Scheduling malleable tasks for mean flow time criterion

M. Caramia¹, M. Drozdowski²

Abstract

In this paper we study scheduling malleable tasks with limited parallelism, for mean flow time criterion. Malleable tasks may use more than one processor at the same time, and the number of processors used may change over time. The maximum number of processors that can be used by some task is limited. We examine the computational complexity of this problem, and present polynomially solvable cases.

Keywords: Deterministic scheduling, malleable tasks, mean flow time.

1 Introduction

Malleable tasks can be executed by more than one processor at the same time. Furthermore, the number of used processors can be changed over the course of a task execution. Malleable task model may be applied to represent parallel applications executed in environments in which migration is possible. For example, on a parallel computer with shared memory a parallel application can create threads. These threads can be executed simultaneously. Operating system assigns the threads to the processors for time quanta in a round-robin fashion, and preempts the threads when the quanta expire. When the load of the computer system is low all the application threads may run in parallel in real time. When the load is increasing, operating system assigns the application threads to fewer processors. Thus, the number of processors used over time can be changed according to the decisions of the operating system. An upper limit on the number of usable processors may exist. This may be either the number of threads created by the application, or a limit imposed by the operating system protecting its resources.

¹Istituto per le Applicazioni del Calcolo, CNR, V.le del Policlinico, 137 - 00161 Rome, Italy
²Institute of Computing Science, Poznan University of Technology, Piotrowo 3A, 60-965 Poznan, Poland. The research of this author has been partially supported by Polish Committee for Scientific Research. Corresponding Author. Email: Maciej.Drozdowski@cs.put.poznan.pl
from overuse. Another example of malleable tasks is in bandwidth allocation. Bandwidth of a communication link is a resource which can be divided among many simultaneously operating channels. The bandwidth assigned to a channel may vary over time. However, the channels have an upper limit on the usable bandwidth (e.g. Peak Cell Rate in ATM networks). A router must divide the bandwidth between the simultaneous communications such that the maximum for each channel is not exceed. Malleable task model for loom production scheduling has been presented in [9]. A single request for production of a certain fabric can be distributed over several looms. The number of looms used during the course of satisfying the request may vary.

The scheduling problem studied in this paper can be formulated in the following way. Set $T$ of $n$ tasks is to be executed on set $P$ of $m$ parallel identical processors. Each task $j \in T$ is defined by the parameters: processing requirement $p_j$, maximum number of processors that can be used $\delta_j$, ready time $r_j$, and deadline $d_j$ which cannot be exceeded in any feasible schedule. Tasks can be suspended, and restarted later without any additional cost. Each task can migrate to a different processor, increase or decrease the number of used processors, also without cost. The only restriction is that no more than $\delta_j$ processors can be used simultaneously. To verify if task $j$ has received the required processing and can be finished, one has to calculate the area occupied by task $j$ in the time × processors space, and compare it with $p_j$. The completion time of task $j$ will be denoted by $c_j$. The objective is the minimization of the mean flow time $\frac{1}{n} \sum_{j=1}^{n} (c_j - r_j)$. Since $\sum_{j=1}^{n} r_j$ is constant for any instance, the minimization of the mean flow time is equivalent to the minimization of $\sum_{j=1}^{n} c_j$. Therefore, in the following discussion we will refer to the minimization of $\sum_{j=1}^{n} c_j$ as to the mean flow time criterion.

Malleable task scheduling has been considered in earlier publications. The first works considering parallel tasks, i.e. tasks executed on several processors simultaneously, seem to be [8], and [1]. Unfortunately, the lack of generally accepted terminology may confuse. It is often the case that the name malleable tasks is applied to parallel tasks that can be executed on several processors, but the number of processors must be selected before the task starts, and cannot be changed during the execution of the task. We follow the naming conventions proposed in [4, 6] where such tasks are called moldable. We do not consider moldable tasks here. The concept of malleable, moldable, and more generally parallel tasks, and the problems of scheduling them have been presented in [3, 4, 6]. The first study of scheduling malleable tasks appeared in [10]. Tasks had due-dates, and the objective was the minimization

\[ \sum_{j=1}^{n} (c_j - r_j) \]
of maximum lateness. This problem can be solved by means of binary search and maximum network flows. Scheduling chains of three malleable tasks for schedule length criterion has been studied in [5]. The first, and the last task in the chain had parallelism limited to one processor ($\delta_j = 1$). The second, central task had unlimited parallelism ($\delta_j \geq m$). This problem has been shown to be NP-hard, and special cases solvable in polynomial time have been identified [5]. However, to our best knowledge not much is known about the problem of scheduling malleable tasks for the mean flow time criterion.

The rest of this paper is organized as follows: In Section 2 we study the complexity of the proposed problem. Section 3 is dedicated to the case of fixed sequences of task completion times. A low-order polynomial time algorithm is proposed in Section 4 for agreeable processing times and parallelism maxima.

2 Complexity of the problem

In this section we demonstrate that the problem of scheduling malleable tasks with bounded parallelism is NP-hard in general.

**Theorem 1** The problem of scheduling malleable tasks with limited parallelism, ready times, and deadlines, for mean flow time criterion is NP-hard.

**Proof.** We start by proving that this problem is in NP. A solution of the problem can be represented as a set of intervals in which the number of processors assigned to the tasks does not change. The number of such intervals is $O(n^2)$ because ready times, completion times, and deadlines define $O(n)$ periods, in which the processor assignment to the tasks changes $O(n)$ times (we discuss it in more detail in the next section). For each interval one has to verify if no task uses more than the admissible number of processors. By summing the amounts of work performed on the tasks in the consecutive intervals one can verify that each task is fully completed. Finally, the mean flow time is calculated by checking the sum of completion times of the tasks.

We will show now a polynomial time transformation from the problem **PARTITION INTO EQUAL CARDINALITY SUBSETS** [7] to a decision version of our problem. **PARTITION INTO EQUAL CARDINALITY SUBSETS** is defined as follows:

**Instance:** a set of $2k$ integers $A = \{a_1, \ldots, a_{2k}\}$, such that $\sum_{j=1}^{2k} a_j = 2B$. 

3
Question: is it possible to partition \( A \) into two disjoined subsets \( A_1, A_2 \) such that \( |A_1| = |A_2| = k \), and \( \sum_{j \in A_1} a_j = \sum_{j \in A_2} a_j = B \)?

The decision version of our problem is defined as follows:

Instance: set \( T \) of \( n \) malleable tasks with processing requirements \( p_j \in Z^+ \), maximum number of usable processors \( \delta_j \in Z^+ \), ready times \( r_j \in Z^+ \), and deadlines \( d_j \in Z^+ \), for \( j = 1, \ldots, n \), integer \( m \), a positive rational number \( y \). Question: is it possible to execute tasks from set \( T \) such that \( \sum_{j=1}^n c_j \leq y \)?

The polynomial time transformation is defined as follows:

\[ m = kMB^2 - B^2; \; n = 2k + 2; \]
\[ \delta_j = MB^2 - a_j B \text{ for } j = 1, \ldots, 2k; \]
\[ p_j = \delta_j + a_j \text{ for } j = 1, \ldots, 2k; \]
\[ r_j = 0, d_j = \infty \text{ for } j = 1, \ldots, 2k; \]
\[ \delta_{2k+1} = m - 1; \; p_{2k+1} = (m - 1)B; \; r_{2k+1} = 0, d_{2k+1} = B; \]
\[ \delta_{2k+2} = m; \; p_{2k+2} = mL; \; r_{2k+2} = B + 1; d_{2k+2} = B + L + 1; \]
\[ y = B + B + 1 + L + k(B + 1) + k(B + 1 + L) + 2k; \]

where \( L > (B + 3)k \), and \( M > k \) are big constants. Tasks \( 1, \ldots, 2k \) will be called partition tasks, task \( 2k + 1, 2k + 2 \) will be called blocking tasks.

Let us assume that there is a partition into equal cardinality subsets. Then, a feasible schedule for our problem can be as the one presented in Fig.1. Task \( 2k + 1 \) is finished at time \( B \), task \( 2k + 2 \) is finished at time \( B + 1 + L \), and the \( k \) tasks corresponding to the elements of set \( A_1 \) are completed at time \( B + 1 \). The \( k \) tasks corresponding to the elements \( j \in A_2 \) complete at times \( B + 1 + L + \frac{p_j}{\delta_j} \) where \( \frac{p_j}{\delta_j} = \frac{3B^2 - a_j (B - 1)}{3B^2 - a_j B} \leq 2 \). Together we obtain mean flow time \( \sum_{j=1}^n c_j = B + B + 1 + L + k(B + 1) + k(B + 1 + L) + \sum_{j \in A_2} \frac{p_j}{\delta_j} \leq y \)

Suppose that a feasible schedule with mean flow time at most \( y \) exists. We will demonstrate that also a partition into equal cardinality subsets must exist. Note that by the selection of their ready times, deadlines, and the shortest execution times \( \frac{p_j}{\delta_j}, \frac{p_{j+1}}{\delta_{j+1}} \), tasks \( 2k + 1 \), and \( 2k + 2 \) must be finished at times \( B \) and \( B + 1 + L \), respectively. This leaves free intervals \([0, B], [B, B + 1], [B + 1 + L, \infty)\) with \( m \) processors, available for the partition tasks \( 1, \ldots, 2k \).

Let us observe that in order to have mean flow time not greater than \( y \), at least \( k \) tasks from the set \( 1, \ldots, 2k \) must be completed before task \( 2k + 2 \), i.e. before time \( B + 1 \). Suppose it is otherwise and \( x > k \) partition tasks are completed after task \( 2k + 2 \). Then, the sum of the completion times for the blocking tasks, and \( x \) partition tasks completed after task \( 2k + 2 \) is at least
\[ B + B + 1 + L + x(B + 1 + L) \geq B + B + 1 + L + (k + 1)(B + 1 + L) > y = B + B + 1 + L + k(B + 1) + k(B + 1 + L) + 2k \text{ because } L > k(B + 3). \]

On the other hand at most \( k \) partition tasks can be completed before time \( B + 1 \). Suppose it is otherwise and \( x > k \) partition tasks are completed before \( B + 1 \). Since for each partition task \( p_j = MB^2 - a_jB \geq MB^2 - B^2 \), these tasks require at least \( x(MB^2 - B^2) \geq (k + 1)(MB^2 - B^2) \) processing, while the available area is \( B + m = B + kMB^2 - B^2 \), which is smaller because \( M > k \). Hence, no more than \( k \) partition tasks can be completed before \( B + 1 \). Together we have that exactly \( k \) partition tasks must be completed before \( B + 1 \). If we denote the set of tasks completed before \( B + 1 \) by \( A_1 \), and the rest as \( A_2 \), then we have \( |A_1| = |A_2| \).

Note that a partition task \( j \) executed in the interval \([B, B + 1]\) can receive at most \( \delta_j \) processing. The remaining part \( p_j - \delta_j = a_j \) must be processed in the interval \([0, B]\). Now we will prove that also \( \sum_{j \in A_1} a_j = \sum_{j \in A_2} a_j = B \). Suppose that it is otherwise, and \( \sum_{j \in A_1} a_j > B \). Then, \( \sum_{j \in A_1} (p_j - \delta_j) = \sum_{j \in A_1} a_j \geq B + 1 \), at least one unit of work must be processed after \( B + 1 \), and the criterion value \( y \) is not met. Note that there is free space in the interval \([B, B + 1]\) in the amount of \( m - \sum_{j \in A_1} \delta_j = kMB^2 - B^2 - kMB^2 + B \sum_{j \in A_1} a_j > B \), but it cannot be exploited by any partition task in \( A_1 \) because the maximum number of processors is already used. Suppose that \( \sum_{j \in A_1} a_j < B \). Then, the total processing requirement of the tasks in \( A_1 \) is \( \sum_{j \in A_1} p_j = \sum_{j \in A_1} (\delta_j + a_j) = \sum_{j \in A_1} (MB^2 - a_j(B - 1)) \geq kMB^2 - (B - 1)(B -}
1) \(= kMB^2 - B^2 + 2B - 1\) which is greater than the space \(B + kMB^2 - B^2\) available in \([0, B+1]\). Hence, tasks in \(A_1\) cannot be feasibly completed before \(B+1\). Thus, we conclude that a feasible schedule not exceeding mean flow time \(y\) exists if \(\sum_{j \in A_1} a_j = \sum_{j \in A_2} a_j = B\), and the answer to partition with equal cardinality subsets is also positive. \(\square\)

3 Fixed sequences

In this section we present a linear programming solution for the case when the sequence of task completions, ready times and deadlines are known. We start the presentation with a simpler case.

3.1 Fixed sequence of completion times

In this paragraph we assume that all tasks are available at time 0, and have not bounding deadlines (e.g. \(\forall j \; d_j = \infty\)). Without loss of generality let us assume that the sequence of task completions is \(c_1 \leq c_2 \leq \ldots \leq c_n\). Let us denote by \(x_{ij}\) the amount of processing that task \(j\) receives in the interval \([c_{i-1}, c_i]\), for \(i = 1, \ldots, n\). For completeness of arguments we assume \(c_0 = 0\). The linear program is as follows:

minimize \(\sum_{i=1}^{n} c_i\)

subject to:

\[
    x_{ij} \leq \delta_j(c_i - c_{i-1}) \quad j = 1, \ldots, n; \; i = 1, \ldots, j
\]  

(1)

\[
    \sum_{j=1}^{n} x_{ij} \leq m(c_i - c_{i-1}) \quad i = 1, \ldots, n
\]  

(2)

\[
    \sum_{i=1}^{j} x_{ij} \geq p_j \quad j = 1, \ldots, n
\]  

(3)

In the above linear program inequalities (1) guarantee that no task \(j\) uses more than \(\delta_j\) processors in the interval \([c_{i-1}, c_i]\). By inequalities (2) tasks processed in the interval \([c_{i-1}, c_i]\) use no more processing than the capacity of the \(m\) processors. Inequalities (3) ensure that all tasks receive necessary processing.

Though the above linear program includes constraints necessary for feasibility of a schedule, it is not known yet if a feasible schedule can be constructed using the solution of (1)-(3). A feasible schedule can be built using
an extension of McNaughton's algorithm proposed in [5] for schedule length criterion ($C_{max}$). We describe the extension for the sake of completeness of the presentation. Tasks with processing requirements $p_j$, and parallelism bound $\delta_j$, can be scheduled in time

$$C_{max} = \max \left\{ \max_j \left\{ \frac{p_j}{\delta_j} \right\}, \frac{1}{m} \sum_{j=1}^{n} p_j \right\}.$$

(4)

This is necessarily a lower bound because no schedule can be shorter than the length of the longest task or the total processing requirement equally distributed on all processors. A schedule of this length is built by using McNaughton’s wrap-around rule. However, here if a task is wrapped it may use more than one processor at the same time. By the selection of $C_{max} \geq \max_j \left\{ \frac{p_j}{\delta_j} \right\}$ it is guaranteed that no task $j$ uses more than $\delta_j$ processors simultaneously. Let us return now to scheduling the pieces $x_{ij}$ of the tasks in the intervals $[c_{i-1}, c_i]$. By constraints (1)-(2), pieces $x_{ij}$ fulfill condition (4) imposed by the extended McNaughton rule, and can be feasibly scheduled in the intervals $[c_{i-1}, c_i]$.

We conclude this section with an example in which we have $m = 4$ processors, and three tasks such that $c_1 \leq c_2 \leq c_3$, $p_1 = 2$, $p_2 = 5$, $p_3 = 4$, and $\delta_1 = 2$, $\delta_2 = 4$, $\delta_3 = 1$. The linear program is as follows:

minimize $c_1 + c_2 + c_3$

subject to:

\[
\begin{align*}
x_{11} &\leq 2c_1 \\
x_{12} &\leq 4c_1 \\
x_{22} &\leq 4(c_2 - c_1) \\
x_{13} &\leq c_1 \\
x_{23} &\leq (c_2 - c_1) \\
x_{33} &\leq (c_3 - c_2) \\
x_{11} + x_{12} + x_{13} &\leq 4c_1 \\
x_{22} + x_{23} &\leq 4(c_2 - c_1) \\
x_{33} &\leq 4(c_3 - c_2) \\
x_{11} &\geq 2 \\
x_{12} + x_{22} &\geq 5
\end{align*}
\]
Figure 2: Illustration to the example in Section 3.1.

\[ x_{13} + x_{23} + x_{33} \geq 4 \]  

(5)

By solving the above linear program we obtain: \( x_{11} = 2, x_{12} = 1, x_{13} = 1, x_{22} = 4, x_{23} = \frac{4}{3}, x_{33} = \frac{5}{3}, c_1 = 1, c_2 = \frac{7}{3}, c_3 = 4 \). The optimal schedule is depicted in Fig.2.

### 3.2 Fixed sequence of all events

When the sequence of \( r_j \)'s, \( d_j \)'s, and \( c_j \)'s is fixed, our problem can be formulated as a linear program. Let us consider simultaneously all such events: ready times, due dates, completion times. We will denote the number of these events by \( l \). Let \( \tau_i \) and \( \tau_{i+1} \) denote the endpoints of an interval determined by two consecutive events, for \( i = 1, \ldots, l - 1 \). Note that \( \tau_i \) is a constant if it represents a ready time, or a deadline. \( \tau_i \) is a variable if event \( i \) is a completion time. Thus, we have the following formulation:

minimize \( \sum_{i=1}^{l} \tau_i \)

subject to:

\[ x_{ij} \leq \delta_j (\tau_i - \tau_{i-1}) \quad i = 1, \ldots, l \]  

(6)

\[ \sum_{j=1}^{n} x_{ij} \leq m(\tau_i - \tau_{i-1}) \quad i = 1, \ldots, l \]  

(7)

\[ \sum_{i=1}^{l} x_{ij} \geq p_j \quad j = 1, \ldots, n \]  

(8)

\[ x_{ij} = 0 \text{ if } \tau_{i-1} < r_j \quad i = 1, \ldots, l \]  

(9)

\[ x_{ij} = 0 \text{ if } \tau_i > d_j \quad i = 1, \ldots, l \]  

(10)
The main difference with respect to the linear program (1)-(3) is that in the above formulation we consider consecutive events which are not necessarily two completion times. Though, the objective function is a sum of time instants of all events, ready times and deadlines are fixed, and the sum of the \(\tau_i\)'s corresponding to them is constant. Therefore, minimizing \(\sum_{i=1}^{\delta} \tau_i\) is equivalent to minimizing \(\sum_{i=1}^{n} c_i\). Furthermore, we force to zero \(x_{ij}\) in inequalities (9) and (10), for those intervals \(i\) which are before the availability of task \(j\), or after the deadline of task \(j\).

Note that for a fixed number of tasks, the number of possible permutations of task completion times, ready times, and deadlines is also fixed. Hence, we have an observation.

Observation 1 The problem of scheduling malleable tasks with ready times and deadlines is solvable in polynomial time for any fixed number of tasks.

4 Agreeable processing requirements and parallelism maxima

In this section we study a special case of agreeable processing requirements and parallelism bounds. For this case a low-order polynomial time algorithm can be given.

By agreeable processing requirements, and parallelism bounds we mean the instances for which tasks can be ordered such that \(d_1 \leq d_2 \leq \ldots \leq d_n\) and \(\delta_1 \leq \delta_2 \leq \ldots \leq \delta_n\). The agreeable feature of an instance can be checked in \(O(n \log n)\) time by sorting the tasks. We also assume \(r_j = 0, d_j = \infty\), for all tasks \(j\). The algorithm can be formulated as follows:

Algorithm Agreeable
1: for \(j:=1\) to \(n\) do
2: assign task \(j\) to the earliest possible time intervals using maximum possible number of processors, i.e. either \(\delta_j\) or all the processors remaining available in a given time interval.

Let us illustrate this algorithm with an example. Processing requirements are given in a vector \(\bar{p} = [2, 4, 4, 5, 7]\), parallelism bounds are given in a vector \(\bar{\delta} = [1, 1, 2, 2, 4]\), \(m = 5\). The schedule built by algorithm Agreeable is shown in Fig.3.
Let us make some observations about the schedules built by algorithm Agreeable. Let $\alpha_j$ denote the number of processors used by task $j$ at the end of its execution in a schedule constructed by algorithm Agreeable.

**Theorem 2** $\alpha_j = \min\{\delta_j, m\}$, for $j = 1, \ldots, n$.

**Proof.** The proof is inductive in nature. The theorem is satisfied for $j = 1$. Assume it is satisfied for tasks $1, \ldots, j$, where $j \geq 1$. Let us consider the time interval $a$ in which task $j$ is executed (cf. Fig.4). Interval $a$ is the earliest possible time where task $j + 1$ can be executed because there are no idle intervals to the left of $a$. Otherwise task $j$ would have been shifted to such earlier intervals.

a) Suppose there are some free processors in interval $a$, and task $j + 1$ fits completely in interval $a$. Let $\tau_a$ denote the length of the sub-interval with free processors within $a$. We have $\tau_a \leq \frac{\nu_j}{\alpha_j}$ because task $j$ may be executed also before the sub-interval with free processors. On the other hand for task $j + 1$
we have $\frac{p_{i+1}}{\delta_{i+1}} \leq \tau_a$ because $j + 1$ fits completely in the interval. Together we get $\frac{p_{i+1}}{\delta_{i+1}} \leq \tau_a \leq \frac{p_i}{\delta_i}$. But due to the agreeable condition $\frac{p_{i+1}}{\delta_{i+1}} \geq \frac{p_i}{\delta_i}$. Consequently, $\frac{p_{i+1}}{\delta_{i+1}} = \tau_a = \frac{p_i}{\delta_i}$, and $\alpha_{j+1} = \delta_{j+1}$. Furthermore, if one task is completely processed in parallel with some other task then they are finished simultaneously.

b) Suppose $j + 1$ does not fit completely in the interval $a$. Thus, $c_{j+1} > c_j$. It follows from the previous case that after completion of task $j$ all processors are free because all tasks executed in parallel with $j$ finish no later than by $c_j$. Hence, $\alpha_{j+1} = \min \{m, \delta_{j+1}\}$. \hfill $\Box$

**Theorem 3** Algorithm Agreeable constructs the optimum schedule in $O(n^2)$ if $\frac{p_1}{\delta_1} \leq \frac{p_2}{\delta_2} \leq \ldots \leq \frac{p_n}{\delta_n}$ and $\delta_1 \leq \delta_2 \leq \ldots \leq \delta_n$.

**Proof.** This proof has inductive nature.

1) Schedule task 1 using $\alpha_1 = \min \{m, \delta_1\}$ processors. Mean flow time $c_1$ is minimum.

2) Suppose an optimum schedule for tasks 1, \ldots, $j$ is constructed by algorithm Agreeable. We schedule task $j + 1$ using algorithm Agreeable. $\sum_{i=1}^{j+1} c_i$ cannot be reduced by:

   a) reducing $\sum_{i=1}^{j} c_i$ because the schedule for tasks 1, \ldots, $j$ is optimal,

   b) reducing only $c_{j+1}$ because it is infeasible.

   Thus, reducing $c_{j+1}$ and increasing $\sum_{i=1}^{j+1} c_i$ is the only way of reducing $\sum_{i=1}^{j+1} c_i$. Suppose we reduce $c_{j+1}$ by $\varepsilon_{j+1}$. This reduces the area available for task $j + 1$ by $\varepsilon_{j+1} \alpha_{j+1}$ which must be compensated for by delaying the completion times of some tasks among 1, \ldots, $j$. Without loss of generality, let them be tasks 1, \ldots, $k$ and their completions are delayed by $\varepsilon_1, \ldots, \varepsilon_k$, respectively. This creates available area of at most $\sum_{i=1}^{k} \varepsilon_i \alpha_i$. This new area can be consumed by task $j + 1$, in exchange for area $\varepsilon_{j+1} \alpha_{j+1}$. Thus, we reduce the completion time of task $j + 1$ by no more than $\frac{\sum_{i=1}^{k} \varepsilon_i \alpha_i}{\alpha_{j+1}} \geq \varepsilon_{j+1}$.

By Theorem 2 and agreeable condition $\alpha_i \leq \alpha_{j+1}$, for $i = 1, \ldots, k$. Hence, we have:

$$\varepsilon_1 + \ldots + \varepsilon_k \geq \frac{\sum_{i=1}^{k} \varepsilon_i \alpha_i}{\alpha_{j+1}} \geq \varepsilon_{j+1}$$

(11)

which means that the increase of the mean flow time by $\varepsilon_1 + \ldots + \varepsilon_k$ exceeds the reduction of $\varepsilon_{j+1}$. This conclusion can be invalidated only if some task(s) $i \in \{1, \ldots, k\}$ use $\alpha_i' > \alpha_{j+1}$ processors. Due to the agreeable condition, and Theorem 2, we have $\alpha_i' \leq \delta_i = \alpha_i \leq \alpha_{j+1}$ and (11) holds.
The complexity of the algorithm is a result of the fact that in step 2 of algorithm Agreeable the number of available processors for task \( j \) changes at most \( n - 1 \) times, and at most this many times the remaining processing requirement of task \( j \) must be recalculated. \( \square \)

5 Conclusions

In this paper we studied a problem of scheduling malleable tasks with bounded parallelism. The problem is \( \text{NP}- \) hard in the presence of ready times and deadlines. For fixed sequence of ready times, deadlines, and task completion times it can be solved in polynomial time by use of linear programming. When processing requirements and parallelism bounds of the tasks are agreeable, a low-order polynomial time algorithm was proposed. Yet, the complexity of a more fragile problem of scheduling malleable tasks with bounded parallelism without ready times and deadlines remains open.

References


