ELSEVIER Parallel Computing 26 (2000) 1195–1211

PARALLEL COMPUTING

www.elsevier.com/locate/parco

Scheduling preemptable tasks on parallel processors with limited availability

Jacek Błażewicz [a], Maciej Drozdowski [a], Piotr Formanowicz [a,*], Wiesław Kubiak [b], Günter Schmidt [c]

[a] *Institute of Computing Science, Poznań University of Technology, Poznań, Poland*
[b] *Faculty of Business Administration, Memorial University of Newfoundland, St. John's, Canada*
[c] *Department of Information and Technology Management, University of Saarland, Saarbrücken, Germany*

Received 1 October 1998; received in revised form 1 May 1999; accepted 1 July 1999

Abstract

It is well known that in the majority of cases the problem of preemptive task scheduling on m parallel identical processors with the objective of minimizing makespan can be solved in polynomial time. For example, for tree-like precedence constraints the algorithm of Muntz and Coffman can be applied. In this paper, this problem is generalized to cover the case of parallel processors which are available in certain time intervals only. It will be shown that this problem becomes NP-hard in the strong sense in case of trees and identical processors. If tasks form chains and are processed by identical processors with a staircase pattern of availability, the problem can be solved in low-order polynomial time for $C_{\max}$ criterion, and a linear programming approach is required for $L_{\max}$ criterion. Network flow and linear programming approaches will be proposed for independent tasks scheduled on, respectively, uniform and unrelated processors with arbitrary patterns of availability for schedule length and maximum lateness criteria. © 2000 Published by Elsevier Science B.V. All rights reserved.

Keywords: Preemptive scheduling; Parallel processors; Limited machine availability; Linear programming; Computational complexity

1. Introduction

In the majority of the previous papers on scheduling it has been assumed that processors are continuously available. In this work, we assume that processors are

* Corresponding author.
E-mail address: piotr@cs.put.poznan.pl (P. Formanowicz).

available only in some intervals of time, or time windows in other words. It is not difficult to justify this assumption [1,9,17,18]. For example, in computer systems some tasks are urgent real-time tasks which are prescheduled on processors and executed in fixed time periods. Thus, free time intervals for lower priority tasks are created. By the same token, tasks with higher priority may block the computer system or part of it, and in this way create intervals of changing processor availability. The load of multiuser computer systems changes during the day and in the week. In big massively parallel systems it is convenient to change the partition of the processors among different types of users according to their 'presence' on the machine. Fluctuations of the processing capacity can be modeled by intervals of different processor availability. Other applications arise in the context of manufacturing systems where certain dynamic activities (tasks) are scheduled in the intervals of processor (machine) availabilities on a rolling horizon basis. Some other reasons for non-availability periods follow maintenance requirements or breakdowns. In this work, we analyze problems of preemptive scheduling tasks on parallel processors available in certain intervals only. Before doing this and presenting related results we will set up the problem more formally (cf. [3]).

Let $\mathscr{P} = \{P_i \mid i = 1, \ldots, m\}$ be the set of *parallel processors*, where by parallel we mean that any task may be processed by any processor. Three types of parallel processors are distinguished: identical, uniform, and unrelated, respectively, where uniform processors differ in their speeds b_i's. Now let $\mathscr{T} = \{T_j \mid j = 1, \ldots, n\}$ denote the set of *tasks*. Task $T_j \in \mathscr{T}$ has processing requirement of p_j time units in case of identical processors, whereas in case of uniform processors the processing time of task T_j on processor P_i is p_j/b_i, where p_j is measured on a standard processor with speed equal to 1. In case of unrelated processors p_{ij} denotes processing time of task T_j on P_i. Task T_j is also characterized by its ready time r_j and due-date d_j. Among the tasks precedence constraints are defined, and in this paper trees, chains, and empty precedence constraints are analyzed.

Processors are available in q different time intervals. Let $0 < t_1 < t_2 < \cdots < t_l < \cdots < t_q$ be the points in time where the availability of processors changes. We will denote by $m^{(l)}$ the number of identical processors available in interval $[t_l, t_{l+1})$ with $m^{(l)} \geqslant 0$. Following [15,17] we will consider the following patterns of processor availability:

1. If all processors are continuously available the pattern is called *constant*.
2. If there are only k or $k-1$ processors available in each interval the pattern is called *zig-zag*.
3. A pattern is called *increasing* (*decreasing*) if for all $l = 1, \ldots, q-1$: $m^{(l)} \geqslant \max_{1 \leqslant u \leqslant l-1} \{m^{(u)}\}$ ($m^{(l)} \leqslant \min_{1 \leqslant u \leqslant l-1} \{m^{(u)}\}$), i.e., the number of processors available in interval $[t_{l-1}, t_l)$ is not more (not less) than the number in interval $[t_l, t_{l+1})$.
4. A pattern is called *staircase* if for all intervals the availability of processor P_i implies the availability of processor P_{i-1}. A staircase pattern is shown in Fig. 1 where the shaded areas represent intervals of non-availability; patterns 1–3 are special cases of 4.

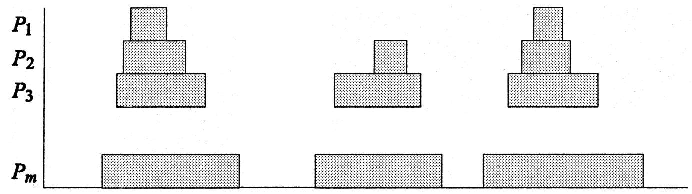

Fig. 1. A staircase pattern of processors availability.

In the paper, optimal preemptive schedules are looked for. The optimality criteria we consider are: schedule length (makespan) $C_{\max} = \max_{T_j \in \mathscr{T}}\{c_j\}$, and maximum lateness $L_{\max} = \max_{T_j \in \mathscr{T}}\{c_j - d_j\}$, where c_j is the time at which task T_j is finished.

To denote our scheduling problem we will use the standard three-field notation [7] with some extensions in the processor field to denote changing patterns of processor availability. These extensions, placed after symbols denoting the type and the number of processors, are as follows: NC_{win}, *zig-zag*, *increasing* (*decreasing*), *increasing zig-zag* (*decreasing zig-zag*), *staircase* and they denote, respectively, arbitrary, zig-zag, increasing (decreasing), increasing (decreasing) zig-zag and staircase patterns of availability. (In the context of flow shop scheduling, symbol h_{ik} denoting numbers of non-availability intervals on particular machines, has been also used [10].)

Preemptive scheduling on parallel processors, as a classical scheduling problem, was analyzed in various settings. Ullman [19] was the first to study this problem from the complexity point of view, proving its strong NP-hardness for precedence constraints and an arbitrary number of identical, continuously available processors. Some easy cases include: tree-like precedence constrained tasks scheduled on identical processors or arbitrary graphs scheduled on two processors [16] and independent tasks scheduled on uniform [8] or unrelated [2,12] processors. Processors with non-availability intervals have been studied in [1,4,15,17,18]. In [4] the following problems have been analyzed: $P, \mathit{decreasing\ zig\text{-}zag}|p_j = 1, \mathit{outforest}|C_{\max}$, $P, \mathit{increasing\ zig\text{-}zag}|p_j = 1, \mathit{inforest}|C_{\max}$, $Pm, NC_{win}|\mathit{pmtn}, \mathit{tree}|C_{\max}$ and $P2, NC_{win}|\mathit{pmtn}, \mathit{prec}, r_j|C_{\max}$. Schmidt [18] demonstrated that a feasible preemptive schedule exists iff

$$\max_{k=1,\dots,m-1}\left\{\frac{\sum_{j=1}^{n} p_j}{\sum_{i=1}^{m} PC_i}, \frac{\sum_{j=1}^{k} p_j}{\sum_{i=1}^{k} PC_i}\right\} \leqslant 1,$$

where $p_1 \geqslant p_2 \geqslant \cdots \geqslant p_n$ for $P, \mathit{staircase}|\mathit{pmtn}|C_{\max}$ and the assumed processor capacities $PC_1 \geqslant PC_2 \geqslant \cdots \geqslant PC_m$, where PC_i is the total processing capacity of processor P_i for a given time horizon. Such a schedule can be constructed in $O(n + m \log m)$ time with the number of preemptions being proportional to the number of intervals of availability. Liu and Sanlaville [15] studied various other patterns of processor availability which they called profiles. They proved that the following problems can be solved in polynomial time: $P, NC_{win}|p_j = 1, \mathit{chains}|C_{\max}$, $P2, NC_{win}|p_j = 1, \mathit{prec}|C_{\max}$, $P, \mathit{decreasing\ zig\text{-}zag}|\mathit{pmtn}, \mathit{outforest}|C_{\max}$, $P, \mathit{increasing}$

$zig\text{-}zag|pmtn,\ inforest|C_{\max}$, $P, NC_{win}|pmtn, chains|C_{\max}$, $P2, NC_{win}|pmtn, prec|C_{\max}$, P, *increasing* $zig\text{-}zag|p_j = 1, inforest|L_{\max}$, $P2, NC_{win}|p_j = 1, prec|L_{\max}$, P, *increasing zig-zag*$|pmtn, inforest|L_{\max}$, $P2, NC_{win}|pmtn, prec|L_{\max}$.

In [1] the problem has been studied in a context of multiprocessor tasks.

In [10,13] problems of scheduling on dedicated processors with non-availability intervals have been analyzed.

The rest of this work is organized as follows. In Section 2 it will be shown that the problem with a staircase pattern of availability is NP-hard in the strong sense for intrees. The case of chains is considered in Section 3. In the case of identical processors with a staircase pattern of availability, the problem can be solved in low-order polynomial time for $C_{\max}$ criterion and a linear programming approach is required for $L_{\max}$ criterion. On the other hand, unrelated processors result in strong NP-hardness of the problem, even in case of two processors. In Section 4 network flow and linear programming approaches will be proposed for independent tasks scheduled on, respectively, uniform and unrelated processors with arbitrary patterns of availability for schedule length and maximum lateness criteria.

2. Trees and staircase pattern

Let us first consider the case of a constant pattern. The problem can be solved by the algorithm of Muntz and Coffman [16]. The algorithm has time complexity of $O(n^2)$ and generates $O(mn)$ preemptions.

Now let us assume we have intervals of non-availability and the pattern is a staircase. Unfortunately, this problem in the case of intrees can be shown to be NP-hard in the strong sense.

Theorem 1. *$P, staircase|pmtn, intree|C_{\max}$ is NP-hard in the strong sense.*

Proof. The transformation is from 3-*partition* [6]. Assume the following instance in this problem:

A set A of $3s$ positive integers $\{a_1, a_2, \ldots, a_{3s}\}$, and an integer bound B such that $B/4 < a_i < B/2$, $i = 1, 2, \ldots, 3s$.

Can A be partitioned into s disjoint subsets $A_1, A_2, \ldots, A_s$, such that $\sum_{a_i \in A_j} a_i = B$ for $j = 1, \ldots, s$?

Define the following instance of the non-availability with trees (NAT) problem for the above 3-partition instance:

tree i for a_i, $i = 1, \ldots, 3s$
leaves of i, $|L_i| = Wa_i$, where $W = 3s + 1$
stem of i, $|S_i| = wa_i$, where $w = 3s$
all jobs have unit processing time (see Fig. 2)
processor system as shown in Fig. 3 with $m = WB$ processors
$C^* = swB + s$ – schedule length
Does there exist a schedule of length C^*?

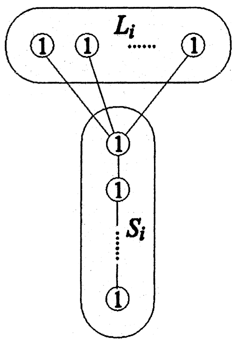

Fig. 2. Tree i for a_i.

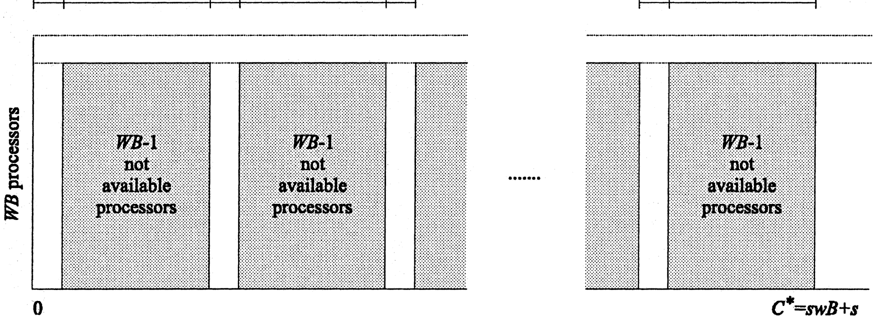

Fig. 3. Processor availability pattern.

$\Rightarrow$ Given 3-partition of I one can easily obtain a feasible schedule with $C_{\max} = C^*$ (cf. Fig. 3). We will omit details.

$\Leftarrow$ Let S^* be a feasible schedule of length C^*. We will show how to obtain a 3-partition of I. Consider set

$$A_s = \{\mathit{all\ leaves\ either\ completed\ or\ preempted\ at}\ w_s = (s-1)wB + s\}.$$

Since there is no idle time in S^*, we have

$$|A_s| \geqslant WB - 3s. \tag{1}$$

(From definition A_q contains only leaves. $3s$ is the maximum number of stems. Each task can be processed by at most one machine at a time. Since there is no idle time in S^* each of WB machines must process any task immediately before w_s. At most $3s$ of these tasks come from stems, so remaining are at least $WB - 3s$ tasks which must be leaves.)

Furthermore, without loss of generality we have

$$A_s = \bigcup_{i=1}^{k} (L_i \cap A_s) \tag{2}$$

for some $k \geqslant 1$. First, we prove that $k = 3$. The proof is by contradiction. Suppose $k > 3$, then $B < \sum_{i=1}^{k} a_i$ and the stems of trees $1, \ldots, k$ could not be completed by C^*. Now suppose $k < 3$, then by (1) and (2) we have

$$WB - 3s \leqslant |A_s| = \sum_{i=1}^{k} |L_i \cap A_s| \leqslant \sum_{i=1}^{k} |L_i| = W \sum_{i=1}^{k} a_i < WB - W$$

and we get a contradiction since $W = 3s + 1$.

Thus $k = 3$, and

$$A_s = \bigcup_{i=1}^{3} (L_i \cap A_s). \tag{3}$$

We now show that $a_1 + a_2 + a_3 = B$.

Again by contradiction, suppose $a_1 + a_2 + a_3 < B$, then by (1) and (3)

$$WB - 3s \leqslant |A_s| = \sum_{i=1}^{3} |L_i \cap A_s| < \sum_{i=1}^{3} |L_i| = W \sum_{i=1}^{3} a_i \leqslant WB - W,$$

which is a contradiction since $W = 3s + 1$.

Now suppose that $a_1 + a_2 + a_3 > B$. Then the stems of the trees could not be completed by C^*. Thus,

$$a_1 + a_2 + a_3 = B. \tag{4}$$

By (4) processor 1 processes jobs from $S_1 \cup S_2 \cup S_3$ only between w_q and C^*, therefore

$$A_s = \{\textit{all leaves either completed or preempted in } \omega_1 = [(s-1)wB + s - 1, w_s]\}.$$

To finish the proof it remains to show that only the leaves from $L_1 \cup L_2 \cup L_3$ are processed on all processors in interval ω_1. Actually, this may not hold for any feasible schedule with C^* (S^* in particular) then, however, we show how to convert S^* into a schedule for which this condition is met.

Consider set

$$A_{s-1} = \{\textit{all leaves either completed or preempted at } w_{s-1} = (s-2)wB + s - 1\} \setminus A_s.$$

Without loss of generality, we have

$$A_{s-1} = \bigcup_{i=4}^{k} (L_i \cap A_{s-1}). \tag{5}$$

Using similar arguments as above for A_s we can show that $k = 6$, and

$$a_4 + a_5 + a_6 \leqslant B \tag{6}$$

and

$$A_{s-1} = \{\textit{all leaves either completed or preempted in } \omega_2 = [(s-2)wB + s - 2, w_{s-1}]\} \setminus A_s. \tag{7}$$

Notice that there can be at most $3(s-1)$ stems processed in parallel in ω_1. Without loss of generality we may assume that the leaves from $L_1 \cup L_2 \cup L_3$ are processed only in ω_1 and ω_2 in S^*. However, since (6) and (7) hold, we can remove all stems from ω_1, if there are any, and replace them by the leaves from $L_1 \cup L_2 \cup L_3$. Thus, without loss of generality, S^* has only jobs from $\bigcup_{i=1}^{3} L_i \cup \bigcup_{i=1}^{3} S_i$ in interval $[(s-1)wB+(s-1), C^*]$, and $a_1+a_2+a_3=B$. Thus, we can 'cut off' interval $[(s-1)wB+(s-1), C^*]$ from S^* and continue by repeating the above arguments. □

3. Chains and staircase pattern

3.1. Minimizing maximum makespan – $P, staircase|chains, pmtn|C_{\max}$

Now let us assume we have processors with intervals of non-availability and tasks form N chains $J_1, J_2, \ldots, J_N$, $N \leqslant n$. In [15] it has been suggested to apply the algorithm of Muntz and Coffman [16] to change a task assignment if one of the following events occurs:
1. an assigned task is completed,
2. the priority of a task has changed, or
3. the availability of a processor has changed.

This approach results in $O(n \log n + nm)$ time complexity and in a number of preemptions generated being $O((n+m)^2 - nm)$. Below we will show how one can improve this result. First, let us prove the following theorem.

Theorem 2. *Preemptive scheduling of chains (on processors with limited availability) for minimizing schedule length can be solved by applying an algorithm for independent tasks.*

Proof. Each chain J_j will be represented by an independent task T_j. The processing time p_j of T_j equals the sum of the processing times of all tasks T_{ij} of J_j, i.e., $p_j = \sum p_{ij}$. Now, assume a feasible schedule is generated for the independent tasks problem. We will show that this schedule represents also a feasible schedule for chains. As each chain refers to an independent task, it is scheduled with all its processing requirements and it is never assigned at the same time to more than one processor. It remains to show that an optimal independent tasks schedule with length $C_{\max}(in)$ is also an optimal chain schedule with length $C_{\max}(ch)$ with $C_{\max}(in) = C_{\max}(ch)$. Assume there is an algorithm which generates a schedule $C_{\max}(ch)' < C_{\max}(ch)$ for the chain problem. As independent tasks can be regarded as chains where each chain consists only of a single task, this algorithm could have been applied also to the independent tasks problem. Then it would have generated also a schedule $C_{\max}(in)' < C_{\max}(in)$; a contradiction. □

Thus, we have the following corollary.

Corollary 1. *Preemptive scheduling of N chains on m identical processors with staircase pattern of availability can be solved in* $O(N + m \log m)$ *time by applying the algorithm by Schmidt* [18]. *The number of induced preemptions is at most* $q - 1$ *where q is the total number of intervals of availability of all processors.*

Unfortunately, a more complicated approach involving linear programming is required in case of $L_{\max}$ criterion. The method is described in the next section.

3.2. Minimizing maximum lateness – $P, NC_{win}|chains, pmtn, r_j|L_{\max}$

In this section, we present a polynomial algorithm for solving the problem of minimizing maximum lateness for chain precedence constraints between tasks, and each task has a ready time. The algorithm uses binary search procedure to find an interval $[L', L'']$ that encloses the optimum value of $L_{\max}$. In each such interval of $L_{\max}$ values the sequence of events (i.e., ready times, due-dates, changes of processor availability) is fixed, and there is a constant number of available processors. Thus a linear programming problem may be applied to solve the problem. As a result, one gets an answer whether or not $L^*_{\max}$ is located in the particular interval. Depending on the answer again a binary search procedure is activated and so on. Since the general procedure is well known, see e.g. [14], we focus here only on its crucial part, namely on the linear programming formulation. This linear program is defined as follows:

$L(j)$ is the number of tasks in chain J_j (thus, $n = \sum_{j=1}^{N} L(j)$); p_{ij}, d_{ij}, r_{ij} denote, respectively, processing time, due-date and release date of T_{ij}; $0 = e_0 < e_1 < \cdots < e_K$ is the list of different moments in $\{r_{ij}, d_{ij} : j = 1, \ldots, N,\ i = 1, \ldots, L(j)\} \cup \{t_l : l = 1, \ldots, q\}$; $p_j = \sum_{i=1}^{L(j)} p_{ij}$ is the processing time of chain J_j;

$$a_l = \begin{cases} 1 & \text{if } e_l \text{ is a due-date,} \\ 0 & \text{otherwise;} \end{cases}$$

$D_{ij} = \{l : r_{1j} \leqslant e_{l-1} + a_{l-1}L_{\max} \text{ and } e_l + a_l L_{\max} \leqslant d_{ij} + L_{\max}\}$;
$R_{ij} = \{l : r_{1j} \leqslant e_{l-1} + a_{l-1}L_{\max} \text{ and } e_l + a_l L_{\max} \leqslant r_{ij}\}$;
$D_j = \{l : r_{1j} \leqslant e_{l-1} + a_{l-1}L_{\max} \text{ and } e_l + a_l L_{\max} \leqslant d_{L(j),j} + L_{\max}\}$;
$y_j^{(l)}$ is the part of chain J_j processed in the interval $[e_{l-1} + a_{l-1}L_{\max}, e_l + a_l L_{\max}]$;
$m^{(l)}$ is the number of processors available in $[e_{l-1} + a_{l-1}L_{\max}, e_l + a_l L_{\max}]$

$$\min L_{\max}; \tag{8}$$

$$y_j^{(l)} \leqslant e_l - e_{l-1} + L_{\max}(a_l - a_{l-1}), \quad j = 1, \ldots, N, \quad l = 1, \ldots, K; \tag{9}$$

$$\sum_{j=1}^{n} y_j^{(l)} \leqslant (e_l - e_{l-1} + L_{\max}(a_l - a_{l-1}))m^{(l)}, \quad l = 1, \ldots, K; \tag{10}$$

$$\sum_{l=1}^{K} y_j^{(l)} = p_j, \quad j = 1, \ldots, N; \tag{11}$$

$$\sum_{l\in D_{ij}} y_j^{(l)} \geqslant \sum_{z=1}^{i} p_{zj}, \quad j=1,\ldots,N, \ i=1,\ldots,L(j); \tag{12}$$

$$\sum_{l\in R_{ij}} y_j^{(l)} \leqslant \sum_{z=1}^{i-1} p_{zj}, \quad i=1,\ldots,L(j), \ \ j=1,\ldots,N; \tag{13}$$

$$y_j^{(l)} = 0, \quad j=1,\ldots,N, \ \ l\notin D_j; \tag{14}$$

$$y_j^{(l)} \geqslant 0, \quad j=1,\ldots,N, \ \ l=1,\ldots,K; \tag{15}$$

$$L' \leqslant L_{\max} \leqslant L''. \tag{16}$$

Any schedule that satisfies (8)–(11), (14)–(16) is an optimal schedule for problem $P, NC_{win}|pmtn, r_j|L_{\max}$ with a considered interval of $L_{\max}$ values, where we set $r_j := r_{1j}$, $d_j := d_{L(j),j}$, and $p_j = \sum_{i=1}^{L(j)} p_{ij}$ for job j in this new problem. Let S be such a schedule. Consider job j that starts at c_S and completes at C_S in S. Let $f_{0j} = c_S < f_{1j} < \cdots < f_{L(j),j} = C_S$ be moments in schedule S such that exactly p_{ij} units of job j are processed in interval $[f_{i-1,j}, f_{ij}]$, $i=1,\ldots,L(j)$. If S meets (12), then $f_{ij} \leqslant d_{ij} + L_{\max}$ for $i=1,\ldots,L(j)$. If S meets (13), then $f_{i-1,j} \geqslant r_{ij}$, $i=1,\ldots,L(j)$. Consequently, if S meets both (12) and (13), then interval $[f_{i-1,j}, f_{ij}]$ falls inside of interval $[r_{ij}, d_{ij} + L_{\max}]$ for $i=1,\ldots,L(j)$. Therefore, by replacing the part of job j done in $[f_{i-1,j}, f_{ij}]$ in S by the ith task of chain J_j we get an optimal schedule for $P, NC_{win}|chains, pmtn, r_j|L_{\max}$ in the considered interval of $L_{\max}$ values.

Changing $L_{\max}$ may cause changes of the sequence of moments e_i when some $d_i + L_{\max}$ becomes equal to, respectively, some release date, starting or ending point of any interval of non-availability. There are at most $(n+q)n$ different intervals $[L', L'']$ for $L_{\max}$ in which the sequence of e_i does not change. Applying binary search procedure we must solve the linear program given by (8)–(16) $O(\log n + \log q)$ times.

In the next section, we will show that this approach cannot be generalized to cover the case of unrelated processors because the problem starts to be strongly NP-hard.

3.3. Chains and unrelated processors – $R2|pmtn, chain|C_{\max}$

In this section, we show that the case of unrelated processors and precedence constraints is strongly NP-hard even for chains and continuously available processors. This result sets a limit on polynomial time solvability (unless $P = NP$) of preemptable scheduling problems with chains.

Theorem 3. *Problem $R2|pmtn, chain|C_{\max}$ is strongly NP-hard.*

Proof. We prove strong NP-hardness of the problem by reduction from 3-partition. The definition of 3-partition is given in the proof of Theorem 1. The instance of $R2|pmtn, chain|C_{\max}$ is constructed as follows:

$n = 8s;$

$p_{1j} = 2sB + 1,\ p_{2j} = B$ for $j = 1, \ldots, s;$

$p_{1j} = B,\ p_{2j} = 2sB + 1$ for $j = s + 1, \ldots, 2s;$

$p_{1j} = a_{j-2s},\ p_{2j} = 2sB + 1$ for $j = 2s + 1, \ldots, 5s;$

$p_{1j} = 2sB + 1,\ p_{2j} = a_{j-5s}$ for $j = 5s + 1, \ldots, 8s.$

Tasks form $1 + 3s$ chains:

$T_1 \prec T_{s+1} \prec \cdots \prec T_j \prec T_{s+j} \prec \cdots \prec T_{2s}$

$T_j \prec T_{j+3s}$ for $j = 2s + 1, \ldots, 5s.$

We ask whether a schedule of length $C^* = 2sB$ exists.

Suppose the answer to 3-partition is positive. Then a feasible schedule is presented in Fig. 4. Conversely, let us assume that a feasible schedule of length at most $2sB$ exists. Observe that due to the selection of the processing times and the schedule length tasks are practically preallocated to processors. For example T_1 must be executed on P_2, T_{s+1} on P_1, T_{2s+1} on P_1, etc. Otherwise no schedule of length at most $2sB$ would exist. Hence, tasks $T_1 \prec T_{s+1} \prec \cdots T_j \prec T_{s+j} \cdots \prec T_{2s}$ must be executed consecutively alternating the processors. This creates free intervals of length B on processors P_1, P_2 alternatingly. The feasible schedule admits no idle times. Consider the first free intervals $[0, B]$ on P_1 and interval $[B, 2B]$ on P_2. Before their starting, the tasks in interval $[B, 2B]$ must have their predecessors completed. Since no idle time is allowed the sum of processing times in interval $[0, B]$ on P_1 must be B. As $B/4 < a_j < B/2$, at most three tasks can be completed in $[0, B]$ on P_1. Their processing requirement can be at most B. Suppose processing time of the tasks completed in interval $[0, B]$ is less than B. To avoid idle time more than three tasks can be started on P_1 in $[0, B]$, but cannot be finished. Then, also successors of the tasks completed in $[0, B]$ on P_1 have smaller total processing time on P_2 in $[B, 2B]$, and an idle time appears. We conclude that tasks completed in $[0, B]$ must have processing time also at least B. Hence exactly three tasks of length B must be completed in $[0, B]$ on P_1. The same applies to their successors in $[B, 2B]$ on P_2. The elements of set A in 3-partition corresponding to the three tasks in $[0, B]$ form set A_1. Analogous reasoning can be applied to intervals $[2Bi, 2Bi + B]$ on P_1 and $[2Bi + B, 2B(i + 1)]$ where $i = 0, \ldots, s - 1$. This completes the proof. □

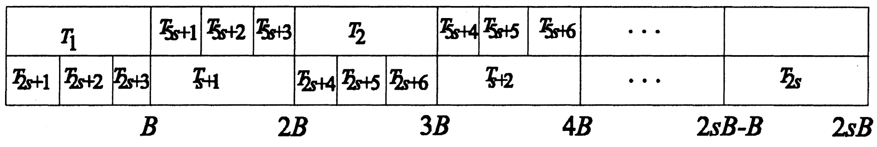

Fig. 4. Illustration to the proof of Theorem 3.

4. Independent tasks

In this section, we will consider independent tasks scheduled on, respectively, uniform processors (Section 4.1) and unrelated processors (Section 4.2) with non-availability intervals. In the first case, minimization of $L_{\max}$ will be solved by a combined strategy: binary search and network flow. In the second case, the network flow approach will be replaced by the so-called *two-phase* method [3].

4.1. Uniform processors

The problem to be analyzed is $Q, staircase|pmtn, r_j|L_{\max}$. We assume that there are k different processor types. Processors of type i have speed $b_i > b_{i+1}$ $(i = 1, \dots, k-1)$. A staircase pattern of processor availability is also assumed. Processors are available in q different intervals, interval l has $m_i^{(l)} \geqslant 0$ $(i = 1, \dots, k)$ processors of type i. Each task T_j, is also available for processing in some restricted interval of time. This interval is determined by the task ready time r_j and due-date d_j (task must not be executed after $d_j + L_{\max}$ which will be called T_j's deadline). The optimality criterion is maximum lateness.

The algorithm we propose for this problem is an extension of the technique proposed in [5,11,20]. The problem is reduced to a sequence of the network flow problems. We describe the construction of the network first. Then we consider the complexity of this algorithm.

The network flow problem solves problem $Q, staircase|pmtn, r_j, d_j|–$, i.e., it finds a feasible schedule provided that one exists. In our case, a deadline should be calculated as $d_j + L_{\max}$ for task T_j and some trial value of the maximum lateness. The network flow algorithm is used then to find the optimal value of the maximum lateness. Suppose the test value of the maximum lateness is given and is equal to $L_{\max}$. This defines $K = 2n + q$ events in the task and processor system. The events are of the following type: ready time of some task, disappearing of some task T_j from the system at time $d_j + L_{\max}$, a change of processor availability at some moment t_l. However, assuming fixed $L_{\max}$, the sequence of the events is also fixed. Let e_l be the time instant at which the lth event takes place, and $\tau_l = e_{l+1} - e_l$ the length of the lth interval between two consecutive events.

The network $G(V,A)$ (cf. Fig. 5) has a source node S_1, and terminal node S_2. Between these two, two layers of nodes are inserted. The first layer consists of n nodes T_j representing the tasks. The second layer has Kk nodes w_{il} $i = 1, \dots, k$, $l = 1, \dots, K$, representing different ranges of processor speeds in the interval $[e_l, e_{l+1}]$. The source node S_1 is connected by an arc of capacity p_j with the node representing task T_j. The capacities of these arcs guarantee that no task receives more processing than required. Nodes representing tasks which can be executed in the interval $[e_l, e_{l+1}]$ are connected with nodes w_{il} $(i = 1, \dots, k)$ by edges with capacity $(b_i - b_{i+1})\tau_l$. These arcs guarantee that no task is processed on any processor longer than possible in the interval of length τ_l. Interval-speed range nodes w_{il} are connected with the terminus S_2 by edges of capacity $(b_i - b_{i+1})\tau_l \sum_{z=1}^{i} m_z^{(l)}$. The constraints imposed by the capacities of the arcs heading to and leaving nodes

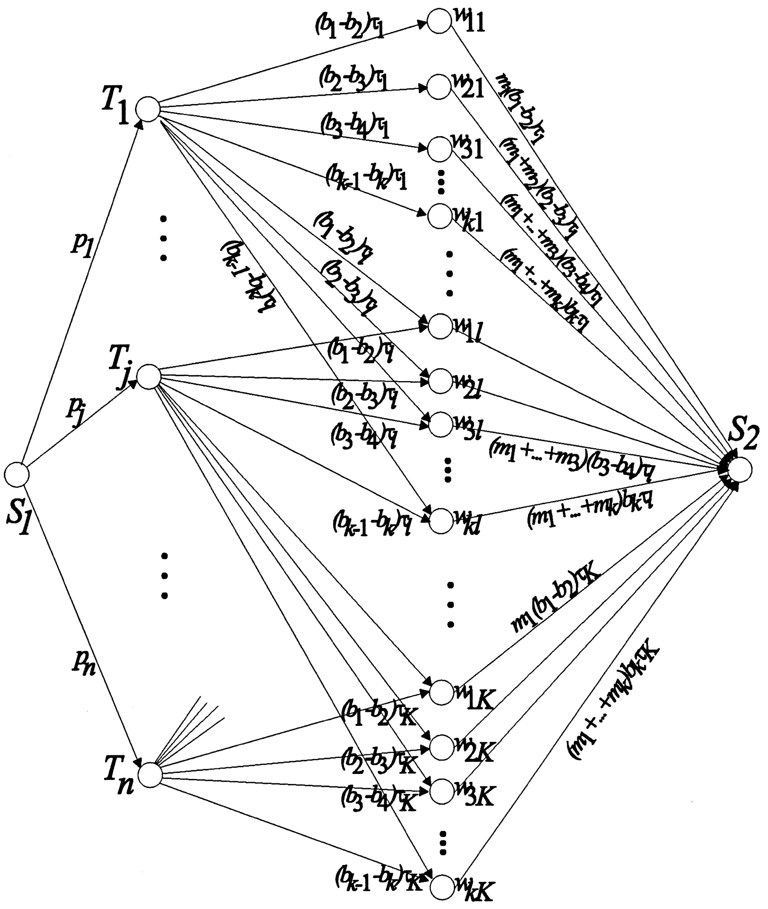

Fig. 5. Network for problem $Q, staircase|pmtn, r_j|L_{\max}$.

w_{il} can be understood as equivalent to the processing capacity constraints imposed in [8] to solve problem $Q|pmtn|C_{\max}$. The maximum flow can be found in $O((n+q)^3)$ time. When the edges joining the source with the task nodes are saturated, then a feasible schedule exists.

The schedule can be constructed on an interval by interval basis. A partial schedule in the interval can be built using the algorithm proposed in [8]. Note that processing capacity constraints imposed in the above paper are fulfilled: it is required there that the longest task should not be longer than the processing capacity of the fastest processor. Here, no task receives more processing in any interval than the capacity of the fastest processor. In [8], it is required that the two longest tasks should not have bigger total processing time than the processing capacity of the two fastest processors. Here, no pair of tasks receives more processing in any interval than the capacity of the two fastest processors. Moreover, in the approach mentioned above, it is required that the three, four, etc., longest tasks should not have bigger total processing time than the processing capacity of the three, four, etc., fastest processors. Here, no three, four, etc., tasks receive more processing in any interval than the capacity of the three, four, etc., fastest processors. More generally, the j longest tasks do not consume more processing time than provided by the j fastest processors in interval l, for $j = 1, \ldots, \sum_{i=1}^{k} m_i^{(l)}$. Finally, total processing requirement assigned to any interval does not exceed processing capacity in the interval due to the capacity constraints on the arcs leaving nodes w_{il} $(i = 1, \ldots, k)$.

Now, let us analyze the algorithm finding the optimum value of the maximum lateness $L_{\max}^*$. With changing value of $L_{\max}$, the sequence of events in the system changes when for some pair of tasks T_i, T_j $r_i = d_j + L_{\max}$. Changing $L_{\max}$ beyond such a value decides whether the pair of tasks can be executed together. The sequence also changes with $L_{\max}$ when for some task T_j its deadline $d_j + L_{\max}$ passes from one interval of processor availability to another. This determines whether a task can be executed on some set of processors or not. Hence, there are $O(n(n+q))$ intervals of $L_{\max}$ where the sequence of events is constant. The network flow algorithm must be called $O(\log n + \log q)$ times in a binary search fashion to determine the interval containing $L_{\max}^*$. Next, the number o of additional calls to the network flow algorithm can be bounded in a way which bears some similarity to the method proposed in 11,20. Yet, due to a different structure of the network and unequal speeds at the processors these results are not immediately applicable here.

Note that after fixing the sequence of the events, the structure of network G also remains fixed, only the values of arc capacities change. Suppose that we already fixed the sequence of events, and $L_{\max}^1$ is the biggest value of the maximum lateness considered in the previous binary search, for which a feasible solution does not exist. For $L_{\max}^1$ the maximum flow in G is $\phi_1 < \phi = \sum_{j=1}^{n} p_j$, where ϕ is the desired value of flow. Now we can find the cut, i.e., the set of arcs with minimum capacity which is bounding the maximum flow. Let us denote by n_l the number of tasks whose processing requirements were not satisfied and which can be processed in interval $[e_l, e_{l+1}]$.

With increasing of $L_{\max}^1$ by δ, the processing capacity of all arcs (w_{il}, S_2) $(i = 1, \ldots, k)$ for some interval $[e_l, e_{l+1}]$ increases by $\delta \sum_{i=1}^{k}(m_i^{(l)} b_i)$. Hence, the maximum possible increase of the capacity in the cut is $\delta \sum_{l=1}^{K} \sum_{i=1}^{k}(m_i^{(l)} b_i)$. However, this increase is possible only if in each interval $[e_l, e_{l+1}]$ the number of tasks which can exploit this increase is sufficiently big. Strictly saying, $n_l \geqslant \sum_{z=1}^{k} m_z^{(l)}$. On the other hand, task T_j which processing requirement is not satisfied can forward additional flow to all the speed ranges of interval l via all arcs (T_j, w_{zl}). Consuming at most $\delta b'_{l1}$ units of the flow, where b'_{lf} is the speed of the fth fastest processor in interval l. Hence, the minimum increase of the cut capacity is $\delta \min_{1 \leqslant l \leqslant K}\{b_{l1}\}$. Furthermore, $2, 3, \ldots, j$ tasks whose processing requirement is not satisfied can forward up to $\delta \sum_{f=1}^{j} b'_{lf}$ units of the flow. In such a case, interval l increases the cut capacity by $\delta \sum_{f=1}^{j} b'_{lf}$. The actual increase is $\mu^1 \delta$ where μ^1 is an integer multiplier satisfying $\min_{1 \leqslant l \leqslant K}\{b'_{l1}\} \leqslant \mu^1 \leqslant \sum_{l=1}^{K} \sum_{i=1}^{k}(m_i^{(l)} b_i)$, and reflecting which tasks can be executed in which interval, which intervals increase their capacity when $L_{\max}$ increases, and which of these combinations have arcs in the cut.

We set $\delta = (\phi - \phi_1)/\mu^1$, $L^2_{\max} := L^1_{\max} + \delta$, and calculate the maximum flow ϕ_2 by the same method (only with capacities changed). If $\phi \neq \phi_2$ we repeat the procedure by calculating $L^{g+1}_{\max} = L^g_{\max} + (\phi - \phi_g)/\mu^g$ for $g = 1, \dots, o$ until we get $\phi = \phi_o$.

Let us analyze the number of iterations that will be performed in the worst case. The problem is to use a right value of the multiplier μ^g, i.e., to find the extension of $L_{\max}$ which gives an exact value of the optimum maximum lateness. The extension of maximum lateness calculated for some value μ^g of the multiplier increases the flow to at least ϕ for all cuts with multipliers greater than μ^g. By binary search over $\sum_{l=1}^{K} \sum_{i=1}^{k} (m_i^{(l)} b_i)$ (integer) values of multipliers one can find the right extension, i.e., the required multiplier of the cut, at which the $L^*_{\max}$ is attained. Thus, the number o of additional calls to the network flow algorithm is $O(\log n + \log q + \log m + \log \max\{b_i\})$, and total algorithm complexity is $O((n+q)^3(\log n + \log q + \log m + \log \max\{b_i\}))$.

4.2. Unrelated processors

In this section, we consider parallel unrelated processors with an arbitrary number of non-availability intervals on each of them. The intervals do not form any particular profile of processor availability. First, we will solve the problem $R, NC_{win}|pmtn|C_{\max}$. This problem can be solved by a slight modification of the two-phase method [3]. In the first phase, a linear programming problem is solved and we use basically the same denotation as in Section 3.2 but now we have independent tasks and unrelated processors so p_{ij} denotes the processing time of task T_j on processor P_i. We may also assume that the moments when availability of processors changes $(t_0, t_1, \dots, t_q)$ are the same as boundaries of intervals in the schedule $(e_0, e_1, \dots, e_K)$, except for the last one, and we have $e_K = C_{\max}$ and $K = q + 1$. Other denotations are as follows:

$x_{ij}^{(l)}$ is the part of task T_j processed on processor P_i in the interval $[e_{l-1}, e_l]$, and

$x_{ij}^{(l)} \in [0, 1]$, $i = 1, 2, \dots, m$; $j = 1, 2, \dots, n$; $l = 1, 2, \dots, K$,

$x_{ij}^{(l)} = 0$ for all l for which processor P_i is non-available in the interval $[e_{l-1}, e_l]$.

Now, we have the following LP problem:

$$\min C_{\max} \tag{17}$$

subject to

$$\sum_{i=1}^{m} p_{ij} x_{ij}^{(l)} \leqslant e_l - e_{l-1}, \quad j = 1, \dots, n, \ \ l = 1, \dots, K; \tag{18}$$

$$\sum_{j=1}^{n} p_{ij} x_{ij}^{(l)} \leqslant e_l - e_{l-1}, \quad i = 1, \dots, m, \ \ l = 1, \dots, K; \tag{19}$$

$$\sum_{i=1}^{m} \sum_{l=1}^{K} x_{ij}^{(l)} = 1, \quad j = 1, \dots, n. \tag{20}$$

Inequalities (18) guarantee that the sum of parts of any task processed in a given interval on all processors does not exceed the length of this interval. Similarly, inequalities (19) guarantee that the sum of parts of all tasks processed on any processor in a given interval is not greater than the length of the interval. Eq. (20) ensure that all tasks are fully executed.

After applying the first phase one gets an optimal assignment of tasks to intervals $[e_{l-1}, e_l]$, $l = 1, \dots, K$. Notice that an order of task parts produced by the first phase does not necessarily constitute a feasible schedule because parts of the same task may overlap. To obtain a feasible solution one must shift task parts within each interval to eliminate cases of overlapped processing of parts of the same task. In the second phase, this problem is solved separately for each interval. We do not present this method here and refer the interested reader to [3].

Now, we will show how a similar approach may be applied to problem $R, NC_{win}|pmtn, r_j|L_{\max}$. The problem can be solved by an iterative method which is a combination of a binary search procedure and the two-phase method. Changing $L_{\max}$ may cause changes of a sequence of moments e_l when some $d_j + L_{\max}$ becomes equal to, respectively, some release date, starting or ending point of any non-availability interval. There are at most $(n+q)n$ intervals $[L', L'']$ of $L_{\max}$ value where the sequence of e_l's does not change. The binary search procedure is used to find an interval of $L_{\max}$ values containing an optimal value of this criterion. In each stage of this procedure only the first stage of the two-phase method is applied. Basically, we will use the same denotation as for problem $R, NC_{win}|pmtn|C_{\max}$ with additional items:

$$a_l = \begin{cases} 1 & \text{if } e_l \text{ is a due-date} \\ 0 & \text{otherwise,} \end{cases}$$
$$A_j = \{l : e_{l-1} + a_{l-1}L_{\max} \geqslant r_j \text{ and } e_l + a_l L_{\max} \leqslant d_j + L_{\max}\},$$
$$B_l = \{j : e_{l-1} + a_{l-1}L_{\max} \geqslant r_j \text{ and } e_l + a_l L_{\max} \leqslant d_j + L_{\max}\},$$

Now, we have the following set of LP problems:

$$\min L_{\max} \tag{21}$$

subject to

$$\sum_{i=1}^{m} p_{ij} x_{ij}^{(l)} \leqslant e_l - e_{l-1} + L_{\max}(a_l - a_{l-1}), \quad j = 1, \dots, n, \quad l \in A_j \setminus \{0\}; \tag{22}$$

$$\sum_{j \in B_l} p_{ij} x_{ij}^{(l)} \leqslant e_l - e_{l-1} + L_{\max}(a_l - a_{l-1}), \quad i = 1, \dots, m, \quad l = 1, \dots, K; \tag{23}$$

$$L' \leqslant L_{\max} \leqslant L''; \tag{24}$$

$$\sum_{i=1}^{m} \sum_{l \in A_j \setminus \{0\}} x_{ij}^{(l)} = 1, \quad j = 1, \dots, n. \tag{25}$$

In the above formulation, inequalities (22) and (23) guarantee that the sum of task parts processed in each interval does not exceed its length. Inequality (24) ensures that the sequence of moments e_i does not change. Inequality (25) guarantees that all tasks are fully executed.

Next, the binary search procedure is applied for finding a proper interval $[L', L'']$.

The second phase of the two-phase method is applied in the last stage of the binary search procedure and gives an optimal schedule which minimizes maximum lateness.

5. Conclusions

In this paper, we have analyzed problems of scheduling preemptable tasks on parallel processors with non-availability periods. It has been shown that such a problem is strongly NP-hard in case of intree precedence constraints, identical processors and the makespan criterion. On the other hand, the problem with chain-like precedence constraints can be solved by the algorithm developed for independent tasks. On the other hand, a polynomial time algorithm based on linear programming has been proposed for the maximum lateness criterion. Moreover, algorithms based on network flows and the two-phase method have been proposed for problems with independent tasks scheduled on, respectively, uniform and unrelated processors subject to $C_{\max}$ and $L_{\max}$ criteria.

Acknowledgements

The authors are grateful to the anonymous referees for their helpful comments. This research has been partially supported by the NATO Collaborative Research Grant CRG 950438 and by CRIT2 project.

References

[1] L. Bianco, J. Błażewicz, P. Dell'Olmo, M. Drozdowski, Preemptive multiprocessor task scheduling with release times and time windows, Annals of Operations Research 70 (1997) 43–55.
[2] J. Błażewicz, W. Cellary, R. Słowiński, J. Węglarz, Deterministyczne problemy szeregowania zadan na rownoleglych procesorach, Cz. I. Zbiory zadan zaleznych, Podstawy sterowania 6 (1976) 155–178.
[3] J. Błażewicz, K. Ecker, E. Pesch, G. Schmidt, J. Węglarz, Scheduling Computer and Manufacturing Processes, Springer, Berlin, 1996.
[4] D. Dolev, M. Warmuth, Scheduling flat graphs, SIAM Journal on Computing 14 (3) (1985) 638–657.
[5] A. Federgruen, H. Groenevelt, Preemptive scheduling of uniform processors by ordinary network flow techniques, Management Science 32 (1986) 341–349.
[6] M.R. Garey, D.S. Johnson, Computers and Intractability, Freeman, San Francisco, 1979.
[7] R.L. Graham, E.L. Lawler, J.K. Lenstra, A.H.G. Rinnoy Kan, Optimization and approximation in deterministic sequencing and scheduling: a survey, Annals of Discrete Mathematics 5 (1979) 287–326.
[8] T. Gonzalez, S. Sahni, Preemptive scheduling of uniform processor systems, Journal of the ACM 25 (1978) 92–101.
[9] K.S. Hong, J.Y.-T. Leung, On-line scheduling of real-time tasks, IEEE Transactions on Computers 41 (10) (1992) 1326–1331.

[10] W. Kubiak, J. Błażewicz, P. Formanowicz, J. Breit, G. Schmidt, Flow shops with limited machine availability, submitted for publication, 1997.
[11] J. Labetoulle, E.L. Lawler, J.K. Lenstra, A.H.G. Rinnoy Kan, Preemptive scheduling of uniform machines subject to release dates, in: W.R. Pulleyblank (Ed.), Progress in Combinatorial Optimization, Academic Press, New York, 1984, pp. 245–261.
[12] E.L. Lawler, J. Labetoulle, Preemptive scheduling of unrelated parallel processors by linear programming, Journal of the ACM 25 (1978) 612–619.
[13] C.-Y. Lee, Minimizing the makespan in the two-machine flowshop scheduling problem with an availability constraint, reaserch report, Department of Industrial and Systems Engineering, University of Florida, 1995.
[14] C.L. Liu, Introduction to Combinatorial Mathematics, McGraw-Hill, New York, 1968.
[15] Z. Liu, E. Sanlaville, Preemptive scheduling with variable profile, precedence constraints and due dates, Discrete Applied Mathematics 58 (1995) 253–280.
[16] R. Muntz, E.G. Coffman, Preemptive scheduling of real-time tasks on multiprocessor systems, Journal of the ACM 17 (1970) 324–338.
[17] E. Sanlaville, G. Schmidt, Machine scheduling with availability constraints, Acta Informatica 35 (1998) 795–811.
[18] G. Schmidt, Scheduling on semi-identical processors, Zeitschrift für Operations Research A 28 (1984) 153–162.
[19] J.D. Ullman, NP-complete scheduling problems, Journal of Computer System Sciences 10 (1975) 384–393.
[20] V.G. Vizing, Minimization of the maximum delay in servicing systems with interruption, USSR Computational Mathematics and Mathematical Physics 22 (3) (1982) 227–233.