ELSEVIER

Available online at www.sciencedirect.com
SCIENCE DIRECT®

European Journal of Operational Research 149 (2003) 377–389

EUROPEAN JOURNAL OF OPERATIONAL RESEARCH

www.elsevier.com/locate/dsw

Scheduling multiprocessor tasks on parallel processors with limited availability

Jacek Błażewicz [a], Paolo Dell'Olmo [b,c], Maciej Drozdowski [a,*], Przemysław Mączka [a]

[a] *Instytut Informatyki, Politechnika Poznańska, Piotrowo 3a, 60-965 Poznań, Poland*
[b] *Dipartimento di Statistica, Probabilità e Statistiche Applicate, Università di Roma "La Sapienza", Piazzale Aldo Moro 5, I-00185 Rome, Italy*
[c] *Istituto di Analisi dei Sistemi ed Informatica C.N.R., Viale Manzoni 30, I-00185 Rome, Italy*

Abstract

In this work we consider the problem of scheduling multiprocessor tasks on parallel processors available only in restricted intervals of time called time windows. The multiprocessor task model applies to modern production systems and parallel applications in which several processors can be utilized in parallel. Preemptable tasks are considered. Polynomial time algorithms are given in three cases: the case of maximum lateness criterion and a fixed number of processors, the case of schedule length criterion when tasks have various ready times and require either one or all processors, and in case of schedule length criterion when the sizes of the tasks are powers of 2.
© 2002 Elsevier Science B.V. All rights reserved.

Keywords: Scheduling; Parallel computing; Multiprocessor tasks; Parallel tasks; Gang scheduling; Co-scheduling; Bandwidth allocation; Time windows

1. Introduction

In a multiprocessor task system it is allowed for some tasks to use several processors in parallel. This assumption is justified by various technological and efficiency reasons: In fault-tolerant systems tasks may be duplicated to obtain results securely [13]. Mutual testing of processors needs at least two processors working in parallel [16]. In parallel computer systems with time-sharing, *co-scheduled* parallel applications, i.e. running on several processors in the same time quantum, utilize processors with higher efficiency than applications scheduled according to other policies [13,18,26]. In bandwidth allocation [1] applications have to reserve sufficient channel capacity for a given amount of time in advance. The required bandwidth plays the role of processors. In the scheduling literature multiprocessor tasks are also referred to as *parallel*, *concurrent*, or *malleable* tasks. The idea of multiprocessor task scheduling is also conveyed by *gang scheduling*. For a more detailed survey of scheduling multiprocessor tasks, we refer the reader to [3,11,25].

* Corresponding author. Tel.: +48-61-665-2124; fax: +48-61-877-1525.
E-mail address: maciej.drozdowski@cs.put.poznan.pl (M. Drozdowski).

0377-2217/03/$ - see front matter © 2002 Elsevier Science B.V. All rights reserved.
doi:10.1016/S0377-2217(02)00760-9

In this work we assume that the availability of processors is restricted, and some machines are available only in certain intervals called time windows. Time windows may appear in case of computer breakdowns or maintenance periods. In any multitasking computer system, and in hard real time systems in particular, urgent tasks have high priority and are pre-scheduled in certain time intervals, thus creating time windows of processor availability. Scheduling in time windows was considered in e.g. [19,22–24].

Now we shall introduce the notation used in the paper. We consider a set $\mathscr{P} = \{P_1, \ldots, P_m\}$ of $m > 1$ parallel identical processors. Processors are accessible in time windows. There are l time windows. Each time window $i = 1, \ldots, l$ is characterized by: s_i—its start time, m_i—the number of processors in window i. The end of one time window is also the beginning of the succeeding window. Without the loss of generality we assume that $s_1 = 0$, $s_i < s_{i+1}$, $m_i \neq m_{i+1}$, and after time s_{l+1} no processor is available. Our definition of time windows is equivalent to the so-called *staircase* patterns of processor availability [23]. In the staircase pattern processors can be renumbered so that the intervals available on processor P_{i+1} are also available on processor P_i (cf. Fig. 1b). In this paper we consider the preemptive schedules, and any processor availability pattern can be converted to a staircase pattern by renumbering the processors within the time windows (see Fig. 1a and b).

The multiprocessor task set is $\mathscr{T} = \{T_1, \ldots, T_n\}$. Tasks are preemptable, which means that they may be suspended and restarted later and possibly on different processors without incurring any additional costs. Nonpreemptable tasks must be continuously executed on the same processor from the beginning until the very end, i.e. they cannot be suspended or migrate. Each task $T_j \in \mathscr{T}$ is defined by: size_j—the number of processors required in parallel, p_j—the processing time, r_j—the ready time, d_j—the due-date. The number of processors simultaneously required by a task will be called the *size* of the task. The set $\mathscr{T}$ of tasks can be divided into subsets according to the sizes of the tasks. Thus, $\mathscr{T} = \mathscr{T}^1 \cup \mathscr{T}^2 \cup \cdots \cup \mathscr{T}^m$, where $\mathscr{T}^i$ is a set of the n_i tasks which are executed by i processors.

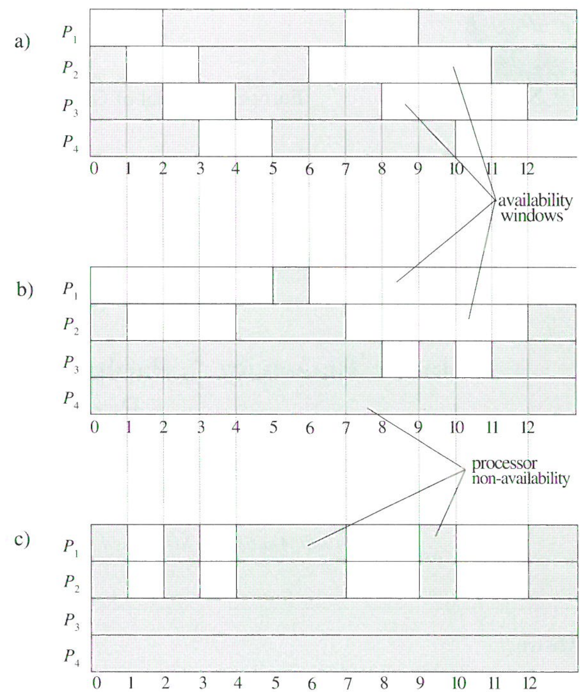

Fig. 1. Converting restricted intervals of availability. (a) The original pattern, (b) availability windows converted into a staircase pattern, (c) availability windows converted into a staircase pattern when allocation across processor partitions is not allowed.

Note that $n = n_1 + n_2 + \cdots + n_m$, and that the number of tasks with the size next to size_i is $n_{\text{size}_{i+1}}$ (not n_{i+1}). For the conciseness of the text we will name the tasks of size i as i-tasks. We assume that tasks are numbered according to their size, i.e. $\text{size}_1 \geqslant \text{size}_2 \geqslant \cdots \geqslant \text{size}_n$. In case of parallel computers with hypercube interconnection or in buddy processor allocation systems [12,17] tasks may not have arbitrary sizes. To avoid the excessive fragmentation of the processor set, tasks have sizes that are multiples of each other. For example, sizes can be powers of 2. This situation will be called a *cube* case. In the cube case only *compact* allocations are allowed, i.e. a k-task can be assigned only to processors $P_{(i-1)k+1}, \ldots, P_{ik}$, where $i = 1, \ldots, m/k$. Compact allocations result in perceiving time windows differently by tasks of different sizes (e.g. see Fig. 1c for 2-tasks). This assumption is discussed in Section 4.

The optimality criteria we consider in this paper are: schedule length $C_{\max} = \max_{T_j \in \mathcal{T}}\{c_j\}$, and maximum lateness $L_{\max} = \max_{T_j \in \mathcal{T}}\{c_j - d_j\}$, where c_j is the completion time of task $T_j \in \mathcal{T}$.

We shall denote the analyzed scheduling problems according to the three-field notation introduced in [15,25] (cf. also [3,11]). Additionally, symbol *win* in the processor field denotes the limited availability of the processors.

Below we shall review the related results on the preemptable scheduling of multiprocessor tasks. **NP**-hard problems will be denoted as **NP**h, and strongly **NP**-hard problems as s**NP**h. When the number of processors is a part of the instance input, scheduling multiprocessor tasks is computationally hard (strictly problem $P|\mathrm{size}_j, pmtn|C_{\max}$ is **NP**h) [10]. Adding precedence constraints makes problem $P3|\mathrm{size}_j, pmtn, \mathrm{chains}|C_{\max}$ s**NP**h, which follows directly from the complexity of problem $P3|\mathrm{size}_j, p_j = 1, \mathrm{chain}|C_{\max}$ [6] for unit execution time nonpreemptable tasks. Mixing precedence constraints and task due-dates makes scheduling multiprocessor tasks s**NP**h even for two processors. Hence, $P2|\mathrm{size}_j, pmtn, \mathrm{chain}|L_{\max}$ is computationally hard, which follows from the proof given for problem $P2|\mathrm{size}_j, p_j = 1, \mathrm{chain}|L_{\max}$ [7]. Consequently, $P2, \mathrm{win}|\mathrm{size}_j, pmtn, \mathrm{chain}|L_{\max}$ is s**NP**h.

When the number of processors is fixed, problem $Pm|\mathrm{size}_j, pmtn|C_{\max}$ can be solved in polynomial time [2]. Restricting possible sizes of the tasks to multiples of each other reduces the problem to case *cube* which is computationally easier because matching task sizes to fit in the processor system is easy [8]. Therefore, the problem of scheduling preemptable tasks on identical hypercubes $P|\mathrm{cube}_j, pmtn|C_{\max}$ can be solved in $O(n^2 \log^2 n)$ time [27], and scheduling on hypercubes with different speeds (problem $Q|\mathrm{cube}_j, pmtn|C_{\max}$) is solvable in $O(n \log n + nm)$ time [9]. When the precedence constraints and ready times are not present simultaneously, scheduling the multiprocessor tasks on two processors is computationally tractable. Thus, problem $P2|\mathrm{size}_j, pmtn, r_j|L_{\max}$ is solvable in polynomial time using a linear programming (LP) formulation, while problem $P2|\mathrm{size}_j, pmtn, \mathrm{prec}|C_{\max}$ can be solved in $O(n^2)$ time [4].

In the following sections we present the polynomial time algorithms for three cases of scheduling multiprocessor tasks in time windows. We start with the most general problem $Pm, \mathrm{win}|\mathrm{size}_j, pmtn, r_j|L_{\max}$ to be considered in Section 2. Then, we present low-order polynomial time algorithms for more specialized problems: $P, \mathrm{win}|\mathrm{size}_j \in \{1, m\}, pmtn, r_j|C_{\max}$ in Section 3, and problem $P, \mathrm{win}|\mathrm{cube}_j, pmtn|C_{\max}$, in Section 4.

2. Pm,win|size$_j$, $pmtn$, r_j|$L_{\max}$

The method we propose here is based on the use of LP and the notion of *processor feasible sets* of tasks. The processor feasible set of tasks is such a set of tasks that can be feasibly executed in parallel on the given number of processors. The number of different processor feasible sets is at most $\sum_{i=1}^{m} \binom{n}{i}$ which is $O(n^m)$, and is polynomially bounded with respect to the input size, provided m is fixed. Let us consider the tasks which can be executed in parallel for some given value L of maximum lateness criterion $L_{\max}$. According to the definition of the maximum lateness, T_j cannot be executed after time $d_j + L$. We will call this value the deadline of T_j. Only tasks with ready times smaller than $d_j + L$ can be executed in parallel with T_j. Also the number of available processors determines the set of tasks which can be executed in parallel. If, due to a change of L, deadline of T_j shifts from one window of processor availability to another, also the admissible processor feasible sets change. Thus, the collection of processor feasible sets changes with changing L in two types of cases: when $r_i = d_j + L$ for two tasks T_i, T_j, and when $s_k = d_j + L$ for some task T_j and window k. The first case may take place $\binom{n}{2}$ times, the second ln times. This defines $O(n^2 + nl)$ intervals $[L_a, L_{a+1}]$ of criterion $L_{\max}$ values for which the collection of processor feasible sets is constant. In each of the $[L_a, L_{a+1}]$ intervals the sequence of events (ready times, deadlines, starting a window) is constant and known. The events are ordered according to the increasing time of their appearance. Let us assume that we know that the optimal value of the maximum lateness satisfies $L \in [L_a, L_{a+1}]$. We will denote by $e_i^a (i = 1, \ldots, 2n + l + 1)$ the time instant at which event i

takes place. However, if event i is representing a deadline of some task T_j then $e_i^a = d_j$ (value of the maximum lateness is not added). Let us associate with each event i value $g_i = 1$ when event i represents a deadline, and $g_i = 0$ otherwise. For each task T_j it is possible to find index u_j of the event associated with task ready time r_j, and index v_j of the event related to T_j's deadline. Hence, T_j can be executed only in the interval $[e^a_{u_j} + g_{u_j}L, e^a_{v_j} + g_{v_j}L]$. Let M_i denote the family of the processor feasible sets of tasks existing between a pair $(i, i+1)$ of events, and M_{ij} the collection of the indices of processor feasible sets in this interval which include task T_j. With a processor feasible set with number k existing between events $(i, i+1)$ we associate variable x_{ik} denoting the duration of executing the processor feasible set. Assuming we know that value $L \in [L_a, L_{a+1}]$, our problem can be formulated as a linear program (LP):

Minimize L

subject to
$$\sum_{i=u_j}^{v_j-1} \sum_{k \in M_{ij}} x_{ik} = p_j \quad j = 1, \ldots, n \tag{1}$$
$$\sum_{k=1}^{|M_i|} x_{ik} \leqslant e^a_{i+1} - e^a_i + L(g_{i+1} - g_i) \quad i = 1, \ldots, 2n + l \tag{2}$$
$$L_a \leqslant L \leqslant L_{a+1} \tag{3}$$
$$x_{ik} \geqslant 0 \quad i = 1, \ldots, 2n + l, \quad k = 1, \ldots, |M_i| \tag{4}$$

In the above LP equalities (1) guarantee that tasks receive the required processing time. Inequalities (2) ensure that processor feasible sets defined for the interval between events $(i, i+1)$ are not executed beyond this interval. The above LP also defines a feasible schedule because in the interval between events $(i, i+1)$ only a processor feasible set with $x_{ik} > 0$ is executed. The LP has $O(n^m(n^2 + nl))$ variables and $O(n + l)$ constraints (not including constraints (4)). Thus, LP can be formulated and solved in polynomial time provided that m is fixed. To identify the interval $[L_a, L_{a+1}]$ of $L_{\max}$ values which contains the optimal value of maximum lateness (cf. constraint (3)) one needs to perform $O(\log n + \log l)$ binary search steps solving a linear program of the type described above. Since the processor system has restricted availability, a feasible solution may not exist. In this case LP is not feasible even in the last acceptable interval of $L_{\max}$ values.

3. $P, \mathrm{win}|\mathrm{size}_j \in \{1, m\}, pmtn, r_j|C_{\max}$

In this section we examine the problem of preemptively scheduling multiprocessor tasks with two possible sizes: 1 or m. Tasks have different ready times. The optimality criterion is schedule length.

Before we go into any further details we give an informal description of the algorithm. When new tasks appear at some ready time, the algorithm must decide which of the ready tasks to execute. In any feasible schedule, if an m-task can be executed in some earlier time interval occupied by 1-tasks, then swapping the m-task with the 1-tasks is feasible and does not change the schedule length. Therefore, m-tasks can be executed as soon as they are ready and m processors are available. 1-tasks, on the other hand, must be scheduled more carefully, because the longest 1-task, and the sum of processing requirements of 1-tasks in the last occupied window determine the schedule length. The algorithm given by Muntz and Coffman [21] is capable of minimizing these two values before reaching the last occupied interval. The algorithm assigns to the tasks processing capacities $\overline{\beta} = [\beta_1, \ldots, \beta_n]$ which can be intuitively understood as fractions of the available processing capability. The capacities are calculated on the basis of the task *height*. The height $h(j)$ of task T_j is the processing time required to complete it. Initially $h(j) = p_j$, for $T_j \in \mathscr{T}^1$. The ready times, and window beginnings are events in the system. Let us sort all $q \leqslant l + n + 1$ events according to their time value, i.e. $e_1 = 0 \leqslant e_2 \leqslant e_3 \cdots \leqslant e_q$. In the following algorithm we denote by A the set of 1-tasks ready by time t, by B the set of m-tasks ready by time t, and by μ_k the number of processors available in interval (e_k, e_{k+1}).

An Algorithm for $P, \mathrm{win}|\mathrm{size}_j \in \{1, m\}, pmtn, r_j|C_{\max}$

1: $t := 0$; $B := \emptyset$; $A := \emptyset$;
2: **for** $k := 1$ **to** $q - 1$ **do**
begin

3: $B := B \cup \{T_j | T_j \notin B, \text{size}_j = m, r_j \leqslant t\}$;
4: $\tau := \min\{\sum_{T_j \in B} p_j, e_{k+1} - e_k\}$; $\varepsilon := \tau$;
5: **if** $\tau \neq 0$ **and** $\mu_k = m$ **then**
begin (*m-tasks are ready and interval k has enough processors*)
6: **while** $B \neq \emptyset$ **and** $\varepsilon > 0$ **do**
if $\varepsilon \geqslant p_j$ **then begin** execute p_j units of T_j in $[e_k, e_k + \tau]$; $\varepsilon := \varepsilon - p_j$; $B := B - \{T_j\}$ **end**
else begin execute ε units of T_j in $[e_k, e_k + \tau]$; $p_j := p_j - \varepsilon$; $\varepsilon := 0$; **end;**
7: $t := t + \tau$;
end;
8: $A := A \cup \{T_j | T_j \notin A, \text{size}_j = 1, r_j \leqslant t\}$;
9: order tasks in A according to nonincreasing $h(j)$;
10: **while** $(e_{k+1} > t)$ **and** $(\exists_{T_j \in A} h(j) > 0)$ **do**
begin
11: CAPABILITIES$(A, k, \overline{\beta})$;
12: calculate times:

$$\tau' := \min\left\{\infty, \min_j \left\{\frac{h(j) - h(j+1)}{\beta_j - \beta_{j+1}} : T_j, T_{j+1} \in A, \beta_j \neq \beta_{j+1}, h(j) > h(j+1)\right\}\right\}$$

(*τ' is the shortest time required for two 1-tasks T_j, T_{j+1} with different heights to become equal with respect to their height*) **if** $\beta_{|A|} > 0$ **then** $\tau'' := h(|A|)/\beta_{|A|}$ **else** $\tau'' := \infty$;
(*τ'' is the time to the earliest completion of any 1-task*)
13: $\tau := \min\{\tau', \tau'', e_{k+1} - t\}$;
14: **for** $T_j \in A$ **do** schedule a piece of length $\tau\beta_j$ of task T_j in interval $[t, t + \tau]$ according to McNaughton rule [20];
15: **for** $T_j \in A$ **do**
begin $h(j) := h(j) - \tau\beta_j$; **if** $h(j) = 0$ **then** $A := A - \{T_j\}$ **end;**
16: $t := t + \tau$;
end;
end;
procedure CAPABILITIES(**in:**X, k;**out:**$\overline{\beta}$);
begin
17: $\overline{\beta} := \overline{0}$; $avail := \mu_k$; (*μ_k is the number of free processors in interval k*)
18: **while** $avail > 0$ **and** $|X| > 0$ **do**
begin
19: $Y{:} =$ the set of the highest tasks in X;
20: **if** $|Y| > avail$ **then**
21: **begin for** $T_j \in Y$ **do** $\beta_j := avail/|Y|$; $avail := 0$; **end;**
else (*tasks in Y cannot use all $avail$ processors*)
22: **begin for** $T_j \in Y$ **do** $\beta_j := 1$; $avail := avail - |Y|$; **end;**
23: $X := X - Y$;
end; (*of **while** loop*)
end; (*of procedure CAPABILITIES*)

Theorem 1. *The above algorithm constructs an optimal schedule for problem* $P,\text{win}|\text{size}_j \in \{1, m\}, pmtn, r_j|C_{\max}$ *and has time complexity* $O(n^2 + nl)$.

Proof. We start with the proof of schedule feasibility. Only the tasks in set A and B are executed. Task T_j is added to one of these sets only if the current time is bigger than r_j (lines 3 and 8). Thus, ready times are observed. m-tasks are executed only in windows with a sufficient number of available processors (line 5). Partial schedules for 1-tasks (built in line 14) are feasible because $\tau\beta_j \leqslant \tau$ for $T_j \in A$, and $\sum_{T_j \in A} \beta_j \tau \leqslant \mu_k \tau$ by the formulation of the procedure CAPABILITIES. Hence, the conditions set in the McNaughton rule [20] are satisfied. A task is removed from the respective set A or B, and stops being processed only when the sum of the pieces of work on the task is equal to the task processing requirement (lines 6, 15). Thus, the schedule is feasible.

Now, we prove the optimality of the schedules. The schedule length is determined by the last completion time of a task finished in the last used window. m-tasks are processed as soon as they appear in the system and a free window is available. Thus, m-tasks are completed in the earliest possible time. 1-tasks executed in the last occupied window are scheduled according to the McNaughton rule [20]. Therefore, the completion time for the tasks in the last window depends on: the longest remaining piece of a single 1-task, and the sum of the processing requirements of all remaining 1-tasks. By lines 19 to 22 the highest (i.e. the longest) 1-tasks always receive the biggest

processing capabilities. By calculation of τ' in line 12 an initially higher task may not reduce its height below some initially lower task. Thus, the remaining processing requirement of the longest 1-task is reduced in the biggest feasible way before reaching the last occupied window. By line 18 no processor remains idle when a task ready to use the processor exists. Thus, also the sum of the remaining processing requirements is minimized in the maximum possible way before reaching the last used window. We conclude that the length of the schedule in the last occupied window is the shortest possible and the schedule is optimal.

The computational complexity of the algorithm can be calculated as follows: Loop 2 is executed $O(n+l)$ times. Lines 3–8 can be executed in time $O(n)$. Line 9 can be executed in $O(n_1^2)$ total time over all algorithm run. This is a result of sorting 1-tasks once and merging the arriving tasks with tasks in set A. The procedure CAPABILITIES is executed in $O(n_1)$ time (assuming $\mu_k \leqslant n_1$). Lines 12–15 are done in $O(n_1)$ time. Thus, loop 10 is executed in time $O(n_1)$. The total running time of the algorithm is $O(n^2 + nl)$. □

When a feasible solution does not exist due to insufficient capacity of the processor system, the algorithm stops with $A \neq \emptyset$ or $B \neq \emptyset$. The above algorithm can be executed on-line because we only need information about the ready tasks, and the time of the next event.

4. $P, \mathrm{win}|\mathrm{cube}_j, pmtn|C_{\max}$

In this section we consider scheduling tasks the sizes of which are multiples of each other. This implies that m is a multiple of the greatest task size. Only compact task assignments are allowed. The schedule length is the optimality criterion. Ready times and due-dates are not taken into account. For the sake of the presentation simplicity we assume that the sizes of the tasks and m are powers of 2. The method we use bears some similarity to the one used in [5,9] to solve problems $Q|\mathrm{size}_j \in \{1,k\}, pmtn|C_{\max}$, $Q|\mathrm{cube}_j, pmtn|C_{\max}$, and in [14] to solve $Q|pmtn|C_{\max}$.

As mentioned earlier, only *compact* assignments are allowed, i.e. k-tasks can be assigned only to processors $P_{(a-1)k+1}, \dots, P_{ak}$, where $a = 1, \dots, m/k$. This assumption has both theoretical and practical justifications. Any interval of a feasible schedule for problem $P, \mathrm{win}|\mathrm{cube}_j, pmtn|C_{\max}$ can be converted to a compact assignment by renumbering the processors within the interval. In real computer systems [12,18] processor partitions are used to eliminate the fragmentation of the resources, and assigning tasks across the partition border is impossible. This may lead to different windows available for tasks of different sizes. For example, when the pattern presented in Fig. 1a is converted to a staircase pattern in Fig. 1b, window [1,4] is available for 2-tasks. This window comprises three subintervals in which different real processors are available. If the processor partitions are $\{P_1, P_2\}$, $\{P_3, P_4\}$, then interval [2,3] is not accessible for 2-tasks because it is an assignment across the partition border. Consequently, when the task assignment across the processor partitions is forbidden, the availability windows depend on the size of the tasks and the bigger the size of the tasks is the more restricted a window can be. Yet, the algorithm presented below can also be applied in this situation. The schedules are optimal because Theorem 3 (presented in the sequel) holds and the amount of processing capacity left after assigning a task of a certain size is the biggest possible.

Let us outline the main stages of the algorithm before presenting the details. First the algorithm calculates a lower bound on the schedule length. Then tasks are scheduled in the order of descending size. The tasks with the same size are scheduled in the order of nonincreasing processing time. When a feasible schedule exists, then its length is equal to the lower bound and thus it is optimal. When a feasible solution does not exist, then there is some task T_j which needs additional processing capacity. In such a case the schedule length is extended so that the processing requirement of T_j can be met. In the following we describe each stage of the algorithm.

4.1. Lower bound on the schedule length

As already stated, we consider windows of the processor availability which are staircase patterns. Therefore, P_1 is the processor capable of accommodating the biggest processing requirement. Processor P_m has the least processing ability. Let us call by *processing capacity* $\mathrm{PC}(i,t)$ the amount of work that can be performed on processor P_i until time t. To calculate the processing capacity of a processor we use function $g(i,k,t)$ which is free processing time that can be used on P_i in window k until time t, defined as follows:

$$g(i,k,t) = \begin{cases} 0 & \text{if } i > m_k \text{ or } t < s_k \\ t - s_k & \text{if } i \leqslant m_k \text{ and } s_k \leqslant t \leqslant s_{k+1} \\ s_{k+1} - s_k & \text{if } i \leqslant m_k \text{ and } s_{k+1} < t \end{cases}$$

The processing capacity is $\mathrm{PC}(i,t) = \sum_{k=1}^{l} g(i,k,t)$. Due to the staircase pattern we have $\mathrm{PC}(1,t) \geqslant \mathrm{PC}(2,t) \geqslant \cdots \geqslant \mathrm{PC}(m,t)$ for all t.

For the simplicity of the presentation let us assume that $\mathrm{size}_1 \geqslant \mathrm{size}_2 \cdots \geqslant \mathrm{size}_n$, and $p_{b+1} \geqslant \cdots \geqslant p_{b+n_k}$, where $b = \sum_{i=(\log_2 k)+1}^{\log_2 m} n_{2^i}$ is the number of tasks with the size bigger than k. A k-task can be assigned only to processors $P_{(a-1)k+1}, \ldots, P_{ak}$ existing in the staircase pattern, where $a = 1, \ldots, m/k$. Hence, the processing capacity available for k-tasks until time t is defined by $\mathrm{PC}(ka,t)$, where $a = 1, \ldots, m/k$. The value of the lower bound on the schedule length must guarantee that the longest task fits on the processor with the greatest processing capacity, the two longest tasks can be accommodated on two processors with the greatest processing capacity, etc. (cf. inequalities (5)). The total processing requirement must fit in the total available processing capacity (cf. (6)). For the completeness of the following argument, and without the loss of generality we assume that $n_k \geqslant m/k$. Let C'_k be the value of the lower bound calculated solely for k-tasks. The above observations can be formulated as the following requirement:

$$\sum_{i=1}^{a} \mathrm{PC}(ki, C'_k) \geqslant \sum_{j=b+1}^{b+a} p_j, \quad a = 1, \ldots, (m/k) - 1 \tag{5}$$

$$\sum_{i=1}^{m/k} \mathrm{PC}(ki, C'_k) \geqslant \sum_{j=b+1}^{b+n_k} p_j \tag{6}$$

Inequalities (5) and (6) determine the capacity available for k-tasks. However, there are tasks of various sizes in our problem. To deal with sizes greater than k we propose to substitute each task T_j of $\mathrm{size}_j > k$, with size_j/k artificial k-tasks with the processing requirement p_j. Let $p_{1k} \geqslant p_{2k} \geqslant \cdots \geqslant p_{n'_k k}$ be the processing times of the real k-tasks and the artificial tasks obtained by the reduction to size k. $n'_k = \sum_{i=(\log_2 k)}^{\log_2 m} n_{2^i} 2^i/k$ is the total number of such tasks. Then, the lower bound on the schedule length for the original k-tasks and the artificial ones can be found as the minimum C_k satisfying the following inequalities:

$$\sum_{i=1}^{a} \mathrm{PC}(ki, C_k) \geqslant \sum_{j=1}^{a} p_{jk}, \quad a = 1, \ldots, (m/k) - 1 \tag{7}$$

$$\sum_{i=1}^{m/k} \mathrm{PC}(ki, C_k) \geqslant \sum_{j=1}^{n'_k} p_{jk} \tag{8}$$

The lower bound on the schedule length can be found as $C = \max_{i=0}^{\log_2 m}\{C_{2^i}\}$. Let us illustrate the above concepts by means of an example.

Example 1. Task data: $n = 8$

Task	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8
Size	4	4	4	2	2	2	1	1
Processing time	4	2	1	3	2	2	4	2

Processor window data: $l = 4$

Window number	1	2	3	4	5
s_j	0	2	4	6	∞
m_j	6	4	8	6	0

The following table presents the processing capacities $\mathrm{PC}(i,t)$ available on the consecutive processors P_i by time t.

Interval of t	$[0,2]$	$[2,4]$	$[4,6]$	$[6,\infty]$
$PC(1,t)$	t	t	t	t
$PC(2,t)$	t	t	t	t
$PC(3,t)$	t	t	t	t
$PC(4,t)$	t	t	t	t
$PC(5,t)$	t	2	$t-2$	$t-2$
$PC(6,t)$	t	2	$t-2$	$t-2$
$PC(7,t)$	0	0	$t-4$	2
$PC(8,t)$	0	0	$t-4$	2

Now, we calculate the lower bound on the schedule length. We have $C_4 = 5.5$, because $PC(4,5.5) + PC(8,5.5) = p_1 + p_2 + p_3 = 7$. For the calculation of C_2 we have the following processing times of the original and artificial tasks of size 2: 4, 4, 3, 2, 2, 2, 2, 1, 1. C_2 must be in interval $[6,\infty]$ to accommodate this amount of work. For this interval we have $PC(2,C_2) + PC\ (4,C_2) + PC(6,C_2) + PC(8,C_2) = 3C_2 \geqslant 21$, i.e. processing capacity is at least equal to the total processing requirement of real and artificial 2-tasks. Hence, we have $C_2 \geqslant 7$. The processing times of real and artificial 1-tasks are: 4, 4, 4, 4, 4, 3, 3, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1, 1, 1, 1. To accommodate this amount of work t must be in interval $[6,\infty]$. In case of C_1 and inequality (8) processing capacity $\sum_{i=1}^{8} PC(i,t) = 6C_1$, must be at least equal to 48. Hence $C_1 = 8 = C$.

Let us analyze the time complexity of calculating bound C. The construction of artificial tasks requires creating at most nm tasks of size 1, $nm/2$ tasks of size 2, etc. Thus, there will be $O(nm)$ artificial tasks altogether. From these tasks we have to select a the longest ones for inequalities (7). This can be done in $O(nm)$ time for all sizes because a is $O(m)$. Thus, the construction of data for the right-hand side of inequalities (7), (8) needs $O(nm)$ time. Functions $g(i,k,t)$ can be calculated in $O(ml)$ time. The processing capacity $PC(i,t)$, as a piecewise-linear function of time t can be constructed in $O(l)$ time. The calculation of C_k in each of inequalities (7) and (8) requires $O(l)$ time. All inequalities can be verified in $O(ml)$ time. Hence, the running time of calculating the lower bound is $O(nm + ml)$.

4.2. Task scheduling rule

Tasks are assigned to processors in descending order of sizes and processing times. The processing capacity that remains after assigning tasks of a certain size is utilized by the tasks of smaller sizes. Therefore, the tasks of bigger sizes should use as little processing capacity as possible. Let us suppose tasks of sizes $m, \dots, 2k$ are already scheduled, and tasks of size k are about to be assigned to the processors. Let $PC(1) \geqslant \cdots \geqslant PC(m)$ denote the remaining processing capacities of the processor system for the schedule length C. Without the loss of generality we assume that $n_k \geqslant m/k$. For the existence of a feasible assignment of k-tasks only (tasks with smaller sizes are not taken into account) it is necessary and sufficient that the following inequalities hold:

$$\sum_{i=1}^{a} PC(ki) \geqslant \sum_{j=b}^{b+a} p_j \quad a = 1, \dots, (m/k) - 1 \tag{9}$$

$$\sum_{i=1}^{m/k} PC(ki) \geqslant \sum_{j=b}^{b+n_k} p_j \tag{10}$$

where $b = \sum_{i=(\log_2 k)+1}^{\log_2 m} n_{2^i}$. The necessity of satisfying the above inequalities is obvious. The sufficiency is demonstrated in the following discussion.

Scheduling rule R

In order to schedule k-task T_j find pair $(P_{kw}, P_{k(w+1)})$ of processors satisfying: $PC(kw) \geqslant p_j > PC(k(w+1))$.

Fill processors $P_{kw+1}, \dots, P_{k(w+1)}$ with $PC(k(w+1))$ units of task T_j in the windows available on $P_{k(w+1)}$.

Assign the remaining processing requirement of T_j equal to $p_j - PC(k(w+1))$ to processors $P_{k(w-1)+1}, \dots, P_{kw}$ from left to right in the windows which are not occupied by T_j on processors $P_{kw+1}, \dots, P_{k(w+1)}$, and are available on P_{kw} (cf. Fig. 2a).

Update the remaining processing capacities: $PC(kw-j) := PC(kw-j) - (p_j - PC(k(w+1)))$, for $j = 0, \dots, k-1$. Construct new composite processors from the remaining intervals of

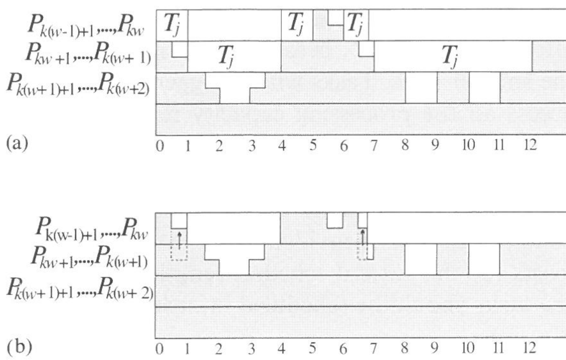

Fig. 2. Scheduling rule. (a) Assignment of a k-task to the processors, (b) restoring of the staircase pattern.

availability on pairs of processors $P_{k(w+1)-j}$ and $\mathrm{PC}_{k(w+2)-j}$, for $j = 0, \dots, k-1$ (cf. Fig. 2b). $\mathrm{PC}(k(w+2)-j) := \mathrm{PC}(k(w+2)-j) + \mathrm{PC}(k(w+1)-j) - \mathrm{PC}(k(w+1))$, for $j = 0, \dots, k-1$. Remove processors $P_{kw+1}, \dots, P_{k(w+1)}$ from data structures holding information about the available processing capacity. Renumber processors $P_{k(w+1)+1}, \dots, P_m$ to $P_{kw+1}, \dots, P_{m-k}$, respectively. Rebuild the windows to respect staircase patterns, if necessary.

Let us comment on the above rule. The assignment of task T_j is feasible because T_j receives the required processing and is executed exactly on k processors. The rule has a special form when $\mathrm{PC}(m) \geqslant p_j$. If this is the case we assume that $\mathrm{PC}(m+k) = 0$, and apply the rule as described above. Since multiprocessor tasks require several processors *simultaneously*, some intervals in the windows on $P_{k(w-1)+1}, \dots, P_{k(w+1)-1}$ may remain idle, because processors P_{kw} and $P_{k(w+1)}$ determined the processing capacity available for a k-task. After the assignment of T_j some intervals on $P_{kw+1}, \dots, P_{k(w+1)-1}$ may still be free but unavailable on $P_{k(w-1)+1}, \dots, P_{kw}$ (cf. Fig. 2b). Such free intervals must be moved to $P_{k(w-1)+1}, \dots, P_{kw}$ in order to restore the staircase pattern. The scheduling rule violates neither inequalities (9) nor (10) as shown in the following theorem.

Theorem 2. *Inequalities* (9) *and* (10) *are invariant when assigning k-tasks, according to scheduling rule R.*

Proof. Let us suppose that initially inequalities (9) and (10) were satisfied, and T_j was assigned to $P_{k(w-1)+1}, \dots, P_{kw}$ and $P_{kw+1}, \dots, P_{k(w+1)}$ by rule R. We begin with the proof of the invariance of inequalities (9).

Due to applying rule R the number of inequalities in (9) is reduced to $(m/k) - 2$. The initial processor $P_{k(w+1)}$ is removed from data structures holding information about the available processing capacity. Values of $\mathrm{PC}(k), \dots, \mathrm{PC}(k(w-1))$ are the same as before applying rule R. The new $\mathrm{PC}(kw)$ is equal to the initial $\mathrm{PC}(kw) + \mathrm{PC}(k(w+1)) - p_j$. The new values of $\mathrm{PC}(k(w+i))$ are the old values $\mathrm{PC}(k(w+i+1))$ for $i = 1, \dots, (m/k) - w - 2$.

As T_j was the longest k-task, its processing time p_j is subtracted on the right-hand side of all inequalities (9). It remains to be shown that the left-hand side is not reduced by more than p_j in any of inequalities (9). Values of $\mathrm{PC}(k), \dots, \mathrm{PC}(k(w-1))$ did not change. Hence, inequalities (9) hold for $a = 1, \dots, w-1$, because the left-hand side did not change. In the new value of the sum on the left-hand side of (9), for $a = w, \dots, (m/k) - 2$, only the new $\mathrm{PC}(kw)$ is different than before applying rule R, and is equal to the initial $\mathrm{PC}(kw) + \mathrm{PC}(k(w+1)) - p_j$. Hence, the left-hand side did not decrease more than p_j, and inequalities (9) hold for $a = i, \dots, (m/k) - 2$.

Inequality (10) holds because p_j has been subtracted on both sides of the inequality. □

The longest k-task can be feasibly scheduled because (9), (10) hold. The above reasoning can be applied inductively to the following k-tasks. Thus, tasks of size k can be feasibly scheduled provided inequalities (9), (10) hold.

After the assignment of k-tasks the processing capacities of the remaining processor system will be used by the tasks of the smaller sizes. The existence of a feasible schedule for these tasks depends on satisfying inequalities (9) and (10) for the new, smaller size tasks. Therefore, the values of the remaining processing capacities on the left-hand side of inequalities (9) and (10) should be as large as possible. In the next theorem we demonstrate that there is no rule which leaves more processing capacity than rule R. For the purposes of the

theorem let $\mathrm{PC}_R(i)$ denote the processing capacity remaining on P_i after assigning k-tasks according to our rule R, and $\mathrm{PC}_O(i)$ according to some alternative method O, while $\mathrm{PC}(i)$ is the processing capacity available for both rules before assigning any k-task.

Theorem 3. *There is no rule O for assigning k-tasks such that the remaining processing capacities satisfy*

$$\sum_{i=1}^{a} \mathrm{PC}_O(ki) > \sum_{i=1}^{a} \mathrm{PC}_R(ki) \qquad (11)$$

for at least one $a \in \{1, \dots, (m/k) - 1\}$.

Proof. Let us suppose, on the contrary, that such a rule exists and it assigns task T_j to processors $P_{qk+1}, \dots, P_{qk+k}$. Our rule assigns T_j to processors $P_{k(w-1)+1}, \dots, P_{kw}$ and $P_{kw+1}, \dots, P_{k(w+1)}$.

If $q < w$ then $\sum_{i=1}^{a} \mathrm{PC}_O(ki) \leqslant \sum_{i=1}^{a} \mathrm{PC}_R(ki)$ for $a = q, \dots, w+1$, and rule O is not better. If $q \geqslant w+1$ then the schedule built by an alternative rule O is infeasible because $p_j > \mathrm{PC}(k(w+1)) \geqslant \mathrm{PC}(kq)$.

Let us suppose O assigns T_j to more than k processors, $P_{k(q-1)+1}, \dots, P_{kq}$ are the last processors used by T_j according to O, and $q \geqslant w+1$. O may feasibly execute at most $\mathrm{PC}(k(w+1))$ units work on $P_{kw+1}, \dots, P_{kq}$. Thus, $x \geqslant p_j - \mathrm{PC}(k(w+1))$ units of work must be assigned to processors $P_1, \dots, P_{kw}$. If rule O uses any of processors $P_1, \dots, P_{k(w-1)}$ then rule R is better for $a = 1, \dots, w-1$ because these processors are not used by our rule. For $a = w$ rule O may not be better than R because amount x units of work assigned to $P_1, \dots, P_{kw}$ by O is at least equal to the amount $p_j - \mathrm{PC}(k(w+1))$ assigned by R. For $a = w+1, \dots, q-1$ rule O is not better because any smaller (than by rule R) consumption of processing capacity on $P_{k(q-1)+1}, \dots, P_{kq}$ according to O must have been compensated for by the use of processing capacity on $P_1, \dots, P_{k(w-1)}$.

Consequently, rule O is either infeasible, or inequality (11) is not satisfied. The induction over the tasks of size k completes the proof. □

The above theorem shows that rule R leaves the biggest possible processing capacity for the tasks of other sizes. Tasks of sizes smaller than k can consume all the processing capacity remaining after scheduling $\mathscr{T}^k$, because k is a multiple of all the smaller sizes. Tasks with a bigger size may not access all the processing capacity remaining after the assignment of the k-tasks because only compact assignments are allowed. Therefore, our rule guarantees the existence of a feasible schedule for smaller size tasks, provided that a feasible schedule exists for the current schedule length. As a result, the tasks should be scheduled in order of descending sizes.

Let us calculate the time complexity of the above scheduling method. Finding a collection of processors with sufficient processing capacities requires $O(\log m)$ time. The schedule for some task may have at most $l+1$ preemptions. Restoring the staircase pattern may require the change of the position of at most $O(l)$ windows. Thus, all tasks can be scheduled in $O(n(l + \log m))$ time.

4.3. Dealing with infeasible schedules

Rule R builds a feasible schedule provided that such a schedule exists for the given schedule length. However, the method of calculating lower bound C is not sufficient to grasp the interactions between the tasks of different sizes. Consequently, schedule with length C may be infeasible. Consider an example.

Example 2. Let $l = 1$, $s_1 = 0$, $s_2 = \infty$, $m_1 = 4$. There are $n = 3$ tasks defined as follows: $\mathrm{size}_1 = \mathrm{size}_2 = 2$, $\mathrm{size}_3 = 1$, $p_1 = 2$, $p_2 = 1$, $p_3 = 2$. From inequalities (7) and (8), we obtain $C_2 = C_1 = C = 2$. The schedule for 2-tasks is shown in Fig. 3a.

Processing capacities that remain in the system after scheduling the 2-tasks are $\mathrm{PC}(1) = \mathrm{PC}(2) = 1$. Inequalities (9) are not satisfied and T_3 cannot be scheduled. There is enough processing capacity to accommodate T_3, yet the capacity is located on two processors in parallel and cannot be exploited by a 1-task. The deficiency of the processing capacity for T_3 is $p_3 - \mathrm{PC}(1) = 1$.

This deficiency must be compensated for by extending schedule length by δ. Suppose the increase of the processing capacities of all processors can be accumulated to compensate for the defi-

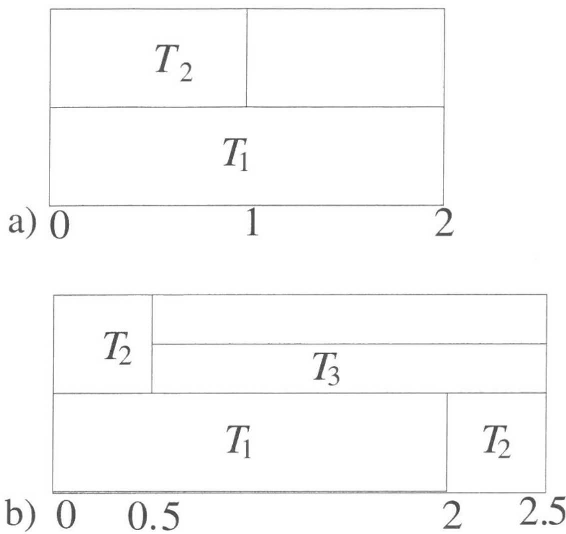

Fig. 3. Example 2. (a) Infeasible schedule, (b) optimal schedule.

ciency. Thus, $m\delta = 1$, and $C = 2.25$. However, the schedule with $C = 2.25$ is infeasible, and we have to extend it again by 1/8. On the other hand, assume that only one processor produces the capacity which will compensate for the deficiency. Then $\delta = 1$, and $C = 3$, but a schedule with this length is not optimal. The optimal schedule is presented in Fig. 3b.

From Example 2 we conclude that there are cases where no feasible schedule with length C exists. The total processing capacity is at least equal to the total processing requirement of the tasks. However, not all this capacity can be used to schedule tasks of a certain size. To calculate the minimum extension making the schedule feasible we have to know the number of processors m' which will produce additional processing capacity usable for compensating the deficiency of processing capacity.

As soon as the tasks of sizes $m, \dots, 2k$ have been scheduled, the infeasibility of the schedule for k-tasks can be identified by verifying inequalities (9). By Theorem 2 a feasible schedule for tasks of size k exists if inequalities (9) hold. Hence, in the case of infeasibility, we have to increase schedule length by some amount δ such that the above inequalities become valid. We calculate the deficiencies of processing capacity in inequalities (9): $\mathrm{DP}(a,k) = \sum_{j=b}^{b+a} p_j - \sum_{i=1}^{a} \mathrm{PC}(ki)$ for $a = 1, \dots, (m/k) - 1$. Suppose inequality for the index a is violated in (9). Then, the extension is $\delta = \mathrm{DP}(a,k)/m'$. If there are more violated inequalities, then we extend the schedule iteratively to satisfy the inequalities one by one. Note that inequality (10) is satisfied by the way of calculating C.

To calculate δ we have to determine m'. For k-tasks m' is in the set $\{1, \dots, m/k\}$. If m' is too small, then the new deficiency of processing capacity is $\mathrm{DP}'(a,k) < 0$, and the schedule is not optimal. If m' is too big then $\mathrm{DP}'(a,k) > 0$ and, if perfect precision of the calculations is possible, the search for a feasible schedule never stops. When the number is right, then the violated inequality in (9) will be satisfied with equality. This means that the exactly required amount of processing capacity is created. Thus, a wrong number m' can be easily recognized. The value of m' results from the number of processors contributing to $\mathrm{PC}(ki)$. This number is a result of creating composite processors by rule R. Since tracing interactions between tasks and windows in rule R is hard we propose to use a binary search over integers in set $\{1, \dots, m/k\}$, to find m'. A faster, but more involved algorithm of determining m' has been proposed in [9].

To verify the correctness of this method observe the following facts: Any increase of processing capacity is created on the processors and the value of the increase accumulated on the left-hand side of inequalities (9) is a multiple of δ. As schedule length increases, also the left-hand sizes of (9) increase because the capacity on the real processors increases, while tasks of size $m, \dots, 2k$ require the same amount of work as before the schedule extension.

Let us analyze the worst-case running time of the approach described above. Each extension made to accommodate a k-task requires rescheduling tasks of sizes $m, \dots, 2k$, which can be done in time $O(nl + n\log m)$ (see Section 4.2). For each violated inequality in the formulation (9), $O(\log m)$ trials must be made to find m'. The number of violated inequalities can be calculated as a sum of geometric sequence. Hence, no more than $2m$ inequalities can be violated over all the task sizes. The extensions may require $O(n(l + \log m)m\log m)$

time in total which is also the worst-case running time of the whole algorithm.

4.4. The algorithm

We summarize the algorithm described above by presenting its pseudocode. In the following formulation we denote by A the set of the violated inequalities (9).

Algorithm for $P, \mathrm{win}|\mathrm{cube}_j, pmtn|C_{\max}$

1: $C := \max\{C_m, \ldots, C_1\}$; (*$C$ is the lower bound on the makespan*)
2: **for** $f := \log_2 m$ **downto** 0 **do** (*assignment of 2^f-tasks*)
begin
3: **if** inequalities (9) do not hold for $a \in A$ **then**
begin
4: **for each** $a \in A$ **do begin** (*loop over violated inequalities*)
begin
5: calculate $\mathrm{DP}(a, 2^f)$; $m_u := m/2^f$; $m' := m_u/2$; $m_l := 1$; flag := **true**;
6: **while** $\mathrm{DP}(a, 2^f) > 0$ **and** flag **do**
begin (*binary search for the number of processors m'*)
7: $\delta := \mathrm{DP}(a, 2^f)/m'$; $C' := C + \delta$;
8: schedule tasks of sizes $m, \ldots, 2^{f+1}$ in windows contained in the interval $[0, C']$ using rule R;
9: calculate new deficiency of processing capacity $\mathrm{DP}'(a, 2^f)$;
10: **if** $\mathrm{DP}'(a, 2^f) > 0$ **then** $m_u := m'$ **else**
11: **if** $\mathrm{DP}'(a, 2^f) < 0$ **then** $m_l := m'$ **else**
12: **begin** flag := **false**; $C := C'$; **end**;
13: $m' := \lfloor \frac{m_u + m_l}{2} \rfloor$;
end; (*of binary search for the number of processors*)
end; (*of the search for the extension accommodating 2^f-tasks*)
end;
14: assign size 2^f tasks using the scheduling rule R;
end;

5. Conclusions

In this work we have considered scheduling multiprocessor tasks in the windows of processor availability. A polynomial time algorithm based on linear programming has been proposed for the general case of the problem and $L_{\max}$ criterion when the number of processors is fixed. Low-order polynomial time algorithms were proposed for problems $P, \mathrm{win}|\mathrm{size}_j \in \{1, m\}, pmtn, r_j|C_{\max}$ and $P, \mathrm{win}|\mathrm{cube}_j, pmtn|C_{\max}$.

References

[1] A. Bar-Noy, R. Canetti, S. Kutten, Y. Mansour, B. Schieber, Bandwidth allocation with preemption, SIAM Journal on Computing 28 (1999) 1806–1828.

[2] J. Błażewicz, M. Drabowski, J. Węglarz, Scheduling multiprocessor tasks to minimize schedule length, IEEE Transactions on Computers 35 (1986) 389–393.

[3] J. Błażewicz, M. Drozdowski, K. Ecker, Management of resources in parallel systems, in: J. Błażewicz, K. Ecker, B. Plateau, D. Trystram (Eds.), Handbook on Parallel and Distributed Processing, Springer, Berlin-Heidelberg, 2000, pp. 263–341.

[4] J. Błażewicz, P. Dell'Olmo, M. Drozdowski, Scheduling multiprocessor tasks on two parallel processors, Proceedings of the 2nd International Conference on Parallel Computing Systems, Ensenada, Mexico, 1999, pp. 165–170.

[5] J. Błażewicz, M. Drozdowski, G. Schmidt, D. de Werra, Scheduling independent multiprocessor tasks on a uniform k-processor system, Parallel Computing 20 (1994) 15–28.

[6] J. Błażewicz, Z. Liu, Scheduling multiprocessor tasks with chain constraints, European Journal of Operational Research 94 (1996) 231–241.

[7] P. Brucker, S. Knust, D. Roper, Y. Zinder, Scheduling UET task systems with concurrency on two parallel identical processors, Mathematical Methods in Operations Research 52 (2000) 369–387.

[8] E.G. Coffman Jr., M.R. Garey, D.S. Johnson, Bin packing with divisible item sizes, Journal of Complexity 3 (1987) 406–428.

[9] M. Drozdowski, Scheduling multiprocessor tasks on hypercubes, Bulletin of the Polish Academy of Sciences, Technical Sciences 42 (1994) 437–445.

[10] M. Drozdowski, On complexity of multiprocessor task scheduling, Bulletin of the Polish Academy of Sciences. Technical Series 43 (1995) 381–392.

[11] M. Drozdowski, Scheduling multiprocessor tasks—An overview, European Journal of Operational Research 94 (1996) 215–230.

[12] D.G. Feitelson, L. Rudolph, Evaluation of design choices for gang scheduling using distributed hierarchical control, Journal of Parallel and Distributed Computing 35 (1996) 18–34.

[13] E.F. Gehringer, D.P. Siewiorek, Z. Segall, Parallel Processing: The Cm* Experience, Digital Press, Bedford, 1987.

[14] T. Gonzales, S. Sahni, Preemptive scheduling of uniform processor systems, Journal of the ACM 25 (1978) 92–101.
[15] R.I. 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.
[16] H. Krawczyk, M. Kubale, An approximation algorithm for diagnostic test scheduling in multicomputer systems, IEEE Transactions on Computers 34 (1985) 869–872.
[17] P. Krueger, T.-H. Lai, V.A. Dixit-Radiya, Job scheduling is more important than processor allocation for hypercube computers, IEEE Transactions on Parallel and Distributed Systems 5 (1994) 488–497.
[18] R.N. Lagerstrom, S.K. Gipp, PScheD: Political scheduling on the CRAY T3E, in: D.G. Feitelson, L. Rudolph (Eds.), Job Scheduling Strategies for Parallel Processing: Proceedings of IPPS'97 Workshop, Lecture Notes in Computer Science, 1291, Springer-Verlag, Berlin, Heidelberg, 1997, pp. 117–138.
[19] Z. Liu, E. Sanlaville, Preemptive scheduling with variable profile, precedence constraints and due dates, Discrete Applied Mathematics 58 (1995) 253–280.
[20] R. McNaughton, Scheduling with deadlines and loss functions, Management Science 6 (1959) 1–12.
[21] R.R. Muntz, E.G. Coffman Jr., Preemptive scheduling of real-time tasks on multiprocessor systems, Journal of the ACM 17 (1970) 324–338.
[22] E. Sanlaville, G. Schmidt, Machine scheduling with availability constraints, Acta Informatica 35 (1998) 795–811.
[23] G. Schmidt, Scheduling on semi-identical processors, Zeitschrift für Operations Research A 28 (1984) 153–162.
[24] G. Schmidt, Scheduling with limited machine availability, European Journal of Operational Research 121 (2000) 1–15.
[25] B. Veltman, B.J. Lageweg, J.K. Lenstra, Multiprocessor scheduling with communications delays, Parallel Computing 16 (1990) 173–182.
[26] J. Zahorjan, E.D. Lazowska, D.L. Eager, The effect of scheduling discipline on spin overhead in shared memory parallel systems, IEEE Transactions on Parallel and Distributed Systems 2 (1991) 180–199.
[27] Y. Zhu, M. Ahuja, On job scheduling on a hypercube, IEEE Transactions on Parallel and Distributed Systems 4 (1993) 62–69.