A 3/2-Dual Approximation Algorithm for Scheduling Independent Monotonic Malleable Tasks

Grégory Mounié, Christophe Rapine, Denis Trystram

▶ **To cite this version:**

Grégory Mounié, Christophe Rapine, Denis Trystram. A 3/2-Dual Approximation Algorithm for Scheduling Independent Monotonic Malleable Tasks. SIAM Journal on Computing, 2007, 37 (2), pp.401–412. 10.1137/S0097539701385995 . hal-00002166v2

HAL Id: hal-00002166
https://hal.science/hal-00002166v2

Submitted on 2 Dec 2004

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

A $\frac{3}{2}$-DUAL 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, 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 height 1 and $\frac{1}{2}$.

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

AMS subject classifications. 15A15, 15A09, 15A23

1. Introduction.

The implementations of large actual applications on parallel and distributed systems are based on algorithmic studies where scheduling and load-balancing issues are the central points to be considered. There exists a very large literature addressing the problem of scheduling efficiently the tasks of a parallel program. This problem corresponds to find for each task a date to start its execution together with a processor location. Tasks in this model correspond to indivisible pieces of the application to be 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 task graphs (unit execution times and unit communication times) and extended by Papadimitriou and Yannakakis [18]. In this model, the communications between tasks allocated to different processors are considered explicitly by the transmission time of a message between them. The communication times between tasks within the same processor are neglected. The scheduling UET-UCT problem is known to be $\mathcal{NP}$-hard in the strong sense [21], and not approximable within a factor of 5/4 of the optimum by any polynomial algorithm [10], unless $\mathcal{P} = \mathcal{NP}$. The best known approximation result is due to Hanen & Munie [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 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. A malleable task is a computational unit which may 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

*ID-IMAG, avenue Jean Kuntzman - ZIRST, 38 330 Montbonnot Saint Martin, France. e-mail Denis.Trystram@imag.fr

† GILCO - INPG, 46 avenue Felix Viallet, 38 031 Grenoble cedex, France. e-mail Christophe.Rapine@gilco.inpg.fr

tasks on a multiprocessor system composed by m identical processors. An instance of the problem is a set $\mathcal{T} = \{T_1, \ldots, T_n\}$ of tasks, together with a set of n functions $t_i : p \to t_{i,p}$ which indicates 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 $\mathcal{P}_i$ of the processors to execute it, under the constraints that:

- Task T_i starts simultaneously its execution on all the processors of $\mathcal{P}_i$, and occupy them without interruption till its completion time $C_i = st_i + t_{i,|\mathcal{P}_i|}$.
- A processor executes at most one task at a time.

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 malleable tasks, which leads to a performance guarantee of $\frac{3}{2} + \varepsilon$ for any $\varepsilon > 0$, in time $\mathcal{O}(nm \log(n/\varepsilon))$. This bound improves all existing practical results for solving this problem.

The organization of this paper is the following: we first present a brief survey on related works and recall the model of MT and its main properties. Then we discuss the principle of our approach, and we present the algorithm and analyze its performance guarantee. Some experimental results are reported at the end of the paper to assert the average behavior of our algorithm compared to the other existing approaches.

2. Preliminaries on Malleable Tasks.

2.1. Related works. The problem of scheduling independent malleable tasks has been extensively studied in the last decade. This interest was motivated among others 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 a $\mathcal{NP}$-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.

Prasanna et al. [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 approximable 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 if the result has an important theoretical interest, this algorithm can not be considered for a practical use.

We are interested in efficient, low complexity, heuristics with good performance guarantee. Most existing works are based on a two-phases approach proposed by Turek, Wolf and Yu [24]. The basic idea is to select in a first step an allotment (the number of 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 multiprocessor tasks. As far as the makespan criterion is concerned, this problem is identical to a 2-dimensional strip-packing problem [1, 3, 12] for independent tasks. It is clear that applying an approximation of guarantee λ for the non-malleable problem on the allotment of an optimal solution provides the same guarantee λ 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 allotment problem such that any λ-approximation algorithm of complexity $\mathcal{O}(f(n,m))$ for the non-malleable (multiprocessor) problem can be adapted into a λ-approximation algorithm of complexity $\mathcal{O}(mnf(n,m))$ for the malleable 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 Steinberg [23], he presented a 2-approximation algorithm for scheduling independent MT. The powerfulness 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 choose an allotment such that the resulting non-malleable 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 $(\frac{3}{2}+\varepsilon)$-approximation 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 explicit.

2.2. Notations and basic properties. The aim of this work is to construct a 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 compute only one task at a time (no time sharing) and that the number of processors allocated to a task remains constant during all its execution. In addition we are looking for non-preemptive schedules with *contiguous* allocation, which means that for each task the set of the subscripts of the processors alloted to it is an interval of $[1,m]$. Their performance guarantee are established in 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 $w_i : p \mapsto w_{i,p} = p.t_{i,p}$ 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 $\mathcal{T}$ when t_i is a decreasing function for any task T_i.*
- *The work monotony is achieved by set of tasks $\mathcal{T}$ when w_i is an increasing function for any task T_i.*

A set of task 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.

From the parallel computing point of view, this monotonic assumptions 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 super-linear speedups. Due to cache effects or scheduling anomalies described by Graham [7], this behavior can not be asserted for all the applications. However, it is a quite reasonable hypothesis, that is expected for most large actual parallel applications, mainly due to the communication overhead. 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 can not 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 $\mathcal{O}(\log m)$ by dichotomic 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 a 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 simplified following property:

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

$$2t_{i,\gamma(i,h)} \geq t_{i,\gamma(i,h)-1} > h \geq t_{i,\gamma(i,h)} > \tfrac{1}{2}\, h$$

3. Description of the scheduling algorithm.

3.1. Principle. The principle of the algorithm is to use the dual approximation technique [9]. A λ-dual approximation algorithm for the MT-scheduling problem takes a real number d as an entry and:

- either delivers a schedule of length at most λd,
- or answers, correctly, that there exists no schedule of length lower than d.

Ludwig and Tiwari [14] proposed a lower bound ω that can be computed in time $\mathcal{O}(mn \log n)$, such that the optimal makespan d^* verifies $\omega \leq d^* \leq 2\omega$. Hence a λ-dual approximation running in time $f(n,m)$ can be converted, by dichotomic search, in a $\lambda(1+\varepsilon)$-approximation running in time $\mathcal{O}(mn \log n + \log(1/\varepsilon) f(n,m))$ for any $\varepsilon > 0$.

We are interested in this article in finding a $3/2$-dual approximation. Let d be the current real number entry for our dual approximation. In the following we assert that a 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 $3/2d$. The idea of the algorithm is to partition the set of tasks into two shelves, one of height d and the other of height $d/2$. As the tasks are independent in both shelves, the scheduling strategy is straightforward after the allotment of the tasks has been determined, and yields directly to a solution of length at most $3/2d$. The main problem to face with 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 detail later.

3.2. Structure of an optimal schedule. To take advantage of the dual approximation paradigm, we have to explicit the consequences of our assumption that a schedule of length lower than d exists. 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 lower than d and the total work is lower than md.*

REMARK 2. *In an optimal solution, if there exists two successive tasks (i.e. tasks allocated successively to a common processor), at least one of these tasks has an execution time lower than $d/2$.*

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 lower than $d/2$. 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, let us detail the successive steps of the algorithm.

1. Remove the set $\mathcal{T}_S$ of tasks whose execution times on one processor are lower than $d/2$.
2. Find for the remaining tasks a processor allotment such that any task has an execution time lower than d. From this allotment, we can partition the tasks into two sets $\mathcal{T}_1$ and $\mathcal{T}_2$, composed respectively of the tasks having an execution time strictly greater than, respectively lower or equal to $d/2$ in the allotment. The crucial point of the algorithm is the choice of this allotment in order that it fulfills some properties to be a good candidate for the 2-Shelves schedule. This choice is done using a Knapsack formulation of the problem.
3. Apply some basic transformations to build a feasible 2-Shelves schedule. The allotment may be modified, but a task can only get less processors than in the initial allotment.
4. Finally insert the set of tasks $\mathcal{T}_S$, determined in the first step, in the schedule.

3.4. Forgetting about the small tasks. Recall that we are looking for a MT-schedule of length at most $3/2d$, assuming that there exists a schedule of length lower than d. As usual in many approximation schemes, since we are interested in a solution at a factor of $3/2$ of the optimal solution, we can "forget" about some small tasks which do not affect the final performance of our algorithm. This small tasks are in our case the set $\mathcal{T}_S$ constituted by the tasks whose sequential execution time is lower than or equal to $d/2$. Let denote by $\mathcal{W}_S$ the sum of the execution times of $\mathcal{T}_S$. Remark that $\mathcal{W}_S$ is a lower bound of the work area of execution of $\mathcal{T}_S$ in any feasible schedule.

LEMMA 1. *If a 2-Shelves schedule of length $3/2d$ exists for $\mathcal{T}\backslash\mathcal{T}_S$ with a work area lower than $md - \mathcal{W}_S$, then a MT-schedule of length at most $3/2d$ can be derived for $\mathcal{T}$ in time $\mathcal{O}(nm)$.*

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 impose that they all finish exactly at time $3/2d$. It creates on each processor an idle time interval between the completion of the task of S_1 and the starting of the one of S_2. We define the *load* of a processor as the sum of the execution time of the tasks allocated to it. By definition the load is equal to $3/2d$ minus the length of the idle time interval on the processor. Now consider the following algorithm to schedule the tasks of $\mathcal{T}_S$:

- Consider the tasks in an arbitrary order $\mathcal{T}_S = \{T_1, \dots, T_k\}$.

– Allocate task T_i to the less loaded processor, at the earliest possible date. Update its load.

The only problem that may occur is that a task T_i can not be scheduled before the tasks of S_2. But at each step, the less loaded processor necessary has a load lower than 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. □

3.5. Partitioning the tasks into two Shelves. In this section, we detail how to fill both shelves S_1 and S_2 by expliciting a first processors allotment for the tasks. Accordind to lemma 1, we can assume that only tasks with sequential execution time strictly greater than $d/2$ remain in $\mathcal{T}$. In order to obtain by the end a 2-Shelves schedule, we look for an allotment satisfying the 3 following constraints:

($C1$) The total work area of the allotment is lower than $\mathcal{W} = md - \mathcal{W}_\mathcal{S}$.

($C2$) The set $\mathcal{T}_1$ of tasks with an execution time strictly greater than $d/2$ in the allotment uses at most m processors. These tasks are intended to be scheduled in S_1.

($C3$) The set $\mathcal{T}_2$ of tasks with an execution time lower than $d/2$ in the allotment uses 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 $3/2d$ which would allow us to build a solution to the MT-problem according to lemma 1. Unfortunately we have no certitude on the existence of such an allotment. To tackle with this point, we relax the allotment problem looking for a solution which verifies only constraints ($C1$) and ($C2$), maybe violating ($C3$).

We use a well-known knapsack algorithm to find such a relaxed allotment. Let us recall briefly this problem: given a set of n items, each one associated to an integral weight w_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 $\mathcal{NP}$-hard [5], however, it admits [15, 19] a pseudo-polynomial algorithm, using dynamic programming, that solves it exactly in time complexity in $\mathcal{O}(nW)$.

The idea is to solve a knapsack problem where the profit of an item-task will be its work. The weight of an item-task will correspond to the number of processors alloted to it. The subset $\mathcal{T}_1$ of tasks solution of the knapsack corresponds to the tasks executed in time strictly greater than $d/2$. Finally, the capacity constraint is that the total number of processors that execute tasks in $\mathcal{T}_1$ is limited by m. The objective is to minimize the total work area W^*. Due to the monotonic assumption, we have only 2 allotments to consider for a task. If it is selected to belong to $\mathcal{T}_1$, clearly $\gamma(i,d)$ is a dominant allotment, otherwise $\gamma(i,d/2)$ is. Notice that, due to remark 1, $\gamma(i,d)$ is lower than m for all the tasks. The problem can be formulated as:

$$\text{find } W^* = \min_{\mathcal{T}_1 \subseteq \mathcal{T}} \left(\sum_{i \in \mathcal{T}_1} w_{i,\gamma(i,d)} + \sum_{i \notin \mathcal{T}_1} w_{i,\gamma(i,d/2)} \right)$$

$$\text{under the constraint } \sum_{i \in \mathcal{T}_1} \gamma(i,d) \leq m$$

If the work area W^* is greater than $\mathcal{W} = md - \mathcal{W}_\mathcal{S}$ then there exists no solution with a makespan lower than d and the algorithm answers 'NO' to the dual approximation. Otherwise, we will detail in the next section how to construct a feasible solution

with a makespan lower than $3/2d$. The lemma below establishes the correctness of this dual approximation:

LEMMA 3.1. *Assuming that there exists a schedule of length lower than d, the Knapsack formulation of the problem delivers in time $\mathcal{O}(nm)$ an allotment satisfying constraints $(C1) - (C2)$.*

Proof. Consider an optimal schedule. As it was already noticed, the total work of tasks of $\mathcal{T}_S$ in this schedule is at least $\mathcal{W}_S$, hence the remaining tasks occupy an area bounded by $\mathcal{W} = md - \mathcal{W}_S$. The allotment of the optimal solution partitions these tasks into two sets $\mathcal{T}_1'$ and $\mathcal{T}_2'$, where $\mathcal{T}_1'$ groups the tasks with execution time strictly greater than $d/2$. By definition any task T_i is alloted to at least $\gamma(i, d)$ processors if it belongs to $\mathcal{T}_1'$, and $\gamma(i, d/2)$ processors if it belongs to $\mathcal{T}_2'$. Finally, as a corollary of remark 2, tasks of $\mathcal{T}_1'$ use less than m processors. It follows that set $\mathcal{T}_1'$ is a feasible solution for the knapsack procedure, with a resulting work area lower than $\mathcal{W}$. By definition it implies that the optimum W^* of the Knapsack procedure is lower than or equal to $\mathcal{W}$. □

3.6. Satisfying constraint $(C3)$. Starting from the allotment found by the Knapsack procedure, we can construct a solution with the tasks of $\mathcal{T}_1$ in S_1 and the others, $\mathcal{T}_2 = \mathcal{T} \backslash \mathcal{T}_1$, 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 3 possible transformations that will reduce this number to less than m. These transformations are applied until the resulting schedule is 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 $[0, d]$, see figure 3.1. The 3 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 lower than $3/4d$ and is alloted 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 $3/4d$ and are each alloted 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 lower than $3/4d$. It is then allocated on top of a task of S_1, if exists, of duration greater than $3/4d$, such that the completion time of T does not exceed $3/2d$.
3. Let q denotes the number of idle processors in S_1. If it exists a task T_i in S_2 such that its execution time on q processors is bounded by $3/2d$, allocate T_i on $\gamma(i, 3/2d)$ processors. According to the resulting execution time, T_i is either scheduled in S_0 or in S_1.

The algorithm to build a feasible solution of length lower than $3/2d$ is then the following:

```
Algorithm BuildFeasible
    • Start from the solution delivers by the Knapsack
      formulation, S0 = ∅, S1 = T1, S2 = T2.
    • While the solution is not feasible
          apply one of the transformation (1), (2) or (3).
```

The end of this section is devoted to prove the lemma 3.2:

LEMMA 3.2. *The algorithm* `BuildFeasible` *delivers a feasible schedule of length at most $3/2d$ in time complexity $\mathcal{O}(nm)$.*

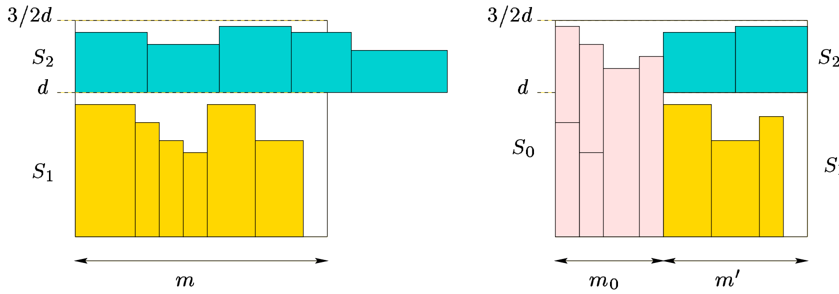

FIG. 3.1. *The 2-Shelves schedule obtained from the allotment phase (left) and the final schedule given by* `BuildFeasible` *(right) with the new area* S_0.

Notice that the transformations ensure by construction that the makespan remains bounded by $3/2d$ at each step of the algorithm (for transformation (1), this is a direct consequence of property 2). In addition a transformation can only decrease the number of processors alloted to a tasks, which asserts due to monotony that the total work area of the schedule remains bounded by $\mathcal{W} = md - \mathcal{W}_{\mathcal{S}}$ at any step of `BuildFeasible`.

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 $m_0 d$. Since the total work area is bounded by $\mathcal{W}$, it is straightforward to remark that the total work area of tasks in S_1 and S_2 is bounded by $m'd - \mathcal{W}_{\mathcal{S}}$. In addition set S_1 requires less than m' processors for the concurrent execution of its tasks. Hence to prove lemma 3.2, we are going to show that while the second shelf S_2 requires more than m' processors, one of the 3 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 1, which makes us able to conclude that the algorithm is a 3/2-dual approximation.

3.6.1. Algorithm `BuildFeasible` delivers a feasible schedule. Consider that none of the transformations can be applied to the current schedule. We have to prove that this solution is feasible, i.e. requires less than m processors, which is equivalent by construction to prove that S_2 requires less than m' processors.

Let q be the number of idle processors in the first shelf S_1. Assume for 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 $\mathcal{W}' = m'd - \mathcal{W}_{\mathcal{S}}$.
2. Any task in S_1 has a duration strictly greater than $3/4d$, except possibly one sequential task whose execution time can be in the range $]d/2, 3/4d]$.
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 $3/2qd$, and hence is alloted 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 $\mathcal{T} \backslash \mathcal{T}_{\mathcal{S}}$ has a sequential execution time strictly greater than $d/2$, all the tasks in S_2 are alloted 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 $3/2d$ when alloted to q processors. Due to the monotonic behavior its current work is at least its

work on q processors, which is strictly greater than $q.(3/2d)$. In particular, we have $3q.t_{i,3q} \geq w_{i,q} > q.(3/2d)$, which implies that the time duration on $3q$ processor 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 explicit 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 3.3. *If the schedule is not feasible, the overall work area $\mathcal{W}_1$ of S_1 is at least $3/4d(m' - q)$, while the overall work area $\mathcal{W}_2$ of S_2 is at least $1/4d(m' + 1)$.*

Proof. The lower bound on $\mathcal{W}_2$ is straightforward, since any task in S_2 has an execution time strictly greater than $d/4$, and tasks of S_2 use at least $m'+1$ processors. The same argument holds for $\mathcal{W}_1$ is no sequential task of duration lower than $3/4d$ exists in the shelf. Otherwise let T be this unique task, with a sequential time t in the range $]d/2, 3/4d]$.

Let us first establish that T can not be the only task scheduled in S_1. Indeed assume for sake of the contradiction that it is the case. If there is no idle processor in S_1 ($q = 0$), we have simply $m' = 1$. Hence at least $\mathcal{W}_1 > d/2$ while the lower bound on S_2 can be rewritten as $\mathcal{W}_2 > d/2$. It contradicts the fact that $\mathcal{W}_1 + \mathcal{W}_2$ is bounded by $m'd = d$. If some idle processors exists, we then have $q = m' - 1 > 0$. As S_2 contains at least one task, $\mathcal{W}_2 > 3/2qd = 3/2(m' - 1)d$. We obtain $m'd > d/2 + 3/2(m' - 1)d = (3m' - 2)d/2$, 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) can not be applied, task T' has an execution time t' strictly greater than $3/2d - t$. Thus considering one processor allocated to T' and the processor executing T, their average load is strictly greater than $3/4d$. Since any non idle processor is occupied by a task in S_1 with an execution time greater than $3/4d$, we obtain $\mathcal{W}_1 > 3/4(m' - q)$. ◻

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

case 1. Assume $q = 0$. In this case lemma 3.3 leads directly to a contradiction. Indeed we have: $m'd \geq \mathcal{W}_1 + \mathcal{W}_2 > 3/4m'd + 1/4(m' + 1)d > m'd$

case 2. Assume $q > 0$. We need a more accurate lower bound on $\mathcal{W}_2$ to conclude. Let k be the number of tasks in S_2. By construction we have

$$\mathcal{W}_2 = \sum_{i \in S_2} w_{i,\gamma(i,d/2)}$$

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

$$\mathcal{W}_2 > \frac{3}{2}qdk \tag{3.1}$$

Second, due to monotony, the work of each task T_i when alloted to one less processors can only increase: $w_{i,\gamma(i,d/2)} \geq w_{i,\gamma(i,d/2)-1}$. 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:

$$\mathcal{W}_2 \ > \ (\sum_{i \in S_2} \gamma(i, d/2) - k)\frac{d}{2} \ \geq \ \frac{1}{2}(m' + 1 - k)d \tag{3.2}$$

Using the lower bound established in lemma 3.3 on $\mathcal{W}_1$ we can rewrite the upper bound $m'd$ on the total work area as $W_2 < m'd/4 + 3/4qd$. Using equation (3.1), we obtain:

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

Using in its turn equation (3.2), we have:

$$2(m' + 1 - k) < m' + 3q \;\Leftrightarrow\; m' < 3q + (2k-2)$$

By transitivity we obtain the following strict inequality:

$$3q(2k-1) < 3q + (2k-2) \;\Leftrightarrow\; 3q(2k-2) < (2k-2) \;\Leftrightarrow\; (3q-1)(k-1) < 0$$

But both k and q are greater than 1, which makes impossible the previous inequality. This concludes the proof of lemma 3.2 by contradiction.

3.6.2. Time Complexity of `BuildFeasible`. We finally establish the time complexity of the algorithm. Each of the 3 transformations either moves a task from S_2 to S_1 or S_0, or from S_1 to S_0. Hence at most 2 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 $\mathcal{T}_S$. If we look at the time complexity of each of the 3 transformations at a step of the algorithm, we have:

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

Since at most $2N$ transformations can be applied, and in particular transformation (3) can be applied at most N times, algorithm `BuildFeasible` has an overall complexity in $\mathcal{O}(N^2 + N \log m)$. To obtain a time complexity in $\mathcal{O}(nm)$, simply notice that N is bounded both by 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. Experiments. We have presented a 3/2-dual approximation for scheduling a set of monotonic independent malleable tasks. The guarantee of 3/2 corresponds to an upper bound of the worst case ratio compared to an optimal solution. In this section we report some experiments to study the average behavior of the algorithm. We compare its performances with three other existing algorithms, namely:

- To serve as a reference, the well-known LPT scheduling algorithm. Recall that LPT (Largest Processing Time first) is a single processor algorithm where the tasks are allocated to the available processors according to the decreasing order of their length [7]. This comparison is done to show the gain obtained in the parallelization of the tasks.
- Our new algorithm with a worst case performance ratio of 3/2, denoted by 3/2-ALG.
- The algorithm from [16] with a worst case ratio of $\sqrt{3}$, denoted by $\sqrt{3}$-ALG.
- The version from Ludwig [13] of the algorithm from Turek, Wolf and Yu [24], which uses Steinberg's strip packing algorithm [23].

Experiments report the performance ratio of these algorithms compared to the lower bound of the optimum proposed by Ludwig [13]. Hence the results are overestimations of real performance ratios.

4.1. Description of the random instances. For the experiments, the time functions of the tasks have been determined randomly. To preserve the monotonic behavior we define the execution time $t_{i,p+1}$ of task T_i on $p+1$ processors as:

$$t_{i,p+1} = \frac{p + X_{[0,1]}}{p+1}\, t_{i,p}$$

where X is a random variable and $X_{[0,1]}$ its restriction to the real interval $[0,1]$. For experiments we have generated X both from a Gaussian distribution $\mathcal{N}(m,\sigma)$ centered on m and with a standard deviation σ, and from a uniform distribution $\mathcal{U}[a,b]$ on a real interval $[a,b]$. The sequential times of the tasks are chosen according to the same distribution. To shorten the presentation of the results, we will report only the case of 32 parallel processors, which is quite representative of the general behavior of the algorithms.

4.2. Distribution of the solutions. Let us first study the distribution of the solutions in order to validate the average measures. From the experiments, it appeared that the ratios of performance of the algorithms hardly vary with the distribution of the time functions. Thus, we decided to present the results for an "average" distribution X following the Gaussian distribution $\mathcal{N}(0.5, 0.5)$. Figure 4.1 presents the distribution of the performance ratios for 3/2-ALG and Ludwig's over 50000 experiments and for 20 tasks.

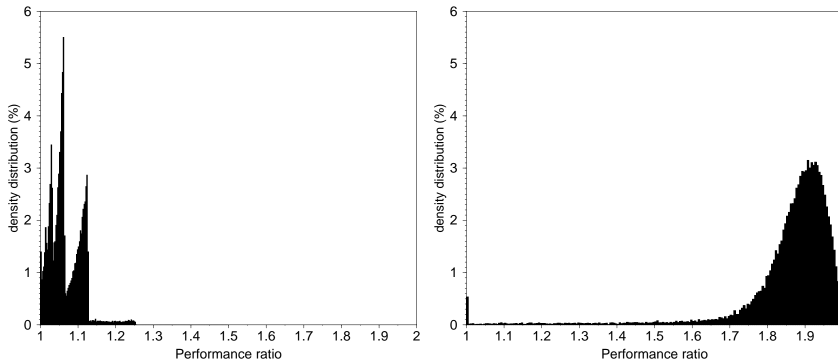

FIG. 4.1. *Distribution of the algorithm performance ratio (left) compared to Ludwig's (right) for a Gaussian distribution* $\mathcal{N}(0.5, 0.5)$ *on* 32 *processors with* 20 *tasks.*

The first remark is that Ludwig's algorithm performance ratio follows a Gaussian-like distribution, with an expected value quite closed to its theoretical upper bound of 2. On the other hand the 3/2-ALG has a small mean value but with a less regular distribution: the range of values is narrower than Ludwig's one, but with at least 2 distinct peaks of probability, emphasing two different behaviors of the algorithms. We interpret these peaks as the use or not of the transformation phases to build a feasible solution after the knapsack allotment procedure.

One interesting behavior of the algorithm occurs for what we can call *perfect parallel tasks*, i.e. tasks with a linear speedup. This is in fact the less favorable case for the algorithm, and figure 4.2 shows that its ratio distribution becomes then a Gaussian-like distribution. We have also reported the distribution of the $\sqrt{3}$-ALG in this situation, which differs in this case from the 3/2-ALG.

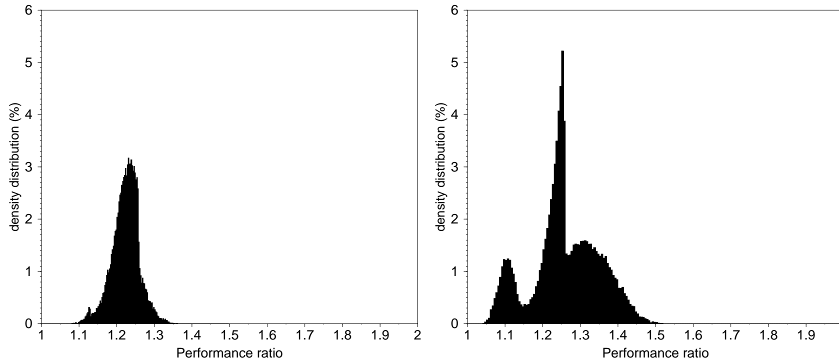

FIG. 4.2. *Distribution of the algorithm (left) compared to* $\sqrt{3}$-ALG *(right) for perfect parallel tasks on* 32 *processors with* 20 *tasks.*

4.3. Average Behavior. We report here the average behavior of the 4 algorithm on 32 processors. Figure 4.3 shows the average performance ratios for a number of tasks from 1 to 150, and time functions using the Gaussian distribution $\mathcal{N}(0.5, 0.5)$. Each points in the following figures represents the mean value over 500 experiments.

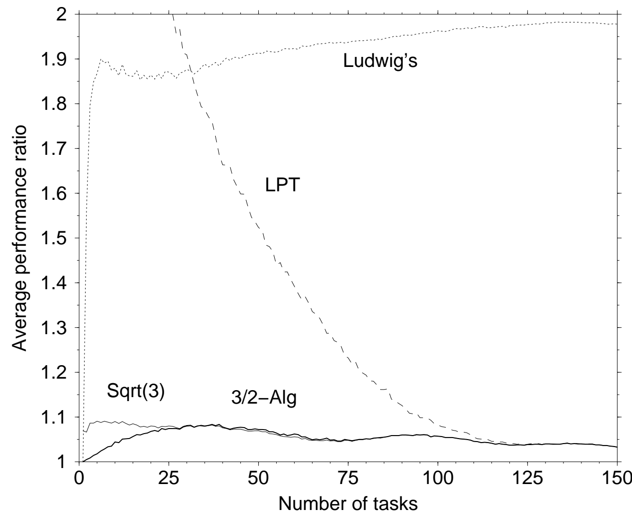

FIG. 4.3. *Average performance ratios of the* 4 *algorithms on* 32 *processors with a Gaussian distribution* $\mathcal{N}(0.5, 0.5)$ *for time functions.*

These results are quite similar for other distributions (normal or uniform) we have tested. In all these experiments, we observe that the use of the Steinberg's algorithm for the strip-packing leads to an average guarantee close to the worst case 2 for Ludwig's algorithm. On the contrary, both other algorithms 3/2 and $\sqrt{3}$-ALG are very close to optimality (beneath 10%). This means that in average, the guarantee is much better than their respective worst cases 3/2 and $\sqrt{3}$. Finally sequential scheduling (LPT) is competitive with parallel scheduling only for more than one hundred malleable tasks. In practise, the use of MT is interesting when the number of tasks is not too large.

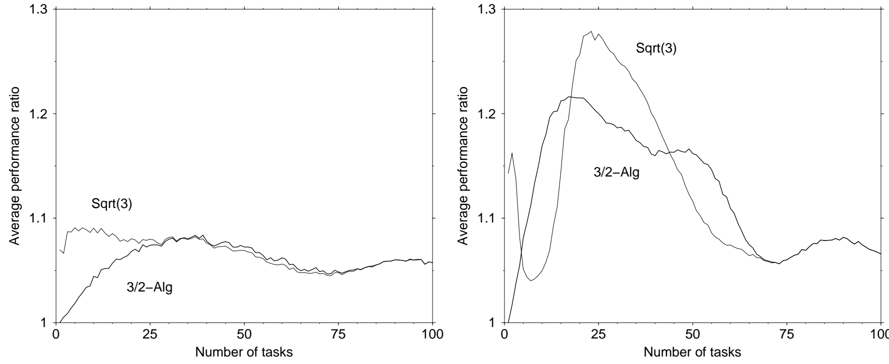

FIG. 4.4. *Average performance ratios of the of 2 dual algorithms on 32 processors with a Gaussian distribution $\mathcal{N}(0.5, 0.5)$ (left) and perfect parallel tasks (right).*

Figure 4.4 focuses on the average performance on the 3/2 and $\sqrt{3}$ algorithms for the previous $\mathcal{N}(0.5, 0.5)$ distributions and for perfect parallel tasks, which appeared in our experiments to be the worst situation for both algorithms: 3/2-ALG can be around 20% over the optimal value, while $\sqrt{3}$-ALG can be around 30%. It appears that these algorithms have quite comparable average performances.

4.4. Worst case behavior. Finally we report in figure 4.5 the worst case ratios encountered over the 500 experiments for the 3/2 and $\sqrt{3}$-ALG for the two previous distributions to compare them to their theoretical guarantees. Off course these results are only indicative: looking back at the performance ratio distributions of the algorithms, the worst cases have a small probability of appearance, which make the reported results quite sensitive. Surprisingly for the "average" distribution $\mathcal{N}(0.5, 0.5)$ of the time functions, the worst ratio of the latter algorithm does not exceed 1.3, which is quite far from its theoretical bound of $\sqrt{3}$, while the 3/2-algorithm performs worse, getting up to 1.4.

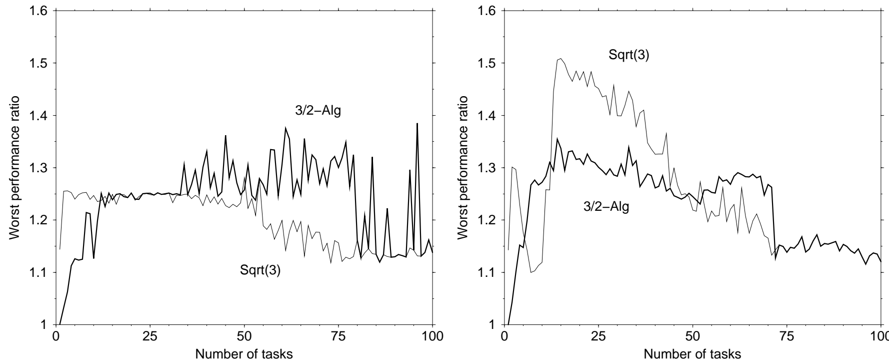

FIG. 4.5. *Worst case performance ratio of the algorithm compared to $\sqrt{3}$-Alg for a gaussian distribution $\mathcal{N}(0.5, 0.5)$ (left) and for perfect parallel tasks (right) on 32 processors with 20 tasks.*

In contrast for perfect parallel tasks the worst case ratio of the 3/2-ALG is only slightly modified, while for the $\sqrt{3}$-ALG it becomes greater than 1.5 for instances around 20 tasks. Again, this range of number of tasks is the most important in practise.

5. 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 $\frac{3}{2} + \varepsilon$. 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 their intrinsic hierarchical behavior which should help in developing good scheduling algorithms for cluster computing. This issue is under investigation.

REFERENCES

[1] R. Baker, E.G. Coffman, and R.L. Rivest. Orthogonal packings in two dimensions. *SIAM Journal on Computing*, 9(4):846–855, 1980.

[2] R. Brent. *The Parallel Evaluation of Arithmetic Expressions in Logarithmic Time*, pages 83–102. Academic Press, New York, 1973.

[3] E.G. Coffman, M.R. Garey, D.S. Johnson, and R.E. Tarjan. Performance bounds for level-oriented two-dimensional packing algorithms. *SIAM Journal on Computing*, 9(4):808–826, 1980.

[4] J. Du and J.Y-T. Leung. Complexity of scheduling parallel tasks systems. *SIAM Journal on Discrete Mathematics*, 2(4):473–487, November 1989.

[5] M.R. Garey and D.S. Johnson. *Computers and intractability: A guide to the theory of NP-completeness*. W.H. Freeman, New York, 1979.

[6] A. Gerasoulis and T. Yang. PYRROS: static scheduling and code generation for message passing multiprocessors. In *Proceedings of the 6th ACM International Conference on Supercomputing*, pages 428–437. ACM, July 1992.

[7] R.L. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics*, 17(2):416–429, March 1969.

[8] C. Hanen and A. Munier. An approximation algorithm for scheduling dependant tasks on m processors for small communication delays. In *Proceedings of IEEE Symposium on Emerging Technologies and Factory Automation*, volume 1, pages 167–189. IEEE, October 1995.

[9] D.S. Hochbaum and D.B. Shmoys. Using dual approximation algorithms for scheduling problems: theoretical and practical results. *Journal of the ACM*, 34:144–162, 1987.

[10] J. Hoogeveen, J.-K. Lenstra, and B. Veltman. Three, four, five, six, or the complexity of scheduling with communication delays. *Operations Research Letters*, 16:129–137, 1994.

[11] K. Jansen and L. Porkolab. Linear time approximation schemes for scheduling problems. In *10th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 99*, pages 490–498, 1999.

[12] C. Kenyon and E. Remila. Approximate strip packing. In *Proc. of 37th FOCS*, pages 31–36, 1996.

[13] W. T. Ludwig. *Algorithms for scheduling malleable and nonmalleable parallel tasks*. PhD thesis, University of Wisconsin - Madison, Department of Computer Sciences, 1995.

[14] W.T. Ludwig and P. Tiwari. Scheduling malleable and nonmalleable parallel tasks. In *Proceedings of the Fifth Annual ACM-SIAM Symposium on Discrete Algorithms*. ACM-SIAM, 1994.

[15] S. Martello and P. Toth. *Knapsack problems: algorithms and computer implementations*. Wiley, New York, 1990.

[16] G. Mounie, C. Rapine, and D. Trystram. Efficient approximation algorithms for scheduling malleable tasks. In *SPAA'99, Eleventh ACM Symposium on Parallel Algorithms and Architectures*, 1999.

[17] M.A. Palis, J-C. Liou, and D.S.L. Wei. Task Clustering and Scheduling for Distributed Memory Parallel Architectures. *IEEE Transactions on Parallel and Distributed Systems*, 7(1):46–55, 1996.

[18] C. Papadimitriou and M. Yannakakis. Towards an architecture-independent analysis of parallel algorithms. *SIAM Journal on Computing*, 19(2):322–328, 1990.

[19] C.H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

[20] G. N. S. Prasanna and B. R. Musicus. Generalised multiprocessor scheduling using optimal control. In *3rd Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 216–228. ACM, 1991.

[21] V.J. Rayward-Smith. UET scheduling with unit interprocessor communication delays. *Discrete Applied Mathematics*, 18:55–71, 1987.

[22] V. Sarkar. *Partitioning and Scheduling Parallel Programs for Multiprocessors.* Pitman, 1989.

[23] A. Steinberg. A strip-packing algorithm with absolute performance bound 2. *SIAM Journal on Computing*, 26(2):401–409, 1997.

[24] J. Turek, J. Wolf, and P. Yu. Approximate algorithms for scheduling parallelizable tasks. In *4th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 323–332, 1992.