Scheduling multiprocessor tasks for mean flow time criterion

Maciej Drozdowski[1 †], Paolo Dell'Olmo[2‡]

[1]Instytut Informatyki, Politechnika Poznańska, Poznań, Poland
[2]Dipartimento di Statistica, Probabilità e Statistiche Applicate, University of Rome "La Sapienza", Rome, Italy, and
Istituto di Analisi dei Sistemi ed Informatica, Rome, Italy

Scope and purpose

Efficient use of scarce resources is of vital importance both in computer and production systems. Scheduling theory attempts to provide guidelines and algorithms for efficient assignment of the resources to the activities. For a long time the classical scheduling theory assumed that a task can be executed only by one processor at the given moment of time. This assumption is too restrictive in the case of parallel computer systems and modern production systems where tasks (e.g. programs) can be processed on several processors in parallel. Therefore, a new scheduling model of multiprocessor tasks has been proposed recently. Multiprocessor tasks may require more than one processor simultaneously. In this work a deterministic multiprocessor task scheduling problem is analyzed. Most of the earlier publications on this subject assumed schedule length (makespan) optimality criterion. Here we consider different optimality criteria which are the mean flow time and the mean weighted flow time. These are important for the user of the computer system rather than for the owner of the computing facility because they are very closely related to the average waiting time observed by the user.

†To whom all correspondence should be addressed. Email: maciej_d@sol.put.poznan.pl
This research has been partially supported by a KBN grant and project CRIT2.
‡dellolmo@iasi.rm.cnr.it

Scheduling multiprocessor tasks for mean flow time criterion

Maciej Drozdowski[1 †], Paolo Dell'Olmo[2‡]

[1]Instytut Informatyki, Politechnika Poznańska, Poznań, Poland
[2]Dipartimento di Statistica, Probabilità e Statistiche Applicate, University of Rome "La Sapienza", Rome, Italy, and
Istituto di Analisi dei Sistemi ed Informatica, Rome, Italy

Abstract

Multiprocessor tasks are executed by more than one processor at the same moment of time. This work considers the problem of scheduling unit execution-time and preemptable multiprocessor tasks on m parallel identical processors to minimize mean flow time and mean weighted flow time. We analyze complexity status of the problem. For parallel identical processors, when tasks have unit execution time and the number of processors is arbitrary the problem is shown to be computationally hard. Constructing an optimal preemptive schedule is also computationally hard in general. Polynomial algorithms are presented for scheduling unit execution time tasks when the number of processors is fixed, or the numbers of simultaneously required processors are powers of 2. The case of preemptable tasks requiring either 1 or m dedicated processors simultaneously is solvable in low-order polynomial time.

Keywords

Deterministic scheduling, parallel computer systems, multiprocessor tasks, mean flow time.

---

†To whom all correspondence should be addressed. Email: maciej_d@sol.put.poznan.pl
This research has been partially supported by a KBN grant and project CRIT2.
‡dellolmo@iasi.rm.cnr.it

Scheduling multiprocessor tasks for mean flow time criterion

Maciej Drozdowski[1 †], Paolo Dell'Olmo[2‡]

[1]Instytut Informatyki, Politechnika Poznańska, Poznań, Poland
[2]Dipartimento di Statistica, Probabilità e Statistiche Applicate, University of Rome "La Sapienza", Rome, Italy, and
Istituto di Analisi dei Sistemi ed Informatica, Rome, Italy

1 Introduction

Parallel processing receives growing attention of the research community and the industry. An important element of parallel processing success is its efficiency. Hence, there is a need for efficient scheduling algorithms.

In this work we assume that applications (or programs) are multiprocessor tasks. Multiprocessor task model assumes that some tasks may require several processors at the same moment of time. Examples of multiprocessor tasks include programs executed on several processors in parallel which vote for the reliable final result [10], mutual testing of processors by each-other [13], file transfers [6] which require two 'processing' elements simultaneously: the sender and the receiver. Multiprocessor tasks are also favorable for efficiency reasons and represent *coscheduled* (also called *gang-scheduled*) tasks [10, 19]. An extensive description of the multiprocessor task concept can be found in [2, 8, 18].

We formulate now the considered scheduling problem. The computer system consists of set $\mathcal{P} = \{P_1, \dots, P_m\}$ of processors.

The task set $\mathcal{T} = \{\mathcal{T}^1, \mathcal{T}^2, \dots, \mathcal{T}^m\}$ is composed of subsets: $\mathcal{T}^1$ of n_1 tasks requiring one processor, $\mathcal{T}^2$ of n_2 tasks requiring two processors in parallel, ..., $\mathcal{T}^m$ of n_m tasks requiring m processors simultaneously, where $n = n_1+n_2+\dots+n_m$. For the sake of conciseness we will address uniprocessor tasks as 1-tasks, biprocessor tasks as 2-tasks, triple-processor tasks as 3-tasks, etc. When tasks are scheduled on parallel identical processors, the number of processors simultaneously required by task T_j will be called task size and

†To whom all correspondence should be addressed. Email: maciej_d@sol.put.poznan.pl
This research has been partially supported by a KBN grant and project CRIT2.
‡dellolmo@iasi.rm.cnr.it

denoted $size_j$. In the case of dedicated processors not the number, but set fix_j of simultaneously required processors is given. Note, that for each task only one possible number or only one set of simultaneously required processors is given. Tasks are independent. In the following we distinguish two possible situations for task processing times. Tasks have either unit execution times (UET) and are *nonpreemptable* or the processing times of the tasks are arbitrary but tasks are *preemptable*. When preemptions are not allowed all tasks must be executed continuously on the same processor(s) from the beginning till the very end. Preemptability means that each task can be suspended, and restarted later (possibly on a different processor) without inducing additional overheads. In the former case $t_j = 1$ for $j = 1, \ldots, n$. In the latter case processing time is denoted t_j for task T_j. Tasks may arrive into the system at different moments of time or may have limited duration of availability for processing. Therefore, task T_j ($j = 1, \ldots, n$) is characterized by ready time r_j and due-date d_j. Weights w_j ($j = 1, \ldots, n$) may be given for the tasks to reflect their relative urgency or value.

The optimality criteria considered are mean flow time: $\sum_{j=1}^{n} c_j$, and mean weighted flow time $\sum_{j=1}^{n} w_j c_j$, where c_j is the completion time of task T_j. C_{max} denotes schedule length (though it is not the optimality criterion analyzed in this work).

To denote considered problems the three-field notation introduced in [11, 18] will be used (cf. also [2, 8]). Symbol P in the first field means that the number of processors is given in the particular instance of the problem. $P2$ means that the number of processors is equal 2 and fixed by the formulation of the problem. According to [18] word $size_j$ in the task field means that tasks may require arbitrary numbers of processors. Word $cube_j$ is used if the numbers of processors required by the tasks are powers of 2. Term fix_j means that tasks require certain sets of processors.

Now, we briefly review related results for classical and multiprocessor tasks scheduling on parallel processors. For the case of mean flow time criterion and parallel identical processors preemption is not profitable [17]. Thus, preemptive and nonpreemptive cases (problems $P||\sum c_j$ and $P|pmtn|\sum c_j$) can be solved by an algorithm based on the Shortest Processing Time (SPT) rule. Introducing different ready times or weights makes the problem computationally hard, therefore the following problems are **NP**-hard: $P2||\sum w_j c_j$, $P2|pmtn|\sum w_j c_j$ [5], $P2|pmtn, r_j|\sum c_j$ [9], $1|pmtn, r_j|\sum w_j c_j$ [14]. UET case is slightly less complex because problems $P|p_j = 1, r_j, d_j|\sum c_j$ [15], $P|p_j = 1|\sum w_j c_j$ [17] can be solved in polynomial time. As far as multiprocessor task case is considered, it has been shown [16] that problem $P2|size_j|\sum w_j c_j$ is strongly **NP**-hard, and problem $P2|size_j|\sum c_j$ is **NP**-hard. In the case of dedicated processors problem $P|fix_j, p_j = 1|\sum c_j$ is **NP**-hard in the strong sense, wich was shown in [12]. For fixed number of processors problems $Pm|fix_j, p_j = 1|\sum w_j c_j$, and $Pm|fix_j, p_j = 1, r_j|\sum c_j$ are polynomially solvable [4].

Further organization of the work is as follows. In Section 2 we consider scheduling UET tasks, and in Section 3 preemptive scheduling is analyzed. Conclusions and a table summing the results are presented in the last section.

2 UET tasks

2.1 $P|size_j, p_j = 1|\sum c_j$

In this section we consider scheduling unit execution time (UET) tasks. First we show that the problem of scheduling UET tasks with mean flow time criterion is strongly **NP**-hard.

Theorem 1 *Problem $P|size_j, p_j = 1|\sum c_j$ is* **NP***-hard in the strong sense.*

Proof. The problem is in **NP**. We prove strong **NP**-hardness by reduction of 3-PARTITION to a decision version of our problem. 3-PARTITION is defined as follows.

3-PARTITION

INSTANCE: Set A of $3q$ integers a_j $(j = 1, \ldots, 3q)$, such that $\sum_{j=1}^{3q} a_j = Bq$ and $B/4 < a_j < B/2$ for $j = 1, \ldots, 3q$.

QUESTION: Can A be partitioned into q disjoint subsets $A_1, \ldots, A_q$ such that $\sum_{a_j \in A_i} a_j = B$ for $i = 1, \ldots, q$?

3-PARTITION can be transformed to our scheduling problem as follows:

$m = B$; $n = 3q$;
$size_j = a_j, t_j = 1$ for $j = 1, \ldots, n$;
$y = \sum_{i=1}^{q}(3i) = \frac{3}{2}(q + q^2)$.

We ask whether a schedule with mean flow time at most equal to y exist. First, assume that the answer to 3-PARTITION is positive. Then, in a feasible schedule with mean flow time at most equal y three tasks T_e, T_f, T_g corresponding to three elements a_e, a_f, a_g forming set A_i $(i = 1, \ldots, q)$, such that $\sum_{a_j \in A_i} a_j = a_e + a_f + a_g = B$, are executed in parallel (cf. Fig.1). Thus, in time unit i tasks corresponding to the elements of set A_i are executed in parallel.

Now suppose a feasible schedule with mean flow time less than or equal y exists. We will show that such a schedule must not be longer than q. Note, that due to the processing requirement no shorter shedule than q may exist. Assume, on the contrary, that the schedule is longer than q but its mean flow time is y or less. Let us denote by n' the number of tasks that are not completed by time q. On the one hand due to not completing the n' tasks before time q mean flow time is reduced by at most qn' in relation to y. On the other hand, these tasks also contribute at least $(q+1)n'$ to the mean flow time. Thus, any schedule completed after q has bigger mean flow time than y. Consequently, the schedule must not be longer than q and must have the form presented in Fig.1. Hence, also the 3-PARTITION instance must have a positive answer. □

2.2 $Pm|size_j, p_j=1|\sum w_j c_j$

The method of solving this problem is based on the concept of a *processor feasible set* of tasks (PFS in short). A collection of tasks is processor feasible when the tasks can be processed in parallel on the given number of processors. The number M of PFSes is at most $\binom{n}{m}$ and is polynomially bounded provided that m is fixed (i.e. it is $O(n^m)$). Note, that the collection of tasks executed in each time unit of any feasible schedule for the considered problem is a processor feasible set.

Let us observe that a greedy algorithm first executing PFSes with the biggest total weight of the included tasks is not optimal in general. To verify this observation consider an example.

Example 1.
Assume k, m, w are positive integers, and $m \geq k > w + 1$. All tasks have unit execution time.

$n = 2k$;
$w_j = 1, size_j = 1$ for $j = 1, \dots, k$;
$w_j = w, size_j = m - 1$ for $j = k + 1, \dots, 2k$.

A greedy algorithm schedules tasks $T_1, \dots, T_k$ in the first time unit, and tasks $T_{k+1}, \dots, T_{2k}$ in the interval $[2, k+1]$. The optimality criterion is $\sum w_j c_j = k + \sum_{i=2}^{k+1} wi$. On the other hand, a schedule in which pairs of tasks (T_i, T_{i+k}) for

$i = 1, \ldots, k$ are executed in parallel has mean flow time $\sum w_j c_j = \sum_{i=1}^{k} (w + 1)i$. The second schedule is better when $w > \frac{k-1}{2}$.

It can be shown by interchange argument that among the tasks with the same $size_j$ tasks with a bigger value of the weight should be processed first. Thus, let us assume that tasks with the same size k are ordered according to nonincreasing value of weight and that T_j^k denotes the j-th k-task, while w_j^k is its weight. The method we use here is based on the technique proposed in [3, 4]. This method reduces the problem to finding a shortest path in some directed graph $G = (V, E)$.

Each of the vertices in G is a tuple $(t, i_1, \ldots, i_m, l_1, \ldots, l_m)$ where $0 \le t \le n$, $i_j = 0, \ldots, n_j$ and $l_j = 0, \ldots, n_j$ for $j = 1, \ldots, m$. Such a vertex represents a partial schedule with i_j tasks of size $j \quad (j = 1, \ldots, m)$ started before or at time t. l_j is the number of j-tasks $\quad (j = 1, \ldots, m)$ completed at time $t + 1$. Thus, numbers $l_1, \ldots, l_m$ represent a processor feasible set of tasks started at t and completed at $t + 1$. $(0, \ldots, 0)$ is a starting vertex representing state of the schedule when no decision about tasks started at time 0 is made yet. Vertex $(n, n_1, \ldots, n_m, 0, \ldots, 0)$ is the terminus of the network representing the situation when all tasks are completed.

There are arcs of two types in E. The first type of arcs links nodes with the same value of time t. The arcs of this type represent a situation that some processor feasible set is started at time t. Let A_j $(j = 1, \ldots, M)$ denote a processor feasible set started at t. Then, an arc exists from node $(t, i_1, \ldots, i_m, 0, \ldots, 0)$ to node $(t, i_1 + l_i, \ldots, i_k + l_k, \ldots, i_m + l_m, l_1, \ldots, l_m)$. Note that l_k $(k = 1, \ldots, m)$ is the number of k-tasks included in A_j. A_j can be scheduled at t if and only if there are still unexecuted tasks of the sizes A_j comprises. This means that such an arc exists if and only if $i_k + l_k \le n_k$ for $k = 1, \ldots, m$. Starting A_j at t bears cost $\sum_{a=1}^{m} \sum_{b=1}^{l_a} (t + 1) w_{i_a + b}^{a}$ in the mean flow time. The second type of arcs joins nodes with different time values: $(t, i_1, \ldots, i_m, l_1, \ldots, l_m)$ with $(t + 1, i_1, \ldots, i_m, 0, \ldots, 0)$. These arcs bear no cost in the mean flow time.

A path in G from node $(0, \ldots, 0)$ to the node $(n, n_1, \ldots, n_m, 0, \ldots, 0)$ represents a feasible schedule. The length of the path is equal to the value of the mean flow time criterion. Hence, the shortest such path in G from $(0, \ldots, 0)$ to $(n, n_1, \ldots, n_m, 0, \ldots, 0)$ represents the optimal schedule. There are $O(n^{2m+1})$ nodes in G, at most $O(n^m)$ arcs direct into any node. Hence, construction of G requires $O(n^{3m+1})$ time. The shortest path can be found in $O(n^{3m+1})$ time. The total time complexity of the proposed method is

$O(n^{3m+1})$. Thus, the problem can be solved in polynomial time provided that the number of processors m is fixed.

2.3 $P|cube_j, p_j=1|\sum w_j c_j$

In this section we analyze the case when task sizes are multiples of each other and m is a multiple of the biggest task size. In particular it is the case of hypercube computers and buddy processor allocation schemes where tasks require number of processors which is a power of 2. This case is simpler than the previous ones because matching sizes of tasks to be processed in parallel can be done in polynomial time [7]. For the simplicity reasons we assume in the further presentation that sizes of tasks are powers of 2.

The algorithm we propose for this problem builds processor feasible sets in the nonincreasing order of the total value of included tasks. The number of processors occupied by the PFS built gradually increases from 1 to m. A pseudocode of the algorithm for problem $P|cube_j, p_j=1|\sum w_j c_j$ is as follows. We will refer to this algorithm as MWS (Merge by Weight and Size) algorithm.

Algorithm MWS

begin
 PFS:=$\emptyset$;
 for i=0 **to** $\log_2 m - 1$ **do**
  begin
   L:=list of 2^i-tasks sorted according to nonincreasing weights;
   PFS':=list of pairs of the highest weight elements
     merged from list L and PFS;
(* in a pair on list PFS' both elements can come from PFS, or both elements may come from L, or one element may come from PFS and one from L, depending on the total weight of the combination *)
   PFS:=PFS';
  end;
 L:=list of $m-$tasks sorted according to nonincreasing weights;
 PFS':=PFS and L merged in the order of nonincreasing weight;
 schedule tasks in the order from list PFS';
end.

Theorem 2 *Algorithm MWS builds optimal schedules for problem $P|cube_j$, $p_j=1|\sum w_j c_j$ in $O(n(\log n + \log m))$ time.*

Proof. Let us first analyze optimality of MWS algorithm. Suppose that in the optimal schedule total weight of the tasks executed in the first time unit is not maximal. Let $\mathcal{T}^*$ denote a set of tasks which can be executed in parallel and has the highest total weight of the included tasks. We will demonstrate that every schedule can be transformed to a one in which $\mathcal{T}^*$ is executed in the first time unit, and that the conversion does not increase the mean weighted flow time.

Let l_k where $k = 2^i, i = 0, \ldots, \log m$ denote the number of $k-$tasks in set $\mathcal{T}^*$. Without loss of generality we assume that $\sum_{k=0}^{\log m} l_k 2^k = m$. Otherwise the desired collection is not the heaviest possible because empty slots are left that can be filled by any task, or the number of the tasks of certain sizes is not big enough. If $\sum_{k=0}^{\log m} l_k 2^k < m$ then there are no tasks of size $2^{\lfloor \log_2 (m - \sum_{k=0}^{\log m} l_k 2^k) \rfloor}, \ldots, 1$. In this situation we add $m - \sum_{k=0}^{\log m} l_k 2^k$ 1-tasks with weight 0.

The first step of the conversion consists in shifting tasks in their initial time units of processing (cf. Fig.2a). Without changing the mean flow time $1-$tasks of set $\mathcal{T}^*$ can be assigned to processors $P_1, \ldots, P_{l_1}$. Analogously 2^k-tasks can be reassigned to processors $P_{a+1}, \ldots, P_{a+l_k 2^k}$ where $a = \sum_{i=0}^{k-1} l_i 2^i$. This transformation will be called move *type one*. The second step consists in shifting tasks included in $\mathcal{T}^*$ from their original time units to the first time unit of the schedule. This shift is feasible when the replacement is wider than the tasks being replaced (cf.Fig.2b). We will call this kind of transformation move *type two*. In the opposite case (the task being replaced is wider than the collection replacing it) move *type three* is applied. The narrower tasks from a latter time unit must replace some task(s) of the same (total) size in an earlier time unit where another task to be shifted is executed. Then, the pair can replace another wider task (or collection of tasks) in the time unit where another task(s) desired for shifting is(are) executed (cf.Fig.2c). In this way all the required tasks can be feasibly collected in one time unit which finally is swapped with the first time unit in the schedule. The first type of moves does not change mean flow time because completion times are not changed. The second and third types of moves do not increase mean flow time because always the tasks with higher weight are executed earlier than in the original schedule. The tasks which due to the swaps are finally executed later must have weights smaller than or equal to the weight of their replacements. Otherwise they would have been selected for set $\mathcal{T}^*$. Hence, the mean flow time has been reduced which contradicts assumption that the schedule is optimal. We conclude that the first time unit must have the biggest possible total weight.

This reasoning can be applied recursively to the following time units of the optimal schedule. Hence, time unit i should have the biggest total weight for the tasks executed in interval $[i, C_{max}]$.

Algorithm MWS builds schedules in which the first time unit has the biggest weight PFS for tasks selected from $\mathcal{T}$. The second time unit has the biggest total weight possible for the remaining tasks, etc.

Construction of list L requires $O(n \log n)$ time over all the algorithm run, each construction of list PFS' (merging) requires $O(n)$ time. The **for** loop of the algorithm is executed $O(\log m)$ times, therefore, algorithm MWS has complexity $O(n(\log n + \log m))$. $\square$

3 Preemptive scheduling

In this section we assume that processing times are arbitrary but tasks are preemptable. We first analyze the complexity of scheduling on parallel processors. Then, we propose a polynomial time algorithm for problem $P|fix_j, pmtn|\sum c_j$ where $|fix_j| \in \{1, m\}$, i.e. when tasks use either one dedicated processor or all m processors.

$P|size_j, pmtn|\sum c_j$

The idea of **NP**-hardness proof for $P|size_j, p_j = 1|\sum c_j$ used in Theorem 1 cannot be applied here. It is because tasks are preemptable. Let us examine an example.

Example 2.

$m = 4, n = 6, n_1 = 4, n_2 = 1, n_3 = 1, n_4 = 0,$
$size_j = 1$ for $j = 1, \ldots, 4$
$size_5 = 2, size_6 = 3$, processing times of all the tasks are 1.

It is clear that partition of the task set into two subsets of equal total sizes does not exist because the sum of sizes is odd. However, this does not imply that the shortest schedule has length 3. A feasible schedule of length 2.25 is shown in Fig.3.

The proof of Theorem 1 was based on the observation that schedule length must be integer. This is no longer true here. From the above example we infer that Theorem 1 cannot be extended to the preemptive case. Yet, **NP**-hardness in the ordinary sense can be shown.

Theorem 3 *Problem* $P|size_j, pmtn|\sum c_j$ *- is* **NP**-*hard.*

Proof. The proof is based on showing a Turing transformation of PARTITION problem to our scheduling problem. PARTITION is defined below.

PARTITION
INSTANCE: Set A of q integers a_j $(j = 1, \ldots, q)$, such that $\sum_{j=1}^{q} a_j = 2B$ and $a_j < B$ for $j = 1, \ldots, q$.
QUESTION: Does set $A' \subset A$ such that $\sum_{a_j \in A} a_j = \sum_{a_j \in A-A'} a_j = B$ exist?
The instance of the scheduling problem is described below.

$m = B, n = q + k;$
$size_j = a_j, t_j = \frac{1}{4}$ for $j = 1, \ldots, q;$
$size_j = m, t_j = C$ for $j = q+1, \ldots, q+k;$
C, k are integers such that $C \gg q$, $k > q$ and $k > \frac{Bq}{2}$.

We will call tasks $T_1, \ldots, T_q$ small, and tasks $T_{q+1}, \ldots, T_{q+k}$ big. We ask whether a schedule with mean flow time not greater than $y = \frac{q+k}{2} + \frac{(k+1)kC}{2}$ exists. The construction we described requires $k > \frac{Bq}{2}$ big tasks which poses a question whether the transformation is polynomial. Transformation is polynomial because description of big tasks requires only the number of tasks, their size and processing time. The length of such a string is $O(\log B + \log q)$.

Suppose the answer to PARTITION is positive. Then a schedule can be built in which tasks corresponding to elements in A' are executed in interval $[0, \frac{1}{4}]$, tasks corresponding to elements in $A - A'$ are executed in $[\frac{1}{4}, \frac{1}{2}]$, and big tasks are executed at the end of the schedule. The value of the mean flow time is $\sum_{j \in A'} \frac{1}{4} + \sum_{j \in A-A'} \frac{1}{2} + \sum_{j=1}^{k} (\frac{1}{2} + Cj) \leq \frac{q}{2} + \frac{k}{2} + \frac{k(k+1)C}{2} = y$.

Now suppose a feasible schedule with mean flow time at most y exists. In such a schedule small tasks must be executed first. Otherwise, if any of big tasks were executed first, the value of the mean flow time would increase in relation to y by at least $C + q(C + \frac{1}{4}) - \frac{q}{2} - (C + \frac{1}{2}) > 0$. Suppose that the last small task is completed at $\frac{1}{2} + \varepsilon$ and $\varepsilon > 0$. Then, mean flow time for big tasks is $k\varepsilon + \frac{k}{2} + \frac{(k+1)kC}{2}$. Now, let us examine the value of ε when the answer to PARTITION is negative. This means that any PFS of tasks uses at most $m - 1$ processors which creates inevitable idle time. The amount of idle time is at least $\frac{1}{2}$ which must be compensated in extending the schedule for small tasks by at least $\frac{1}{2B}$. Thus, $\varepsilon \geq \frac{1}{2B}$. The mean flow time for small tasks must be at least $\frac{q}{4}$, which would be the case if all small tasks were processed in parallel. Hence, total mean flow time would be at least $\frac{q}{4}+\frac{k}{2B}+\frac{k}{2}+\frac{k(k+1)C}{2}$. If $k>\frac{Bq}{2}$ then the total mean flow time is bigger than y. Thus, negative answer to PARTITION implies negative answer to our scheduling problem. And vice versa, positive answer to the scheduling problem implies that all small tasks must be finished before $\frac{1}{2}+\frac{1}{2B}$ which is possible only if total idle time created in the part of the schedule for small tasks is less than $\frac{1}{2}$ which is possible only when some PFS uses all processors. Thus, the answer in PARTITION is positive. $\square$

$P|fix_j, pmtn|\sum c_j$ and $|fix_j| \in \{1, m\}$

It is assumed here that tasks require either one dedicated processor or all m processors simultaneously. In the schedule for this problem one can distinguish intervals of time where $1-$tasks are executed and intervals of time where $m-$tasks are executed. In the intervals of one of the above two types interruption of tasks execution is not profitable which follow standard scheduling theory results [17]. Closer examination of the order among tasks of the same size leads to the following conclusion.

Theorem 4 *In the schedule for $P|fix_j, pmtn|\sum c_j$ where $|fix_j| \in \{1, m\}$, tasks of the same size are ordered according to SPT rule.*

Proof. The proof can be done by the interchange argument. $\square$

However, the order between m-tasks and 1-tasks is not obvious. It seems that our problem has some similarity with $1|out-tree|\sum c_j$. It is because of the following properties of the optimal schedules:
1. preemption during processing m-tasks is not profitable,
2. properties mentioned in Theorem 4,
3. m-tasks should be started at the beginning of the schedule or at the end of some other task (this feature will become clear in the further discussion).

Due to Theorem 4 schedule of 1-tasks is fixed. m-tasks can be inserted in this schedule only in time moments where some 1-task(s) complete. Therefore, schedule of 1-tasks can be transformed into a chain of artificial tasks $T_1' \rightarrow T_2' \rightarrow \ldots \rightarrow T_l'$ representing intervals between completions of the consecutive original 1-tasks. The chain of artificial tasks representing the schedule of 1-tasks and m tasks can be combined into an out-tree with an artificial 0-processing time task as a root. Thus, it seems that our problem can be reduced to $1|out-tree|\sum c_j$. Problem $1|out-tree|\sum c_j$ is solvable in $O(n\log n)$ time [1] or cf. e.g.[2]. However, the similarity to $1|out-tree|\sum c_j$ is superficial. Let us examine an example.

Example 3.

$m=2, n=3$,
$fix_1=\{P_1\}$, $fix_2=\{P_2\}$, $t_1 == t_2=3$,
$fix_3=\{P_1,P_2\}$, $t_3=2$.

In the schedule for 1-tasks T_1 and T_2 are executed in parallel. Hence, the chain representing this schedule consist of a single artificial task of length $t'=3$. The algorithm for $1|out-tree|\sum c_j$ must decide which of the tasks should be execute first T_3, or the artificial task representing 1-tasks schedule. The algorithm selects T_3 to start the schedule because $t_3<t'$. This results in mean flow time 12. On the other hand, if T_3 is executed last, mean flow time is 11. We conclude from this example that our problem is not equivalent to $1|out-tree|\sum c_j$.

Let us analyze yet another example to verify other possible positions of m-tasks in relation to 1-tasks.

Example 4.

$m=2, n=9$,
$fix_1=fix_3=fix_5=fix_7=\{P_1\}, fix_2=fix_4=fix_6=\{P_2\}$,
$t_1=\ldots=t_3=1$, $t_4=\ldots=t_6=2$, $t_7=3$;
$fix_8=fix_9=\{P_1,P_2\}$, $t_8=1, t_9=2$.

The order of $1-$tasks is SPT order, shown in Fig.4a. The question is where to insert the $2-$tasks. Suppose, T_8 starts the partial schedule for $T_1,\ldots,T_8$. This increases mean flow time by 8. The same would happen if T_8 were executed last. Yet, when T_8 is inserted after T_1 and T_2 mean flow time increases by 7. Suppose T_8 is executed right after T_1 and T_2. By Theorem 4 task T_9 must be executed after T_8. When T_9 is executed immediately after T_8, mean flow time is increased by 14. However, if it was executed after task T_6, or at the end of the schedule the mean flow time would grow by 10.

It can be concluded from Example 4 that the position of m-tasks in relation to 1-tasks is not governed by a simple rule. There can be many equally good insertion points of an m-task. To find a right position of an m-task we propose to calculate a simple piece-wise linear function which specifies the increase of the mean flow time if m-task T_j were inserted into some given initial

schedule S for tasks $T_1, \ldots, T_{j-1}$ at time t. If in the following discussion initial schedule S is clearly identified by the context we denote such a function $f(j,t)$. Where distinction between the initial schedules for tasks $T_1, \ldots, T_{j-1}$ is needed this function will be denoted $f(S,j,t)$ (cf. proof of Theorem 5). Function f can be seen as a superposition of two other functions of t:
a) the increase of the mean flow time due to starting T_j at t,
b) a decreasing function which reflects the number of tasks completed before starting T_j.
The value of $f(j,t)$ decreases by $t_j k_i$ at the points $x_i\,(i = 1, \ldots, q)$ of the initial schedule where k_i tasks are completed. Thus, $f(j,t)$ can be calculated using the following rules:

(i) $f(j,0) = jt_j$;

(ii) $f(j,t)$ increases linearly with t, provided $t \neq x_i\,(i = 1, \ldots, q)$;

(iii) $f(j,t)$ decreases by $k_i t_j$ when $t = x_i\,(i = 1, \ldots, q)$.

It can be observed that $f(j,t)$ has local minima only at $0, x_1, \ldots, x_q$. Thus, we can restrict further examination of insertion points for m-tasks to these $q+1$ time moments. Furthermore, it is the reason for inserting m-tasks only at the end of some other tasks, which was alluded to before Example 3. Function $f(j,t)$ and its minima can be calculated in $O(n)$ time. Functions $f(8,t)$ and $f(9,t)$ (when T_8 is starting at time 1) for the Example 4 are depicted in Fig.4b. We are able now to formulate an algorithm for problem $P|fix_j, pmtn|\sum c_j$ where $|fix_j| \in \{1, m\}$. Without loss of generality we assume that $|fix_1| \leq \ldots \leq |fix_n|$.

Algorithm for $P|fix_j, pmtn|\sum c_j$ where $|fix_j| \in \{1, m\}$.

begin
  build an SPT schedule for $1-$tasks $T_1, \ldots, T_{n_1}$;
  order $m-$tasks according to SPT rule;
  for j=1 **to** n_m **do**
    begin
      construct $f(n_1 + j, t)$ and find its minimum $f(n_1 + j, t')$;
      insert m-task T_j into the schedule for tasks $T_1, \ldots, T_{j-1}$ at t';
    end;
end.

The algorithm has complexity $O(n^2)$. The optimal schedule for Example 4 is shown in Fig.4c. From the form of example functions in Fig.4b one observes that there may be many equally good positions for inserting $m-$tasks into the initial schedule for 1-tasks. Does the selection of T_j insertion point contribute to the minima achieved by functions $f(j+1,t),\dots,f(n,t)$? We are going to demonstrate that it is not the case.

Theorem 5 *The insertion point t' for $m-$task T_j is immaterial for the minimum of $f(j+1,t),\dots,f(n,t)$, provided that $f(j,t')$ is a minimum of $f(j,t)$.*

Proof. Suppose $m-$task T_j can be inserted in two places t' and t'', where $t' < t''$, and there are no other equally good insertion points between t' and t''. We select t'' and show that $f(j+1,t)$ has the same minimum if t' were selected for the insertion of T_j. The proof will be given by contradiction.

Function $f(j,t)$ has minima at time moments where some tasks finish in the schedule for tasks $T_1,\dots,T_{j-1}$. Therefore, we can restrict our considerations to these points in time. Let $l_i \quad (i=1,\dots,a)$ denote lengths of intervals between two consecutive completions of some task(s), in the schedule for tasks $T_1,\dots,T_{j-1}$ and included in $[t',t'']$. Thus, $t''=t'+\sum_{i=1}^{a} l_i$. k_i is the number of tasks finished at the end of interval i. Since t',t'' are equivalent for the insertion of T_j we have $f(j,t')=f(j,t'+\sum_{i=1}^{a} l_i)$. By the method of $f(j,t)$ construction we have

$$f(j,t'+\sum_{i=1}^{a} l_i)=f(j,t')+\sum_{i=1}^{a} l_i - t_j\sum_{i=1}^{a} k_i,$$

and from this

$$\sum_{i=1}^{a} l_i - t_j\sum_{i=1}^{a} k_i = 0. \tag{1}$$

Let $f(S_1,j+1,t)$ denote the function returning the increase in the mean flow time due to processing task T_{j+1} when T_j was started at t'. Similarly $f(S_2,j+1,t)$, denotes the increase of the mean flow time incurred by T_{j+1} provided that T_j was started at t''.

Suppose T_j is inserted at t', then the local minimum of $f(S_1,j+1,t)$ achieved at the end of some interval $x\in\{1,\dots,a-1\}$ is

$$f(S_1,j+1,t'+t_j+\sum_{i=1}^{x} l_i)=f(S_1,j+1,t')+t_j-t_{j+1}+\sum_{i=1}^{x} l_i - t_{j+1}\sum_{i=1}^{x} k_i. \tag{2}$$

Alternatively, T_j can be started at t''. Then,

$$f(S_2,j+1,t''+t_j)=f(S_1,j+1,t')+\sum_{i=1}^{a} l_i - t_{j+1}\sum_{i=1}^{a} l_i + t_j - t_{j+1} \tag{3}$$

Suppose starting point t' is better than t''. This means that there exist some point $t'+t_j+\sum_{i=1}^{x} l_i$ where

$$f(S_1, j+1, t' + t_j + \sum_{i=1}^{x} l_i) < f(S_2, j+1, t'' + t_j)$$

By substitution from equations (2), (3) we get:

$$t_j - t_{j+1} + \sum_{i=1}^{x} l_i - t_{j+1} \sum_{i=1}^{x} k_i < \sum_{i=1}^{a} l_i - t_{j+1} \sum_{i=1}^{a} l_i + t_j - t_{j+1}$$

and from (1)

$$\sum_{i=1}^{x} l_i - t_{j+1} \sum_{i=1}^{x} k_i < -(t_{j+1} - t_j) \sum_{i=1}^{a} k_i$$

By reformulation of this inequality we obtain

$$(t_{j+1} - t_j) \sum_{i=1}^{x} k_i + (t_{j+1} - t_j) \sum_{i=x+1}^{a} k_i < t_{j+1} \sum_{i=1}^{x} k_i - \sum_{i=1}^{x} l_i$$

$$(t_{j+1} - t_j) \sum_{i=x+1}^{a} k_i < t_j \sum_{i=1}^{x} k_i - \sum_{i=1}^{x} l_i$$

Since m-tasks are ordered by SPT rule, $t_{j+1} \geq t_j$ and the left-hand side of the above inequality is nonnegative. The right-hand side must be negative because $f(j, t') < f(j, t' + \sum_{i=1}^{x} l_i) = f(j, t') + \sum_{i=1}^{x} l_i - t_j \sum_{i=1}^{x} k_i$. Otherwise time moment $t' + \sum_{i=1}^{x} l_i < t''$ would be an alternative minimum, and would be selected for alternative insertion point of T_j. Thus, we get a contradiction. We conclude that the minimum of $f(S_1, j+1, t)$ may not be smaller than minimum of $f(S_2, j+1, t)$.

Analogously, suppose starting point t'' is better than t'. This would mean $\forall_x f(S_1, j+1, t' + t_j + \sum_{i=1}^{x} l_i) > f(S_2, j+1, t'' + t_j)$, but substituting a for x we get $f(S_1, j+1, t' + t_j + \sum_{i=1}^{a} l_i) = f(S_2, j+1, t'' + t_j)$ by equations (2), (3).

This reasoning can be applied recursively to all pairs of points in the initial schedule for tasks $T_1, \ldots, T_{j-1}$, where function $f(j, t)$ has minima, and to consecutive pairs of m-tasks. This concludes the proof. □

4 Conclusions

In this work we analyzed scheduling multiprocessor tasks for mean flow time and weighted mean flow time criteria. The results obtained are collected in the following table.

Problem	Complexity
Unit execution time	
$P\|size_j, p_j=1\|\sum c_j$	strongly **NP**-hard
$Pm\|size_j, p_j=1\|\sum w_j c_j$	$O(n^{3m+1})$
$P\|cube_j, p_j=1\|\sum w_j c_j$	$O(n(\log n + \log m))$
Preemptive scheduling	
$P\|size_j, pmtn\|\sum c_j$	**NP**-hard
$P\|fix_j, pmtn\|\sum c_j$ where $\|fix_j\| \in \{1, m\}$	$O(n^2)$

Further research may include analysis of preemptive scheduling multiprocessor tasks requiring processors according to $cube_j$ model, scheduling with mean weighted flow time criterion, or may include problems where release times and due-dates are present.

References

[1] Adolphson, D. and Hu, T.C., Optimal linear ordering, *SIAM Journal of Applied Mathematics*, 1973, **25**, 403-423.

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

[3] Błażewicz, J., Kubiak, W. and Szwarcfiter J., Scheduling independent fixed-type tasks, in: *Advances in Project Scheduling*, eds R.Słowiński, J.Węglarz, Elsevier Science Publ., Amsterdam, 1989, 225-236.

[4] Brucker, P. and Krämer, A., Polynomial algorithms for resource-constrained and multiprocessor task scheduling problems, *European Journal of Operational Research*, 1996, **90**, 214-226.

[5] Bruno, J., Coffman Jr., E.G. and Sethi, R., Scheduling independent tasks to reduce mean finishing time, *Communications of the ACM*, 1974, **17**, 382-387.

[6] Coffman Jr., E.G., Garey, M.R., Johnson, D.S. and Lapaugh, A.S., Scheduling file transfers, *SIAM Journal on Computing*, Aug. 1985, **3**, 744-780.

[7] Coffman Jr., E.G., Garey, M.R., and Johnson, D.S., Bin packing with divisible item sizes, *Journal of Complexity*, 1987, **3**, 406-428.

[8] Drozdowski, M., Scheduling multiprocessor tasks - An overview, *European Journal of Operational Research*, 1996, **94**, 215-230.

[9] Du, J., Leung, J.Y-T. and Young, G.H., Minimizing mean flow time with release time constraints, Technical Report, Computer Science Program, University of Texas at Dallas, 1988.

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

[11] Graham, R.I., Lawler, E.L., Lenstra, J.K., Rinnoy Kan, A.H.G., Optimization and approximation in deterministic sequencing and scheduling: a survey, *Annals of Discrete Mathematics* 1979, **5**, 287-326.

[12] Hoogeveen, J.A., van de Velde, S.L., Veltman, B., Complexity of scheduling multiprocessor tasks with prespecified processor allocations, *Discrete Applied Mathematics*, 1994, **55**, 259-272.

[13] Krawczyk, H. and Kubale, M., An approximation algorithm for diagnostic test scheduling in multicomputer systems, *IEEE Transactions on Computers*, 1985, **34**, No.9, 869-872.

[14] Labetoulle, J., Lawler, E.L., Lenstra, J.K. and Rinnoy Kan, A.H.G., Preemptive Scheduling of Uniform Machines Subject to Release Dates, in: *Progress in Combinatorial Optimization*, ed W. R. Pulleyblank, Academic Press, New York, 1984, 245-261.

[15] Simons, B., Multiprocessor scheduling of unit-time jobs with arbitrary release times and deadlines, *SIAM Journal on Computing*, 1983, **12**, 294-299.

[16] Lee, C.-Y., Cai, X., Scheduling multiprocessor tasks without prespecified allocations, 1996, private communication.

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

[18] Veltman, B., Lageweg, B.J. and Lenstra, J.K., Multiprocessor scheduling with communications delays, *Parallel Computing*, 1990, **16**, 173-182.

[19] Zahorjan, J., Lazowska, E.D., and Eager, D.L., The effect of scheduling discipline on spin overhead in shared memory parallel systems, *IEEE Transactions on Parallel and Distributed Systems*, 1991, **2**, No.2, 180-199.

AUTHORS' BIOGRAPHIES

Maciej Drozdowski is an Associate Professor in the Instytut Informatyki, Politechnika Poznańska, Poznań, Poland. His fields of research include scheduling, especially for parallel and distributed computer systems, combinatorial optimization, complexity analysis, and performance evaluation of computer systems. He received M.S. degree in Control Engineering, Ph.D. in Computer Science, in years 1987, 1992, respectively. In 1997 he defended a habilitation in Computer Science.

Paolo Dell'Olmo is an Associate Professor in the Dipartimento di Statistica, Probabilità e Statistiche Applicate, University of Rome "La Sapienza", Rome, Italy, and a fellow at Istituto di Analisi dei Sistemi ed Informatica, Rome, Italy. His research interests are mainly directed to combinatorial optimization and applications of the mathematical models to real life transportation systems. In particular, his studies include scheduling theory, complexity, coloring and ordering problems on graphs, and air traffic management.

Fig.1

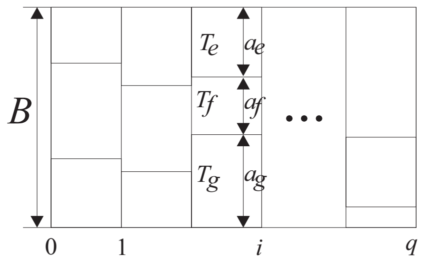

Figure 1: A feasible schedule for the proof of Theorem 1, when 3-PARTITION exists.

Fig.2a

Fig.2b

Fig.2c

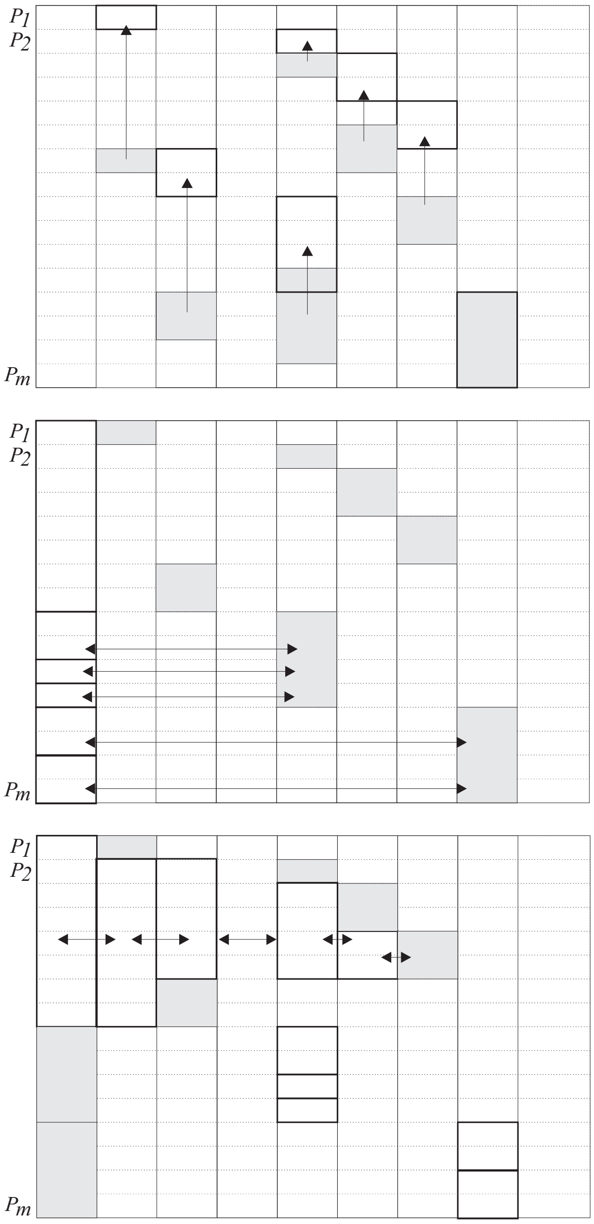

Figure 2: Transformation of the schedule for problem $P|cube_j, p_j = 1|\sum w_j c_j$. a) type 1, b) type 2, c) type 3 of a move.

Fig.3

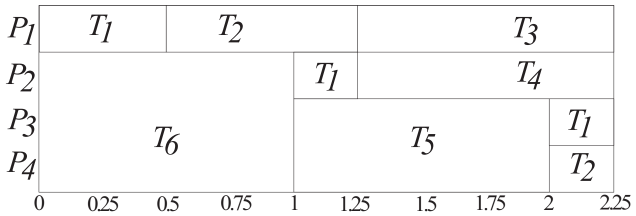

Figure 3: Feasible schedule for Example 2.

Fig.4a

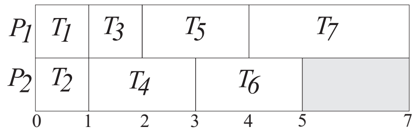

Fig.4b

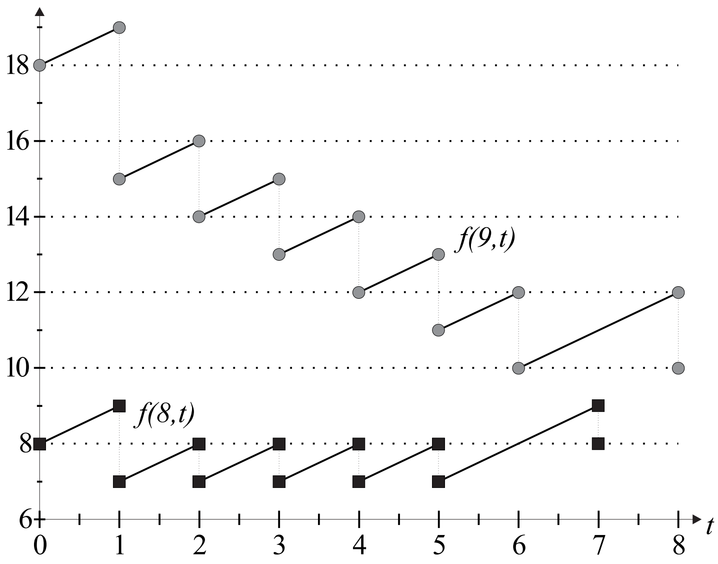

Fig.4c

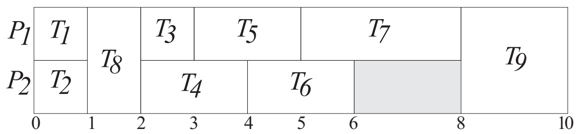

Figure 4: Example 4: a) SPT schedule for 1-tasks, b) $f(8,t)$ and $f(9,t)$ for T_8 started at 1, c) optimal schedule for all tasks.