# New Applications of the Muntz and Coffman Algorithm

Maciej Drozdowski

**Instytut Informatyki**
**Politechnika Poznańska**
**ul. Piotrowo 3a**
**60-965 Poznań**
**Poland**
**telephone: (4861)6652124, (4861)6652366**
**fax: (4861)8771525**
**email: maciej_d@sol.put.poznan.pl**

# New Applications of the Muntz and Coffman Algorithm

Maciej Drozdowski*

**Abstract**

Muntz and Coffman proposed an algorithm to solve the problem of scheduling preemptable tasks either with arbitrary precedences on two processors, or tasks with tree-like precedences on an arbitrary number of processors, for the schedule length criterion. In this work, we demonstrate that this well-known algorithm has interesting features which extend its application to many other scheduling problems. Three deterministic scheduling problems for preemptable tasks are considered. Though these problems have diverse formulations, they have at least one thing in common: Basically their optimization algorithms boil down to the Muntz-Coffman algorithm. The foundations of the Muntz-Coffman algorithm versatility are established.

**Keywords:** Deterministic scheduling, preemptive scheduling.

*Instytut Informatyki, Politechnika Poznańska, ul. Piotrowo 3a, 60-965 Poznań, Poland. The research has been partially supported by a KBN grant.

# 1 Introduction

At the end of the 60's Muntz and Coffman proposed an algorithm that solves two problems of scheduling preemptable tasks under the schedule length criterion: The problem of scheduling tasks with arbitrary precedences on two parallel identical processors [16], and the problem of scheduling tasks with tree-like precedence constraints on an arbitrary number of parallel identical processors [17]. This algorithm has features which make it applicable to many other deterministic scheduling problems with preemptable tasks. Such problems include, e.g., tasks available for processing in restricted time intervals, processors available in time windows of availability, tasks which can be executed on more than one processor at the same time, the lateness criterion, etc. In this work we discuss such problems and their optimization algorithms derived from the Muntz and Coffman (MC) algorithm. In Section 2 the classical form of the MC algorithm is discussed. In Section 3 three other problems solvable by algorithms based on MC are presented. Conclusions are given in the last section.

In the remainder of this section, we introduce the following notation and terminology to be used in this work. The processing environment, e.g. computer system, $\mathcal{P} = \{P_1, \ldots, P_m\}$ consists of $m$ parallel processors. Processors are either *identical* or *uniform*. In the latter case processors differ in their speeds $s_i$. Without loss of generality we assume that $s_1 \geq s_2 \geq \ldots \geq s_m$. Processors may be available in restricted time intervals (some processors may be blocked). These intervals will be called *time windows* (or simply windows). There are $p$ time windows, each identified by a time interval $[b_l, e_l]$, for $l = 1, \ldots, p$, and the number $m_l$ of processors available. We assume that neighboring windows differ in the set of available processors. The sets of available processors are nonempty (completely blocked windows are skipped). In the case of uniform processors we denote by $s_{il}$ the speed of $i$-th processor available in window $l$. In each window $l$ processors are ordered such that $s_{1l} \geq s_{2l} \geq \ldots s_{m_l l}$.

Task set $\mathcal{T} = \{T_1, \ldots, T_n\}$ has $n$ elements which represent work to be performed. Precedence constraints may exist among tasks, e.g. when the results of one task are the input data for some other task. $T_i \prec T_j$ will mean that task $T_i$ must be completed before the processing of task $T_j$ can start. The set of all such relations in the task system constitute a precedence constraint graph (PCG). Tasks may arrive to the system at different times or may have desired time limits by which they should be completed. Therefore, a task is

characterized by a ready time and a duedate. We will denote the ready time of task $T_j$ by $r_j$ for $j = 1, \ldots, n$. Analogously, $d_j$ denotes the duedate of $T_j$. In this work we assume that tasks are *preemptable*. When preemptions are not allowed all tasks must be executed continuously on the same processor from the beginning till the very end. Preemptability means that each task can be suspended, and restarted later (possibly on a different processor) without additional overheads. Processing time of $T_j$ is denoted $t_j$. We say that a task is *ready* when it has not been completed, it has arrived, and all its predecessors are finished. In parallel computer systems it may be allowed, or required, to execute a task on many processors simultaneously. Tasks of this kind are called multiprocessor tasks [1, 3, 5, 18] (also parallel tasks or malleable tasks). When $T_j$ must be executed on some number of processors in parallel, we will denote this number by $size_j$. When $T_j$ can be executed on some number of processors in parallel, but the number of the processors is not fixed in advance, we will denote by $\delta_j$ the maximum number of processors that $T_j$ is allowed to use. In this last case, processing time depends on the number of assigned processors. It is assumed that tasks have linear speedup. Linear speedup means that execution of $T_j$ on $m'$ processors lasts $t_j/m'$ units of time. Moreover, when such a task is preempted the amount of performed work is equal to the total area that the task occupied, i.e. duration of the execution interval multiplied by the number of processors used. Task $T_j$ is considered finished when the sum of all such areas is equal to $t_j$.

Let $c_j$ denote the completion time of task $T_j$. Two optimality criteria will be analyzed: the schedule length $C_{max} = \max_j\{c_j\}$, and the maximum lateness $L_{max} = \max_j\{c_j - d_j\}$,

The three-field notation introduced in [11, 18, 7] will be used to refer to problems. In particular, *win* in the processor field means that processors have windows of availability. The word $size_j$ in the task field means that tasks require several processors simultaneously, with the number of required processors fixed. $spdp - lin - \delta_j$ means that task $T_j$ has linear speedup when assigned up to $\delta_j$ processors ($j = 1, \ldots, n$), and that using more than $\delta_j$ processors is forbidden.

# 2 The Muntz-Coffman (MC) algorithm

Originally, the MC algorithm was proposed to solve problems $P2|pmtn, prec|C_{max}$, and $P|pmtn, in - tree|C_{max}$ [16, 17]. Out-tree cases can be reduced

to in-tree cases by reading schedules backwards. The particular structure of precedence constraints and processor numbers result from the fact that the MC algorithm can be considered as a continuous version of the algorithms given by Hu [14] to solve the problem $P|p_j = 1, tree|C_{max}$, and by Coffman and Graham [4] for the problem $P2|p_j = 1, prec|C_{max}$, i.e. the same problems with nonpreemptable unit-execution-time tasks.

The MC algorithm uses two concepts: task level, and processing capability. *Task level* (or height) is the length of the longest path in the PCG starting at the considered task and finishing at its furthest successor. Processing time of the task itself is included in the level. Observe that the level changes while processing a task. For example, a task with level 0 is finished. Intuitively, height can be thought of as an indicator of the task urgency. We will denote the level (or height) of task $T_j$ by $h(j)$. For simplicity of presentation, we assume that $h(1) \geq h(2) \geq \ldots \geq h(n)$.

*Processing capability* $\beta_j$ is a fraction of the total processing power assigned to task $T_j$ in some time interval. For example, a task with $\beta_j = 0.5$ in an interval of length 1 is executed 0.5 units of time.

In the pseudocode of the MC algorithm we denote by $\overline{\beta} = [\beta_1, \ldots, \beta_n]$ a vector of processing capabilities for all tasks, and by $t$ the current time moment in the schedule being constructed. For simplicity of presentation, $t_j$ denotes the remaining processing time of $T_j$.

**MC algorithm**

1: $t := 0$; **for all** $T_j \in \mathcal{T}$ **do** calculate $h(j)$;
2: **while** $\exists_{j \in \mathcal{T}} t_j > 0$ **do**
**begin**
2.1: construct set $Q$ of ready tasks;
2.2: capabilities($Q, \overline{\beta}$); (*find capabilities for ready tasks*)
2.3: calculate times:
$\tau' := \min\{\infty, \min_{T_j, T_{j+1} \in Q}\{\frac{h(j)-h(j+1)}{\beta_j - \beta_{j+1}} : \beta_j \neq \beta_{j+1}, \text{and } h(j) > h(j+1)\}$
(* the shortest time required for different heights of two tasks $T_j, T_{j+1}$ to become equal*)
$\tau'' := \min_{T_j \in Q}\{\frac{t_j)}{\beta_j} : \beta_j > 0\}$;
(* the time to the earliest completion of any task*)
2.4: $\tau := \min\{\tau', \tau''\}$;
2.5: schedule $\tau\beta_j$ piece of task $T_j$ in interval $[t, t+\tau]$ according to McNaughton's wrap-around rule [15], for $T_j \in Q$;
2.6: **for** $T_j \in Q$ **do** $h(j) := h(j) - \tau\beta_j, t_j := t_j - \tau\beta_j$; remove tasks with $t_j = 0$ from $Q$ and from $\mathcal{T}$;

2.7: $t := t + \tau$;
**end**; (* end of the algorithm *)

**procedure** capabilities(**in:**$X$;**out:**$\overline{\beta}$); (* $X$ - a set of ready tasks *)
**begin**
3.1: $\overline{\beta} := \overline{0}$; $avail := m$; (* $avail$ is the number of free processors *)
3.2: **while** $avail > 0$ **and** $\mid X \mid > 0$ **do**
**begin**
3.2.1: construct set $T$ of the highest tasks in $X$ with $h(j) > 0$;
3.2.2: **if** $\mid T \mid < avail$ **then**
**begin**
3.2.3: **for** $T_j \in T$ **do** $\beta_j := 1; avail := avail - \mid T \mid$;
**end**
**else** (* tasks in $T$ can use all $avail$ processors *)
**begin**
3.2.4: **for** $T_j \in T$ **do** $\beta_j := \frac{avail}{\mid T \mid}$; $avail$:=0;
**end**;
3.2.5: $X := X - T$;
**end**; (* of **while** loop *)
**end**; (* of procedure capabilities *)

*Description of the MC algorithm.* The algorithm builds a schedule interval by interval in **while** loop 2. The end of an interval is reached when the level of one task is reduced to that of another task (after time $\tau'$), or when some task is finished (after time $\tau''$). As tasks are processed their level and the remaining processing requirement $t_j$ is decreased in line 2.6. Finished tasks are removed from $\mathcal{T}$. Procedure capabilities assigns in line 3.2.3 one processor ($\beta_j = 1$) to the highest tasks as long as their number is not greater than the number of available processors. The remaining tasks, if there are any, share the remaining processors equally (line 3.2.4).

*Complexity of the MC algorithm.* The **while** loop 2 can be executed at most $2n - 1$ times because at most $n$ tasks can be finished and at most $n - 1$ times level of one task may be decreased to the level of some other task (cf. Observation 1). Procedure capabilities requires $O(\min\{n, m\})$ time if we initially order tasks according to their heights which can be done in $O(n \log n)$ time. Times $\tau'$ and $\tau''$ can be calculated in $O(n)$ time in line 2.3. Building a partial schedule in line 2.5 using the McNaughton's wrap-around rule [15] requires $O(n)$ time. Line 2.6 requires $O(n)$ time. The total algorithm complexity is $O(n^2)$ (we assume for simplicity that $m \leq n$).

Below we analyze properties of the MC algorithm.

**Observation 1** *When two tasks become equal their levels remain equal until one or both of the tasks complete.*

**Proof.** This attribute of the algorithm is a result of two facts: Tasks with equal levels receive the same capabilities, and a task with an initially higher level may not reduce it below the level of some other task with an initially lower level (which is guaranteed by lines 2.3 and 2.4). □

Note that this property extends also to certain successors of the two tasks. Suppose that $h(i) = h(k)$ for tasks $T_i, T_k$, and $T_i \prec T_j$. Without loss of generality let us assume that $T_i$ is completed earlier than $T_k$. After the completion of $T_i, h(j) = h(k)$.

**Observation 2** *The ordering of ready tasks according to their heights does not change.*

**Proof.** An immediate consequence of Observation 1. □

For the purposes of the following key theorem recall that $h(1) \geq \ldots \geq h(n)$.

**Theorem 3** *The MC algorithm is optimal in simultaneously minimizing the partial sums $\sum_{i=1}^{j} h(i)$ for $j = 1, \ldots, m$, and the total amount of remaining work $\sum_{j=1}^{n} t_j$.*

**Proof.** We start with the second part of this theorem. $\sum_{j=1}^{n} t_j$ is obviously minimal until some idle time appears for the first time in the interval $[t, t + \tau]$. $Q$ is the set of tasks executed in $[t, t + \tau]$. An idle time in MC means that $|Q| < m$, and there are no other ready tasks. Moreover, all the tasks executed after $t + \tau$ are successors of the tasks in $Q$. Therefore, any algorithm processing more tasks in $[0, t + \tau]$ must be infeasible. Thus, the idle time cannot be avoided, and $\sum_{j=1}^{n} t_j$ is minimal possible.

Suppose, that MC is not optimal in minimizing $S(j)_{MC} = \sum_{i=1}^{j} h(i)$ for $j = 1, \ldots, m$. Thus, there are index $q$ and time $t$ starting from which some better algorithm $A$ has $S(q)_A = \sum_{i=1}^{q} h(i) < S(q)_{MC}$, and $S(j)_A = \sum_{i=1}^{j} h(i) = S(j)_{MC}$ for $j = 1, \ldots, q-1$. This means that from moment $t$ on task $T_q$ is processed faster than in the MC algorithm, and it receives a bigger value of $\beta_q$ than in MC. Suppose $T_q$ was assigned $\beta_q$ in line 3.2.3 by algorithm MC. As it is maximum possible value, algorithm $A$ builds infeasible schedule. The same reasoning applies if $T_q$ was assigned $\beta_q$ in line 3.2.4 by algorithm MC, and $|T| = 1$. Suppose $T_q$ was assigned $\beta_j$ in line 3.2.4, and $|T| > 1$.

Then there is at least one more task $T_a$ with the same height. If $A$ gives a higher $\beta_q$ than MC at the cost of $\beta_a$ and if $a < q$, then $S(a)_A > S(a)_{MC}$ and algorithm $A$ is no better than MC. If $A$ gives a higher $\beta_q$ than MC at the cost of $\beta_a$ and $a > q$, then $T_q$ immediately stops being the $q$-th highest task, $h(a) > h(q)$, and tasks $T_a$, $T_q$ should be swapped in the ordering according to their height. After swapping, $S(q)_A > S(q)_{MC}$ and algorithm $A$ is not better than MC. □

The MC algorithm has already been extended to solve many scheduling problems. For example, problems of scheduling on uniform processors $Q|pmtn|C_{max}$, $Q2|pmtn, prec|C_{max}$ [13] can be solved by an algorithm derived from MC. A generalized algorithm for $Q|pmtn|C_{max}$ is considered in the next section. We finish this section with an example demonstrating that the algorithm complexity is indeed $O(n^2)$, even for very simple in-trees.

**Example 1**

Let $m = 2$, and $n > 2$, $t_1 = 1, t_2 = 2, t_3 = 3, \ldots, t_j = j, \ldots, t_{n-2} = t_{n-1} = n-2$, $t_n = 1$. Tasks $T_1, \ldots, T_{n-1}$ precede $T_n$, hence we have an in-tree. We calculate levels: $h(n-1) = h(n-2) = n-1, h(n-3) = n-2, \ldots, h(1) = 2$. In the first interval of the schedule, and the first iteration of **while** loop 2 tasks $T_{n-1}$ and $T_{n-2}$ obtain $\beta_{n-1} = \beta_{n-2} = \frac{2}{2}$. The other tasks receive 0 processing capability. In line 2.4 $\tau' = 1$ is selected for the length $\tau$ of the interval, and $T_{n-1}, T_{n-2}$ are executed in parallel. At the beginning of the next iteration (also interval) $h(n-1) = h(n-2) = h(n-3)$ and $\beta_{n-1} = \beta_{n-2} = \beta_{n-3} = \frac{2}{3}$. The other tasks are not executed. The length of the interval is $\frac{3}{2}$. In the third interval four tasks are executed: $T_{n-1}, \ldots, T_{n-4}$, with $\beta_{n-1} = \ldots = \beta_{n-4} = \frac{2}{4}$. The interval has length 2. Each of the following intervals adds one more task to be processed. Finally, in interval (and iteration) $n-2$, tasks $T_{n-1}, \ldots, T_1$ receive capabilities $\beta_{n-1} = \ldots = \beta_1 = \frac{2}{n-1}$. In the last interval of length 1 only $T_n$ is executed. The schedule built above is presented in Fig.1.

In each of the intervals the McNaughton's wrap-around rule is used which results in the complexity proportional to the number of considered tasks. In the first interval we had 2 tasks, in the second 3 tasks, ..., in $j$-th interval $j+1$ tasks, ..., in interval $n-2$ we had $n-1$ tasks. Thus, the total complexity of the algorithm is $O(\sum_{i=1}^{n-2}(i+1)) = O(n^2)$. □

# 3 Derivatives of the Muntz-Coffman algorithm

## 3.1 $Q, win|pmtn|C_{max}$

In this section we consider the problem of preemptive scheduling on uniform processors available in time windows, for the schedule length criterion. Tasks are independent. The number of available processors is not stable in general, yet in each of the time windows it is fixed and known a priori.

Let us outline the main ideas of the algorithm solving this problem. The length of the schedule is determined by the pieces of tasks which remain to be scheduled in the last occupied window $[b_p, e_p]$. Let $t_1 \geq t_2 \geq \ldots t_n$ be the processing requirements of tasks that remain to be completed in $[b_p, e_p]$. Let $s_{1p} \geq s_{2p} \geq \ldots s_{m_p p}$ be the speeds of uniform processors available in window $p$. The following reasoning can be used to determine the length of the schedule in the last window: The fastest processor must be able to execute the longest remaining task, the two fastest processors must be able to execute the two longest tasks, etc. Finally, all processors together must be able to accommodate all the tasks remaining in the last window. Thus, the length of the partial schedule in the last interval is determined by the lower bound:

$$C = \max\{\max_{i=1}^{m-1}\{\sum_{j=1}^{i} t_j / \sum_{j=1}^{i} s_{jp}\}, \sum_{j=1}^{n} t_j / \sum_{j=1}^{m} s_{jp}\}. \quad (1)$$

If the above equation holds, then the pieces of the tasks can be feasibly scheduled, e.g., according to the Gonzales and Sahni (GS) algorithm for $Q|pmtn|C_{max}$ [10]. The GS algorithm is presented in the following part of this section. It follows from equation (1) that the optimization algorithm for our problem $Q, win|pmtn|C_{max}$ should minimize $\max_{i=1}^{m-1}\{\sum_{j=1}^{i} t_j\}$, and $\sum_{j=1}^{n} t_j$ before reaching the last occupied interval. Observe that the MC algorithm has exactly this required feature (see Theorem 3). Below we present a modified version of the MC algorithm which solves problem $Q, win|pmtn|C_{max}$.

**Algorithm for** $Q, win|pmtn|C_{max}$

1: $t := 0; l := 1$; **for all** $T_j \in \mathcal{T}$ **do** calculate $h(j)$; order tasks according to their heights;
2: **while** $\exists_{T_j \in \mathcal{T}} t_j > 0$ **do**
**begin**
2.1: capabilities$(l, \mathcal{T}, \overline{\beta})$; (*find capabilities for the current window $l$*)
2.2: calculate times:

$\tau' := \min\{\infty, \min_{T_j, T_{j+1} \in Q}\{\frac{h(j)-h(j+1)}{\beta_j - \beta_{j+1}} : \beta_j \neq \beta_{j+1}, \text{and } h(j) > h(j+1)\}$
$\tau'' := \min_{T_j \in \mathcal{T}}\{\frac{t(j)}{\beta_j} : \beta_j > 0\}$;
$\tau''' := e_l - t$;
(* $\tau'''$ is the time to the completion of the current time window*)

2.3: $\tau := \min\{\tau', \tau'', \tau'''\}$; **if** $\tau = \tau'''$ **then** $l := l + 1$;

2.4: schedule $\tau\beta_j$ piece of task $T_j$ in interval $[t, t+\tau]$ according to the Gonzales-Sachni (GS) algorithm [10] for $T_j \in \mathcal{T}$;

2.5: **for** $T_j \in \mathcal{T}$ **do** $h(j) := h(j) - \tau\beta_j, t_j := t_j - \tau\beta_j$; remove tasks with $t_j = 0$ from $\mathcal{T}$;

2.6: $t := t + \tau$;

**end**;(* end of the algorithm *)

*Description of the algorithm for $Q, win|pmtn|C_{max}$.* Observe that in this problem there is no need to select the ready tasks in each interval (iteration) because all tasks are initially ready. We have no precedences and height of a task is equal to the remaining processing time. The above algorithm differs from the original MC algorithm in few details: Procedure capabilities takes into account changing number of currently available processors and differing speeds. Therefore, index $l$ of the current window is passed as a parameter to procedure capabilities in line 2.1. Line 3.1 in this procedure should look like
3.1: $\overline{\beta} := \overline{0}$; $avail := m_l$;
The assignment of processing capacities takes into account differing processors speeds. Thus, lines 3.2.3 and 3.2.4 in procedure capabilities shoud be
3.2.3: **for** $T_j \in T$ **do** $\beta_j := \frac{1}{|T|}\sum_{i=m_l-avail+|T|}^{m_l-avail+1} s_{il}; avail := avail- \mid T \mid$;
3.2.4: **for** $T_j \in T$ **do** $\beta_j := \frac{1}{|T|}\sum_{i=m_l-avail+1}^{m_l} s_{il}; avail := 0$;
Finally, the algorithm recalculates the current processing capabilities assignment also at the end of the current time window. For this purpose $\tau$ is selected in line 2.3 not to exceed time $\tau'''$ remaining to the end of the current time window. Note that the order of the tasks according to their height does not change, i.e. $h(1) \geq h(2) \geq \ldots h(n)$ is an invariant of the algorithm.

*A note on applying the Gonzales and Sahni (GS) algorithm.* Line 2.4 building partial schedules in intervals $[t, t+\tau]$ needs additional explanation. Below we present the basics of the GS algorithm. Let $S(i) = \sum_{j=1}^{i} t_j$, and let $PC(i) = Cs_i$ denote *processing capacity* of some processor $P_i$ in interval with length $C$. If we order processors such that $s_1 \geq s_2 \geq \ldots \geq s_m$, and tasks such that $t_1 \geq t_2 \geq \ldots \geq t_n$, then a feasible schedule exists provided

that

$$S(i) \leq \sum_{h=1}^{i} PC(h) \text{ for } i = 1, \ldots, m-1, \text{ and } S(n) \leq \sum_{h=1}^{m} PC(i). \qquad (2)$$

Inequalities (2) determine a lower bound on schedule length (cf. equation (1)). Algorithm GS schedules tasks one by one from the longest to the shortest in such a way that inequalities (2) are always satisfied by the remaining tasks and the remaining processing capacity. To assign tasks to processors GS algorithm uses the following rule. For a task with processing requirement $t_j$ find a pair of processors $P_i$ and $P_{i+1}$ such that $PC(i) \geq t_j > PC(i+1)$. Find a moment of time $x$ such that task $T_j$ is executed in interval $[0, x]$ on processor $P_i$ and in interval $[x, C]$ on $P_{i+1}$. From the two remaining processor intervals ($[x, C]$ on $P_i$ and $[0, x]$ on $P_{i+1}$) create a new composite processor with processing capacity $PC(i) + PC(i+1) - t_j$. In the boundary case $x$ can be equal $C$ or 0, then $T_j$ uses one processor without interruption. On the other hand, when $i+1$ is greater than the number of available processors (original or composite) one should assume $PC_{i+1} = 0$.

Now, let us translate the GS algorithm to our problem of constructing an interval of schedule in line 2.4. The length of the partial schedule is $\tau$. Pieces of tasks are $\beta_j \tau$ for $j = 1, \ldots, n$. Now $S(i) = \sum_{j=1}^{i} \beta_j \tau$, $PC(i) = s_{il}\tau$. We must guarantee that inequalities (2) hold. Assume it holds for $i = 1, \ldots, j$ for some $j \geq 0$, and tasks $T_{j+1}, \ldots, T_{j+|T|}$ received equal processing capabilities, where $|T|$ is defined in procedure capabilities. The capabilities of the tasks are $\beta_{j+1} = \ldots = \beta_{j+|T|} = \frac{1}{|T|} \sum_{i=m_l-avail+1}^{m_l-avail+x} s_{il} = \frac{1}{|T|} \sum_{i=j+1}^{j+x} s_{il}$ where $x = \min\{avail, |T|\}$. Since $|T| \geq x$, and $s_{(j+1)l} \geq \ldots \geq s_{m_l l}$ no task receives bigger processing capability than $s_{(j+1)l}$. Indeed, $\beta_{j+1} = \frac{1}{|T|} \sum_{i=j+1}^{j+x} s_{il} \leq \frac{x s_{(j+1)l}}{|T|} \leq s_{(j+1)l}$. Thus, inequality $S(j+1) \leq \sum_{h=1}^{j+1} PC(h)$ also holds. Analogously, $\sum_{i=j+1}^{j+a} \beta_i = \frac{a}{|T|} \sum_{i=j+1}^{j+x} s_{il} \leq \frac{a \frac{x}{a} \sum_{i=j+1}^{j+a} s_{il}}{|T|} \leq \sum_{i=j+1}^{j+a} s_{il}$ for $a = 1, \ldots, |T|$. Hence, also the remaining inequalities hold in formula (2) for the pieces of tasks assigned to interval $[t, t+\tau]$, and partial schedule can be built by the GS algorithm.

*Complexity of the algorithm for $Q, win|pmtn|C_{max}$.* Line 1 can be executed in $O(n \log n)$ time. The **while** loop 2 can be executed at most $2n-1+p$ times because it is executed at most $2n-1$ times as in the original MC algorithm, and there are $p$ time windows at the end of which capabilities must be recalculated. Procedure capabilities requires $O(\min\{n, \max_i\{m_i\}\})$ time.

Times $\tau'$ and $\tau''$ can be calculated in $O(n)$, and $\tau'''$ in $O(1)$ time in line 2.2. The original GS algorithm needs sorting the pieces of tasks assigned to interval $[t, t+\tau]$, according to their processing requirements. However, in our problem pieces of the tasks are already sorted because tasks with higher height receive bigger processing capability. Therefore, the order of pieces according to value of $\tau\beta_j$ coincides with the order of heights of the original tasks. This order is invariant in our algorithm. Hence, building a partial schedule in line 2.4 requies $O(n)$ time. Line 2.5 requires $O(n)$ time. The total complexity (assuming $\max_i\{m_i\} \leq n$) is $O(n(n+p))$.

The above algorithm can be also applied to schedule in time windows chains of preemptive tasks, i.e. for problem $Q, win|chains, pmtn|C_{max}$. It is because each chain can be considered as one long preemptive task with processing time equal to the sum of processing requirements in the chain. Then, our algorithm can be applied to such a reformulated problem. Furthermore, it is not difficult to extend our algorithm to problem $Q2, win|prec, pmtn|C_{max}$ (cf. [13]). Surprisingly, the case of arbitrary precedence constraints and time windows is computationally hard already for an arbitrary number of identical processors and trees (strictly saying $P, win|tree, pmtn|C_{max}$ is NP-hard) [2]. The hardness comes from the need of simultaneous matching trees of tasks in two directions: time and processor number. We finish presentation of the algorithm with an example.

**Example 2**

$p = 5$, the windows of processor availability are: $[b_1, e_1] = [0, 1], m_1 = 2, s_{11} = 3, s_{21} = 1$, $[b_2, e_2] = [1, 2], m_2 = 3, s_{12} = 3, s_{22} = 2, s_{32} = 1$, $[b_3, e_3] = [2, 4], m_3 = 1, s_{13} = 1$, $[b_4, e_4] = [4, \infty], m_4 = 2, s_{14} = 2, s_{24} = 1$. We have $n = 4$, the processing requirements are: $t_1 = 8, t_2 = 6, t_3 = 4, t_4 = 2$. In the following table we present levels of the tasks $\overline{h} = [h(1), \ldots, h(4)]$ (at the beginning of each algorithm iteration), capability assignments $\overline{\beta}$, and the length of the intrval $\tau$. The schedule has length $C_{max} = 6 + \frac{2}{3}$ and is shown in Fig.2. □

| $\overline{h}$ | $\overline{\beta}$ | $\tau$ |
|---|---|---|
| $[8, 6, 4, 2]$ | $[3, 1, 0, 0]$ | 1 |
| $[5, 5, 4, 2]$ | $[2.5, 2.5, 1, 0]$ | $\frac{2}{3}$ |
| $[\frac{10}{3}, \frac{10}{3}, \frac{10}{3}, 2]$ | $[2, 2, 2, 0]$ | $\frac{1}{3}$ |
| $[\frac{8}{3}, \frac{8}{3}, \frac{8}{3}, 2]$ | $[\frac{1}{3}, \frac{1}{3}, \frac{1}{3}, 0]$ | 2 |
| $[2, 2, 2, 2]$ | $[\frac{3}{4}, \frac{3}{4}, \frac{3}{4}, \frac{3}{4}]$ | $\frac{8}{3}$ |

## 3.2 $P|spdp - lin - \delta_j, r_j, pmtn|C_{max}$

In this section we extend the MC algorithm in yet another dimension. We consider special case of multiprocessor tasks with ready times. Multiprocessor tasks may require more than one processor at the same moment of time. Here we assume that the number of required processors is not fixed before executing the task, and can be changed during its execution. Furthermore, task $T_j$ preserves linear speedup up to $\delta_j$ processors. Thus, $T_j$ executed $\frac{t_j}{m}$ units of time on $m \leq \delta_j$ processors is considered finished. Using more processors than $\delta_j$ is prohibited. Since the number of used processors is not predetermined the height of the task must be calculated differently than in the original MC algorithm. Here the height is equal to the shortest remaining execution time, i.e. $h(j) = \frac{t_j}{\delta_j}$ for $T_j$, where $t_j$ is the remaining piece of work on $T_j$. As tasks have various ready times we may distinguish intervals of time between consecutive ready times. At each ready time new task(s) appear in the processing environment. After the last ready time we remain with some pieces of tasks to be completed. Thus, the length of the schedule after the last ready time determines the length of the whole schedule. It is analyzed in the following Proposition.

**Proposition 4** *Length of the optimal schedule for problem $P|spdp - lin - \delta_j, pmtn|C_{max}$ is*

$$C_{max} = \max\{\max_{T_j \in \mathcal{T}}\{\frac{t_j}{\delta_j}\}, \frac{1}{m}\sum_{j=1}^{n} t_j\}$$

**Proof.** Note that $C_{max}$ calculated in the above way is also a lower bound on the schedule lenght because $\max_{T_j \in \mathcal{T}}\{\frac{t_j}{\delta_j}\}$ is the time required to finish the longest task, while $\frac{1}{m}\sum_{j=1}^{n} t_j$ is the time needed to finish all the tasks provided all processors are equally loaded. A schedule of that length can be constructed by extending McNaughton's wrap-around rule [15]. In the McNaughton's algorithm tasks are scheduled one after another starting at (say) processor $P_1$. A task which completion spans beyond the calculated $C_{max}$ is wrapped-around and the excess of work is shifted to the next processor at time 0. We do the same in our case, however, it is possible to wrap-around the same task several times. In this way some task(s) can be executed at the same time on many processors. A schedule is feasible because all tasks receive the required processing and no task $T_j$ uses more than $\delta_j$ processors which is guaranteed by the first term in the formula stating $C_{max}$. □

From Proposition 4 we know that the length of the schedule after the last ready time is determined by the longest piece of a single task, and the sum of all remaining work. Thus, the optimization algorithm for our problem should minimize the longest remaining processing requirement of any task, and the total remaining work before reaching the last ready time. MC algorithm is able to achieve this (cf. Theorem 3). Let $Q_k$ denote the set of tasks available for processing at the ready time $r_k$, and $l$ the number of different ready times. Below we present an adapted version of the MC algorithm.

**Algorithm for** $P|spdp-lin-\delta_j, r_j, pmtn|C_{max}$

1: $t := 0$; group tasks with the ready time $r_k$ in set $Q_k$; order tasks in $Q_k$ according to nonincreasing heights, $k = 1, \ldots, l$;
2: **for** $k := 1$ **to** $l$ **do**
**begin**
2.1: order tasks in $Q_k$ according to nonincreasing values of $h(j)$;
2.2: **while** $(r_{k+1} > t)$ **and** $(\exists_{T_j \in Q_k} h(j) > 0)$ **do**
**begin**
2.2.1: capabilities$(Q_k, \overline{\beta})$;
2.2.2: calculate times:
$\tau' := \min\{\infty, \min_{T_j, T_{j+1} \in Q_k}\{\frac{h(j)-h(j+1)}{\frac{\beta_j}{\delta_j} - \frac{\beta_{j+1}}{\delta_{j+1}}} : \frac{\beta_j}{\delta_j} \neq \frac{\beta_{j+1}}{\delta_{j+1}}, h(j) > h(j+1)\}$
(* the shortest time required for two tasks $T_j, T_{j+1}$ with different heights to become equal*)
**if** $\beta_{|Q_k|} > 0$ **then** $\tau'' := \frac{h(|Q_k|)}{\beta_{|Q_k|}/\delta_{|Q_k|}}$ **else** $\tau'' := \infty$;
(*the time to the earliest completion of any task*)
2.2.3: $\tau := \min\{\tau', \tau'', r_{k+1} - t\}$;
2.2.4: schedule $\tau\beta_j$ piece of task $T_j$ in interval $[t, t+\tau]$ according to extended McNaughton's wrap-around rule (Proposition 4) for $T_j \in Q_k$;
2.2.5: $h(j) := h(j) - \frac{\tau\beta_j}{\delta_j}$ for $T_j \in Q_k$;
2.2.6: $t := t + \tau$;
**end**;
2.3: $Q_{k+1} := Q_{k+1} \cup \{T_j : T_j \in Q_k, h(j) > 0\}$;
**end**; (* end of the algorithm *)

*Description of the algorithm for* $P|spdp-lin-\delta_j, r_j, pmtn|C_{max}$. As in the previous section we have no precedences and height of a task is equal to the remaining processing time. We explicitly recalculate processing capability assignment with each ready time, because at ready times new tasks appear, possibly with heights sufficiently big to execute them. The end of the interval built in each iteration of **while** loop 2.2 results either $(\tau')$ from two tasks with

height initially different becoming equal, or ($\tau''$) from finishing the lowest task in $Q_k$ (which is also the last task in $Q_k$ by line 2.1), or ($r_{k+1}-t$) reaching a new ready time. In order to observe different values of $\delta_j$ procedure capabilities must be modified in the following lines:
3.2.2: **if** $\sum_{T_j \in T} \delta_j < avail$ **then**
3.2.3: **for** $T_j \in T$ **do** $\beta_j := \delta_j$; $avail := avail - \sum_{T_j \in T} \delta_j$;
3.2.4: **for** $T_j \in T$ **do** $\beta_j := \delta_j \frac{avail}{\sum_{T_j \in T} \delta_j}$; $avail$:=0;
By this method of calculating capabilities, it is guaranteed that tasks with the biggest height are preferred, and no processor is idle as long as there is a task ready to use it. Furthermore, tasks of equal height receive the same capability and remain equal till their completion. More on feasibility and optimality of the schedule built by the above algorithm can be found in [6].

*Complexity of the algorithm for* $P|spdp-lin-\delta_j, r_j, pmtn|C_{max}$. Again, the internal **while** loop 2.2 will be performed at most $O(n)$ times. Line 2.2.4 requires $O(n)$ time to schedule the tasks, and procedure capabilities can be executed in $O(\min\{n, m\})$ time. Therefore, the complexity of the algorithm is $O(n^2)$. Below we solve an example instance of the considered problem.

**Example 3**

The tasks data is the following: $n = 5, t_1 = 5, t_2 = 2, t_3 = 3, t_4 = 2, t_5 = 1; \delta_1 = 2, \delta_2 = 1, \delta_3 = 2, \delta_4 = 2, \delta_5 = 1; r_1 = r_2 = 0, r_3 = r_4 = r_5 = 1$. We have $m = 4$ processors. In the following table we present levels of the tasks $\overline{h} = [h(1), \ldots, h(4)]$, capability assignments $\overline{\beta}$, and interval lengths $\tau$ for each iteration of the algorithm. $x$ in the height vector means that the task is not ready yet. The schedule has length $C_{max} = 3.5$ and is shown in Fig.3. □

| $\overline{h}$ | $\overline{\beta}$ | $\tau$ |
|---|---|---|
| $[2.5, 2, x, x, x]$ | $[2, 1, 0, 0, 0]$ | 1 |
| $[1.5, 1, 1.5, 1, 1]$ | $[2, 0, 2, 0, 0]$ | 0.5 |
| $[1, 1, 1, 1, 1]$ | $[1, 0.5, 1, 1, 0.5]$ | 2 |

Before the end of this section let us observe that if we set $\delta_j = 1$, then this algorithm solves problem $P|r_j, pmtn|C_{max}$, which in turn is equivalent to $P|pmtn|L_{max}$, when the schedule is read backwards. Note, that for $P|pmtn|L_{max}$ the algorithm by Horn [12] is referred to in the deterministic scheduling literature. After modifications the algorithm presented above can be applied to solve two equivalent problems $P2|size_j, r_j, pmtn|C_{max}$ and $P2|size_j, pmtn|L_{max}$ [7]. Let us remind that $size_j$ means that tasks have fixed and known a priori numbers of simultaneously required processors.

## 3.3 $P|spdp - lin - \delta_j, pmtn, chain|C_{max}$

In this section we consider problem of scheduling chains of multiprocessor tasks which have no predefined number of processors to be simultaneously used. However, for each task $T_j$ the number of usable processors is bounded from above by $\delta_j$.

This problem has practical origin in parallel processing. *Parallelism profile* is a function of time stating the number of processors used by a parallel application. Parallelism profile is measured on a computer with unbounded (in reality: sufficiently big) number of processors. The problem is to execute in the shortest possible time $n$ parallel applications with known parallelism profile on $m$ parallel identical processors. From the scheduling point of view parallelism profile is a chain of multiprocessor tasks each with a bounded number of processors used in parallel. Furthermore, *bulk synchronous processing* (BSP) model of parallel computations translates itself directly to a chain of multiprocessor tasks. In BSP model computation is a sequence of parallel computations and synchronizations which can be considered as, respectively, multiprocessor tasks, and sequential tasks with $\delta_j = 1$. This model can be applied to scheduling other activities with a profile of the resource demand known in advance.

Now we return to our scheduling problem. It has been shown [9] that the case of three-operation chains with $\delta_h = \delta_t = 1, \delta_p = m$, where $h$ is the first, $p$ is the central, and $t$ is the last operation, is strongly **NP**-hard even if $n < m$. Yet, if the optimal schedule for the $m-1$ longest chains is known then the optimal schedule for all $n$ tasks can be found by a modified version of the MC algorithm. The reader is kindly referred to [9] for the original idea of the problem and its solution, the complexity analysis, polynomially solvable cases, theoretical and experimental evaluation of heuristics for the problem. In the following discussion we concentrate on the use of the MC algorithm to schedule $r > m$ arbitrary length chains of multiprocessor tasks provided that some initial schedule $S$ for the $m-1$ longest chains is given. The algorithm can be understood as filling idle periods in schedule $S$ with the tasks of the remaining chains.

We concentrate on chains here rather than on particular tasks. $h(j)$ of chain $j$ is the sum of the remaining processing requirements of its components. Let us assume that we have set $\mathcal{C}$ of $r > m$ chains ordered in descending order of the heights, thus $h(1) \geq h(2) \geq \ldots \geq h(r)$. By a *high chain* we mean a chain which is strictly higher than the $m$-th chain. The set of all high

chains is denoted by $\mathcal{H}$, i.e. $\mathcal{H} = \{j : h(j) > h(m)\}$. The remaining chains in set $\mathcal{L} = \{j : h(j) \leq h(m)\}$ are called *low chains*. Note, that $|\mathcal{H}| \leq m-1$ and $|\mathcal{L}| + |\mathcal{H}| \geq m$.

Schedule $S$ for high chains is a sequence of $p$ intervals in which the numbers of processors assigned to the tasks do not change. Interval $k$ ($k = 1, \ldots, p$) is defined by its length $L_k$, set $Q_k$ of tasks executed in it, and the number of processors $proc_k(j)$ occupied by task $T_j \in Q_k$ in it. Interval $k$ is called *compact* if the number of processors available in $k$ is zero, or all high chains which have not been finished in the preceding intervals are executed in $k$. It is possible to convert any interval of $S$ into a compact one (see [8] for details). In the pseudocode of the algorithm for $P|spdp - lin - \delta_j, pmtn, chain|C_{max}$ the following notation is used:
$k$ - interval index,
$Q_k$ - the set of tasks from high chains processed in interval $k$ of $S$,
$proc_k(j)$ - the number of processors used by task $T_j$ from high chain $j$ in interval $k$ of $S$,
compact($t$,$S$) - procedure compacting the interval of $S$ that starts at $t$,
$head(j)$ - the task heading chain $j$.

**Algorithm for** $P|spdp - lin - \delta_j, pmtn, chain|C_{max}$

1: $t := 0$; compact(0,$S$); $\mathcal{A} := \{j : j \in \mathcal{L}, h(m) = h(j)\}$;
2: **while** $\mathcal{H} \neq \emptyset$ **and** $h(m) > 0$ **do**
**begin**
2.1: $\overline{\beta} := \overline{0}$; let $k$ be the interval of $S$ starting at $t$;
2.2: $\beta_j$:=$proc_k(j)$ for $T_j \in Q_k$; $avail := m - \sum_{T_j \in Q_k} proc_k(j)$;
2.3: **if** $avail > 0$ **then** $\beta_j := \frac{avail}{|\mathcal{A}|}$ for $T_j \in \{T_l : T_l = head(q), q \in \mathcal{A}\}$;
2.4: calculate times:
$\tau' := \min\{\infty, \min_{j \in \mathcal{H}}\{\frac{h(j)-h(m)}{\beta_{head(j)} - \beta_{head(m)}} : \beta_{head(j)} > \beta_{head(m)}\}\}$;
(* the shortest time required for the height of a high chain to drop to the height of the $m$-th highest chain *)
**if** $\beta_{head(m)} > 0$ **then** $\tau'' := \frac{h(m) - h(|\mathcal{H}|+|\mathcal{A}|+1)}{\beta_{head(m)}}$ **else** $\tau'' := \infty$;
(* the time required for the $m$-th highest chain to reach the height of chain $|\mathcal{H}| + |\mathcal{A}| + 1$, the highest one in $\mathcal{L} - A$ *)
$\tau''' = \sum_{i=1}^{k} L_i - t$;
(*the time to the completion of the current interval in schedule $S$*)
$\tau := \min\{\tau', \tau'', \tau'''\}$;

2.5: schedule piece $\tau\beta_i$ of $T_i = head(j)$ in interval $[t, t+\tau]$, using extended McNaughton's rule (Proposition 4), for $j \in \mathcal{C}$;
2.6: $h(j) := h(j) - \tau\beta_j$ for $j \in \mathcal{C}$; $t := t + \tau$;
$t_i := t_i - \tau\beta_i$, remove tasks with $t_i = 0$ from chain $j$ and from $\mathcal{T}$, for $T_i = head(j)$ and $j \in \mathcal{C}$;
2.7: $\mathcal{B} := \{j : j \in \mathcal{H}, h(j) = h(m)\}$; $\mathcal{H} := \mathcal{H} - \mathcal{B}$;
$\mathcal{A} := \mathcal{A} \cup \mathcal{B} \cup \{j : j \in \mathcal{L} - \mathcal{A}, h(j) = h(m)\}$;
2.8: **if** $\mathcal{B} \neq \emptyset$ **then begin** remove $j \in \mathcal{B}$ from the end of $S$ starting at time $t$; compact$(t, S)$ **end;**
**end**;
3: **if** $h(m) > 0$
**then** schedule the tasks remaining in chain $j$ as a single task of length $h(j)$ in interval $[t, t + \sum_{j\in\mathcal{T}} t_j/m]$ using McNaughton's rule [15], for $j \in \mathcal{C}$
**else** concatenate the remaining part of $S$ starting at $t$;
(* end of the algorithm *)

*Description of the algorithm for* $P|spdp-lin-\delta_j, pmtn, chain|C_{max}$. Unlike in the preceding sections, procedure capabilities has been incorporated in the rest of the code. The algorithm builds subintervals in the intervals of the initial schedule $S$. The compactness of these subintervals is maintained during the whole algorithm run by lines 1 and 2.8. Tasks from high chains receive the same processing capabilities as in the original schedule $S$. Thus, the part of the schedule for high chains remains intact. The remaining *avail* free processors are shared by tasks from the highest low chains (line 2.3). In line 2.4 length of the subinterval is calculated such that for the current values of capabilities, no high chain may reduce its height below the level of chains in set $\mathcal{A}$ (time $\tau'$), no chain from $\mathcal{A}$ may reduce its height below the level of chains in set $\mathcal{L} - \mathcal{A}$ (time $\tau''$). In line 2.5 pieces of tasks heading chains are scheduled. Tasks from low chain $j$ consume $\tau\beta_j$ units of time. Hence, if $t_i < \tau\beta_j$ for $T_i \in head(j)$, then after completing $T_i$ its successor is immediately started. In line 2.6 heights of the processed chains and the remaining processing requirements are appropriately decreased. In line 2.7 chains from the set of high and low chains which became equal in height with the $m$-th highest chain are shifted to set $\mathcal{A}$. If $h(m) > 0$ in line 3 then all high chains have been shifted to set $\mathcal{A}$, and $|\mathcal{A}| \geq m$. Therefore, McNaughton's rule can be applied to whole remaining chains treated as single tasks of length $h(j)$, for $\in \mathcal{C}$. Note that in this case all processors finish simultaneously, and since no available processor was left idle during the whole algorithm run,

the schedule must be optimal. In the opposite case ($h(m) = 0$ in line 3), all low tasks are completed and the remaining part of the schedule is the initial schedule $S$ for high chains. In this case optimality of the schedule depends on optimality of schedule $S$ for high tasks. The proof of the algorithm correctness and optimality of the schedules it builds is presented in [8].

Optimal schedule $S$ for high chains can be obtained by applying linear programming to any feasible permutation $\pi$ of task completions and selecting the best one. The method of constructing optimal schedule for the given permutation $\pi$ is presented in [8]. Suboptimal schedule $S$ also can be used.

*Complexity of the algorithm for* $P|spdp-lin-\delta_j, pmtn, chain|C_{max}$. Note that the number of chains $r$ is $O(n)$. Procedure compact requires $O((m+n)p)$ time in line 1 (cf. [8]). Loop **while** 2 can be executed at most the number of times the events mentioned in line 2.4 take place. A high chain may have its height reduced to the level of chains in set $\mathcal{A}$ at most $m-1$ times. Chains from $\mathcal{L}-\mathcal{A}$ my join set $\mathcal{A}$, $O(n-m)$ times. Some interval of schedule $S$ may finish $p+n$ times, where $n$ are the additional intervals which may appear as results of running procedure compact. Lines 2.5-2.7 require $O(n)$ time. Line 2.8 requires $O(mp(m+n))$ time over all the algorithm run time because the condition in line 2.8 is satisfied at most $O(m)$ times and compacting needs $O((m+n)p)$ time. Line 3 requires $O(n)$ time. Thus, the total complexity of the algorithm is $O((m-1+n+p+n-m)n+(n+m)mp)$. Since $m < n$, the complexity is $O(n(n+mp))$.

Let us note that schedules built by the above algorithm (also the optimal ones) have a crucial property: low tasks are executed on one processor at a time, and their parallelism is not exploited. As pointed out in [9], it is important information for designers of parallel systems: there are optimal schedules where only at most $m-1$ tasks take advantage of their parallelism.

# 4 Conclusions

In this work we demonstrated that the MC algorithm is a very efficient 'vehicle' for solving many diverse scheduling problems. Thus, MC algorithm seems to be one of the fundamental algorithms of deterministic scheduling theory. The reason for its potency has been identified in Theorem 3. At the end of this work one may ask 'where else can we go with this algorithm?' That is the question.

## Acknowledgement

The author expresses his thanks to the Referees for their suggestions on improving quality of this work.

## References

[1] Błażewicz J, Drabowski M and Węglarz J. Scheduling multiprocessor tasks to minimize schedule length. *IEEE Transactions on Computers* 1986; **35**(5): 389-393.

[2] Błażewicz J, Drozdowski M, Formanowicz P, Kubiak W and Schmidt G. Scheduling preemptable tasks on parallel processors with limited availability, *Parallel Computing* 2000; **26**(9):1195-1211.

[3] Błażewicz J, Ecker K, Pesch E, Schmidt G and Węglarz J. *Scheduling Computer and Manufacturing Processes*, Springer-Verlag: Heidelberg, 1996.

[4] Coffman EG Jr. and Graham RL. Optimal scheduling for two-processor systems, *Acta Informatica* 1972; **1**(3): 200-213.

[5] Drozdowski M. Scheduling multiprocessor tasks - An overview, *European Journal of Operational Research* 1996; **94**(2): 215-230.

[6] Drozdowski M. Real-time scheduling of linear speedup parallel tasks, *Information Processing Letters* 1996; **57**(1): 35-40.

[7] Drozdowski M. *Selected problems of scheduling tasks in multiprocessor computer systems*, Series: Monographs, No.321, Poznań University of Technology Press, Poznań, (1997), (see also `http://www.cs.put.poznan.pl/~maciejd/h.ps`).

[8] Drozdowski M. Scheduling parallel applications with known parallelism profile, Technical Report RA-002/2001, Institute of Computing Science, Poznań University of Technology, 2001.

[9] Drozdowski M and Kubiak W. Scheduling parallel tasks with sequential heads and tails, *Annals of Operations Research* 1999; **90**: 221-246.

[10] Gonzalez T and Sahni S. Preemptive scheduling of uniform processor systems, *Journal of the ACM* 1978; **25**(1): 92-101 .

[11] Graham RI, Lawler EL, Lenstra JK and Rinnoy Kan AHG. Optimization and approximation in deterministic sequencing and scheduling: A survey, *Ann. Discrete Math.* 1979; **5**: 287-326.

[12] Horn WA, Some simple scheduling algorithms, *Naval Research Logistics Quarterly* 1974, **21**: 177-185.

[13] Horvath EC, Lam S and Sethi R. A level algorithm for preemptive scheduling, *Journal of the ACM* 1977; **24**(1): 32-43.

[14] Hu TC. Parallel sequencing and assembly line problems, *Operations Research* 1961; **9**: 841-848.

[15] McNaughton R. Scheduling with deadlines and loss functions, *Management Science* 1959; **6**: 1-12.

[16] Muntz RR and Coffman EG Jr., Optimal Preemptive Scheduling on Two-Processor Systems, *IEEE Transactions on Computers* 1969; **18**(11): 1014-1020.

[17] Muntz RR and Coffman EG Jr., Preemptive scheduling of real-time tasks on multiprocessor systems, *Journal of ACM* 1970; **17**(2): 324-338.

[18] Veltman B, Lageweg BJ and Lenstra JK, Multiprocessor scheduling with communications delays, *Parallel Computing* 1990; **16**: 173-182.

List of figure captions

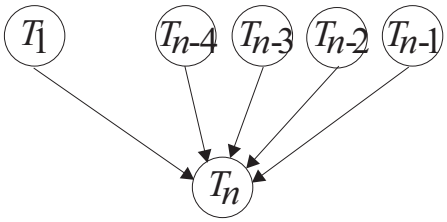
Fig. 1 Example 1. a) PCG, b) schedule.

Fig. 2 Optimal schedule for Example 2.

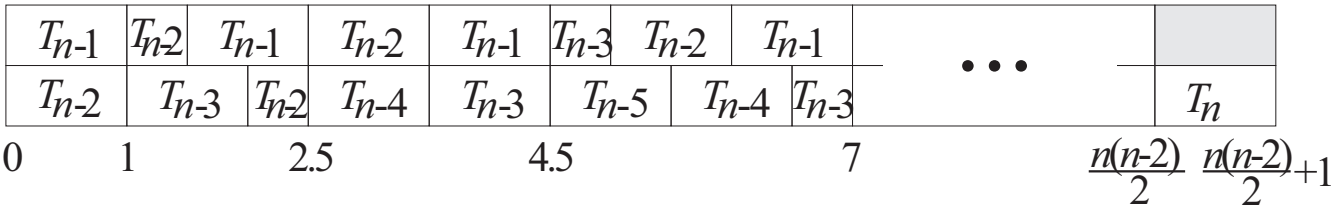
Fig. 3 Example 3 optimal schedule.

Fig.1

a)


$T_1$
$T_{n-4}$
$T_{n-3}$
$T_{n-2}$
$T_{n-1}$
$T_n$

b)


$T_{n-1}$ $T_{n-2}$ $T_{n-1}$ $T_{n-2}$ $T_{n-1}$ $T_{n-3}$ $T_{n-2}$ $T_{n-1}$ ...
$T_{n-2}$ $T_{n-3}$ $T_{n-2}$ $T_{n-4}$ $T_{n-3}$ $T_{n-5}$ $T_{n-4}$ $T_{n-3}$ $T_n$
0 1 2.5 4.5 7 $\frac{n(n-2)}{2}$ $\frac{n(n-2)}{2}+1$

Maciej Drozdowski, JOS-170-E

Fig.2

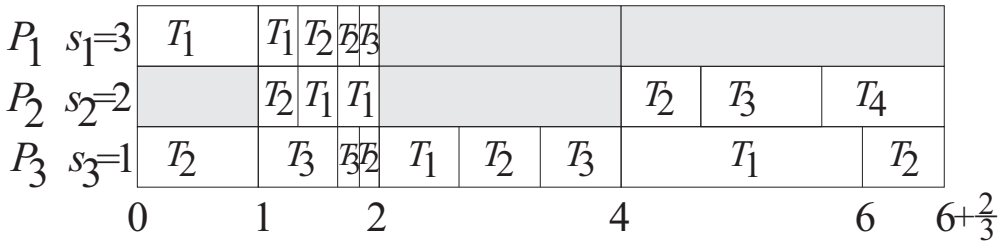


Fig.3