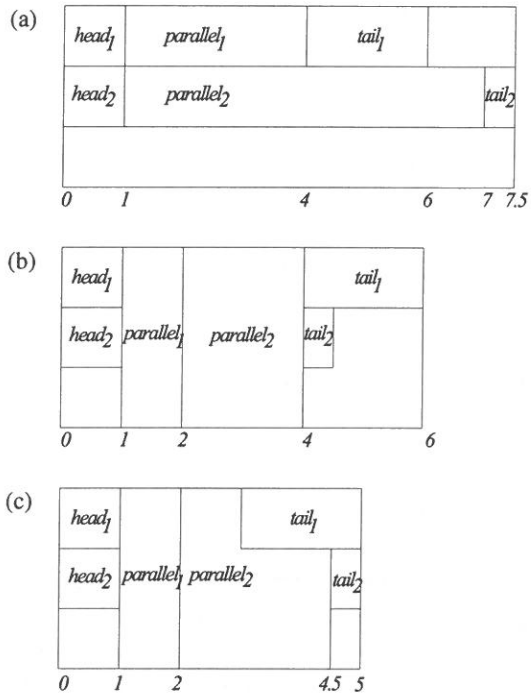

Figure 1. Example schedule: (a) sequential tasks, (b) multiprocessor tasks, (c) parallel tasks.

and $P|any1, pmtn|C_{max}$ for the problem without tails and without heads, respectively. This notation extends the one introduced in [11] (see also [19]) to classify machine scheduling problems.

Although our model allows parallel parts to use more than one processor at a time, we show that it is also rooted in earlier results obtained on scheduling the DAG model of tasks. In particular, [16] shows that level schedules are optimal for preemptive scheduling tasks with arbitrary execution times and tree-like precedence constraints. In [6] and [9], more general precedence constraints and unit-time execution tasks are considered. However, neither model applies directly to our situation. First of all, the DAG representation of parallel parts becomes known only after a schedule is complete. Secondly, the level schedules are not optimal in our model since any given DAG representation of parallel parts results in a DAG representation of tasks which is not tree-like.

The paper is organized as follows. In section 2, we present a linear program to find an optimal schedule for a given sequence of the completion times of heads and the start times of tails. In section 3, we show a case with fewer tasks than processors to be NP-hard in the strong sense. Section 4 gives an efficient, polynomial-time merging algorithm to obtain an optimal schedule for n tasks if an optimal schedule for tasks

longer than the mth longest task is given. The algorithm builds an optimal schedule with at most $m - 1$ tasks running their parallel parts on more than one processor at a time, the remaining tasks run their parallel parts as if they were *sequential*. Therefore, there always exist optimal schedules with only a few tasks exploiting the parallel processing capability of a parallel system. This seems to be of great importance for any scheduling rule used to schedule tasks in such systems. Section 5 discusses some cases solvable in polynomial time, and presents a heuristic which is optimal in probability. Finally, section 6 analyzes the worst-case performance of three heuristics for the problem, and presents results of computational experiments with these heuristics.

2. Linear program for $P|1any1, pmtn|C_{max}$

In this section, we present a polynomial-time algorithm for the case with given sequence of the completion times of heads and the start times of tails.

Denote by H_j and T_j the completion time of $head_j$ and the start time of $tail_j$, respectively. For task j, $head_j$ can only be executed by H_j, $parallel_j$ can only be executed between H_j and T_j, and $tail_j$ can only be executed after T_j. Suppose that a sequence of such events, i.e. completion and start times, is given for the tasks in $\mathcal{T}$. This sequence creates $2n + 1$ periods, some may be empty. A period determines a set of tails, heads, and parallel parts that are available for processing in it; any two different periods define two different sets.

We formulate a linear program which finds the shortest feasible schedule for a given sequence π of $s(j)$'s and $f(j)$'s, where

$s(j)$: the index of the last period where $head_j$ is allowed to appear,
$f(j)$: the index of the first period where $tail_j$ is allowed to appear, $(f(j) > s(j))$,
x_i : the length of period i,
u_{ij} : the amount of task j processed in period i.

The formulation is as follows:

LP:
$$\text{minimize } C_{max} = \sum_{i=1}^{2n+1} x_i$$

$$\text{subject to } \sum_{i=1}^{s(j)} u_{ij} = h_j, \quad j = 1, \ldots, n, \tag{1}$$

$$\sum_{i=s(j)+1}^{f(j)-1} u_{ij} = p_j, \quad j = 1, \ldots, n, \tag{2}$$

$$\sum_{i=f(j)}^{2n+1} u_{ij} = t_j, \quad j = 1, \ldots, n, \tag{3}$$

$$u_{ij} \leq x_i, \quad i = 1,\ldots,s(j), f(j),\ldots,2n+1, \; j = 1,\ldots,n, \tag{4}$$

$$\sum_{j=1}^{n} u_{ij} \leq m x_i, \quad i = 1,\ldots,2n+1, \tag{5}$$

$$u_{ij} \geq 0, \quad i = 1,\ldots 2n+1, \; j = 1,\ldots,n,$$

$$x_i \geq 0, \quad i = 1,\ldots,2n+1.$$

Constraints (1), (2) and (3) guarantee that all tasks are fully executed. Constraints (4) guarantee that no sequential part of any task exceeds the length of any period in which it is executed. Constraints (5) guarantee that the demand for processing does not exceed the capacity of any period.

LP has $2n + 1 + (2n + 1)n$ variables and $O(n^2)$ constraints. Hence, it can be formulated and solved in polynomial time. From the solution to LP, we can build a schedule by concatenating partial schedules for consecutive periods $1,\ldots,2n + 1$ built in this order. A schedule in each period can be obtained according to McNaughton's wrap-around rule [15], where each parallel part is treated as a sequential task for which the overlap of processing on different processors is allowed. This way of building schedules for periods is always feasible due to constraints (4) and (5). Thus, a solution to LP can be converted into a solution to $P|1any1, pmtn|C_{max}$ for a given sequence π. Since for each task the completion of its head must precede the starting of its tail, the number of such sequences equals $(2n)!/2^n$. Furthermore, for fixed m and $n \leq m$, problem $Pm|1any1, pmtn|C_{max}$ can be solved in polynomial time. The next section addresses the time complexity of problem $P|1any1, pmtn|C_{max}$.

3. Complexity of $P|1any1, pmtn|C_{max}$

In this section, we show that problem $P|1any1, pmtn|C_{max}$ is NP-hard in the strong sense.

Theorem 1. Problem $P|1any1, pmtn|C_{max}$ is NP-hard in the strong sense.

Proof. We consider a decision version of problem $P|1any1, pmtn|C_{max}$, where we ask if there is a feasible schedule with $C_{max} \leq y$. This decision version is in NP because it is enough to guess an optimal sequence of $s(j)$'s and $f(j)$'s and use LP to check if there exists a schedule not longer than y for the sequence. We now show that the 3-PARTITION problem [8] pseudo-polynomially transforms to $P|1any1, pmtn|C_{max} \leq y$.

3-PARTITION
Instance. Set A of $3q$ elements, a positive integer weight a_i for each $i \in A$, a positive integer bound b such that for each $i \in A$: $b/4 < a_i < b/2$, and $\sum_{i \in A} a_i = bq$.
Question. Can A be partitioned into q disjoint sets $A_1,\ldots,A_q$ such that $\sum_{i \in A_j} a_i = b$ for $j = 1,\ldots,q$?

For a given instance of 3-PARTITION, we define a set of $n = 4q$ tasks, $1,\ldots,q$, $q+1,\ldots,4q$. Tasks $1,\ldots,q$ will be called *profile-defining* tasks. Tasks $q+1,\ldots,4q$ will be called *filling* tasks. The tasks are defined as in table 1.

Table 1

Task	h_j	p_j	t_j
1	1	BB'	$q-1+(q-1)B$
j	$j+(j-1)B$	BB'	$q-j+(q-j)B$
q	$q+(q-1)B$	BB'	0
$q+1$	0	Ma_1	Qa_1
$q+j$	0	Ma_j	Qa_j
$q+3q$	0	Ma_{3q}	Qa_{3q}

In table 1, $B' = Mb$, $B = Qb$, $Q = q(3q+1)$, $M = 7q^3(3q-1)+1$.

We ask if there exists a schedule on $m = B' + q$ processors which is not longer than $y = q + Bq$. The transformation is pseudo-polynomial.

Our proof bears some resemblance to the proofs using profiles, see [6,9,18], where the profile is a function defining processors available at any point in time. In these proofs, the profile either makes a part of an instance or can readily be emulated by nonpreemptable tasks and precedence constraints between them. Our proof follows the latter; however, it is more involved since the task preemptability and relatively weak (chain) precedence constraints between the three possible parts of a task make the profile emulation (done by scheduling the profile-defining tasks) more difficult. A rough idea of the profile emulation is as follows. Our choice of M and Q makes sure that the total time in $[0, y]$ when a profile-defining job j is not executed does not exceed $\varepsilon = Bq/m$, and ε can be made as small as we want by choosing M sufficiently large. Moreover, the total duration of all nonoverlapping intervals with the tails of filling jobs is close enough to y (notice that no more than $3q$ processors can work in parallel on these tails, which are made small enough in comparison to the number of processors m by our choice of M). Thus, intuitively, any schedule of the profile-defining jobs which is not longer than y becomes "close" enough to the profile shown in figure 2(a).

Suppose that sets $A_1,\ldots,A_q$ make up the required partition of set A. Then, the schedule of the profile-defining tasks is given in figure 2(a). The filling tasks in A_j have their parallel parts executed between the parallel parts of profile-defining tasks

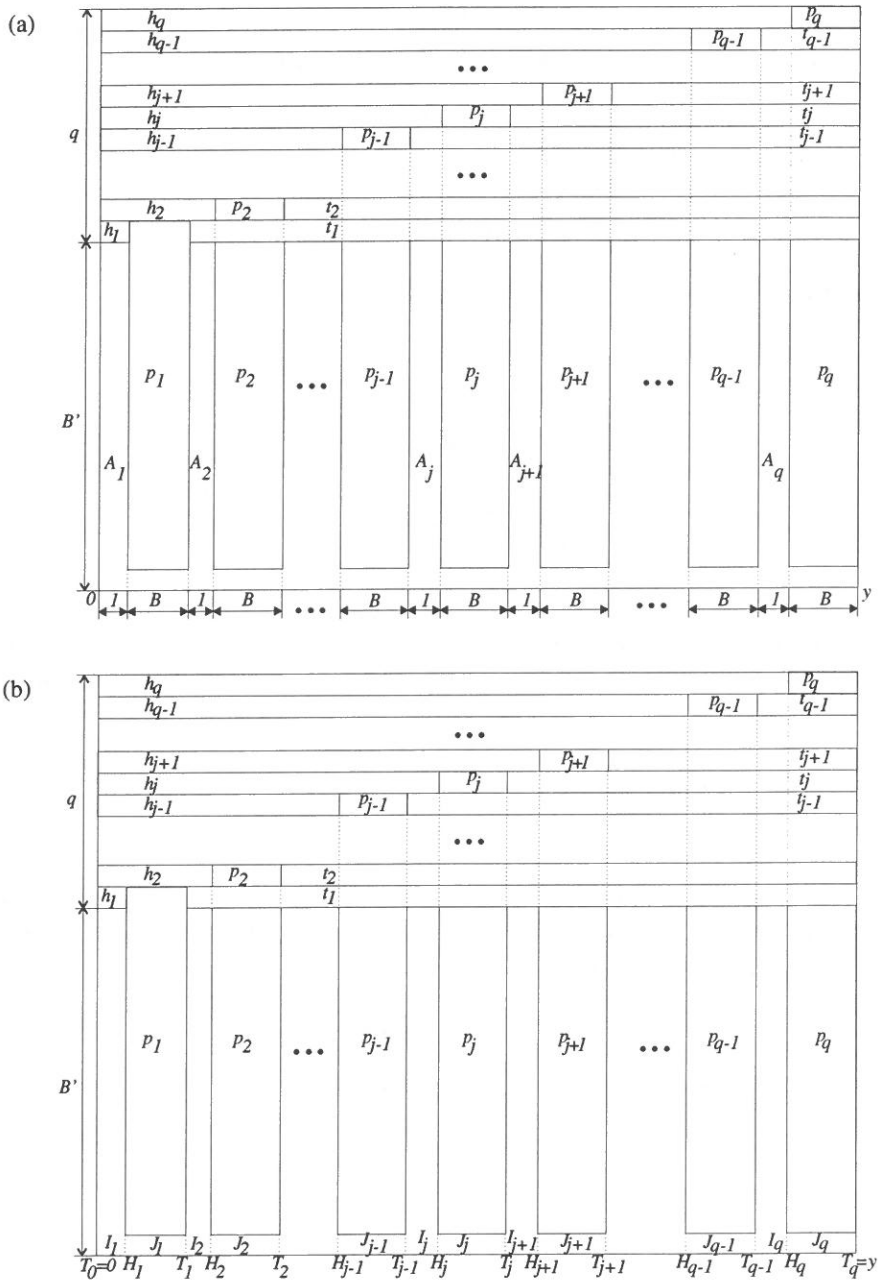

Figure 2. The proof of theorem 1.

$j-1$ and j for $j = 2,\ldots,q$. A_1 is scheduled between time 0 and the parallel part of profile-defining task 1. The tails of the tasks in A_j are executed in parallel with the parallel part of profile-defining task j for $j = 1,\ldots,q$.

Suppose that a schedule not longer than y exists. We will show that then the required partition of A exists. Let us start with some simple observations.

Observation 2. No schedule shorter than y exists. Furthermore, no schedule of makespan y allows for idle time.

Proof. Notice that the total processing time equals my. □

Observation 3. We have $\varepsilon(7q-1) < 1$ and $M > (1+2\varepsilon)3q^2$.

Proof. Notice that

$$\varepsilon(7q-1) = \frac{bq^2(3q+1)(7q-1)}{b(7q^3(3q+1)+1)+q} < 1$$

and

$$(1+2\varepsilon)3q^2 = 3q^2\left(1+\frac{2bq^2(3q+1)}{b(7q^3(3q+1)+1)+q}\right) < 3q^2(1+1) < M.$$

□

Lemma 4. Let S be a schedule with makespan y. Let H_j, T_j be the completion time of the head and the start time of the tail, respectively, for profile-defining task j (cf. figure 2(b)). Then,

$$0 \le H_j - h_j \le \varepsilon, \qquad j = 1,\ldots,q, \tag{6}$$

$$0 \le (y - t_j) - T_j \le \varepsilon, \quad j = 1,\ldots,q, \tag{7}$$

and $T_q = y$ in (7).

Proof. We prove (6) only. The proof of (7) is similar and will be omitted. Consider a profile-defining task j. Let $parallel_j$ start at s_j and complete at f_j in S. Suppose that (6) does not hold for j. Obviously, $0 \le H_j - h_j$, so $H_j - h_j > \varepsilon$. Then,

$$f_j - s_j \le y - (H_j + t_j) < y - (h_j + \varepsilon + t_j), \tag{8}$$

where $\varepsilon = Bq/m$. By the definitions of h_j and t_j, we have

$$y - (h_j + t_j) - \varepsilon = B - \varepsilon. \tag{9}$$

From (8) and (9), we have

$$f_j - s_j < B - \varepsilon = \frac{p_j}{B'} - \varepsilon. \tag{10}$$

On the other hand,

$$f_j - s_j \ge \frac{p_j}{m}. \tag{11}$$

From (10) and (11),

$$\frac{p_j}{m} < \frac{p_j}{B'} - \varepsilon$$

or

$$\varepsilon < p_j\left(\frac{1}{B'} - \frac{1}{m}\right) = p_j\frac{m - B'}{mB'} = \frac{BB'(B' + q - B')}{mB'} = \frac{Bq}{m} = \varepsilon.$$

Thus, we get a contradiction. Consequently, we must have $H_j - h_j \le \varepsilon$.

By observation 2, no idle time is allowed in $[0, y]$; moreover, at most $n - 1 < m$ tails can be finished at y. Thus, $parallel_q$ must finish at y and $T_q = y$. □

Lemma 5. Let S be a schedule with makespan y. Let H_j, T_j be the completion time of the head and the start time of the tail, respectively, of profile-defining task j. Then,

$$1 \le H_{j+1} - T_j \le 1 + 2\varepsilon \tag{12}$$

and

$$B - 2\varepsilon \le T_j - H_j \le B, \tag{13}$$

where $T_q = y$ in (13).

Proof. We begin with proving the right-hand side of (12). By lemma 4, we have

$$H_{j+1} - T_j \le h_{j+1} + \varepsilon - (y - t_j - \varepsilon) \tag{14}$$

for $j = 1,\ldots,q-1$. By the definition of h_{j+1}, t_j and from (14), we have

$$H_{j+1} - T_j \le h_{j+1} + t_j - y + 2\varepsilon = y + 1 - y + 2\varepsilon = 1 + 2\varepsilon.$$

For $j = 0$, we have $H_1 \le h_1 + \varepsilon = 1 + \varepsilon < 1 + 2\varepsilon$.

Now we prove the left-hand side of (12). By lemma 4, we have

$$H_{j+1} - T_j \ge h_{j+1} + t_j - y = y + 1 - y = 1$$

for $j = 1,\ldots,q-1$. For $j = 0$, we have $H_1 \ge 1$. This ends the proof of (12). Inequalities (13) can be obtained from lemma 4 and equation (9). □

Obviously, when ε tends to 0, then $H_j - h_j$ tends to 0 (see (6)), $(y - t_j) - T_j$ tends to 0 (see (7)), $H_{j+1} - T_j$ tends to 1 (see (12)), and $T_j - H_j$ tends to B (see (13)). From lemma 5 (see (12)), the parallel parts of the profile-defining tasks may not mix.

Lemma 6. No more than $B + \varepsilon(q-1)$ capacity in $J_j = [H_j, T_j]$ is occupied by tails of the filling tasks.

Proof. Without loss of generality, we may assume that the tail and the head of profile-defining task j are processed on processor j only. Therefore, by lemma 4, the total capacity not occupied by the heads and tails of profile-defining tasks $1,\ldots,j-1$, $j+1,\ldots,q$ on processors $1,\ldots,j-1,j+1,\ldots,q$ in J_j does not exceed $\varepsilon(q-1)$. By lemma 5, processors $j, q+1,\ldots,q+B'$ offer not more than $(B'+1)B$ capacity. Therefore, the total capacity available in J_j is at most $(B'+1)B + \varepsilon(q-1)$. The parallel part of j needs BB' of this capacity. Consequently, at most $B + \varepsilon(q-1)$ is left for the tails of the filling tasks. □

Lemma 7. No more than $(1 + 2\varepsilon)3q$ capacity in $I_j = [T_{j-1}, H_j]$ is occupied by the tails of filling tasks.

Proof. Follows from lemma 5 and the fact that each tail uses at most one processor at a time. □

Now, let us define recursively the following sets of filling tasks:

$A_q = \{j$: the parallel part of j is processed in $I_q = [T_{q-1}, H_q]\}$;

$A_{q-k} = \{j$: the parallel part of j is processed in $I_{q-k} = [T_{q-k-1}, H_{q-k}]$ and $j \notin A_{q-k-1} \cup \ldots \cup A_q\}$ for $k = 1, \ldots, q-1$.

We have the following lemma.

Lemma 8.

$$\sum_{j \in A_{q-k}} a_j = b \quad \text{for } k = 0, \ldots, q-1.$$

Proof. By induction on k. Consider $k = 0$. Assume $\sum_{j \in A_q} a_j > b$. Then $\sum_{j \in A_q} Qa_j \geq B + Q$, and the tails of tasks in A_q require at least $B + Q$ processing. By lemmas 6 and 7, the maximum capacity available for tails in intervals $I_q = [T_{q-1}, H_q]$ and $J_q = [H_q, y]$ is at most $3q(1 + 2\varepsilon)$ and $B + \varepsilon(q-1)$, respectively. Together it is equal to $B + \varepsilon(7q-1) + 3q$. By observation 3, we have $B + \varepsilon(7q-1) + 3q < B + 3q + 1$. This capacity, however, is not big enough to accommodate $B + Q$ of tails processing requirement because $Q = q(3q+1)$. Hence, $\sum_{j \in A_q} a_j \leq b$.

Assume $\sum_{j \in A_q} a_j < b$. Then, the processing requirement of parallel parts in I_q is not more than $\sum_{j \in A_q} Ma_j \leq M(b-1) = B' - M$. By lemma 5, the minimum capacity of interval I_q is B'. The demand for processing tails in I_q may not exceed $(1 + 2\varepsilon)3q$ by lemma 7, and the demand for processing parallel parts in I_q may not exceed $B' - M$. By observation 3, we have $M > (1 + 2\varepsilon)3q^2$. Thus, the total demand for processing in I_q is smaller than the available processing capacity in I_q; consequently, idle time is inevitable and a feasible schedule of makespan y does not exist.

Now, assume that the lemma holds for any l, $k - 1 \geq l \geq 0$. We prove that it holds for k. We have $\sum_{j \in A_{q-l}} a_j = b$ for $l = 0, \ldots, k-1$. Assume that $\sum_{j \in A_{q-k}} a_j > b$. Then the tails of the tasks in A_{q-k} demand at least $\sum_{j \in A_{q-k}} Qa_j \geq B + Q$. Thus, the tails in $A_{q-k} \cup \ldots \cup A_q$ demand at least $kB + B + Q$ units of processing time. By lemma 6, intervals $J_{q-k}, \ldots, J_q$ provide at most $(k+1)[B + \varepsilon(q-1)]$ processing capacity for the tails of filling tasks. By lemma 7, intervals $I_{q-k}, \ldots, I_q$ provide at most $(k+1)(1 + 2\varepsilon)3q$ processing capacity for the tails of filling tasks. Together, it is (by observation 3): $(k+1)(B + 3q + \varepsilon(7q-1)) < (k+1)(B + 3q + 1) \leq kB + B + Q$, because $Q = q(3q+1)$. Hence, tails cannot be feasibly scheduled, and $\sum_{j \in A_{q-k}} a_j \leq b$.

Assume that $\sum_{j \in A_{q-k}} a_j < b$. Then, the parallel parts of A_{q-k} require at most $\sum_{j \in A_{q-k}} Ma_j \leq M(b-1) = B' - M$ processing capacity, and all the parallel parts of tasks

in $A_{q-k} \cup \ldots \cup A_q$ demand at most $(k+1)B' - M$ processing capacity. By lemma 7, the demand for processing tails in $I_{q-k},\ldots,I_q$ may not exceed $(k+1)(1+2\varepsilon)3q$. The minimum capacity of $I_{q-k},\ldots,I_q$ is $(k+1)B'$. Since $M > (1+2\varepsilon)3q^2$, this results in some idle time and a schedule of makespan y does not exist. Thus, the lemma holds for k.

Therefore, we have $\sum_{j \in A_{q-l}} a_j = b$ for $k = 0,\ldots,q-1$. □

From lemma 8, we have that sets $A_1,\ldots,A_q$ form the required partition. This finishes the proof of theorem 1. □

Theorem 1 has important consequences for *multiprocessor* scheduling problems; it proves that problem $P|size_j, pmtn, chain|C_{max}$, where each multiprocessor task uses a fixed number of processors in parallel, is NP-hard in the strong sense. Another consequence is that preemptive scheduling of parallel tasks with a given *parallelism profile* is NP-hard in the strong sense (parallelism profile is the number of processors used over time, and it is measured in a computer with an unbounded number of processors [10]). In the following sections, we identify some polynomially solvable cases of problem $P|1any1, pmtn|C_{max}$.

4. $Pm|1any1, pmtn|C_{max}$ and $m \leq n$

In this section, we consider the case where the number of tasks n is not less than the number of processors m. We present a polynomial-time algorithm, MERGE, that merges an optimal schedule S of at most $m-1$ highest tasks with the remaining tasks to get an optimal schedule S' for the whole set $\mathcal{T}$. Intuitively, MERGE can be described as filling the periods of processor inactivity in S with the tasks of equal height. This algorithm exploits both ideas developed in [6] for scheduling unit execution time tasks with precedence constraints and the method for preemptive scheduling tasks with arbitrary processing times and tree-like precedence constraints based on the concept of *processing capabilities* [16]. We first introduce some necessary notation.

By *height* $h(j)$ of task j, we mean the time that remains to *finish* it provided that it is executed on one processor only. This value is $h_j + t_j + p_j$ initially, and changes during the execution of j. Let us assume that tasks are ordered in descending order of their initial heights, thus $h(1) \geq h(2) \geq \cdots \geq h(n)$. By a *high* task, we mean a task which is strictly higher than the mth task in $\mathcal{T}$. The set of all high tasks is denoted by $\mathcal{H}$, i.e. $\mathcal{H} = \{j : h(j) > h(m)\}$. Note that there can be at most $m-1$ high tasks. The tasks in set $\mathcal{L} = \mathcal{T} - \mathcal{H}$ are called *low* tasks.

Schedule S of high tasks is a sequence of intervals $1,\ldots,l$. For any interval, the number of processors assigned to a task does not change. Thus, interval k $(k = 1,\ldots,l)$ uniquely determines set Q_k of tasks executed in it, the number of processors $proc_k(j)$ occupied by task $j \in Q_k$ in it, and its length L_k. We say that interval k is *compact* if either the number of available processors in k (equal to $m - \sum_{i \in Q_k} proc_k(i)$) is zero or all high tasks which have not been finished in intervals $1,\ldots,k-1$ are in k.

We can convert any interval k into a compact one as follows. If $m - \sum_{i \in Q_k} proc_k(i) > 0$ in interval k and there is a task j in intervals $k+1,\ldots,l$ which is not in interval k, then we move to interval k a piece $q_j = \min\{l_j, L_k\}$ of task j from intervals $k+1,\ldots,l$, where l_j is the total time spent on j in $k+1,\ldots,l$. Then, we schedule the piece moved to interval k on a single processor, starting at the beginning of k. Next, we remove from intervals $k+1,\ldots,l$ the piece of j transferred to k. For $l_j > L_k$, the removal is done as follows. Let interval $k_j > k$ be the first interval such that task j occupies at least q_j units in intervals $k+1,\ldots,k_j$, i.e. $\sum_{r=k+1,\, j \in Q_r}^{k_j} proc_r(j) L_r \geq q_j$. Calculate $0 < t_j \leq L_{k_j}$ such that $\sum_{r=k+1,\, j \in Q_r}^{k_j - 1} proc_r(j) L_r + proc_{k_j}(j)(L_{k_j} - t_j) = q_j$. Then, remove j from intervals $k+1,\ldots,k_j - 1$ and from k_j between t_j and L_{k_j}. Notice that this creates at most one additional interval. This interval is obtained by splitting interval k_j at t_j, for $0 < t_j < L_{k_j}$. A pair $(\sum_{r=1}^{k_j - 1} L_r + t_j, j)$, for $0 < t_j < L_{k_j}$, will be called a *split* of interval k_j.

In the presentation of the MERGE algorithm, the following notation is used:

k	: interval index;
l	: the number of intervals in the initial schedule S of high tasks;
Q_k	: the set of high tasks in interval k of S;
$proc_k(j)$	: the number of processors occupied by high task j in interval k of S;
T_b	: the current time moment in S;
avail	: the number of processors available for low tasks;
β_j	: the processing capability of task j;
τ	: the duration of the current processing capabilities assignment;
compact(t, S)	: procedure making compact the interval of S that starts at t.

Algorithm MERGE

Input: A set $\mathcal{L} \subset \mathcal{T}$, such that $\forall_{j \in \mathcal{L}} h(j) \leq h(m)$;
A set $\mathcal{H} \subset \mathcal{T}$, such that $\forall_{j \in \mathcal{H}} h(j) > h(m)$ and $|\mathcal{H}| + |\mathcal{L}| \geq m$;
An optimal schedule S for $\mathcal{H}$ on m processors.

Output: An optimal schedule S' for $\mathcal{T}$ on m processors.

begin

Step 1. compact(0, S); $T_b := 0$;

Step 2. $A := \{j : j \in \mathcal{L}, h(m) = h(j)\}$;

Step 3. **while** $\mathcal{H} \neq \emptyset$ **and** $h(m) > 0$ **do**
begin (* consider the interval of S starting at T_b *)
3.1. Let k be the interval of S starting at T_b;
3.2. $\beta_j := 0$ for $j \in \mathcal{T}$;
3.3. $\beta_j := proc_k(j)$ for $j \in Q_k$;
3.4. $avail := m - \sum_{j \in Q_k} proc_k(j)$;

3.5. **if** $avail > 0$ **then** $\beta_j := avail/|A|$ for $j \in A$;
3.6. for the current values of β_j calculate the following times:
(a) L'_k – the length of interval k;
(b) $\tau' := (h(m) - h(j^*))/\beta_m$ if $\beta_m > 0$ – the shortest time required for the mth highest task to reach the height of the highest task j^* in $\mathcal{L} - A$ (if $\mathcal{L} - A = \emptyset$, then assume $h(j^*) = 0$);
(c) $\tau'' := \min_{j \in \mathcal{H}}\{(h(j) - h(m))/(\beta_j - \beta_m) : \beta_j > \beta_m\}$ – the shortest time required for the height of a high task to drop to the height of the mth highest task;
$\tau := \min\{L'_k, \tau', \tau''\}$;
3.7. Assign $\beta_j \tau$ piece of task j to interval $[T_b, T_b + \tau]$ for $j \in \mathcal{T}$;
3.8. Reduce $h(j)$ by $\tau\beta_j$ for $j \in \mathcal{T}$;
3.9. $T_b := T_b + \tau$;
3.10. $B := \{j : j \in \mathcal{H}, h(j) = h(m)\}$; $C := \{j : j \in \mathcal{L} - A, h(j) = h(m)\}$;
$A := A \cup B \cup C$; $\mathcal{H} := \mathcal{H} - B$;
3.11. **if** $B \neq \emptyset$ **then** remove $j \in B$ from the end of S starting at T_b;
3:12. compact(T_b, S)
end;

Step 4. **if** $h(m) > 0$ **then** assign $h(j)$ piece of $j \in \mathcal{T}$ to interval $[T_b, T_b + \sum_{j \in \mathcal{T}} h(j)/m]$;

Step 5. In each interval, schedule assigned pieces according to McNaughton's rule. Do not change the part of S starting at T_b;

end (* of algorithm MERGE *)

High level description. Interval 1 of initial schedule S is made compact in line 1. Intervals of S are considered one by one. The compact interval of S starting at T_b is considered in the while loop 3 (lines 3–3.12). In this interval, MERGE creates subintervals where processing capabilities assigned to tasks do not change. The subintervals are built in lines 3.1–3.7. In each, the high tasks remain with the same processing capabilities (line 3.3) as in the original interval. The number of processors left free by these tasks is calculated in line 3.4. These processors are fairly shared by the highest low tasks (line 3.5). Length τ of the subinterval calculated in line 3.6 prevents the height of some initially higher task from dropping below the height of some initially lower task inside the subinterval. A feasible assignment of tasks to each subinterval is obtained in line 3.7. The reduction of task heights in line 3.8 reflects the amount of processing the tasks receive in the subinterval. All high tasks that fall to $h(m)$ are transferred to A in line 3.10. The low tasks with heights equal to $h(m)$ are added to A. Any task that has been deleted from $\mathcal{H}$ is also deleted from the remaining part of the schedule for high tasks in line 3.11. The compactness of the next interval to be considered is restored in line 3.12. Hence, the compactness of the interval of S that starts at T_b is maintained throughout the whole MERGE run. Upon completing the while loop in line 3, we have $\mathcal{H} = \emptyset$, or $h(m) = 0$. In the former case, all tasks are low and there are at least m highest among them with their heights equal to $h(m)$.

They are assigned to the last subinterval in line 4. In the latter case, all low tasks have been completed, and the remaining intervals of S can be concatenated with the new schedule (line 5). The schedule for the subintervals constructed in loop 3–3.12 is obtained by applying McNaughton's wrap-around rule (line 5).

Lemma 9. The following are invariants of the while loop in line 3:

(1) $h(j) = h(m) \geq 0$ for $j \in A$,
(2) $h(j) > h(m)$ for $j \in \mathcal{H}$,
(3) $h(m) > h(j) \geq 0$ for $j \in \mathcal{L} - A$,
(4) $|\mathcal{H}| + |A| \geq m$.

Proof. We first prove that (1) holds. Line 3.5 sets capabilities of all tasks in A equal, and thus, line 3.8 reduces the height of each task in A by the same amount. Consequently, all tasks in A have the same height equal to $h(m)$ at the beginning of line 3.10. Since set A is extended in line 3.10 by adding to it only those tasks from $\mathcal{H}$ and $\mathcal{L} - A$ whose heights are equal to $h(m)$, then $h(j) = h(m)$ for $j \in A$ at the end of the loop. Furthermore, the values τ' and τ are set in line 3.6 to prevent $h(m)$ from becoming negative. Thus, (1) holds.

Next, we prove (2). Since line 3.10 moves all tasks with their heights equal to $h(m)$ from $\mathcal{H}$ to A, it suffices to show that no task in $\mathcal{H}$ is lower than $h(m)$ in line 3.10. This holds since the values of τ'' and τ are set to prevent the height of any high task from dropping below the height of the mth highest task. Thus, (2) also holds.

Finally, we show that (3) holds. It suffices to show that no task in $\mathcal{L} - A$ is higher than $h(m)$ in line 3.10. This must hold since the values τ' and τ are set to prevent the mth highest task from dropping below the height of the highest task in $\mathcal{L} - A$. Moreover, $\beta_j = 0$ for $j \in \mathcal{L} - A$. Therefore, (3) holds. (4) holds since nothing is ever removed from A, and any task removed from $\mathcal{H}$ is added to A in line 3.10. □

Lemma 10. In line 3.5, $|A| \geq avail$ and $\beta_j \leq 1$ for $j \in A$.

Proof. For each interval k that starts at T_b, its compactness is guaranteed by line 3.12. Hence, for k in line 3.1, either $avail = m - \sum_{j \in \mathcal{H}} proc_k(j) = 0 < |A|$ or $\mathcal{H} - Q_k = \emptyset$. In the former case, the lemma holds. In the latter, at least $|\mathcal{H}|$ processors are used by high tasks. Consequently, at most $m - |\mathcal{H}|$ processors are available to tasks in A. By the definition of $\mathcal{H}$ and lemma 9, $|A| + |\mathcal{H}| \geq m$. Thus, $|A| \geq m - |\mathcal{H}| \geq avail$ and the lemma holds. □

Lemma 11. The while loop 3 is executed no more than $n + lm$ times.

Proof. In line 3.6, $\tau = \tau'$ at most $n - m + 1$ times, $\tau = \tau''$ at most $m - 1$ times, and $\tau = L'_k$ at most lm times. We now prove that the last claim holds. A compact interval in

line 3.12 starts at T_b and ends at E_b for some moment E_b. E_b either equals $\sum_{r=1}^{k} L_r$, for some original interval k or equals $\sum_{r=1}^{k-1} L_r + t_j$, for some split $(\sum_{r=1}^{k-1} L_r + t_j, j)$ of some original interval k or equals the completion time of some high task. Moreover, for any original interval k, inequalities $\sum_{r=1}^{k-1} L_r < E_b < \sum_{r=1}^{k} L_r$ with $E_b = \sum_{r=1}^{k-1} L_r + t_j$, for some split $(\sum_{r=1}^{k-1} L_r + t_j, j)$ of some original interval k, hold at most $m-1$ times in line 3.12. Otherwise, there would be two different splits (t, j) and (t', j), $t < t'$, of original interval k with the same high task j. Thus, E_b would assume value t on one pass through line 3.12 and value t' on some later pass through the same line. This, however, cannot happen because by the definition of split, there is no task j between t and $\sum_{r=1}^{k} L_r$ when $E_b = t$. Therefore, split (t', j) would have not been created by procedure compact. Finally, the completion time of a high task either equals $\sum_{r=1}^{k} L_r$, for some original interval k or equals $\sum_{r=1}^{k-1} L_r + t_j$, for some split $(\sum_{r=1}^{k-1} L_r + t_j, j)$ of some original interval k. Therefore, $\tau = L'_k$ at most $l + l(m-1)$times.

Thus, τ assumes at most $n + lm$ different values and, consequently, the while loop 3 is executed no more than $n + lm$ times. □

Theorem 12. MERGE is a correct optimization algorithm.

Proof. By lemma 11, MERGE halts. Thus, it suffices to prove that it provides a feasible and optimal schedule S'.

The tasks assigned to the interval $[T_b, T_b + \tau]$ in line 3.7 meet the following conditions:

(i) The tasks in Q_k have the same processing capabilities as in S. They occupy $\sum_{j \in Q_k} \beta_j \tau = \tau(m - avail)$ of the interval's capacity.

(ii) The tasks in A get no more than τ processing each, by lemma 10. They occupy $\sum_{j \in A} \beta_j \tau = \tau \sum_{j \in A} \beta_j = \tau |A| avail/|A| = \tau\, avail$ of the interval's capacity.

(i) and (ii) make sure that a feasible schedule on m processors can be obtained by McNaughton's wrap-around rule in $[T_b, T_b + \tau]$. Also, the precedence relations among parallel part and sequential operations are not violated for any task, because no task in A receives more than one processor at a time. Notice that this schedule leaves no idle time in $[T_b, T_b + \tau]$.

In line 4, if $h(m) > 0$, then $\mathcal{H} = \emptyset$. By lemma 9, $|A| \geq m$ and $h(j) = h(m)$ for $j \in A$. Consequently, $\sum_{j \in \mathcal{T}} h(j)/m \geq \max_{j \in \mathcal{T}} \{h(j)\} = h(m)$, and a feasible schedule on m processors can be obtained by McNaughton's wrap-around rule in $[T_b, T_b + \sum_{j \in \mathcal{T}} h(j)/m]$. Therefore, if the while loop 3 finishes with $h(m) > 0$, then the final schedule has no idle time inserted and so it is optimal. The theorem holds. It also holds if $h(m) = 0$ and $\mathcal{H} = \emptyset$ in line 4. For $h(m) = 0$ and $\mathcal{H} \neq \emptyset$ in line 4, the makespan of S equals the makespan of S'. Again, the theorem holds. □

Lemma 13. MERGE has time complexity $O((n + lm)(n + m(l + m)))$.

Proof. By lemma 11, the while loop in line 3 is executed at most $n + lm$ times. Lines 3.1–3.10 can be executed in time $O(n)$. In line 3.11, the removal of a high task from the intervals following the current one can be done in $O(l + m)$ time. Notice that the number of such intervals never exceeds $l + m - 1$. Since there are at most $m - 1$ high tasks, line 3.11 requires $O(m(l + m))$ time during the whole MERGE run. Procedure compact(T_b, S) in line 3.12 can be implemented to run in $O(m(l + m))$ time. Hence, the while loop in line 3 requires $O((n + lm)(n + m(l + m))$ time. Line 4 can be executed in $O(n)$ time, and line 5 in $O(n(l + m - 1 + n))$ time. Therefore, the time complexity of the algorithm is $O((n + lm)(n + m(l + m))$. □

Theorem 14. There is an optimal schedule with only high tasks using more than one processor at a time.

Proof. The theorem follows immediately from the fact that $\beta_j \leq 1$ for $j \in A$ in the while loop in line 3, see lemma 7. □

The quality of schedule S for high tasks is of great importance because algorithm MERGE ends up with either an optimal schedule or a schedule not longer than S. Obviously, even suboptimal schedule S can be used by MERGE. Note that for fixed m, problem $Pm|1any1, pmtn|C_{max}$ can be solved in polynomial time as follows. An optimal schedule for $m - 1$ or less high tasks can be obtained by applying LP to any feasible permutation of their $s(j)$ and $f(j)$ indices, see section 2. A complete optimal schedule for all n tasks can then be found by MERGE. Since the number of the permutations does not exceed fixed $(2m)!/2^m$, this approach leads to a polynomial time algorithm.

MERGE can be used to solve a more general problem with chain precedence constraints between parallel tasks. This holds even for the problem with an upper bound on the number of processors a parallel task is allowed to use in parallel, i.e. problem $P|any, pmtn, chain|C_{max}$. We have the following corollary.

Corollary 15. MERGE solves problem $P|any, pmtn, chain|C_{max}$ for a given optimal schedule S for chains higher than the mth highest chain.

Proof. Corollary 15 follows from the fact that tasks in $\mathcal{L}$ are scheduled using at most one processor at a time. □

In the following section, we identify some special cases for which polynomial algorithms exist.

5. Polynomially solvable cases of $P|1any1, pmtn|C_{max}$

This section presents special cases of $P|1any1, pmtn|C_{max}$ for which low-order polynomial time algorithms exist. First, we consider problem $P|1any1, pmtn|C_{max}$

with tasks without tails. The algorithm we give is based on McNaughton's and the earliest completion time rules. We call it XMECT (eXtended McNaughton – Earliest Completion Time) for short. Although we define it for tasks without tails, it can also be applied to optimally schedule tasks without heads, i.e., to problem $P|any1, pmtn|C_{max}$. Let us assume that $h_1 \geq h_2 \geq \ldots \geq h_n$. The algorithm is as follows.

Algorithm XMECT

begin

Step 1. $j := 1$; (* scheduling heads – beginning *)

Step 2. **while** $h_j > (1/(m-j))\sum_{i=j+1}^{n} h_i$ **do**
(* $head_j$ is longer than the optimal makespan for heads $head_{j+1},\ldots,head_n$ scheduled on processors $j+1,\ldots,m$ *)
begin
2.1. schedule whole $head_j$ on processor j starting at 0;
2.2 $j := j+1$;
end;
(* heads $1,\ldots,j-1$ are scheduled on processors $1,\ldots,j-1$ *)

Step 3. schedule heads $head_j,\ldots,head_n$ on processors $j,\ldots,m$ according to McNaughton's wrap-around-rule; (* scheduling heads – end *)
(* scheduling parallel parts – beginning *)

Step 4. schedule parallel parts as soon as their heads are finished so that there be no idle time between them, use as many processors as available.

end. (* scheduling parallel parts – end *)

High level description. The heads are scheduled first. They are divided into two subsets (line 2): the set of long heads and the set of short heads. The long heads are assigned to one processor each (line 2.1), while the short heads are scheduled according to McNaughton's wrap-around rule (line 3) on the remaining processors. The parallel parts are started as soon as possible and using all available processors, i.e. not occupied by heads (line 4).

We have the following theorem.

Theorem 16. XMECT solves $P|1any, pmtn|C_{max}$ to optimality in $O(n \log n)$ time.

Proof. We start from the second part of the theorem. To find the order of heads, $O(n \log n)$ time is needed for sorting. The while loop in line 2 can be executed in time $O(\min\{n, m\})$. Lines 3 and 4 can be executed in time $O(n)$. The whole algorithm can be executed in $O(n \log n)$ time.

Suppose XMECT gives suboptimal schedules. Then, there exists an instance for which it fails to give an optimal schedule. Let us consider such an instance I^*, with

the smallest number of tasks n^*. Apply XMECT to I^* to get schedule S. Since S is suboptimal, there is some idle time in S. Let t be the time when the latest idle interval begins in S. Delete all the tasks that finish by t. Let I^{**} denote the instance with the tasks left after the removal. Now, we can distinguish two cases.

Case 1. $I^{**} = I^*$. Then, $t = 0$ and $n < m$. Thus, the total idle time inserted in any schedule for I^* is at least $(m - n)h_n$. Since XMECT inserts exactly $(m - n)h_n$ idle time into S, S is optimal.

Case 2. $I^{**} \subset I^*$. Then, the number of tasks in I^{**} is smaller than n^*. Thus, $C^{**} = C^{**}_{opt}$, where C^{**} and C^{**}_{opt} are the makespan of the schedule obtained by XMECT for I^{**} and the optimal makespan for I^{**}. On the other hand, we have $I^* = I^{**} \cup \{T_{i^*+1}, \ldots, T_n\}$ for some i^*. Therefore $C^{**} = C^*$. Obviously, $C^*_{opt} \geq C^{**}_{opt}$ because $I^{**} \subset I^*$. Thus, we get $C^*_{opt} \geq C^*$ and this is possible only if $C^*_{opt} = C^*$. Hence, we get a contradiction with the assumption that $C^*_{opt} < C^*$, and XMECT solves $P|1any, pmtn|C_{max}$ to optimality. □

Theorem 17. If $\mathcal{H} = \emptyset$, then $P|1any1, pmtn|C_{max}$ can be solved in $O(n)$ time.

Proof. MERGE reduces to lines 4 and 5 in this case. □

Theorem 18. Problem $P2|1any1, pmtn|C_{max}$ can be solved in $O(n^2)$ time using the MERGE algorithm.

Proof. For $m = 2$, there is at most one high task. The optimal schedule for one task is easy to find. It has $l \leq 3$ intervals. By lemma 9, the time complexity of MERGE is $O(n^2)$ in this case. □

Now, we consider $P3|1any1, pmtn|C_{max}$.

Theorem 19. Problem $P3|1any1, pmtn|C_{max}$ can be solved in $O(n^2)$ time by the MERGE algorithm.

Proof. For $m = 3$, the number of high tasks cannot be greater than two. Thus, it is sufficient to find an optimal schedule for two tasks. Without loss of generality, we can assume that there is at most one tail and at most one head in an instance of two tasks 1 and 2. We can distinguish two cases.

Case 1. The head and the tail belong to the same task. Without loss of generality, we assume that task 2 has no sequential operations. The optimal schedules are shown in figures 3(a) and 3(b).

Case 2. The head and the tail belong to different tasks. Without loss of generality, we assume that task 1 has a head, and task 2 has a tail. We can assume $h_1 +$

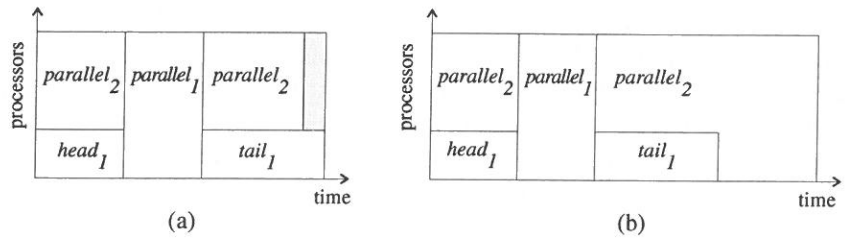

Figure 3. Optimal schedules for two tasks only:
the tail and the head belong to task 1.

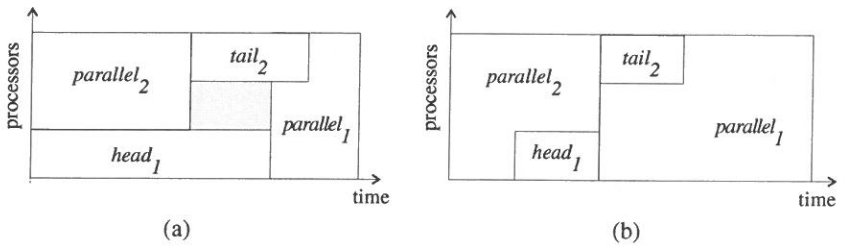

Figure 4. Optimal schedules for two tasks only:
task 1 has a head, task 2 has a tail.

$p_1/(m-1) \geq t_2 + p_2/(m-1)$. The opposite case can be dealt with by renaming tail for head, head for tail and by reading a schedule from its end. For $p_2/(m-1) \leq h_1$, an optimal schedule is presented in figure 4(a). The opposite, $p_2/(m-1) > h_1$, implies $p_1/(m-1) > t_2$ with an optimal schedule depicted in figure 4(b). Thus, we have shown how to obtain the optimal schedule for two tasks only. The complexity of identifying a proper case and building the corresponding schedule is $O(1)$.

Such a schedule has $l \leq 6$, which results in the time complexity of $O(n^2)$ for MERGE. □

Theorem 20. Problem $P|1any1, pmtn|C_{max}$ can be solved in $O(n \min\{\log n, m\})$ time if

$$\frac{1}{m}\sum_{j=1}^{n}(h_j + t_j) \geq \max_j\{h_j + t_j\}.$$

Proof. The construction of an optimal schedule is done in two steps. First, heads and tails are scheduled. Then, parallel parts are inserted into the resulting schedule.

To do the first step, we replace $head_j$ and $tail_j$ of task j with compound task j with execution time $h_j + t_j$, for $j = 1,\ldots,n$. For such compound tasks, McNaughton's wrap-around rule produces a schedule without idle time in $[0, \sum_{j=1}^{n}(h_j + t_j)/m]$. In this schedule, we can identify, for compound task j, a moment of time when exactly h_j units of this task are executed. Let us call such moments *cracks*. This step can be implemented to run in $O(n)$ time. The second step consists in inserting parallel parts

of the tasks on all processors in the cracks of the compound tasks. All tasks executed at the crack of compound task j ($j = 1,\ldots,n$) are preempted and $parallel_j$ is inserted on all processors for p_j/m units of time. Then, the original schedule resumes. It can be observed that the insertions of parallel parts do not introduce idle times in the schedule. These insertions on m processors can be done in $O(\min\{\log n, m\})$ time each. Hence, the time complexity is $O(n \min\{\log n, m\})$. □

In the following lemma, we show that the algorithm described in theorem 20 is asymptotically optimal in probability.

Lemma 21. For $h_j + t_j$ ($j = 1,\ldots,n$) being independent and uniformly distributed over interval $[A, B]$ with $A \geq 0$,

$$\lim_{n\to\infty} P\left[\frac{1}{m}\sum_{j=1}^{n}(h_j + t_j) \geq \max_j\{h_j + t_j\}\right] = 1$$

for $n \geq 4m$.

Proof. The proof follows from [4, p. 415, inequality (3.7)]. □

Before presenting another polynomially solvable case, let us introduce some notation. Let us consider $2n$ numbers: h_j and $C - t_j$ for $j = 1,\ldots,n$, where $C = \max_{j\in\mathcal{T}}(p_j/m + t_j + h_j)$. We order them in ascending order, $0 = e_0 \leq e_1 = \min_j\{h_j\} \leq e_2 \leq \cdots \leq e_{2n} = \max_j\{C - t_j\} \leq e_{2n+1} = C$, and define sets $RS_i = \{j : h_j > e_{i-1}$ or $C - t_j < e_i\}$ and $RP_i = \{j : h_j \leq e_{i-1}$ and $C - t_j \geq e_i\}$ for $i = 1,\ldots,2n$. We assume that $p_j/(C - h_j - t_j) = 0$ for $C = h_j + t_j$. We have the following theorem.

Theorem 22. Problem $P|1any1, pmtn|C_{max}$ can be solved in $O(n^2)$ time if

$$\forall 2 \leq i \leq 2n \quad |RS_i| + \sum_{j\in RP_i}\frac{p_j}{C - h_j - t_j} \geq m. \tag{15}$$

Proof. We prove that if (15) holds, then it is possible to build an optimal schedule without unnecessary idle times. The main idea of the proof is shown in figure 5. There, if $parallel_j$ is processed at speed $\beta_j = p_j/(C - h_j - t_j)$ for $C - h_j - t_j$ time units, then j will finish at C. Notice that $C - h_j - t_j \geq 0$.

For parts of tasks in $[e_{i-1}, e_i]$, $i = 2,\ldots,2n$ in figure 5, we build a feasible schedule of length $C_i = ((e_i - e_{i-1})/m)(|RS_i| + \sum_{j\in RP_i} p_j/(C - h_j - t_j)) \geq e_i - e_{i-1}$. Since the sequential operations of tasks in $[e_{i-1}, e_i]$ are not longer than $e_i - e_{i-1} \leq C_i$, this schedule can be obtained by McNaughton's rule in $O(n)$ time. Thus, a feasible schedule without idle time for $[e_1, e_{2n}]$ of figure 5 can be obtained in $O(n^2)$ time.

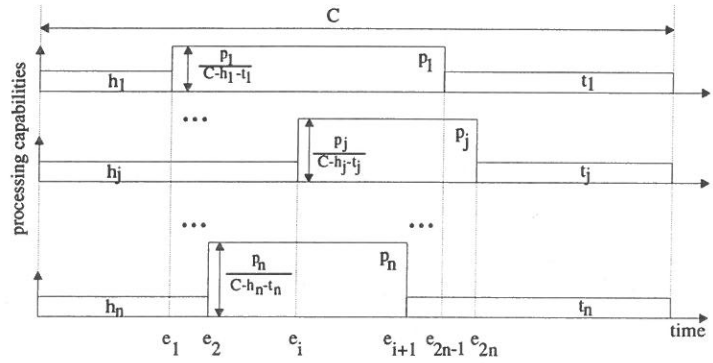

Figure 5. The proof of theorem 22.

Schedules for parts of tasks from $[e_0, e_1]$ and $[e_{2n}, e_{2n+1}]$ of figure 5 can be obtained by McNaughton's wrap-around rule as well. However, if these intervals are non-empty and $n < m$, idle time is unavoidable. □

6. Heuristics for $P|1any1, pmtn|C_{max}$

In this section, we present heuristics for problem $P|1any1, pmtn|C_{max}$ and analyze their performance. By the worst-case performance ratio of heuristic H, we mean the ratio $S^H = C^H_{max}/C^*_{max}$, where C^H_{max} is the makespan of the schedule obtained by H, and C^*_{max} is the optimal makespan. We consider the following three heuristics:

Heuristic H1

Step 1. Schedule all heads according to McNaughton's rule.

Step 2. Schedule all parallel parts. Use all processors for each parallel part. Start the earliest parallel part when the latest head finishes.

Step 3. Schedule all tails according to McNaughton's rule. Start the earliest tail when the latest parallel part finishes.

Heuristic H2

Step 1. Consider tasks without their tails i.e., for each task j consider only $head_j$ and $parallel_j$. Schedule such a set of tasks according to XMECT to obtain schedule S_1.

Step 2. Schedule tails only according to McNaughton's rule to obtain schedule S_2.

Step 3. Concatenate S_1 and S_2.

Heuristic H3

Step 1. Schedule the compound tasks (see theorem 20 for definition) according to McNaughton's rule.

Step 2. Insert $parallel_j$ on all processors at the crack point for task $j = 1,\ldots,n$.

The following three theorems give tight bounds on the worst-case performance ratio of the above heuristics.

Theorem 23. Heuristic H1 can be implemented to run in $O(n)$ time. The worst-case performance ratio of H1 satisfies $S^{H1} \leq 3$ and this bound is tight.

Proof. The tight worst-case bound for H1 is 3 because in each of the three steps, H1 builds a partial schedule not longer than the optimum. To verify the tightness consider the following instance: $n = 3$, $h_1 = A$, $p_1 = B$, $p_2 = B$, $t_2 = A$, $p_3 = (m-2)(A+B)$, $t_1 = h_2 = t_3 = h_3 = 0$. The optimal schedule has makespan $C^*_{max} = A + B$, while H1 builds a schedule with makespan $C^{H1}_{max} = 2A + (1 - 2/m)(A + B) + 2B/m$. Thus, $\lim_{m\to\infty} S^{H1} = (3A + B)/(A + B)$ which approaches 3 as B tends to 0. □

Theorem 24. Heuristic H2 can be implemented to run in $O(n \log n)$ time. The worst-case performance ratio of H2 satisfies $S^{H2} \leq 2$ and this bound is tight.

Proof. The complexity of obtaining S_1 and S_2 is $O(n \log n)$ (cf. algorithm XMECT). Note that neither S_1 nor S_2 is longer than C^*_{max}. Thus, $S^{H2} \leq 2$. To see that this bound is tight, consider the following instance: $n = m$, $h_1 = A$, $t_1 = B$, $p_1 = B$, $h_2 = \cdots = h_m = A + B$, $t_2 = \cdots = t_m = p_2 = \cdots = p_m = 0$, where A, B are positive. It can be observed that an optimal schedule has makespan $C^*_{max} = h_1 + t_1 + p_1/m = A + B + B/m$. In the schedule built by H2, heads are processed first without preemption. As soon as $head_1$ finishes, $parallel_1$ starts and it is executed on one processor only. $parallel_1$ and $head_2, \ldots, head_m$ finish at $h_2 = A + B$ when $tail_1$ starts. Thus, this schedule has makespan $C^{H2} = h_1 + p_1 + t_1 = A + 2B$. Hence, we have $\lim_{m\to\infty} S^{H2} = (A + 2B)/(A + B)$. Furthermore, when $h_1 = A$ tends to 0, then S^{H2} tends to 2. □

Theorem 25. H3 can be implemented to run in $O(n \min\{\log n, m\})$ time, has the worst-case bound $S^{H3} \leq 2$ and this bound is tight.

Proof. The $O(n \min\{\log n, m\})$ complexity of H3 has been shown in theorem 20. The makespan of the schedule obtained in line 1 may not be greater than C^*_{max}. In line 2, the makespan is increased by exactly $\sum_{j=1}^{n} p_j/m$. Such an increase must not be greater than C^*_{max}. Hence, $S^{H2} \leq 2$.

To show that the bound is tight, consider the following: $m = n$, $p_1 = t_1 = 0$, $h_2 = \cdots = h_m = t_2 = \cdots = t_m = h_1/(2m)$, $p_2 = \cdots = p_m = h_1(1 - 1/m)$, where h_1 is the processing time of $head_1$. In the optimal schedule, $head_j$ starts at moment 0 on processor j; $head_1$ finishes at h_1, while $head_2, \ldots, head_m$ finish at $h_1/(2m)$. Next, $parallel_2, \ldots, parallel_m$ are executed on $m - 1$ processors finishing by $h_1 - h_1/(2m)$. Then, $tail_2, \ldots, tail_m$ follow on processors $2, \ldots, m$. Thus, the optimal schedule has

makespan $C^*_{max} = h_1$. H3 builds the schedule as follows. In step 1, a schedule is built in which only two processors are occupied: on one processor $head_1$ is executed, on the other $head_2$, $tail_2$,..., $head_m$, $tail_m$ are executed. Step 2 inserts $parallel_2$,..., $parallel_m$ which increases makespan to $C^{H3}_{max} = h_1 + \sum_{j=2}^{m} p_j/m = h_1 + h_1(1-1/m)^2$. Hence, $\lim_{m\to\infty} S^{H3} = 2$. □

The above three heuristics were implementated in Borland Pascal 7.0, and run on a 486 PC with MS-DOS 6.22. Their results were compared against a lower bound $\max\{\max_{j\in\mathcal{T}}\{h_j + p_j/m + t_j\}, (1/m)\sum_{j=1}^{n}(h_j + p_j + t_j)\}$ on the optimal makespan. Processing times of heads and tails, h_j and t_j, were randomly generated from the uniform distribution in [0, 1]. Processing times of parallel parts, p_j, were randomly generated from the uniform distribution in [0, ρ], where $\rho > 0$. The results of computational experiments are collected in figures 6–8. Each point represents an average relative (with respect to the lower bound) error in 1000 experiments, over 130 thousand instances were solved. In figure 8, the horizontal axis represents values of ρ.

Figures 6–8 show that the worst performance on average has been observed when the ratio n/m tends to 1 and/or ρ tends to 2. Still, the average performance of H1, H2, and H3 is not worse than 1.5, 1.25, and 1.4, respectively. For large (relative to m) values of n and small (relative to n) values of m, all heuristics build schedules close to optimum, see figures 6 and 7. For large m and constant n, only H3 tends to building schedules close to optimum, see figure 7, while both H1 and H2 converge to 1.1. Then, however, the sequential parts, $head_j$ and $tail_j$, dominate the schedule and the length of a schedule built by either H1 or H2 tends to $\max_{j\in\mathcal{T}} h_j + \max_{j\in\mathcal{T}} t_j$, while the length of a schedule built by H3 tends to $\max_{j\in\mathcal{T}}\{h_j + t_j\}$. A similar situation takes place when $\rho < 0.1$, see figure 8.

To conclude, H1 always builds on average worse schedules than H2. Furthermore, there seem to exist two values α and ω, such that H3 performs on average better than H2 for $n/m \le \alpha$ and $n/m \ge \omega$, while H2 performs on average better than H3 for n/m between α and ω. Our computations have shown that a good estimate for α is somewhere between 0.3 and 0.5, and that a good estimate for ω is 1.7. Finally, H3 performs on average better than H2 for $\rho \le 0.3$, and H2 does better otherwise. This suggests that H2 and H3 complement each other and should thus be used together with better solution chosen as an output of the combined heuristic.

7. Concluding remarks

This paper has considered the problem of scheduling parallel tasks with sequential heads and tails to minimize makespan. The parallel tasks are different from the multiprocessor tasks because they allow the number of processors processing a task to change with time. They are also different from the DAG model of parallel computations because they allow more than one processor at a time to process a task. Despite this difference, we show that the concept of profiles developed in [6] for the

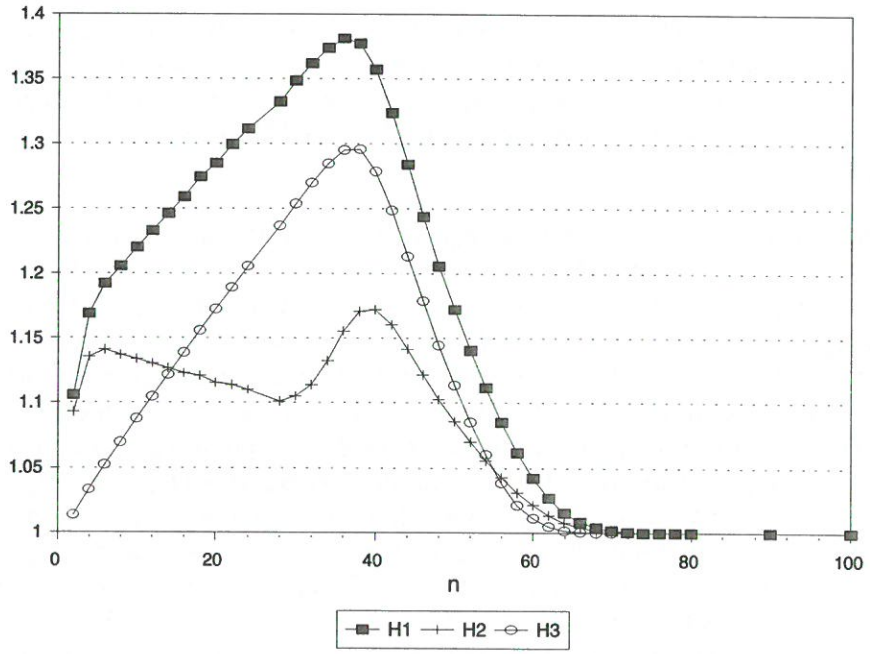

Figure 6. Average relative distance from the lower bound for H1, H2, H3 versus the number of tasks, $m = 32$, $\rho = 1$.

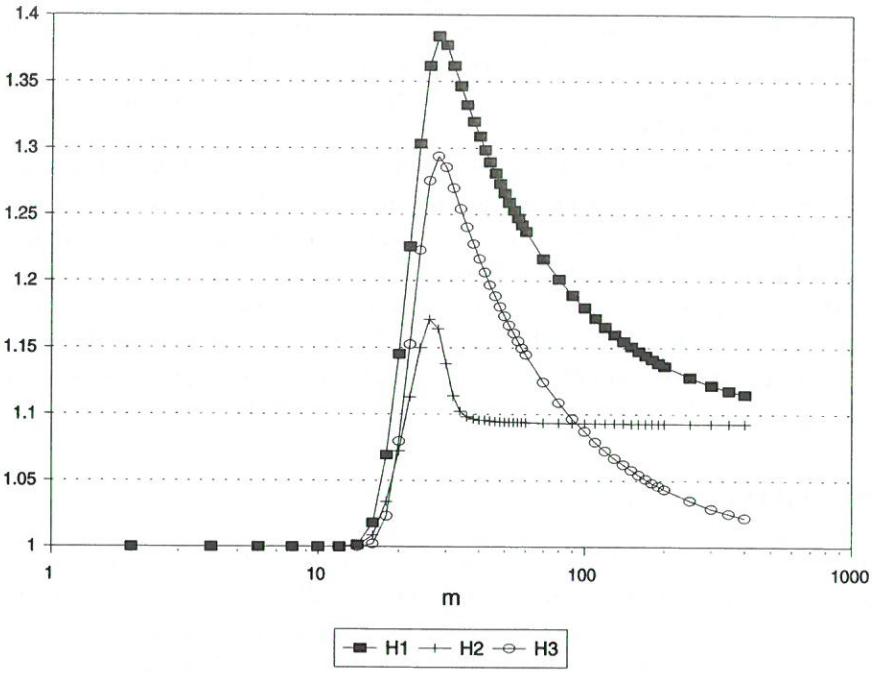

Figure 7. Average relative distance from the lower bound for H1, H2, H3 versus the number of processors, $n = 32$, $\rho = 1$.

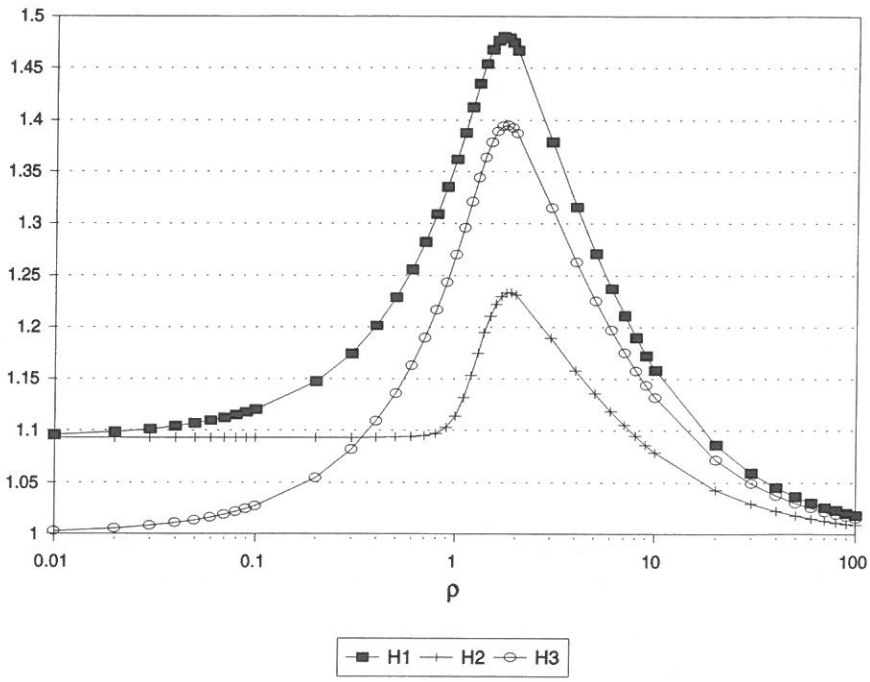

Figure 8. Average relative distance from the lower bound for H1, H2, H3 versus ρ, $m = n = 32$.

DAG model can be extended to prove that the problem of scheduling parallel tasks is NP-hard in the strong sense. This holds even if the number of tasks does not exceed the number of processors. We show that this case remains crucial for the time complexity of the scheduling problem, since any optimal schedule for tasks longer than the mth longest task can be extended to an optimal schedule for all tasks in polynomial time. We show that this follows from earlier results obtained in [6] and [19]. In the resulting optimal schedule for all tasks, only those longer than the mth longest task use the parallel processing capability of parallel systems, all the remaining tasks are scheduled as sequential. This property of some optimal schedules seems to be of importance for scheduling large number of parallel tasks in real parallel systems.

The complexity of scheduling tasks with one sequential operation, either head or tail, remains unknown.

References

[1] L. Bianco, J. Błażewicz, P. Dell'Olmo and M. Drozdowski, Scheduling preemptive multiprocessor tasks on dedicated processors, Performance Evaluation 20(1994)361–371.

[2] J. Błażewicz, P. Dell'Olmo, M. Drozdowski and M.G. Speranza, Scheduling multiprocessor tasks on three dedicated processors, Inf. Proc. Letters 41(1992)275–280, Corrigendum Inf. Proc. Letters 49(1994)269–270.

[3] J. Błażewicz, M. Drabowski and J. Węglarz, Scheduling multiprocessor tasks to minimize schedule length, IEEE Trans. Comput. C-35(1986)389–393.
[4] J.L. Bruno and P.J. Downey, Probabilistic bounds on the performance of list scheduling, SIAM J. on Computing 15(1986)409–417.
[5] E.G. Coffman, Jr., *Computer and Job-shop Scheduling Theory*, Wiley, 1976.
[6] D. Dolev and M. Warmuth, Profile scheduling of opposing forests and level orders, SIAM J. on Algebraic and Discrete Methods 6(1985)665–687.
[7] J. Du and J.Y-T. Leung, Complexity of scheduling parallel task systems, SIAM J. on Discrete Mathematics 2(1989)473–487.
[8] M.R. Garey and D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W.H. Freeman, San Francisco, 1979.
[9] M.R. Garey, D.S. Johnson, R.E. Tarjan and M. Yannakakis, Scheduling opposing forests, SIAM J. on Algebraic and Discrete Methods 4(1983)72–93.
[10] D. Ghosal, G. Serazzi and S. Tripathi, The processor working set and its use in scheduling multiprocessor systems, IEEE Transactions on Software Engineering 17(1991)443–453.
[11] R.L. Graham, E.L. Lawler, J.K. Lenstra and A.H.G. Rinnooy Kan, Optimization and approximation in deterministic sequencing and scheduling: A survey, Ann. Discrete Math. 5(1979)287–326.
[12] J.L. Gustafson, R.E. Benner, M.P. Sears and T.D. Sullivan, A radar simulation for a 1024-processor hypercube, in: *Proceedings of Supercomputing 1989*, ACM Press, New York, 1989, pp. 96–105.
[13] M. Kubale, The complexity of scheduling independent two-processor tasks on dedicated processors, Inf. Proc. Letters 24(1987)141–147.
[14] E.L. Lloyd, Concurrent task systems, Oper. Res. 29(1981)189–201.
[15] R. McNaughton, Scheduling with deadlines and loss functions, Management Science 6(1959)1–12.
[16] R.R. Muntz and E.G. Coffman, Jr., Preemptive scheduling of real-time tasks on multiprocessor systems, Journal of ACM 17(1970)324–338.
[17] K.C. Sevcik, Application scheduling and processor allocation in multiprogrammed parallel processing systems, Performance Evaluation 19(1994)107–140.
[18] J.D. Ullman, NP-complete scheduling problems, J. Comput. Syst. Sci. 10(1975)384–393.
[19] B. Veltman, B.J. Lageweg and J.K. Lenstra, Multiprocessor scheduling with communication delays, Parallel Computing 16(1990)173–182.