A $\frac{3}{2}$-APPROXIMATION ALGORITHM FOR SCHEDULING INDEPENDENT MONOTONIC MALLEABLE TASKS∗

GREGORY MOUNIE†, CHRISTOPHE RAPINE‡, AND DENIS TRYSTRAM†

Abstract. A malleable task is a computational unit that may be executed on any arbitrary number of processors, whose execution time depends on the amount of resources allotted to it. This paper presents a new approach for scheduling a set of independent malleable tasks which leads to a worst case guarantee of $\frac{3}{2} + \varepsilon$ for the minimization of the parallel execution time for any fixed $\varepsilon > 0$. The main idea of this approach is to focus on the determination of a good allotment and then to solve the resulting problem with a fixed number of processors by a simple scheduling algorithm. The first phase is based on a dual approximation technique where the allotment problem is expressed as a knapsack problem for partitioning the set of tasks into two shelves of respective heights 1 and $\frac{1}{2}$.

Key words. scheduling, malleable tasks, polynomial approximation, performance guarantee

AMS subject classifications. 90B35, 68W25, 68W40, 68Q85

DOI. 10.1137/S0097539701385995

1. Introduction. The implementation of applications on parallel and distributed systems requires sophisticated algorithms for scheduling the tasks of the parallel programs. There exists a very large literature addressing this problem. It corresponds to determining a date for each task to start its execution together with a processor location. Usually, the tasks correspond to indivisible pieces of the application that are executed sequentially on a processor. The standard communication model for scheduling the tasks of a parallel program is the delay model introduced by Rayward-Smith [21] for UET-UCT (unit execution times and unit communication times) task graphs and extended by Papadimitriou and Yannakakis [18]. In this model, the communications between tasks executed on different processors are considered explicitly by the time for transferring a message between them. The communication times between tasks within the same processor are neglected. The scheduling UET-UCT problem is known to be $NP$-hard in the strong sense [21], and it is not approximable within a factor of $5/4$ of the optimum by any polynomial algorithm [10], unless $P = NP$. The best known approximation result is due to Hanen and Munier [8], whose algorithm is within a factor of $7/3$ of the optimum for small communication delays. Among the various possible approaches, the most commonly used is to consider the tasks of the program at the finest level of granularity and apply some adequate clustering heuristics to reduce the relative communication overhead [22, 6, 17]. The main drawback of such an approach is that the communications are taken into account explicitly: they are expressed assuming a model of the underlying architecture of the system. A good alternative is to consider the malleable tasks model (denoted MT), where the communication times are considered implicitly by a function representing the parallel execution time with the penalty due to the management of the parallelism (communication, synchronization, etc.). A malleable task is a computational unit which may

∗Received by the editors March 5, 2001; accepted for publication (in revised form) December 8, 2006; published electronically May 16, 2007.
†ID-IMAG, 51 avenue Jean Kuntzman, 38 330 Montbonnot Saint Martin, France (Denis.Trystram@imag.fr, Gregory.Mounie@imag.fr).
‡GILCO-INPG, 46 avenue Felix Viallet, 38 031 Grenoble cedex, France (Christophe.Rapine@gilco.inpg.fr).
be executed on several processors with a running time that depends on the number of processors allotted to it.

In this paper, we are interested in scheduling a set of \( n \) independent malleable tasks on a multiprocessor system composed of \( m \) identical processors. An instance of the problem is a set \( T = \{T_1, \ldots, T_n\} \) of tasks, together with a set of \( n \) functions \( t_i : p \rightarrow t_{i,p} \) which represents the processing time of task \( T_i \) when executed on \( p \) processors. A solution (scheduling) consists in finding for each task \( T_i \) a starting time \( st_i \) and a subset \( P_i \) of the processors to execute it, with the following constraints.

- Task \( T_i \) starts its execution simultaneously on all the processors of \( P_i \) and occupies them without interruption until its completion time \( C_i = st_i + t_{i,|P_i|} \).
- A processor executes at most one task at a time.

A task will be represented as a rectangle in the Gantt chart. This study is restricted to algorithms that provide consecutive processors. It is clear that the optimal allotment could use processors that are not consecutive. However, since we make use of a lower bound of the true optimum in our proof of the approximation bound, we see that this restriction does not have a substantial impact on what results can be achieved.

The objective is to minimize the makespan defined as the maximum completion time over all the tasks. Our main contribution is to propose a new method for scheduling independent monotonic malleable tasks. The analysis leads to a performance guarantee of \( \frac{3}{2} + \varepsilon \) for any constant \( \varepsilon > 0 \), in time \( O(nm \log(n/\varepsilon)) \). This bound improves all existing practical results for solving this problem. Such a method should be used as a basis for other scheduling problems like MT with precedence constraints or multiobjective scheduling analysis.

The organization of this paper is the following: we first briefly survey related work and recall the model of MT and its main properties. Then, we discuss the principle of our approach and present the algorithm and analyze its performance guarantee.

2. Preliminaries on malleable tasks.

2.1. Related work. The problem of scheduling independent malleable tasks has been extensively studied in the last decade. Among other reasons, interest in this problem was motivated by the problem of scheduling jobs in batch processing. Classical scheduling problems (i.e., with sequential tasks) are a particular case of the MT scheduling, and hence their complexity results apply directly to MT problems. It implies that scheduling independent MT is an \( \mathcal{NP} \)-hard problem \([5]\), in the ordinary sense if \( m \) is fixed. Du and Leung \([4]\) studied more precisely the complexity for MT scheduling problems, establishing that the problem with arbitrary precedence constraints is strongly \( \mathcal{NP} \)-hard for 2 processors, and scheduling independent MT is strongly \( \mathcal{NP} \)-hard for 5 processors.

Srinivasa Prasanna and Musicus \([20]\) developed an approach based on optimal control theory for a continuous version of malleable tasks, leading to optimal solution assuming the same particular parallel time function for all the tasks.

Jansen and Porkolab \([11]\) proposed a polynomial approximation scheme based on a linear programming formulation for scheduling independent malleable tasks. The complexity of the scheme, although linear in the number of tasks, is high independently of the accuracy of the approximation due to an exponential factor in the number of processors. Thus, even though the result is of great theoretical interest, this algorithm cannot be considered for a practical use.

We are interested in efficient, low complexity heuristics with a good performance guarantee. Most previous work is based on a two-phase approach proposed by Turek, Wolf, and Yu \([24]\). The basic idea is to select first an allotment (the number of
SCHEDULING INDEPENDENT MONOTONIC MALLEABLE TASKS

403

processors allotted to each task) and in a second step to solve the resulting non-

malleable scheduling problem, which is a classical scheduling problem of multiproces-

sor tasks. As far as the makespan criterion is concerned, this problem is related to a

2-dimensional strip-packing problem [1, 3, 12] for independent tasks. It is clear that

applying an approximation of guarantee \( \lambda \) for the nonmalleable problem on the allot-

ment of an optimal solution provides the same guarantee \( \lambda \) for the malleable problem

if ever an optimal allotment can be found. Two complementary ways for solving the

problem have been proposed, focusing either on the allotment (first phase) or on the

scheduling (second phase).

- Turek, Wolf, and Yu proposed a polynomial selection algorithm for the al-

lotment problem such that any \( \lambda \)-approximation algorithm of complexity

\( \mathcal{O}(f(n,m)) \) for the nonmalleable (multiprocessor) problem can be adapted

into a \( \lambda \)-approximation algorithm of complexity \( \mathcal{O}(mnf(n,m)) \) for the mal-

leable problem. Ludwig [13, 14] improved the complexity of the allotment

selection in the special case of monotonic tasks. Based on this result and on

the 2-dimensional strip-packing algorithm of guarantee 2 proposed by Stein-

berg [23], he presented a 2-approximation algorithm for scheduling indepen-

dent MT. The power of this approach is also its main limitation: any

improvement in the approximation of the strip-packing problem directly applies

to the MT problem, but the performance guarantee of the approach is limited

by the best known result for strip-packing.

- The other way corresponds to choosing an allotment such that the resulting

nonmalleable problem is not a general instance of strip-packing, and hence

better specific approximation algorithms can be applied. Using Knapsack as

an auxiliary problem for the selection of the allotment, this technique leads

to a \( (\sqrt{3} + \varepsilon) \)-approximation for monotonic tasks [16].

We focus in this paper on the second approach and show how a \( (3/2 + \varepsilon) \)-approxi-

mation algorithm can be obtained for any \( \varepsilon > 0 \). The basic idea is to determine an

allotment such that the tasks can be partitioned into two shelves of respective heights

d and \( d/2 \) for some deadline \( d \) to be determined.

2.2. Notation and basic properties. The aim of this work is to construct an

MT schedule for a set of \( n \) independent malleable tasks that minimizes the maximum

completion time over all the \( m \) processors. Recall that we assume that a processor can

calculate only one task at a time (no time sharing) and that the number of processors

dedicated to a task remains constant during all its execution. In addition we are

looking for nonpreemptive schedules with contiguous allocation, which means that for

each task the set of the subscripts of the processors allotted to it is an interval of \([1, m]\).

Their performance guarantee is established with respect to an optimal solution, which

may be contiguous or not.

2.2.1. Monotonic assumptions. We define the work function \( w_i \) of a task \( T_i \),

which corresponds to its computational area in the Gantt chart representation of a

schedule, as \( w_i : p \mapsto w_{i,p} = p \times t_{i,p} \) for \( p \leq m \). According to the usual behavior

of parallel programs, we will assume that the tasks are monotonic: allocating more

processors to a task decreases its execution time and increases its work.

Definition 2.1 (monotony).

- The time monotony is achieved by a set of tasks \( T \) when \( t_i \) is a decreasing

function for any task \( T_i \).

- The work monotony is achieved by a set of tasks \( T \) when \( w_i \) is an increasing

function for any task \( T_i \).
A set of tasks is monotonic if the two previous conditions are fulfilled.

Notice that an instance of the MT problem can always be transformed to fulfill the time monotony property, replacing the functions \( t_i \) by \( t'_i : p \mapsto \min\{t_{i,q} | q = 1, \ldots, p\} \).

This transformation does not affect the optimal solution of the scheduling.

Due to cache effects or scheduling anomalies described by Graham [7], the work monotony cannot be asserted for all the applications. However, it is a quite reasonable hypothesis, which is expected for most large actual parallel applications, mainly due to the communication overhead. From the parallel computing point of view, this monotonic assumption may be interpreted by the well-known Brent’s lemma [2], which states that the parallel execution of a task achieves some speedup if it is large enough, but does not lead to superlinear speedups.

We give below one useful definition for the presentation of our algorithm, together with two basic properties implied by the monotonic behavior of the tasks.

**Definition 2.2 (canonical number of processors).** Given a real number \( h \), we define for each task \( T_i \) its canonical number of processors \( \gamma(i, h) \) as the minimal number of processors needed to execute task \( T_i \) in time at most \( h \). If \( T_i \) cannot be executed in time less than \( h \) on \( m \) processors, we set by convention \( \gamma(i, h) = +\infty \).

Notice that if the set of tasks is monotonic, the canonical number of processors can be found in time \( O(\log m) \) by bisection search. In addition \( w_{i,\gamma(i,h)} \) is also the minimal work area needed to execute \( T_i \) in time less than \( h \).

**Property 1.** Given a real number \( h \), if \( \gamma(i, h) < +\infty \), the execution time of task \( T_i \) on its canonical number of processors satisfies the inequality

\[
 h \geq t_{i,\gamma(i,h)} > \frac{\gamma(i,h) - 1}{\gamma(i,h)} h.
\]

**Proof.** For short let us denote by \( p \) the canonical number of processors of task \( T_i \) for the given deadline \( h \). If \( p = 1 \), the inequality is clearly satisfied. Otherwise the monotonic behavior of the tasks implies that \( w_{i,p} \geq w_{i,p-1} \), i.e., \( p \times t_{i,p} \geq (p - 1) \times t_{i,p-1} \). By definition of the canonical number of processors, \( t_{i,p-1} > h \), which proves Property 1.

As a corollary, if the canonical number of processors is at least 2 for a task, we have the following simplified property.

**Property 2.** Given a real number \( h \), if \( \gamma(i, h) \in [2, m] \), we have

\[
 2t_{i,\gamma(i,h)} \geq t_{i,\gamma(i,h)-1} > h \geq t_{i,\gamma(i,h)} > \frac{1}{2} h.
\]

3. Description of the scheduling algorithm.

3.1. Principle. The principle of the algorithm is to use the dual approximation technique [9]. A \( \lambda \)-dual approximation algorithm for the MT scheduling problem takes a real number \( d \) as an input and

- either delivers a schedule of length at most \( \lambda d \), or
- answers correctly that there exists no schedule of length at most \( d \).

Ludwig and Tiwari [14] proposed a lower bound \( \omega \) that can be computed in time \( O(mn \log n) \) such that the optimal makespan \( d^* \) verifies \( \omega \leq d^* \leq 2 \omega \).

Hence a \( \lambda \)-dual approximation running in time \( f(n, m) \) can be converted, by bisection search, in a \( \lambda(1 + \varepsilon) \)-approximation running in time \( O(\min(n \log n + \log(1/\varepsilon)f(n, m)) \) for any \( \varepsilon > 0 \).

We are interested in this article in finding a 3/2-dual approximation. Let \( d \) be the current real number input for our dual approximation. In the following we assert that an MT schedule of length lower than \( d \) exists: thus we have to show how it is possible
to build a schedule of length at most $3d/2$. The idea of the algorithm is to partition the set of tasks into two shelves, one of height $d$ and the other of height $d/2$. Since the tasks are independent in both shelves, the scheduling strategy is straightforward after the allotment of the tasks has been determined and yields directly a solution of length at most $3d/2$. The main problem we face is to choose the tasks in each shelf in order to obtain a feasible solution. The way to determine the partition will be described in section 3.5.

3.2. Structure of an optimal schedule. To take advantage of the dual approximation paradigm, we have to make explicit the consequences of our assumption that there exists a schedule of length at most $d$. We state below some straightforward properties of such a schedule. They should give the insight for the construction of our solution.

Remark 1. In an optimal solution, the execution time of each task is at most $d$, and the total work is at most $md$.

Remark 2. In an optimal solution, if there exist two successive tasks (i.e., tasks executed successively on a common processor), at least one of these tasks has an execution time at most $d/2$.

The basic idea of the solution that we propose comes from the analysis of the shape of an optimal schedule. From Remark 2 the tasks whose execution times are strictly greater than $d/2$ do not use more than $m$ processors, and hence can be executed concurrently. The other tasks can be executed in time at most $d/2$. Thus, we are looking for a schedule in two shelves: $S_1$ of height $d$ and $S_2$ of height $d/2$.

3.3. Algorithm. Starting from the idea of constructing a schedule in two shelves, we sketch below the main steps of the dual approximation algorithm; the full details follow in the next sections. The input consists in a set of tasks ($T$), the number of processors ($m$), and a guess of the optimal makespan ($d$).

\begin{itemize}
\item 3/2 Dual Approximation $(T, m, d)$
\begin{enumerate}
\item Determine the set $T_S = \{T_i | t_{i,1} \leq d/2\}$ of tasks whose sequential time is at most $d/2$. Compute $W_S = \sum_{T_i \in T_S} t_{i,1}$.
\item Search for an allotment for $T \setminus T_S$ such that
\begin{itemize}
\item the total work is at most $md - W_S$,
\item each task has an execution time at most $d$,
\item the tasks whose execution times are strictly greater than $d/2$ require at most $m$ processors altogether.
\end{itemize}
If such an allotment does not exist, return NO.
\item Convert the allotment into a feasible 2-Shelves schedule of length at most $3d/2$ by calling algorithm BuildFeasible (see section 3.6).
\item Insert the set of tasks $T_S$ in the schedule using a greedy algorithm (see section 3.4) and return YES and the schedule.
\end{enumerate}
\end{itemize}

Sections 3.4–3.6 detail the implementation of these steps. Steps 1 and 4, presented together in section 3.4, simply mean that we can forget about sufficiently small tasks to build the 2-Shelves schedule. They can be implemented in time complexity $O(nm)$.

Step 2 is the critical point of the algorithm. It consists in choosing a first allotment that fulfills some properties to be a good candidate for the 2-Shelves schedule. We need
an oracle to either find such an allotment or deliver a certificate that no schedule of length at most $d$ exists.

This oracle is implemented in time $O(nm)$ using a knapsack formulation of the problem described in section 3.5.

Finally section 3.6 presents the algorithm $\text{BuildFeasible}$ which applies several simple transformations to the initial allotment in order to build a feasible schedule with two shelves of respective heights $d$ and $d/2$. Its time complexity is also in $O(nm)$, which leads to an overall complexity in $O(nm)$ for the dual approximation scheme.

### 3.4. Forgetting about small tasks.

Recall that we are looking for an $MT$ schedule of length at most $3d/2$, assuming that there exists a schedule of length at most $d$. Since we are interested in an approximation of the optimal solution, we can “forget” about some small tasks which do not affect the final performance of the algorithm. These small tasks are the set $\mathcal{T}_S$ of the tasks whose sequential execution time is at most $d/2$. Let us denote by $W_S$ the sum of the execution times of $\mathcal{T}_S$. We remark that $W_S$ is a lower bound of the work area of execution of $\mathcal{T}_S$ in any feasible schedule.

**Lemma 3.1.** If there exists a 2-Shelves schedule of length $3d/2$ for $\mathcal{T}\setminus\mathcal{T}_S$ with a work area at most $md - W_S$, then an MT schedule of length at most $3d/2$ can be derived for $\mathcal{T}$ in time $O(nm)$.

**Proof.** Consider a 2-Shelves schedule composed of shelves $S_1$ and $S_2$. We can modify the starting time of the tasks of $S_2$, which is currently $d$, to require that they all finish exactly at time $3d/2$. It creates on each processor an idle time interval between the completion of the tasks of $S_1$ and the starting of the tasks of $S_2$. We define the *load* of a processor as the sum of the execution times of the tasks computed by it. By definition the load is equal to $3d/2$ minus the length of the idle time interval on the processor. Now consider the following algorithm to schedule the tasks of $\mathcal{T}_S$ (see Figure 3.1):

- Consider the tasks in an arbitrary order $\mathcal{T}_S = \{T_1, \ldots, T_k\}$.
- Allocate task $T_i$ to the least loaded processor, at the earliest possible date. Update its load.

The only problem that may occur is that a task $T_i$ cannot be completed before the tasks of $S_2$. But at each step, the least loaded processor has a load at most $d$; otherwise it would contradict the fact that the total work area of the tasks is bounded by $md$. Hence, the idle time interval on this processor has a length at least $d/2$ and can contain the task $T_i$. \[\square \]
3.5. Partitioning the tasks into two shelves. In this section, we detail how to fill both shelves $S_1$ and $S_2$ by specifying an initial allotment of processors for the tasks. According to Lemma 3.1, we assume that only tasks with a sequential execution time strictly greater than $d/2$ remain in $T$.

In order to obtain a 2-Shelves schedule, we look for an allotment satisfying the following three constraints:

(C1) The total work area of the allotment is at most $W = md - W_S$.
(C2) The set $T_1$ of tasks with an execution time strictly greater than $d/2$ in the allotment uses a total of at most $m$ processors. These tasks are intended to be scheduled in $S_1$.
(C3) The set $T_2$ of tasks with an execution time at most $d/2$ in the allotment uses a total of at most $m$ processors. These tasks are intended to be scheduled in $S_2$.

Such an allotment clearly defines a 2-Shelves schedule of length at most $3d/2$ which would allow us to build a solution for the MT problem according to Lemma 3.1. Unfortunately, we have no certitude on the existence of such an allotment. Therefore, we relax the allotment problem looking for a solution which verifies only constraints (C1) and (C2), but might violate (C3).

Due to the monotonic assumption, we have only two allotments to consider for a task. If it is selected to belong to $T_1$, clearly $\gamma(i, d)$ is a dominant allotment; otherwise $\gamma(i, d/2)$ is. According to Remark 1, we note that $\gamma(i, d)$ is at most $m$ for all the tasks. To determine if such a relaxed allotment exists, we can solve the following optimization problem:

\[
\text{find } W^* = \min_{T_1 \subseteq T} \left( \sum_{i \in T_1} w_{i, \gamma(i, d)} + \sum_{i \notin T_1} w_{i, \gamma(i, d/2)} \right)
\]

under the constraint $\sum_{i \in T_1} \gamma(i, d) \leq m$.

This problem is in fact a well-known knapsack problem. Let us recall it briefly: given a set of $n$ items, each one associated to an integral weight $\omega_i$ and a profit $v_i$, and a knapsack with a total weight capacity $W$, find a subset of the tasks which can be contained by the knapsack with the maximal profit. This problem is $NP$-hard [5]; however, it admits [15, 19] a pseudopolynomial algorithm, using dynamic programming, that solves it exactly in time complexity in $O(nW)$.

Here, imagine that all the tasks are initially allotted to their canonical number of processors to respect the $d/2$ threshold. The problem is then to determine the set $T_1$ using at most $m$ processors such that the total weight is minimal. Hence, the profit of an item-task will correspond to the work saving obtained by executing the task just to respect the threshold $d$ instead of $d/2$, i.e., $v_i = w_{i, \gamma(i, d/2)} - w_{i, \gamma(i, d)}$. The weight of an item-task will be its canonical number of processors needed to respect the threshold $d$, $\omega_i = \gamma(i, d)$, while the capacity $W$ of the knapsack is $m$. Using this notation, the problem can be rewritten as the following knapsack problem:

\[
\text{find } W^* = \sum_{i \in T} w_{i, \gamma(i, d/2)} - \max_{T_1 \subseteq T} \sum_{i \in T_1} v_i
\]

under the constraint $\sum_{i \in T_1} \omega_i \leq m$. 


If the work area $W^*$ is greater than $W = md - W_S$, then there exists no solution with a makespan at most $d$, and the algorithm answers “NO” to the dual approximation. Otherwise, we will describe in detail in the next section how to construct a feasible solution with a makespan at most $3d/2$. Lemma 3.2 establishes the correctness of this dual approximation.

**Lemma 3.2.** Assuming that there exists a schedule of length at most $d$, the knapsack formulation of the problem delivers an allotment satisfying constraints (C1) and (C2) in time $O(nm)$.

**Proof.** Consider an optimal schedule. As already noted, the total work of tasks of $T_S$ in this schedule is at least $W_S$; hence the remaining tasks occupy an area bounded by $W = md - W_S$. The allotment of the optimal solution partitions these tasks into two sets $T'_1$ and $T'_2$, where $T'_1$ groups the tasks with execution time strictly greater than $d/2$. By definition any task $T_i$ is allotted to at least $\gamma(i, d)$ processors if it belongs to $T'_i$, and at least $\gamma(i, d/2)$ processors if it belongs to $T'_2$. Finally, as a direct consequence of Remark 2, the tasks of $T'_1$ use at most $m$ processors. It follows that $T'_1$ is a feasible solution for the knapsack procedure, with a resulting work area at most $W$. By definition it implies that the optimum $W^*$ of the knapsack procedure is at most $W$. □

### 3.6. Satisfying constraint (C3).

Starting from the allotment found by the knapsack procedure, we can construct a solution with the tasks of $T_1$ in $S_1$ and the others, $T_2 = T \setminus T_1$, in $S_2$. No more than $m$ processors are used to schedule the tasks in $S_1$, but it may happen that more than $m$ processors are needed in $S_2$. We then apply three possible transformations that will reduce this number to less than $m$. These transformations are applied until the resulting schedule becomes a feasible solution on $m$ processors. These transformations modify the shape of the 2-Shelves solution we are looking for, creating a new area $S_0$ whose processors are continuously busy in the time interval $[0, d]$, as depicted in Figure 3.2. The three transformations are the following (note that these transformations can be applied in any order):

1. If a task $T$ in $S_1$ has an execution time at most $3d/4$ and is allotted to $p > 1$ processors, allocate $T$ to $p - 1$ processors in $S_0$.
2. If $T$ and $T'$ in $S_1$ have an execution time lower than $3d/4$ and are each allotted to 1 processor, allocate $T$ and $T'$ to the same processor in $S_0$. A special case happens if $T$ is the only remaining sequential task of execution time at most $3d/4$.
3. Let $q$ denote the number of idle processors in $S_1$. If there exists a task $T_i$ in $S_2$ whose execution time on $q$ processors is bounded by $3d/2$, allocate $T_i$ on $\gamma(i, 3d/2)$ processors. According to the resulting execution time, $T_i$ is either scheduled in $S_0$ if $t \geq d$ or in $S_1$ otherwise.

Finally, the algorithm to build a feasible solution of length at most $3d/2$ is the following:

**Algorithm BuildFeasible**

- Start from the solution delivered by the Knapsack, $S_0 = \emptyset, S_1 = T_1, S_2 = T_2$.
- While the solution is not feasible
  - apply one of the transformations (1), (2), or (3).

The end of the section is devoted to the proof of Lemma 3.3.
SCHEDULING INDEPENDENT MONOTONIC MALLEABLE TASKS

Lemma 3.3. The algorithm BuildFeasible delivers a feasible schedule of length at most \(3d/2\) in time complexity \(O(nm)\).

Figure 3.2. The 2-Shelves schedule obtained from the allotment phase (left) and the final schedule given by BuildFeasible (right) with the new area \(S_0\).

Let \(m_0\) be the number of processors used to schedule the tasks of set \(S_0\) in the final solution. We denote by \(m' = m - m_0\) the remaining processors for the 2-Shelves schedule composed of \(S_1\) and \(S_2\). By construction any processor in \(S_0\) completes after deadline \(d\), which implies a work area greater than \(m_0d\). Since the total work area is bounded by \(W\), it is straightforward to remark that the total work area of tasks in \(S_1\) and \(S_2\) is bounded by \(m'd - W_2\). In addition the set \(S_1\) requires less than \(m'\) processors for the concurrent execution of its tasks. Hence, to prove Lemma 3.3, we are going to show that while the second shelf \(S_2\) requires more than \(m'\) processors, one of the three transformations can be applied. It is clear that the schedule restricted to \(S_1\) and \(S_2\) on \(m'\) processors, if feasible, verifies the conditions of application of Lemma 3.1. Thus, we can conclude that the algorithm is a \(3/2\)-dual approximation.

3.6.1. Algorithm BuildFeasible delivers a feasible schedule. Suppose that none of the transformations can be applied to the current schedule. We have to prove that this solution is feasible, i.e., requires at most \(m\) processors, which is equivalent by construction to proving that \(S_2\) requires at most \(m'\) processors.

Let \(q\) be the number of idle processors in the first shelf \(S_1\). Assume for the sake of contradiction that the second shelf \(S_2\) requires \(m_2 > m'\) processors. We have the following structure for the current schedule:

1. The total work area of tasks in \(S_1 \cup S_2\) is bounded by \(W' = m'd - W_2\).
2. Any task in \(S_1\) has a duration strictly greater than \(3d/4\), except possibly one sequential task whose execution time can be in the range \([d/2, 3d/4]\).
3. Any task in \(S_2\) has a duration strictly greater than \(d/4\).
4. Any task in \(S_2\) has a work area greater than \(3qd/2\), and hence is allotted to at least \(3q + 1\) processors.

The second point is a direct consequence of the fact that neither transformation (1) nor (2) can be applied. The third point is a corollary of Property 2: since any task of \(T \setminus T_3\) has a sequential execution time strictly greater than \(d/2\), all the tasks in \(S_2\) are allotted to \(\gamma(i, d/2) \geq 2\) processors. The last point comes from the fact that, due to
transformation (3), any task in $S_2$ has a duration greater than $3d/2$ when allotted to $q$ processors. Due to the monotonic assumption, the task’s current work is at least its work on $q$ processors, which is strictly greater than $q(3d/2)$. In particular, we have $3q t_3 q = w_{i,q} > q(3d/2)$, which implies that the time duration on $q$ processors is strictly greater than $d/2$. Thus by definition $\gamma(i, d/2) > 3q$.

To obtain a contradiction to the assumption that the current schedule is not a feasible solution, we will derive some lower bounds of the work area in $S_1$ and $S_2$ which will contradict the fact that their sum is bounded by $m’d$. We start by giving the lower bound used for the tasks in $S_1$, together with a very simple first lower bound for $S_2$.

**Lemma 3.4.** If the schedule is not feasible, the overall work area $W_1$ of $S_1$ is greater than $3d(m’-q)/4$, while the overall work area $W_2$ of $S_2$ is at least $d(m’+1)/4$.

**Proof.** The lower bound on $W_2$ is straightforward, since any task in $S_2$ has an execution time strictly greater than $d/4$, and the tasks of $S_2$ use at least $m’+1$ processors. The same argument holds for $W_1$ if no sequential task of duration at most $3d/4$ exists in the shelf. Otherwise let $T$ be this unique task, with a sequential time in the range $[d/2, 3d/4]$.

Let us first establish that $T$ cannot be the only task scheduled in $S_1$. Indeed, assume for the sake of contradiction that it is the case. If there is no idle processor in $S_1$ ($q = 0$), we simply have $m’ = 1$. Hence at least $W_1 > d/2$ while the lower bound on $S_2$ can be rewritten as $W_2 > d/2$. It contradicts the fact that $W_1 + W_2$ is bounded by $m’d = d$. If there exist some idle processors, then we have $q = m’ - 1 > 0$. As $S_2$ contains at least one task, $W_2 > 3qd/2 = 3(m' - 1)d/2$. We obtain $m’d > d/2 + 3(m’-1)d/2 = (3m’ -2)d/2$, which implies $2 > m’$, contradicting $q > 0$.

Hence, at least another task $T'$ is partially scheduled in $S_1$ together with $T$. Since transformation (2) cannot be applied, task $T'$ has an execution time $t'$ strictly greater than $3d/2 - t$. Thus, considering the processor executing $T'$ and the processor executing $T$, their average load is strictly greater than $3d/4$. Since any other non-idle processor is occupied by a task in $S_1$ with an execution time greater than $3d/4$, we obtain $W_1 > 3(m’-q)/4$. 

To conclude that a non-feasible schedule leads to a contradiction, we distinguish between two cases, depending on whether or not there exist some idle processors in $S_1$. 

**Case 1.** Assume $q = 0$. In this case Lemma 3.4 leads directly to a contradiction. Indeed we have $m’d \geq W_1 + W_2 > 3md/4 + (m’+1)d/4 > m’d$. 

**Case 2.** Assume $q > 0$. We need a more accurate lower bound on $W_2$ to complete the proof. Let $k$ be the number of tasks in $S_2$. By construction we have

$$W_2 = \sum_{i \in S_2} w_{i,\gamma(i,d/2)}.$$

We can express the work of each task in two different ways. First, using the fact that this work is greater than $3qd/2$, we obtain

$$W_2 > \frac{3}{2} qdk.$$  

Second, due to monotony, the work of each task $T_i$ when allotted on fewer processors never increases: $w_{i,\gamma(i,d/2)} \geq w_{i,\gamma(i,d/2)-1}$. By definition, the execution time on $\gamma(i,d/2) - 1$ processors is strictly greater than $d/2$, which implies that $w_{i,\gamma(i,d/2)-1} > (\gamma(i,d/2) - 1)d/2$. We have

$$W_2 > \left( \sum_{i \in S_2} \gamma(i,d/2) - k \right) \frac{d}{2} \geq \frac{1}{2} (m’ + 1 - k)d.$$
Using the lower bound established in Lemma 3.4 on $W_1$ we can rewrite the upper bound $m'd$ on the total work area as $W_2 < m'd/4 + 3qd/4$. Using (3.1), we obtain

$$6qk < m' + 3q \iff 3q(2k - 1) < m'.$$

Using (3.2), we get

$$2(m' + 1 - k) < m' + 3q \iff m' < 3q + (2k - 2).$$

By transitivity we obtain the following strict inequality:

$$3q(2k - 1) < 3q + (2k - 2) \iff 3q(2k - 2) < (2k - 2) \iff (3q - 1)(k - 1) < 0.$$

But both $k$ and $q$ are greater than 1, which contradicts the previous inequality. This concludes the proof of Lemma 3.3 by contradiction.

### 3.6.2. Time complexity of BuildFeasible.

We finally establish the time complexity of the algorithm. Each of the three transformations either moves a task from $S_2$ to $S_1$ or $S_0$, or from $S_1$ to $S_0$. Hence at most two transformations can be applied to any task. Let $N$ be the number of tasks to deal with in our problem, after the elimination of the “small” tasks of $T_S$. If we look at the time complexity of each of the three transformations at a step of the algorithm, we have the following:

- The first two transformations can be implemented in time complexity $O(N)$ by a simple scan of the tasks in $S_1$.
- Transformation (3) can also be implemented in time $O(N)$ scanning the tasks of $S_2$. The determination of $\gamma(i, 3d/2)$ for the elected task $T_i$ can be computed in time $O(\log m)$ by a bisection search.

Since at most $2N$ transformations can be applied, algorithm BuildFeasible has an overall complexity in $O(N^2 + N \log m)$. To obtain a time complexity in $O(nm)$, simply notice that $N$ is bounded by both $n$ and $2m$: indeed any of the $N$ tasks has a sequential execution time greater than $d/2$ for a total work bounded by $md$. Monotony implies that $N \leq 2m$.

### 4. Conclusion.

We have presented in this paper a new algorithm for scheduling a set of independent malleable tasks. It improves significantly the best bound known at this time, with a performance guarantee of $\frac{3}{2} + \varepsilon$ for any $\varepsilon > 0$ in time complexity $O(nm \log(n/\varepsilon))$. The basic idea was to focus on the first phase of allotment using a knapsack formulation of the problem.

The natural continuation of this work is to study the scheduling of other structures of precedence graphs with malleable tasks. We believe that a similar analysis in two phases with a sophisticated allotment algorithm should lead to good approximation algorithms.

Another promising feature of MT is its intrinsic hierarchical behavior which should help in developing good scheduling algorithms for cluster computing. This issue is under investigation.

### REFERENCES

