Minimizing Weighted Mean Completion Time for Malleable Tasks Scheduling

Olivier Beaumont, Nicolas Bonichon, Lionel Eyraud-Dubois, Loris Marchal

A Normal Form for Scheduling Work Preserving Malleable Tasks and its Applications

Olivier Beaumont and Loris Marchal

[1] INRIA and University of Bordeaux
olivier.beaumont@labri.fr
[2] CNRS and ENS Lyon
loris.marchal@ens-lyon.fr

Abstract. Malleable tasks are jobs that can be scheduled with preemptions on a varying number of resources. We focus on the special case of work-preserving malleable tasks, for which the area of the allocated resources remains constant and equal to the sequential processing time. The quantity of resources allocated to each task is bounded, and preemptions are allowed. We provide a normal form for the schedule of such malleable tasks, and prove that any valid schedule can be turned into this normal form, based on the completion times of the tasks. We present two applications of this normalization. First, we show that on average, the number of preemptions per task is bounded by 3. Second, we show that the use of the normal form dramatically reduces the search space for a number of NP-complete problems on malleable tasks, for objectives like the maximum tardiness or the weighted sum of completion times.

1 Introduction

Parallel Tasks. Parallel tasks model has been introduced in order to deal with the complexity of explicitly scheduling communications. In the most general setting, a parallel task comes with its completion time on any number of resources (see [9,7,8] and references therein for complete surveys) . The tasks may be either rigid (when the number of processors for executing a dedicated code is fixed), moldable (when the number of processors is fixed for the whole execution, but may take several values) or malleable [4](when the number of processors may change even during the execution due to preemptions). In this paper, we concentrate on the special case of malleable tasks that are work-preserving, *i.e.* parallel tasks such that the overall work (that corresponds to the area allocated to a task in a Gantt Chart) does not depend on the number of processors allocated to a task. This corresponds to ideal parallel tasks. Although malleability requires advanced capacities of the runtime environment, it may be well suited to multicore machines and it may be used in other context, such as sharing a large bandwidth out of a server between small connexions. It is worth noting that in the context of multicore machines as in the context of bandwidth sharing, it is natural to assume that a given task cannot make use of all available resources (because to be work-preserving, a task should be allocated on a single multicore processor, or because the bandwidth achievable between a large capacity server and a distant node is typically bounded by the bandwidth of the distant node).

Therefore, it is natural to associate to a task the maximal number of resources that it can use simultaneously, that will be denoted by δ in the rest of the paper. Although this paper is written in the perspective of parallel task scheduling, we would like to underline that the use of quality of service mechanisms [5,1,12] for TCP bandwidth sharing makes that presented result can be adapted to the simultaneous transfer of files in large scale distributed networks [2].

We focus on the following scheduling problem: (i) the system is made of P identical processors on which (ii) n malleable work-preserving tasks $T_1, \ldots, T_n$ are to be scheduled. Task T_i may be scheduled on any number q of processors such that $q \leq \delta_i$, its running time on q processors is given by $p_i(q) = p_i/q$ and preemption is allowed without any cost.

Contributions. The main contribution of this paper is to propose a normal form for malleable task scheduling. More precisely, we prove in Section 2 that we can transform any valid schedule into a normal form schedule, that preserves the completion time of all tasks. Therefore, it can be used to reduce the search space [11] when solving any optimization problem involving malleable work-preserving tasks, provided that the objective function only involves task completion times. This normal form has several advantages : first, we prove in Section 3 that the overall number of preemptions for n tasks in any solution under the normal form is bounded by $3n$. Then, we prove in Section 4 that it is possible to find the optimal solution for $P|pmtn; var; p_i(q) = p_i/q, \delta_i| \sum w_i C_i$ using integer linear programs having of order $O(n)$ variables.

Related Works. Let n and P denote respectively the number of tasks and the number of processors. Most of scheduling problems for independent malleable work-preserving tasks are polynomially solvable when the goal is to maximize the makespan. For instance, $P|var; p_i(q) = p_i/q, \delta_i, r_i|C_{\max}$ (even in presence of release dates) can be solved in time $O(n^2)$ [6]. The maximum lateness problem $P|var; p_i(q) = p_i/q, \delta_i, r_i|C_{\max}$ is also solvable in time $O(n^4 P)$ [7]. It is worth noting that the algorithm we propose in Section 2 enables to solve this problem in $O(n \log n)$ time if all release dates are equal to zero. On the other hand, problems related to the weighted sum of completion times are NP-Complete [11]. Using the Graham notation extended for parallel tasks by Drowdzoski [7], this corresponds to the problem $P|pmtn; var; p_i(q) = p_i/q, \delta_i| \sum w_i C_i$, that is NP-complete as a generalization of $P|pmtn| \sum w_i C_i$ [10].

2 Normalization of the schedule

The main contribution of this paper is to propose a new normal form for malleable tasks such that (i) the area p_i allocated to a task T_i does not depend on the number of processors ($p_i(q) = p_i/q$), (ii) preemptions are allowed and (iii) the quantity of resources allocated to T_i at any time cannot be larger than δ_i.

In order to present our normalization process, we start with a valid schedule $\mathcal{O}$ for such malleable tasks. For the sake of simplicity, we assume that in $\mathcal{O}$, tasks are sorted by non-decreasing order of completion time C_i: $C_1 \leq C_2 \leq \cdots \leq C_n$. In order to produce the normalized schedule, we only rely on the C_i: in the modified schedule, the completion time of each task will be preserved. The

normalized schedule is built, based on the C_i by Algorithm 1, called WATER-FILLING algorithm. In the following, we present the intuition of the algorithm, prove its correctness, and then describe the some properties of the normalized schedule.

In the following, we consider that the schedule is organized by *columns*. Each column correspond to a time-slice of the schedule between two task completion: we call column k the time interval $[C_{k-1}, C_k]$ between the completion of tasks T_{k-1} and T_k (or between time 0 and C_0 when $k = 1$). We denote by $l_k = C_k - C_{k-1}$ the length, or duration, of column k. In each column, we consider that the number of processors allocated to one task is almost constant: it may vary at most one time from some value q to $q - 1$.

The algorithm proceeds by allocating some resources to task T_1, then to task T_2, etc. until all tasks have been scheduled. After scheduling task $T_1, \ldots, T_i$, A_k^i records the *area* that have been already been allocated in column k. Using this notation, $\lceil A_k^i/l_k \rceil$ is the height of column k, *i.e.* the number of the processors allocated to tasks $T_1, \ldots, T_i$, at the beginning of the column, and $\lfloor A_k^i/l_k \rfloor$ is the height at the end of column k. In the algorithm, when scheduling a new task T_i, V_i represents the amount of the task that remains to be scheduled, expressed in serial time (in the beginning, $V_i = p_i$).

The rationale behind the WATER-FILLING algorithm is the following: what may cause task T_{i+1} to be impossible to allocate once tasks $T_1, \ldots, T_i$ have been allocated is that there is not enough time steps when the quantity of available processors is large enough. Thus, the goal of the WATER-FILLING algorithm is to maximize the overall quantity of available resources, as formally stated in Lemma 1 after each allocation. To achieve this, the WATER-FILLING algorithm tends to balance, as much as possible, the height of columns 1 to $i + 1$. The typical situation before and after the allocation of a task T_{i+1} is depicted on Figure 2. We recall that tasks are considered by increasing completion time C_i. When T_i is considered, the algorithm first search for a maximal set of contiguous columns that may accommodate T_i ($\text{AVAILABLE}(k, T_i) \geq V_i$) if all had the same height (up to a difference of 1 due to integer constraint, see ALLOCATEAREA), *i.e.* the same quantity of used resources. If true, then $\text{ALLOCATEAREA}(k, T_i, V_i)$ returns such an allocation and if not, the WATER-FILLING algorithm assigns as much resources as possible in column i (done by $\text{ALLOCATECOL}(k, T_i, V_i)$) and then restarts one column on the left. It is worth noting that when $\text{ALLOCATEAREA}(k, T_i, V_i)$ is called, contiguous columns between $left_i$ and $right_i$ are *de facto* merged, since tasks allocated later (those with $j > i$) will "see" a single column of constant height (again up to a difference of 1 due to integer constraints). This property will be very helpful when proving the correctness of the WATER-FILLING algorithm (Theorem 1) and when building an integer linear program that computes the optimal solution in Section 4.

In what follows we prove that above algorithm returns a valid allocation if the completion times correspond to any valid solution. The proof relies on the following lemma, stating that after allocating a series of tasks, the area that remains available for the next task is maximized, whatever its limit δ on simultaneous resource usage. Note that in the following, we do not use the completion time C_i anymore, but we use the length of the columns l_i instead, which identically describes the solution.

Algorithm 1: WATER-FILLING algorithm for malleable tasks.

```
for i = 1 ... n do
    k ← i
    V_i ← p_i
    while AVAILABLE(k, T_i) < V_i and k > 0 do
        ALLOCATECOL(k, T_i, V_i)
        k ← k − 1
    if k = 0 then
        return no valid solution
    ALLOCATEAREA(k, T_i, V_i)
return current allocation
```

Algorithm 2: AVAILABLE(k, T_i).

```
amount ← 0
MaxHeight ← min(P, ⌊A_k^{i−1} / l_k⌋ + δ_i)
for j = 1 ... k do
    amount ← amount + max(MaxHeight × l_j − A_j^{i−1}, 0)
return amount
```

Algorithm 3: ALLOCATECOL(k, T_i, V_i).

```
alloc ← min(P × l_k − A_k^{i−1}, δ_i × l_k, V_i)
V_i ← V_i − alloc
A_k^i ← A_k^{i−1} + alloc
```

Lemma 1. *Let $(l_1, l_2, \ldots, l_n)$ denote a sequence of time interval length, $T_i = (\delta_i, p_i)_{1 \leq i \leq n}$ denote a sequence of tasks and P denote the bound on available resources. We consider a valid schedule $\mathcal{O}$ for the sequence of time intervals (such as task T_i completes at the end of the i^{th} interval), and we denote by O_j^i the area in column j occupied by tasks $T_1, \ldots, T_i$. Then, let us denote by H_j^i the occupied area in column j by tasks $T_1, \ldots, T_i$, if allocated using the* WATER-FILLING *algorithm (Algorithm 1). We have*

$$\forall \delta, \quad \sum_{k=1}^{m+1} \min(\delta, P - H_k^m) l_k \geq \sum_{k=1}^{m+1} \min(\delta, P - O_k^m) l_k.$$

In particular, above lemma enables to prove the following theorem

Theorem 1. *The* WATER-FILLING *algorithm (Algorithm 1) finds a valid allocation if one exists.*

We first prove this theorem using the previous lemma, and then move to the proof of the lemma itself.

Proof (Theorem 1). Let us assume by contradiction that Algorithm 1 fails to compute a valid solution and let us denote by T_{m+1} the first task that cannot be allocated using Algorithm 1.

Algorithm 4: AllocateArea($right_i, T_i, V_i$).

$left \leftarrow right + 1$
$amount \leftarrow 0$
$\Rightarrow$ *add columns on the left until we reach (or exceed) the required amount of resource:*
while $amount < V_i$ **do**
  $left \leftarrow left - 1$
  $maxHeight \leftarrow \lfloor A^{i-1}_{left-1} / l_{left} \rfloor$
  for $k = left, \ldots, right$ **do**
    $A^i_k \leftarrow maxHeight \times l_k$;
  $amount \leftarrow \sum_{k=left}^{right} \left(A^i_k - A^{i-1}_k\right)$
$\Rightarrow$ *if the allocated resources exceeds the demand, decrease the allocation:*
$finalMinHeight \leftarrow \lfloor maxHeight - (amount - V_i) / \sum_{k=left}^{right} l_k \rfloor$
for $k = left, \ldots, right$ **do**
  $A^i_k \leftarrow finalMinHeight \times l_k$
$\Rightarrow$ *allocate the remaining resources, from left to right:*
$remainder \leftarrow V_i - \sum_{k=left}^{right} \left(A^i_k - A^{i-1}_k\right)$
$k \leftarrow left$
while $remainder > 0$ **do**
  $A^i_k \leftarrow A^i_k + \max(remainder, l_k)$
  $remainder \leftarrow remainder - l_k$
  $k \leftarrow k + 1$
$\Rightarrow$ *Define the left and right bounds of the columns involved in* AllocateArea
$left_i \leftarrow left$
$right_i \leftarrow right$

Let us consider the execution of Algorithm 1 at iteration $i = m+1, j = m+1$. Clearly, Available $(l+1, T_{m+1}) < V_{m+1}$, otherwise, we would have been able to pack task T_{m+1}. Thus, AllocateCol $(m+1, T_{m+1}, p_{m+1})$ allocates $\min(\delta_{m+1}, P - H^m_{m+1})$ to T_{m+1} in column $m+1$. Similarly, for next iterations of the j loop, Available $(j, T_{m+1}) < V_{m+1}$, otherwise, we would have been able to pack task T_{m+1}, so that AllocateCol (j, T_{m+1}, V_{m+1}) allocates $\min(\delta_{m+1}, P - H^m_j)$ to T_{m+1} in column j.

Therefore, since Algorithm 1 fails to find a solution, it means that $V_{m+1} > 0$ at the end of the j loop, so that

$$\sum_{j=1}^{m+1} \min(\delta_{m+1}, P - H^m_j) l_j < p_{m+1}.$$

On the other hand, let us consider a valid allocation $\mathcal{O}$ of tasks $T_1, \ldots, T_{m+1}$ and let us denote by O^m_j the occupied capacity in column j by tasks $T_1, \ldots, T_m$ in allocation $\mathcal{O}$.

Using 1, we know that $\forall \delta$, then

$$\sum_{k=1}^{m+1} \min(\delta, P - H^m_k) l_k \geq \sum_{k=1}^{m+1} \min(\delta, P - O^m_k) l_k$$

so that, in particular, for $\delta = \delta_{m+1}$,

$$\sum_{k=1}^{m+1} \min(\delta_{m+1}, P - H_k^m) l_k \geq \sum_{k=1}^{m+1} \min(\delta_{m+1}, P - O_k^m) l_k.$$

Moreover, $\forall j$, the resources allocated to T_{m+1} in $\mathcal{O}$ cannot exceed $\min(\delta_{m+1}, P - O_k^m)$ by construction. Since $\mathcal{O}$ is a valid solution for Tasks $T_1, \ldots, T_m, T_{m+1}$, then $\sum_{k=1}^{m+1} \min(\delta_{m+1}, P - O_k^m) l_k \geq p_{m+1}$ and therefore

$$\sum_{k=1}^{m+1} \min(\delta_{m+1}, P - H_k^m) l_k \geq \sum_{k=1}^{m+1} \min(\delta_{m+1}, P - O_k^m) l_k \geq p_{m+1},$$

which contradicts the assumption and achieves the proof of the theorem. □

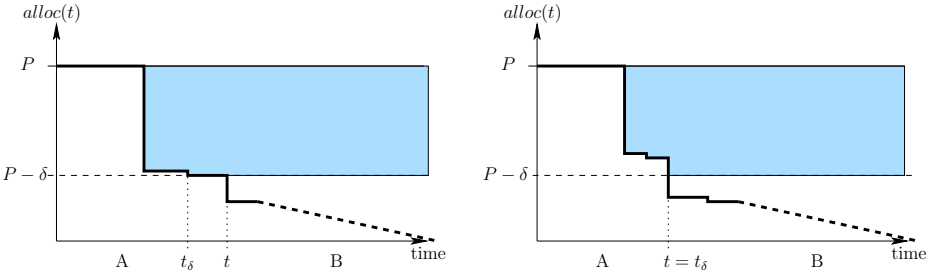
Let us now turn back to the proof of the Lemma.

Proof (Lemma 1). For a given m, let us consider the function $alloc(t)$, which gives the number of processors allocated at time t after scheduling the first m tasks with the WATER-FILLING algorithm. This function is depicted on Figure 2.

For a given time t, we denote by k the column where t lies (*i.e.* $\sum_{j=1}^{k-1} l_j \leq t < \sum_{j=1}^{k} l_j$), and we write $H_k^m = q \times l_k + r$ with $r < k_k$. Then,

$$alloc(t) = \begin{cases} q = \lfloor H_j^i / l_i \rfloor & \text{if } t - l_i > r, \\ q + 1 & \text{otherwise.} \end{cases}$$

Let us now concentrate on the first instant t_δ when $alloc(t)$ intersects with the horizontal line of height $P - \delta$. There are two possibilities, as depicted on Figure 1.



(a) First case. (b) Second case.

Fig. 1. Two cases for t_δ in the proof of Lemma 1.

If t_δ corresponds to an instant when the number of allocated resources is decreased by 1 in the middle of a column due to integer constraints, then we define t as the first instant $t \geq t_\delta$ such that $alloc(t) < P - \delta$ (see Figure 1(a)). Otherwise, if t_δ corresponds to a standard end of column (the completion of some task), then $t = t_\delta$ (see Figure 1(b)).

Note that the quantity $\sum_{k=1}^{m+1} \min(\delta, P - H_k^i) l_k$ is equal to the area above both curves (shaded on the figures). We split this sum into two parts, considering first the columns before time t, denoted by $\mathcal{C}_A$, and second the columns after time t, denoted by $\mathcal{C}_B$

$$\sum_{k=1}^{m+1} \min(\delta, P - H_k^i) l_k = \sum_{k \in \mathcal{C}_A} \min(\delta, P - H_k^i) l_k + \sum_{k \in \mathcal{C}_B} \min(\delta, P - H_k^i) l_k.$$

We consider both contributions, starting with the second one:

- In the second sum, $k \in \mathcal{C}_B$, the area is bounded by δ (since $\min(\delta, B - H_k^i) = \delta$), which is a natural upper bound of this quantity. Thus, no other allocation can have a strictly larger contribution after t.
- In the first sum, $k \in \mathcal{C}_A$, the area is bounded by $alloc(t)$: $\min(\delta, P - H_k^i) = P - H_k^i$. Thus we have

$$\sum_{k \in \mathcal{C}_A} \min(\delta, P - H_k^i) l_k = P \times \left(\sum_{k \in \mathcal{C}_A} l_k \right) - \left(\sum_{k \in \mathcal{C}_A} H_k^i \right)$$

  We focus on the resources already allocated in columns of set $\mathcal{C}_A$ ($\sum_{k \in \mathcal{C}_A} H_k^i$), and see how their amount could be reduced. There are two cases for tasks allocated in $\mathcal{C}_A$:
  - They may be tasks whose completion time is smaller than or equal to t. In this case, they must be allocated completely before t (in $\mathcal{C}_A$).
  - Otherwise, they may be some task T_i whose deadlines is after t. In this case, the WATER-FILLING algorithm has allocated these tasks to their maximum capacity δ_i in the columns of $\mathcal{C}_B$.

  Indeed, there is no third choice, since t corresponds to the beginning of a new column, and therefore, there is no task T_j which has been allocated using ALLOCATEAREA on an interval comprising t,*i.e.* such that $t \in [left_j, right_j]$. Indeed, in this case, the columns between $left_j$ and $right_j$ would have been merged together.
  In both cases (task completing before or after t), all the resources already allocated in the columns of set $\mathcal{C}_A$ cannot be allocated later than t by any other allocation. Thus, the available resources for a task with limitation δ cannot be improved before t either.

This achieves the proof of Lemma 1. □

3 Number of preemptions induced by Water-Filling Algorithm

In this section, we prove that the overall number of preemptions for n malleable tasks scheduled using WATER-FILLING algorithm is bounded by $3n$. Therefore, since any valid schedule can be turned into a WATER-FILLING schedule, the search for optimal malleable schedule can always be restricted to schedules inducing at most 3 preemptions on average, whatever the objective function is.

To obtain this result, we first prove in Section 3.1 that the number of changes in the number of resources allocated to a task is bounded by 3 on average. Then, we prove in Section 3.2 that, given this property, it is indeed possible to assign processors to tasks so that at most 3 preemptions per task take place on average.

3.1 A bound on the number of variations in the number of allocated resources

The following theorem establishes a bound on the number of resource changes in a schedule.

Theorem 2. *The overall number of changes in the quantity of resources allocated to a task using* Water-Filling *algorithm is bounded by* $3n$*, where* n *denotes the number of tasks.*

Proof. For a given task, we do not consider that there is a change in the number of allocated resources when the task is scheduled for the first time and for the last time, so that the number of changes is closely related to the number of preemptions, as we will prove it in Section 3.2.

In what follows, after the allocation of tasks $T_1, \ldots, T_i$ using Water-Filling Algorithm, we denote by

- N_i the overall number of changes over time in the quantity of allocated resources for tasks $T_1, \ldots, T_i$,
- M_i the number of changes over time in the quantity of available resources after the allocation of tasks $T_1, \ldots, T_i$.

The following lemma expresses the relation between these two notations.

Lemma 2.

$$\forall i \geq 1, \quad N_{i+1} + M_{i+1} \leq N_i + M_i + 3.$$

Proof. Let us consider the allocation of task T_{i+1} using Water-Filling Algorithm. Typically, the allocation consists in two phases, as depicted on Figure 2. During the first phase, that corresponds to columns $right_{i+1} + 1$ to $i + 1$, the quantity of resources allocated to T_{i+1} is exactly δ_i, so that no change occurs.

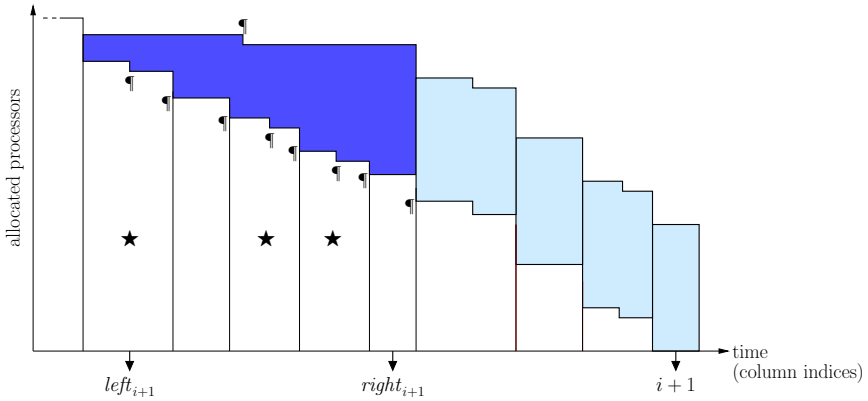


Fig. 2. Allocation of task T_{i+1} and number of changes induces.

During the second phase, that corresponds to columns $left_{i+1}$ to $right_{i+1}$, let us denote by k' the number of columns between $left_{i+1}$ and $right_{i+1}$ that

contains of small step (a change of 1 in the number of available resources over time) and by k the number of columns between $left_{i+1}$ and $right_{i+1}$ without such a small step. On Figure 2, the $k' = 3$ columns with a small step are marked with a ★ while the $k = 2$ columns without a step are left empty. Then, each small step, as well as each column, induces a new change in the number of processors for T_{i+1}, and there is potentially a new small step at the top of the area (see Figure 2, where each change for T_{i+1} is indicated with a ¶ mark). This gives the following relation on the N_i and M_i:

$$N_{i+1} = N_i + 2k' + k + 1 \text{ and } M_{i+1} = M_i - 2k' - k + 2,$$

so that $N_{i+1} + M_{i+1} \leq N_i + M_i + 3$, which achieves the proof of the lemma.

When no task is schedule, we have $N_0 = M_0 = 0$, so that the previous lemma gives $N_0 + M_n \leq 3n$. The observation that $M_n \geq 1$ concludes the proof of the theorem. □

3.2 Bound on the number of preemptions

The following theorem states that using Water-Filling Algorithm, it is possible to allocate processors to tasks such that the overall number of preemptions is exactly the number of changes in the number of processors allocated to tasks. When combining it with Theorem 2, we obtain the final result on the overall number of preemptions. The straightforward proofs are provided in the companion research report [3].

Theorem 3. *Let N be the total number of changes in the amount of processors allocated to tasks, in the schedule produced by* Water-Filling *algorithm. Then, there is an allocation of tasks to processor with no more than N preemptions.*

Corollary 1. *The overall number of preemptions induced by* Water-Filling *algorithm is bounded by $3n$, where n denoted the number of tasks.*

4 A linear program to compute the completion times, once given the shape of the schedule

In this section, we present a linear programming approach to compute the optimal completion times, for a variety of objectives which can be expressed linearly with completion times, such as weighted sum of completion times, maximum lateness, or makespan. We construct a linear program that is able to optimize a given function of the completion times, but which relies on a fixed shape of the schedule. Some information should be provided: (i) the ordering of the completion times and (i) the columns $left_i$ and $right_i$ in the normalized schedule, as computed by AllocateArea, which define the interval when task T_i is started but not using its maximal number of processors δ_i.

In order to explain the linear program formulation, we need to study the structure of the schedule produced by the Water-Filling algorithm. We recall that the schedule of a task T_i is composed of two parts: a first part, in columns $left_i, \ldots, right_i$, where the number of processors allocated to T_i is increased (but

stay smaller than δ_i), and a second part, where the number of processors allocated to T_i is constant and equal to δ_i. We focus on the first part, which resembles a triangle when depicted in Figure 2: its bottom line follows the decreasing number of allocated processors before scheduling T_i, while its top line is flat (up to a small step of one). Once such a triangle has been allocated, all the covered columns behave similarly for the rest of the allocation process: they are *de facto* merged. Thus, they present an tree structure: a triangle may covers several triangle, which in turn cover other triangles. Some columns may not be covered by any triangle, however, to simplify the notations, we will consider that there is a virtual triangle at the bottom of each column.

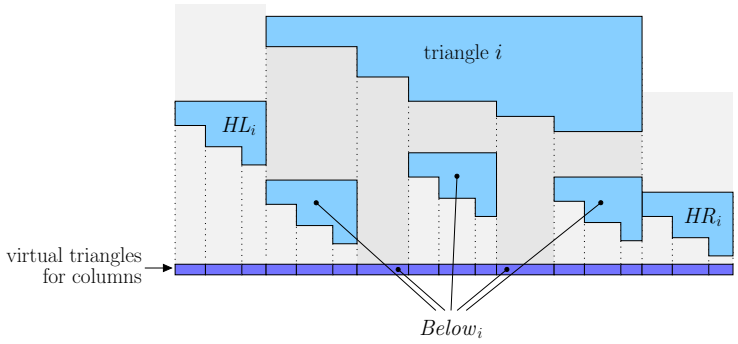


Fig. 3. Hierarchical structures of the schedule.

We now present the notations that will be used to write the linear program, which are illustrated on Figure 3, and how they can be computed. We recall that $left_i$ is the first column where task T_i is allocated to some processor, while $right_i$: the last column where task T_i is allocated strictly less than its maximum number of processor δ_i. In the following, we call "triangle i" the area allocated by AllocateArea for task T_i.

- $Below_i$ is the list of the triangles which are immediately below triangle i:

$$j \in Below_i \Leftrightarrow \begin{cases} j < i,\ left_i \leq left_j \leq right_j \leq right_i, \text{ and} \\ \forall j < k < i,\ \ (left_k > left_j \text{ or } right_k < right_j) \end{cases}$$

  Moreover, we assume that $Below_i$ is sorted by increasing $left_i$.
- HL_i is the highest triangle that spans over column $left_{i-1}$ when tasks $T_1, \ldots, T_{i-1}$ are allocated: $HL_i = \max\{j < i,\ \ right_j = left_i - 1\}$.
- HR_i is the highest triangle that spans over column $right_i + 1$ when tasks $T_1, \ldots, T_{i-1}$ are allocated: $HR_i = \max\{j < i,\ \ left_j = right_i + 1\}$: the last triangle

Some of the above triangles might be virtual; virtual triangles are assumed to have indices smaller than any tasks.

The variables used in this linear program are:

- C_i, the completion time of task T_i;
- H_i, the average height of columns $left_i$ to $right_i$, weighted by their duration, after scheduling task T_i.

We are able to now list the constraints of the linear program.

- We first ensure that for a column which is not present in any triangle, its height does not exceed P (note that this is not really constraint since it does not depends on the variables):

$$\forall k \text{ such that } \forall i,\ k < \mathit{left}_i \text{ or } k > \mathit{right}_i, \quad \sum_{k \leq j \text{ and } \mathit{right}_j < k} \delta_j \leq P \tag{1}$$

- We check that the triangles correctly describe the solution. We first check that the number of processors allocated to a task in the first column of its triangle is positive, and the number of processors allocated to its last column is less than δ_i. The *head* and *tail* are used to get the first and last element of a list.

$$H_i - \left(H_{\mathit{head}(\mathit{Below}_i)} + \sum_{\mathit{head}(\mathit{Below}_i) < j < i \text{ and } \mathit{right}_j < \mathit{left}_i} \delta_j \right) > 0 \tag{2}$$

$$H_i - \left(H_{\mathit{tail}(\mathit{Below}_i)} + \sum_{\mathit{tail}(\mathit{Below}_i) < j < i \text{ and } \mathit{right}_j < \mathit{right}_i} \delta_j \right) \leq \delta_i \tag{3}$$

- The height of the triangle i is smaller than (or equal to) the height of the previous column before the insertion of T_i:

$$H_i \leq H_{HL_i} + \sum_{HL_i < j < i \text{ and } \mathit{right}_j < \mathit{left}_{i-1} \text{ and } j \geq \mathit{left}_{i-1}} \delta_j \tag{4}$$

- The height of the triangle i is larger than the height of the following column after inserting T_i:

$$H_i > H_{HR_i} + \sum_{HR_i < j < i \text{ and } \mathit{right}_j < \mathit{right}_{i+1} \text{ and } j \geq \mathit{right}_{i+1}} \delta_j \tag{5}$$

- For all maximal triangles, after inserting the following tasks, the height of the column should not exceed P:

$$\forall i \text{ such that } \forall j > i,\ \mathit{left}_j > \mathit{left}_i \text{ or } \mathit{right}_j < \mathit{right}_i, \quad H_i + \sum_{\mathit{right}_j < \mathit{left}_i \text{ and } j > i} \delta_j \leq P \tag{6}$$

- The total amount of resources dedicated to task T_i is equal to p_i:

$$\begin{aligned} &\forall i, \quad H_i \times (C_{\mathit{right}_i} - C_{\mathit{left}_i - 1}) \\ &- \sum_{k \in \mathit{Below}_i} \left(H_k + \sum_{k < j < i \text{ and } \mathit{right}_j < \mathit{left}_k \text{ and } j \geq \mathit{right}_k} \delta_j \right) \times (C_{\mathit{right}_k} - C_{\mathit{left}_k - 1}) \\ &\qquad + \delta_i \times (C_i - C_{\mathit{right}_i}) = p_i \end{aligned} \tag{7}$$

The objective of the linear program has to be provided, and depend on optimization problem. All objective functions which are linear with the C_i may be use, such as the weighted flow time ($\sum_i w_i C_i$), the maximum lateness ($\max_i C_i - d_i$), or the makespan ($\max C_i$).

In the companion research report [3], we prove that from a solution of this linear program, if we input the completion times of tasks to the WATER-FILLING algorithm, it produces the same solution. Due to lack of space, we do not include here the straightforward proof.

5 Conclusion

We have introduced a new normal form for the schedules of malleable work preserving tasks and presented several applications to bound the average number of preemptions and to dramatically reduce the search space for a number of NP-complete problems, for objectives like the maximum tardiness or the weighted sum of completion times. An important open issue is the extension of this normal schedule to both deadlines (done in this paper) and release times.

References

1. Dirk Abendroth, Hans van den Berg, and Michel Mandjes. A versatile model for tcp bandwidth sharing in networks with heterogeneous users. *AEU - International Journal of Electronics and Communications*, 60(4):267 – 278, 2006.
2. O. Beaumont and H. Rejeb. On the importance of bandwidth control mechanisms for scheduling on large scale heterogeneous platforms. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–12. IEEE, 2010.
3. Olivier Beaumont and Loris Marchal. A normal form for scheduling malleable tasks and its applications. Research report, INRIA, 2011.
4. J. Blazewicz, M.Y. Kovalyov, M. Machowiak, D. Trystram, and J. Weglarz. Preemptable malleable task scheduling problem. *IEEE Transactions on Computers*, pages 486–490, 2006.
5. Allen B. Downey. Tcp self-clocking and bandwidth sharing. *Computer Networks*, 51(13):3844 – 3863, 2007.
6. M. Drozdowski. New applications of the Muntz and Coffman algorithm. *Journal of Scheduling*, 4(4):209–223, 2001.
7. M. Drozdowski. Scheduling Parallel Tasks–Algorithms and Complexity, chapter 25. Handbook of SCHEDULING Algorithms, Models and Performance Analysis, 2004.
8. P.F. Dutot, G. Mounié, and D. Trystram. Handbook of Scheduling, chapter Scheduling Parallel Tasks-Approximation Algorithms, 2004.
9. DG Feitelson. Scheduling parallel jobs on clusters. *High Performance Cluster Computing*, 1:519–533.
10. M. R. Garey and D. S. Johnson. *Computers and Intractability, a Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1979.
11. R. Sadykov. On scheduling malleable jobs to minimise the total weighted completion time. In *Proceedings of the 13th IFAC Symposium on Information Control Problems in Manufacturing, Moscow, Russia*, pages 1497–1499. IFAC-PapersOnLine, 2009.
12. Yujie Zhu, Aravind Velayutham, Oyebamiji Oladeji, and Raghupathy Sivakumar. Enhancing tcp for networks with guaranteed bandwidth services. *Computer Networks*, 51(10):2788 – 2804, 2007.