Scheduling Monotone Moldable Jobs in Linear Time*

Klaus Jansen  Felix Land

November 25, 2021

Abstract

A moldable job is a job that can be executed on an arbitrary number of processors, and whose processing time depends on the number of processors allotted to it. A moldable job is monotone if its work doesn’t decrease for an increasing number of allotted processors. We consider the problem of scheduling monotone moldable jobs to minimize the makespan.

We argue that for certain compact input encodings a polynomial algorithm has a running time polynomial in $n$ and $\log m$, where $n$ is the number of jobs and $m$ is the number of machines. We describe how monotony of jobs can be used to counteract the increased problem complexity that arises from compact encodings, and give tight bounds on the approximability of the problem with compact encoding: it is \text{NP}-hard to solve optimally, but admits a PTAS.

The main focus of this work are efficient approximation algorithms. We describe different techniques to exploit the monotony of the jobs for better running times, and present a $(\frac{3}{2} + \varepsilon)$-approximate algorithm whose running time is polynomial in $\log m$ and $\frac{1}{\varepsilon}$, and only linear in the number $n$ of jobs.

1 Introduction

In classical scheduling models, the input consists of a description of the available processors, and a set of jobs with associated processing times. Each processor can process one job at any point in time. Additional constraints may be part of the model. One way to model complex, parallelizable tasks are moldable jobs, which have a variable parallelizability $[4]$. Formally, we are given a set $J$ consisting of $n$ jobs and a number $m$ of processors. The processing time $t_j(1)$ on one processor is given for each job $j$, as well as the speedup $s_j(k)$ that is achieved

*Research was in part supported by German Research Foundation (DFG) project JA 612/16-1.

1Some authors use the term malleable.
when executing it on \( k > 1 \) processors. The processing time on \( k \) processors then is given as \( t_j(k) = \frac{t_j(1)}{w_j(k)} \). The goal is to produce a schedule that assigns for each job a starting time and a number of allotted processors such that the makespan, i.e. the completion time of the last job, is minimized.

Without restriction, we assume that the speedup is non-decreasing, or equivalently, the processing time is non-increasing in the number of processors. A job is called monotone if its work function \( w_j(k) = k \times t_j(k) \) is non-decreasing. This is a reasonable assumption, since an increased number of processors requires more communication. Monotony helps when designing algorithms \([1, 20, 22]\). Sometimes even stronger assumptions are made, e.g. that the speedup functions are concave \([24, 23, 15]\).

Since the problems considered here are \( \text{NP}-\text{hard} \), we will discuss approximation algorithms. An algorithm for a minimization problem is \( c \)-approximate if it produces a solution of value at most \( c \cdot \text{OPT}(I) \) for each instance \( I \). The number \( c \geq 1 \) is called its approximation guarantee.

We pay extra attention to the encoding length \( \|I\| \) in dependence of the number \( m \) of processors. The running time of most algorithms is polynomial in \( m \) \([1, 25, 12, 13, 22, 14]\). Many authors expect that the values \( t_j(k) \), \( k \in \{1, \ldots, m\} \) are explicitly given as a list, such that \( m = O(\|I\|) \). Under this assumption, these algorithms’ running time is polynomial in the input size. On the other hand, more compact encodings are conceivable in many cases. Sometimes it is assumed that the processing time function is of a special form, e.g. linear \([7]\) or a power function \([19]\), which can be described with a constant number of coefficients. Since the number of processors is encoded in \( \log m \) bits, the aforementioned algorithms can have a running time that is exponential in the input length when compact encoding is used.

It is our main goal to develop fully polynomial algorithms for instances with compact input encoding, i.e. algorithms with a running time polynomial in \( \log m \). Such algorithms will outperform algorithms whose running time is polynomial in \( m \) for large values of \( m \) (super-polynomial in the input size). Only few known algorithms are fully polynomial in this sense \([20, 23, 9]\). Since we do not want to stipulate a certain form of speedup functions, we assume that the running times \( t_j(k) \) can be accessed via some oracle in constant time.

**Previous Results** It is known that finding an exact solution without monotony is \( \text{NP}-\text{hard} \) \([4]\). If the jobs are monotone, it is only known that finding an exact solution is weakly \( \text{NP}-\text{hard} \) \([10]\). Both results even hold for a constant number of processors. As a consequence, they also hold with compact input encoding. The problem complexity actually depends on the used input encoding. It is known that there is no polynomial time algorithm for scheduling of parallel jobs with approximation guarantee less than \( \frac{3}{2} \), unless \( \text{P} = \text{NP} \). This can be deduced from

---

\(^2\text{For a proof that concave speedup functions imply monotony, see [15].}\)
a reduction from the partition problem to scheduling of parallel jobs [3]. By setting $t_j(k) = t_j$ if $k \geq \text{size}(j)$ and $t_j(k) = \infty$ otherwise, we can extend the result to scheduling of moldable jobs, although the resulting work functions are not monotone. However, this reduction is polynomial only if we use a compact encoding for the resulting instance. Furthermore, if we allow algorithms to be polynomial in $m$, the produced instances can be optimally solved: since the reduction is one-to-one, we can go back to the original partition instance, solve it via dynamic programming in time $O(nm)$, and convert the solution back to the scheduling setting. Indeed, without compact encoding, the problem admits a PTAS [14].

Considering approximate algorithms, Belkhale and Banerjee [1] found a 2-approximate algorithm for scheduling monotone moldable tasks. This approximation guarantee was later matched without monotony by an algorithm due to Turek, Wolf, and Yu [25]. The running time was later improved by Ludwig and Tiwari [18]. Their algorithm for the case of monotone jobs was the first to achieve a running time polynomial in $\log m$, namely $O(n \log^2 m)$. Mounié, Rapine, and Trystram improved the approximation guarantee with monotony to $\sqrt{3} + \varepsilon \approx 1.73$, with arbitrarily small $\varepsilon > 0$, also with polylogarithmic dependence on $m$. They later presented a $(\frac{3}{2} + \varepsilon)$-approximate algorithm with running time $O(nm \log \frac{1}{\varepsilon})$ [21, 22]. A PTAS with running time polynomial in $m$ was subsequently developed that does not require monotony [14]. Finally, a $(\frac{3}{2} + \varepsilon)$-approximate algorithm with polylogarithmic dependence on $m$ that also does not assume monotone jobs was developed by Jansen [9].

**Our Contribution** We improve the understanding of scheduling monotone moldable jobs in several ways. In Section 2 we resolve the complexity of the considered problem.

**Theorem 1.** It is NP-complete to decide whether a set of monotone jobs can be scheduled with a given makespan.

We proceed to describe an extremely efficient FPTAS for the case that the number of machines is large enough in Section 3.

**Theorem 2.** There is an FPTAS for the case that $m \geq 8n\frac{n}{\varepsilon}$ with a running time of $O(n \log^2 m (\log m + \log \frac{1}{\varepsilon}))$.

In combination with the PTAS by Jansen and Thöle [14], this yields a PTAS for scheduling of monotone moldable jobs with compact encoding of running times. The algorithm by Jansen [9] achieves the same approximation guarantee in the more general case without monotony, but has a significantly worse running time. In particular, our new algorithm’s running times are polynomial in $\frac{1}{\varepsilon}$, while Jansen’s algorithm is doubly exponential in $\frac{1}{\varepsilon}$. 

3
Table 1: Running times of our \((\frac{3}{2} + \varepsilon)\)-dual algorithms.

| Algorithm | \(T(n, m, \varepsilon)\) |
|-----------|-------------------|
| Section 4.2.5 | \(O(n(\log m + n \log \varepsilon m))\) |
| Section 4.3 | \(O\left(n\left(\frac{1}{\varepsilon} \log \frac{m}{\varepsilon} + \log^3(\varepsilon m)\right) + \log n\right)\) |
| Section 4.3.3 | \(O\left(n\frac{1}{\varepsilon} \log m\left(\frac{\log m}{\varepsilon} + \log^3(\varepsilon m)\right)\right)\) |

In Section 4 we first describe the \((\frac{3}{2} + \varepsilon)\)-approximate algorithm due to Mounié, Rapine, and Trystram [22], and improve its running time to fully polynomial. We further present techniques to gradually reduce the dependence of the running time on the number \(n\) of jobs.

**Theorem 3.** For each \(T\) given in Table 1, there is a \((\frac{3}{2} + \varepsilon)\)-approximate algorithm with running time \(O(n \log^2 m + \log \frac{1}{\varepsilon} T(n, m, \varepsilon))\).

We make repeated use of a technique we call compression. It reduces the number of processors used by a job in exchange for a bounded increase in the running time. Compression allows us to approximate processor numbers for jobs that are allotted to a large number of processors. This enables the use of various rounding techniques. The intermediate solution then may use more than \(m\) processors, before the jobs are finally compressed such that they require at most \(m\) processors. This is similar to models with resource augmentation (see e.g. [2]), except that we can use additional processors only for jobs that are allotted to a large number of processors.

## 2 NP-Completeness of Monotone Moldable Job Scheduling

In this section we discuss Theorem 1. To be precise, we consider the problem of deciding whether a given instance of scheduling monotone moldable jobs can be scheduled with makespan at most \(d\), where \(d\) is also part of the input.

*Proof of Theorem 1.* We first argue that the problem is in NP by giving a non-deterministic polynomial procedure for solving the problem: first, guess the number of processors allotted to each job in an optimal schedule. Since these numbers are at most \(m\), we can guess them one bit at a time in \(n \log m\) steps. Afterwards, we guess the order in which the jobs start. This is a list of \(n\) numbers in \(\{1, \ldots, n\}\), which can be encoded in \(n \log n\) bits. Again, we can guess this encoding in \(n \log n\) steps. We now use list scheduling to schedule the jobs in this order while respecting the previously guessed processor counts. This procedure is clearly possible in polynomial time and uses \(n(\log m + \log n)\) (binary) guessing steps.

We give a reduction from 4-PARTITION to prove that our scheduling problem is strongly NP-hard. Recall that an instance of 4-PARTITION contains a set \(A =\)
\{a_1, \ldots, a_{4n}\} of natural numbers and a number \(B\), and remains \(\text{NP}\)-hard even when all numbers are strictly between \(\frac{B}{5}\) and \(\frac{B}{3}\) [6]. We construct an instance of the scheduling problem as follows. First, we assume that \(\sum_{i=1}^{4n} a_i = nB\), otherwise we output a trivial no-instance. Next, we scale the numbers such that \(a_i \geq 2\) for each \(i \in \{1, \ldots, 4n\}\). The number of machines will be \(m = n\). Now we create a job \(j_i\) for each number \(a_i\), which has processing time \(t_{ji}(k) = ma_i - k + 1\). Note that these functions are monotonically decreasing. Also \(ma_i \geq 2m > 2k\) for each \(k < m\) and therefore the work satisfies

\[
w_{ji}(k + 1) = (k + 1) \times t_{ji}(k + 1) = (k + 1)(ma_i + 1) - (k + 1)^2 = k(ma_i + 1) - k^2 + (ma_i + 1) - 2k - 1 = w_{ji}(k) + ma_i - 2k > w_{ji}(k),
\]

i.e the jobs are strictly monotone. The target makespan is \(d = nB\). It remains to show that this is indeed a reduction, i.e. that a schedule with makespan \(nB\) exists if and only if the instance of 4-PARTITION is a yes-instance.

First assume that there is a schedule with makespan \(d\). The total work of all jobs is at least \(\sum_{i=1}^{4n} w_{ji}(1) = \sum_{i=1}^{4n} ma_i = md\). Due to the strict monotony, our schedule must allot exactly one processor to each job, and all machines have load \(d\), see Fig. 1 for an example of such a schedule. The numbers corresponding to the jobs on one machine sum up to \(B\), and because they are strictly between \(\frac{B}{5}\) and \(\frac{B}{3}\), there are exactly four such numbers. Therefore, there is a solution to the instance of 4-PARTITION.

On the other hand, if the instance of 4-PARTITION is a yes-instance, a schedule as depicted in Fig. 1 is easily constructed from a solution.

This shows that scheduling monotone jobs is \(\text{NP}\)-hard in the strong sense. Furthermore, using a complexity result for 4-PARTITION [11], there is no algorithm that solves this problem exactly in time \(2^{o(n)} \times \|I\|^{O(1)}\), unless the Exponential Time Hypothesis fails.

![Figure 1: Structure of a schedule with makespan \(nB\)](image-url)
3 An FPTAS for Large Machine Counts

In this section, we present a fully polynomial approximation scheme (FPTAS) for the case that \( m \geq 8n^\frac{\varepsilon}{\varepsilon} \), as stated in Theorem 2. An FPTAS finds a \((1 + \varepsilon)\)-approximate solution in time polynomial in the input length and \( \frac{1}{\varepsilon} \) for each \( \varepsilon \in (0, 1] \). The case where \( m \) is much larger than \( n \) is the most interesting case in our setting with compact input encoding, because otherwise \( m \) is polynomial in the input. Furthermore, this allows us to focus on the case \( m < 8n^\frac{\varepsilon}{\varepsilon} \) in the following chapters.

The algorithm itself is a dual approximate algorithm [8]. A \( c \)-dual approximate algorithm accepts a number \( d \) in addition to the instance as input. It will output a solution with makespan at most \( cd \), provided that a solution with makespan \( d \) exists. Otherwise, it may reject the instance. It is well known that a \( c \)-approximate dual algorithm with running time \( T(n, m) \) can be turned into a \((c + \varepsilon)\)-approximate algorithm with running time \( O(T'(n, m) + \log \frac{1}{\varepsilon} \times T(n, m)) \), where \( T'(n, m) \) is the running time of an estimation algorithm with arbitrary but constant estimation ratio.

An estimation algorithm with estimation ratio \( \rho \) computes a value \( \omega \) that estimates the minimum makespan within a factor of \( \rho \), i.e. \( \omega \leq \text{OPT} \leq \rho \omega \). Here, we use an algorithm due to Ludwig and Tiwari [18] with running time \( T'(n, m) = O(n \log^2 m) \). Although they do not explicitly state this, their algorithm can be trivially turned into one with estimation ratio 2:

Their algorithm computes an allotment \( a \) which allots to each job \( j \in J \) a number \( a_j \) of processors, and this allotment minimizes the value

\[
\omega = \min \left( \frac{1}{m} \sum_{j \in J} w_j(a_j), \max_{j \in J} t_j(a_j) \right)
\]

among all allotments. Therefore \( \omega \leq \text{OPT} \). On the other hand, the list scheduling algorithm, applied to the instance with the fixed allotment \( a \), produces a schedule of makespan at most \( 2\omega \) [5], so \( \text{OPT} \leq 2\omega \).

For our algorithm, we specify \( c = 1 + \varepsilon \), resulting in a \( 1 + 2\varepsilon \) approximation ratio. Our algorithms will frequently schedule jobs using the least number of processors such that its processing time is below a threshold \( t \). Therefore let \( \gamma_j(t) = \min\{p \in [m] \mid t_j(p) \leq t\} \), see also the work of Mounié, Rapine, and Trystram [22]. Note that \( \gamma_j(t) \) can be found in time \( O(\log m) \) by binary search.

The algorithm is extremely simple: allot \( \gamma_j((1 + \varepsilon)d) \) processors to each job \( j \) and schedule them simultaneously. If this schedule requires more than \( m \) machines, reject.

Clearly, the running time for the dual approximate algorithm is \( O(n \log m) \). The final algorithm therefore requires \( O(n \log m (\log m + \log \frac{1}{\varepsilon})) \) time.
3.1 Analysis

It remains to show that the algorithm is indeed \((1 + \varepsilon)\)-dual approximate. The produced schedule clearly has makespan at most \((1 + \varepsilon)d\) by the definition of \(\gamma_j((1 + \varepsilon)d)\). To prove that the algorithm only rejects if there is no schedule with makespan \(d\), we will argue that the produced schedule requires at most \(m\) processors, i.e. \(\sum_{j \in J} \gamma_j((1 + \varepsilon)d) \leq m\), provided that \(d \geq d^*\), where \(d^*\) is the optimal makespan.

To this end, we consider the same algorithm with a more complex allotment rule, which uses compression of jobs, our main technique for exploiting the monotony of the work functions. Compression reduces the number of processors allotted to a job in exchange for a bounded increase of its processing time.

**Lemma 4.** If \(j\) is a job that uses \(b \geq \frac{1}{\rho}\) machines in some schedule, where \(\rho \in (0, 1/4]\), then we can free \(\lceil b\rho \rceil\) machines and the schedule length increases by at most \(4\rho t_j(b)\).

We call the value \(\rho\) the compression factor.

**Proof.** Formally, the statement of the lemma is \(t_j(\lfloor b(1 - \rho) \rfloor) \leq (1 + 4\rho) t_j(b)\). For the proof we set \(b' = \lfloor b(1 - 2\rho) \rfloor \leq b\). Since \(b \geq \frac{1}{\rho}\) we have \(b\rho \geq 1\) and thus \(b' \leq \lfloor b(1 - \rho) \rfloor\). This implies \(t_j(\lfloor b(1 - \rho) \rfloor) \leq t_j(b')\) Because our jobs are monotonic we have

\[t_j(b') \times b' = w_j(b') \leq w_j(b) = t_j(b) \times b. \quad (3)\]

Hence (and because \(1 - 2\rho \geq 1/2\)) it follows that

\[t_j(b') \leq t_j(b) \times \frac{b}{b'} \leq t_j(b) \times \frac{b}{b(1 - 2\rho)} = t_j(b) \times \left(\frac{1 - 2\rho}{1 - 2\rho} + \frac{2\rho}{1 - 2\rho}\right) \leq (1 + 4\rho) t_j(b) \quad (4)\]

and the lemma follows. \(\square\)

Our second allotment rule has two steps.

(i) Allot \(a_j = \gamma_j(d)\) processors to each job \(j\).

(ii) Compress each job that is allotted to at least \(\frac{4}{\varepsilon}\) processors with a factor of \(\rho = \frac{5}{4}\).

Note that \(\rho \leq \frac{1}{4}\) because we assumed \(\varepsilon \leq 1\). According to Lemma 4, each job has processing time at most \((1 + \varepsilon)d\) with this allotment rule.
We claim that the resulting schedule requires at most $m$ processors. Assume that this is not the case after the first step, i.e. $\sum_{j \in J} a_j > m$, otherwise the statement clearly holds. Then the number of required processors is still bounded:

**Lemma 5.** If $d \geq d^*$ we have $\sum_{j \in J} a_j < m + n$.

**Proof.** Let $J' = \{ j \in J \mid a_j = 1 \}$ and assume the statement holds if we remove the jobs in $J'$ and their allotted machines, i.e. $\sum_{j \in J \setminus J'} a_j < (m - |J'|) + (n - |J'|)$. Then $\sum_{j \in J} a_j = \sum_{j \in J \setminus J'} a_j + |J'| < m + n - |J'| \leq m + n$.

It is therefore sufficient to show the statement for jobs with $a_j > 1$. Assume that $d \geq d^*$. Then there is a schedule with makespan at most $d^*$. For each job $j$ let $a^*_j$ be the number of allotted processors in this schedule. Then $a_j \leq a^*_j$. Using the monotony of the work function, we have

$$
\left( \sum_{j \in J} (a_j - 1) \right) d = \sum_{j \in J} (a_j - 1) d
\leq \sum_{j \in J} (a_j - 1) t_j (a_j - 1)
= \sum_{j \in J} w_j (a_j - 1)
\leq \sum_{j \in J} w_j (a_j) \leq \sum_{j \in J} w_j (a^*_j) \leq md.
$$

Therefore $(\sum_{j \in J} a_j) - n = \sum_{j \in J} (a_j - 1) < m$, proving the lemma. □

Now partition the jobs into narrow and wide jobs, $J = J_N \cup J_W$. The wide jobs are those that are compressed in the second step, i.e. $J_W = \{ j \in J \mid \gamma_j(d) \geq \frac{1}{p} \}$ and $J_N = J \setminus J_W$. Let $\alpha = \sum_{j \in J_w} a_j$ and $\beta = \sum_{j \in J_N} a_j$.

In the second step, at least $\rho \alpha$ processors are freed. By definition of the narrow jobs we have $\beta \leq n \frac{1}{\rho} = 4 \frac{n}{\varepsilon} \leq \frac{m}{2}$. Since we assumed that $\alpha + \beta > m$ we have $\alpha > \frac{m}{2}$. Therefore $\rho \alpha > \rho \frac{m}{2} \geq n$. According to Lemma 5 we also have $\alpha + \beta \leq m + n$. It follows that $(1 - \rho) \alpha + \beta \leq m$, proving our claim.

To prove the claim about the first allotment rule, we note that it cannot use more processors for any job, because it picks the minimum number of allotted processors when we target a makespan of $(1 + \varepsilon)d$. Therefore, our algorithm is $(1 + \varepsilon)$-dual approximate, proving Theorem 2.

### 3.2 A PTAS for the General Case

For the general case we can still achieve a PTAS. When $m \geq 8 \frac{n}{\varepsilon}$, simply use the previously described algorithm. Otherwise apply the algorithm by Jansen and Thöle [14]. It is $(1 + \varepsilon)$-approximate and has a running time polynomial in $n$ and $m$ (but exponential in $\frac{1}{\varepsilon}$). Since we use this algorithm only in the case that $m < 8 \frac{n}{\varepsilon}$, the running time is polynomial even when a compact input encoding is used for the processing times.
4 A Linear \((\frac{3}{2} + \varepsilon)\)-Approximation

Unfortunately, the running time of the PTAS used in the last section is rather prohibitive. We therefore want to develop more efficient algorithms. The algorithms we present in this section are modifications of the algorithm due to Mounié, Rapine, and Trystram [22] which has running time \(O(nm)\). We aim for a fully polynomial algorithm with a running time that depends only linearly on the number \(n\) of jobs.

We achieve this goal with several modification to the original algorithm. Before explaining our enhancements, we describe the original algorithm by Mounié, Rapine, and Trystram. In section 4.2 we show how to modify the algorithm such that the running time is logarithmic in the number \(m\) of of machines. With these modifications, the dependence of the running time on \(n\) actually increases. We can however improve upon the general idea in section 4.3.

4.1 The Original Algorithm

The algorithm is \(\frac{3}{2}\)-dual approximate. Let \(d\) be the target makespan. The algorithm is based on the observation that all jobs with a running time larger than \(d\) in a feasible schedule of makespan \(d\) can be executed in parallel.

**Removing the Small Jobs** We partition the jobs into small and big jobs, \(J = J_S(d) \cup J_B(d)\). Small jobs are jobs that complete in time \(\frac{d}{2}\) on one machine, i.e. jobs \(j \in J\) with \(t_j(1) \leq \frac{d}{2}\). All other jobs will be denoted as big jobs. We remove the small jobs from the instance, they will be re-added in a greedy manner at the end of the algorithm. Let \(W_S(d) = \sum_{j \in J_S(d)} t_j(1)\) be the total work of these jobs.

**Finding a Preliminary Schedule** In this step, a schedule for the big jobs is constructed by placing them in two shelves, shelf \(S_1\) with processing time \(d\) and shelf \(S_2\) with processing time \(\frac{d}{2}\). The shelves are scheduled after each other for a total makespan of \(\frac{3}{2}d\). Since a feasible schedule of this type may not exist, we allow the second shelf to use more than \(m\) processors, see Fig. 2. We call such a schedule a two-shelf-schedule. A two-shelf-schedule can be found by solving a knapsack problem where shelf \(S_1\) uses at most \(m\) processors and the profit of each job is the amount of work saved when it is scheduled in shelf \(S_1\) instead of shelf \(S_2\), i.e. \(v_j(d) = w_j(\gamma_j(\frac{d}{2})) - w_j(\gamma_j(d))\). Note that the monotony implies that \(v_j(d) \geq 0\). If \(\gamma_j(d)\) is undefined for any job \(j\), i.e. \(t_j(m) > d\), then we can safely reject \(d\). Each job \(j \in J_B(d)\) for which \(\gamma_j(\frac{d}{2})\) is undefined, i.e. \(t_j(m) > \frac{d}{2}\), must be scheduled in \(S_1\). Those jobs can be easily handled by removing them from the knapsack problem and reducing the capacity accordingly. In order to keep the notation simple, we will however assume that no such jobs exist. We denote the
Figure 2: Example of an infeasible two-shelf-schedule

The knapsack problem

$$\max_{J' \subseteq J_B(d)} \sum_{j \in J'} v_j(d) \quad \text{s.t.} \quad \sum_{j \in J'} \gamma_j(d) \leq m$$

(6)

by KP($J_B(d), m, d$) and the profit of an optimal solution by OPT$_{KP}(J_B(d), m, d)$. Solving the knapsack problem requires time $O(nm)$ with a standard dynamic programming approach. Calculating the required values of $\gamma_j$ can be done beforehand in time $O(n \log m)$.

Let $J'$ be a solution to this knapsack problem. Then we can create a two-shelf-schedule by placing the jobs $J'$ in $S_1$ and the jobs $J_B(d) \setminus J'$ in $S_2$. The work of this schedule is

$$W(J', d) = \sum_{j \in J'} w_j(\gamma_j(d)) + \sum_{j \in J_B(d) \setminus J'} w_j(\gamma_j(d))$$

$$= \sum_{j \in J_B(d)} w_j(\gamma_j(d)) - \sum_{j \in J'} v_j(d).$$

(7)

At this point, we will reject $d$ if $W(J', d)$ is found to be larger than $md - W_S(d)$. The correctness of this step is shown in the following lemma.

Lemma 6 ([22]). If there is a schedule for all jobs with makespan $d$, then there is a solution $J'$ to KP($J_B(d), m, d$) with $W(J', d) \leq md - W_S(d)$.

Proof. We only consider the big jobs in the feasible schedule, and their total work $W_B$ is bounded from above by $md - W_S(d)$, because $W_S(d)$ is a lower bound on the work of the small jobs. The jobs with processing time larger than $\frac{d}{2}$ induce a feasible solution $J'$ to the knapsack problem. Then

$$W(J', d) = \sum_{j \in J_B(d)} w_j(\gamma_j(d)) - \sum_{j \in J'} v_j$$

$$= \sum_{j \in J'} w_j(\gamma_j(d)) + \sum_{j \in J_B(d) \setminus J'} w_j(\gamma_j(d))$$

$$\leq W_B \leq md - W_S(d),$$

(8)
Figure 3: An example of a schedule after applying the transformation rules

so \( J' \) is the claimed solution. \( \square \)

Otherwise, the following step will transform the two-shelf-schedule into a feasible schedule for all jobs.

**Lemma 7** ([22]). Let \( J' \subseteq J_B(d) \) be a solution of \( \text{KP}(J_B(d), m, d) \) with \( W(J', d) \leq md - W_S(d) \). Then we can find a schedule for all jobs in \( J \) with makespan \( \frac{3}{2}d \) in time \( O(n \log n) \).

### 4.1.1 Proof of Lemma 7

**Obtaining a Feasible Schedule** We exhaustively apply three transformation rules that move some jobs to a new shelf \( S_0 \). The jobs in \( S_0 \) will be scheduled concurrent to \( S_1 \) and \( S_2 \) in \( \frac{3}{2}d \) time units, creating a three-shelf-schedule. For an example, see Fig. 3.

(i) If job \( j \in S_1 \) has processing time \( t_j(\gamma_j(d)) \leq \frac{3}{4}d \) and \( \gamma_j(d) > 1 \), then allocate \( j \) to \( \gamma_j(d) - 1 \) processors in \( S_0 \).

(ii) If two jobs \( j, j' \in S_1 \) each have processing time at most \( \frac{3}{4}d \) and \( \gamma_j(d) = \gamma_{j'}(d) = 1 \), then schedule \( j \) and \( j' \) sequentially on one processor in \( S_0 \). If there is only one job \( j \) with \( t_j(\gamma_j(d)) \leq \frac{3}{4}d \) and \( \gamma_j(d) = 1 \), a special case applies [21]: if also a job \( j' \) in \( S_1 \) with \( t_{j'}(\gamma_{j'}(d)) > \frac{3}{4}d \) and \( t_j(\gamma_j(d)) + t_{j'}(\gamma_{j'}(d)) \leq \frac{3}{2}d \) exists, \( j \) is scheduled on top of \( j' \) in \( S_0 \). Conceptually, \( j' \) is split into two parts. One is allocated to one processor in \( S_0 \), while the remaining part remains in \( S_1 \).

(iii) Let \( p_0 \) and \( p_1 \) denote the number of processors required by \( S_0 \) and \( S_1 \), respectively. Let \( q = m - (p_0 + p_1) \). If there is \( j \in S_2 \) with \( t_j(q) \leq \frac{3}{2}d \), allocate \( j \) with \( p = \gamma_j(\frac{3}{2}d) \leq q \) processors to \( S_0 \) if \( t_j(p) > d \) and to \( S_1 \) otherwise.

The resulting schedule is indeed feasible.

**Lemma 8** ([22]). When none of the transformation rules are applicable, the schedule uses at most \( m \) processors.
Since the transformation rules do not increase the number of processors allotted to any job, the work does not increase. In particular, if the two-shelf-schedule has total work at most \( md - W_S(d) \), then the resulting schedule also has total work at most \( md - W_S(d) \).

Note that any job’s allotment changes twice only if it is moved from \( S_2 \) to \( S_1 \) by rule (iii) and then from \( S_1 \) to \( S_0 \) by rules (i) or (ii). Therefore, the transformations can be exhaustively applied to a two-shelf-schedule in time \( O(n \log n) \): Scan the shelf \( S_1 \) and classify each job into one of three categories

1. If \( t_j(\gamma_j(d)) \leq \frac{3}{4}d \) and \( \gamma_j(d) > 1 \),
2. If \( t_j(\gamma_j(d)) \leq \frac{3}{4}d \) and \( \gamma_j(d) = 1 \), and
3. If \( t_j(\gamma_j(d)) > \frac{3}{4}d \).

Jobs of the first category can immediately be moved to \( S_0 \) according to rule (i). Jobs of the second category are stored and moved to \( S_0 \) in pairs as described in rule (ii). This may leave one unpaired job at the end. Jobs of the third category are stored in a min-heap with the processing time \( t_j(\gamma_j(d)) \) as key. This heap can be used to pair unpaired jobs from the second category in accordance to the special case of rule (ii). Scanning shelf \( S_1 \) in this fashion can be done in time \( O(n \log n) \). Afterwards, we process shelf \( S_2 \) and check if rule (iii) applies. If it does, and the job should be moved to \( S_1 \), we immediately categorize it and apply rules (i) and (ii) to it. This requires time \( O(\log n) \) per job in \( S_2 \). We also need to compute some of the values \( \gamma_j(d) \), \( \gamma_j(\frac{d}{2}) \), and \( \gamma_j(\frac{3}{2}d) \) for each job \( j \in J_B(d) \), resulting in an overall running time of \( O(n(\log n + \log m)) \).

**Re-adding the Small Jobs** We modify the schedule such that the free time on each processor is adjacent: jobs in \( S_0 \) and \( S_1 \) start as early as possible, while the jobs in \( S_2 \) finish at time \( \frac{3}{2}d \). We now allocate the small jobs to the free time intervals with a next-fit approach\(^3\). The small jobs are processed in an arbitrary order. The current job \( j \) gets assigned to the next machine with load at most \( \frac{3}{4}d - t_j(1) \).

**Lemma 9.** Using a next-fit algorithm, all small jobs can be integrated into a three-shelf-schedule, provided its total work is at most \( md - W_S(d) \).

**Proof.** Assume for the sake of contradiction that a small job cannot be added. Then all machines were discarded for not fitting some job, i.e. their load is larger than \( \frac{3}{4}d - t_j(1) \) for a small job \( j \). Since \( j \) is small, i.e. \( t_j(1) \leq \frac{d}{2} \), those machines load was larger than \( \frac{3}{4}d - \frac{1}{2}d = \frac{1}{4}d \). Thus the total load of all machines is larger than \( md \), a contradiction. \( \square \)

\(^3\)in the work of Mounié, Rapine, and Trystram [22], a different allocation rule is used, but the proof is similar.
This can be implemented to run in linear time. We first group adjacent processors that have the same amount of free time. Since the amount of free time only changes when a different job is scheduled on a processor, there can be at most $O(n)$ many groups. The groups are processed in arbitrary order. If the free time of a group is at least $t_j(1)$, where $j$ is the current job, the group is split into two groups, the first containing one processor, and the second containing the other processors. The current job is then added to the processor in the first group, and its free is updated. Otherwise the whole group is discarded. This is possible in time linear in the number of small jobs and groups, achieving an overall running time of $O(n)$.

4.2 Knapsack with Compressible Items

The dominating part of the algorithm is the solution of the knapsack problem. One might be tempted to use one of the known FPTASs for the knapsack problem to find a solution with slightly suboptimal profit. However, the profit of the knapsack problem can be much larger than the work of the schedule, such that a small decrease of the profit can increase the work of the schedule by a much larger factor. Instead, we treat the processor counts approximately, despite the fact that the the available number of processors imposes a hard constraint. We will employ compression to compensate for an increased processor usage of the solution.

For a cleaner notation, we define the knapsack problem with compressible items: an instance of this problem is a tuple $(I, I^c, C, \rho)$, where $I$ is a set of items. An item $i \in I$ has size $s(i)$ and profit $p(i)$. The items $I^c \subseteq I$ can be compressed with factor $\rho$, and $C$ denotes the capacity. A feasible solution to this instance is a set $I' \subseteq I$ such that

$$\sum_{i \in I' \cap I^c} (1 - \rho) s(i) + \sum_{i \in I' \setminus I^c} s(i) \leq C.$$  \hspace{1cm} (9)

We denote the maximum profit of the instance by $\text{OPT}(I, I^c, C, \rho)$.

4.2.1 Simple Application to the Scheduling Problem

Identifying the jobs as items is straightforward: Let $J^C = \{ j \in J_B(d) \mid \gamma_j(d) \geq \frac{1}{\rho} \}$ be the compressible jobs, where the compression factor $\rho$ will be defined later depending on the desired accuracy $\varepsilon$. Set the knapsack sizes as $s(j) = \gamma_j(d)$ and the profits as $p(j) = v_j(d)$ for $j \in J_B(d)$. If we have a solution $J'$ to the knapsack problem $(J_B(d), J^C, m, \rho)$, we can compress the jobs in $J' \cap J^C$ such that $J'$ fits on $m$ processors. Their processing time increases by a factor of at most $(1 + 4\rho)$, so $J'$ is a feasible solution to KP$(J_B(d), m, (1 + 4\rho)d)$.

To make up for the increased makespan, we use the following corollary to Lemma 7 with $d' = (1 + 4\rho)d$. 

13
Corollary 10. Let \( d' \geq d \), and \( J' \subseteq J_B(d) \) be a feasible solution of the knapsack problem \( \text{KP}(J_B(d), m, d') \) with \( W(J', d) \leq md' - W_S(d) \). Then we can, in time \( O(n \log n) \), find a schedule with makespan at most \( \frac{3}{2}d' \).

Proof. Let \( J'' = J' \cap J_B(d') \). Then \( J'' \) is a feasible solution of \( \text{KP}(J_B(d'), m, d') \). Define \( \Delta W = W_S(d') - W_S(d) \). Note that, when targeting makespan \( \frac{3}{2}d' \), each job in \( J \setminus J'' \) will be scheduled on one processor in shelf 1 and shelf 2, since these are exactly the jobs that have \( t_j(1) \leq \frac{d'}{2} \). Therefore, they can be moved between shelves without affecting the work of the schedule, implying \( W(J'', d') = W(J', d') \). Using also the fact that the work functions are monotone, we obtain

\[
W(J'', d') = W(J', d') = \sum_{j \in J'} w_j(\gamma_j(d')) + \sum_{j \in J_B(d') \setminus J'} \frac{w_j(\gamma_j(d'))}{2} \\
\leq \sum_{j \in J'} w_j(\gamma_j(d)) + \sum_{j \in J_B(d') \setminus J'} \frac{w_j(\gamma_j(d))}{2} \\
= \sum_{j \in J'} w_j(\gamma_j(d)) + \sum_{j \in J_B(d) \setminus J'} \frac{w_j(\gamma_j(d))}{2} \\
\quad - \sum_{j \in J_B(d) \setminus J_B(d')} w_j(\gamma_j(d)) \\
= W(J', d) - \Delta W \\
\leq md' - W_S(d) - \Delta W \\
= md' - W_S(d').
\]

Applying Lemma 7 to \( J'' \) yields the desired schedule. \( \square \)

To obtain a \( \left( \frac{3}{2} + \varepsilon \right) \)-dual approximate algorithm, we set \( \rho = \frac{1}{6}\varepsilon \). Concerning the prerequisites of Corollary 10, it is sufficient if the profit of \( J' \) is at least \( \text{OPT}(I, 0, C, 0) = \text{OPT_{KP}}(J_B(d), m, d) \), because then \( W(J', d) \leq md - W_S(d) \), unless no schedule with makespan \( d \) exists.

**Algorithm 1:** Scheduling of monotone moldable jobs using knapsack with compressible items

1. \( \rho \leftarrow \frac{1}{6}\varepsilon \), \( d' \leftarrow (1 + 4\rho)d \), \( J^C \leftarrow \{ j \in J_B(d) \mid \gamma_j(d) \geq \frac{1}{\rho} \} \)
2. for \( j \in J_B(d) \) do
   3. Precompute \( \gamma_j(\frac{d}{2}), \gamma_j(d), \gamma_j(\frac{d'}{2}), \gamma_j(d'), \gamma_j(\frac{3d'}{2}) \)
4. Find solution \( J' \) to \( (J_B(d), J^C, m, \rho) \)
5. Apply Lemma 7 to \( J' \cap J_B(d') \), obtain schedule for jobs \( J \) with makespan \( \frac{3}{2}d' \)
6. if schedule is infeasible then reject \( d \)
7. else return the schedule
Note that the algorithm schedules the jobs with $\gamma_j(d')$ processors. Compression was only used to show that shelf 1 has at most $m$ processors with these processor counts.

How one can find a solution to is discussed in the rest of this section.

### 4.2.2 Separating Compressible and Incompressible Items

Let $I = I_1 \cup I_2$ be a partition of the items. Consider an optimal solution $I^* \subseteq I$ to the knapsack problem $(I, I^c, C, \rho)$, let $\alpha \geq \sum_{i \in I \cap I_1} s(i)$ be the space available for items in $I_1$ in the solution, and $\beta \geq \sum_{i \in I \cap I_2} s(i)$ the space available for the other items. Then we can solve the knapsack problems for $I_1$ and $I_2$ separately.

**Lemma 11.** $\text{OPT}(I, I^c, C, \rho) \leq \text{OPT}(I_1, I^c \cap I_1, \alpha, \rho) + \text{OPT}(I_2, I^c \cap I_2, \beta, \rho)$, and equality holds if $\alpha + \beta = C$.

**Proof.** Let $I_1^* \subseteq I_1$ and $I_2^* \subseteq I_2$ be optimal solutions of $(I_1, I^c \cap I_1, \alpha, \rho)$ and $(I_2, I^c \cap I_2, \beta, \rho)$, respectively. Assume for the sake of contradiction that

\[
\text{OPT}(I, I^c, C, \rho) > \text{OPT}(I_1, I^c \cap I_1, \alpha, \rho) + \text{OPT}(I_2, I^c \cap I_2, \beta, \rho). \tag{11}
\]

Recall that $I^*$ is our optimal solution to $(I, I^c, C, \rho)$. Then

\[
\sum_{i \in I \cap I_1} p(i) + \sum_{i \in I \cap I_2} p(i) = \sum_{i \in I^*} v_j(d) = \text{OPT}(I, I^c, C, \rho) \tag{12}
\]

\[
> \text{OPT}(I_1, I^c \cap I_1, \alpha, \rho) + \text{OPT}(I_2, I^c \cap I_2, \beta, \rho),
\]

therefore we either have $\sum_{i \in I \cap I_1} b(i) > \text{OPT}(I_1, I^c \cap I_1, \alpha, \rho)$ and $\sum_{i \in I \cap I_2} s(i) \leq \alpha$, or $\sum_{i \in I \cap I_2} b(i) > \text{OPT}(I_2, I^c \cap I_2, \beta, \rho)$ and $\sum_{i \in I \cap I_2} s(i) \leq \beta$, contradicting the optimality of $I_1^*$ and $I_2^*$.

If $\alpha + \beta = C$, then the set $I_1^* \cup I_1^*$ is a feasible solution of $(I, I^c, C, \rho)$, hence

\[
\text{OPT}(I, I^c, C, \rho) \geq \text{OPT}(I_1, I^c \cap I_1, \alpha, \rho) + \text{OPT}(I_2, I^c \cap I_2, \beta, \rho) \tag{13}
\]

and equality must hold. \[\square\]

In our case, we partition $I = I^c \cup (I \setminus I^c)$, i.e. $\alpha$ is the space available for compressible items. To utilize this lemma, we need to know values for $\alpha$ and $\beta$. Let $\beta_{\text{max}}$ be an upper bound on the space used by incompressible jobs. We can enumerate all $\beta_{\text{max}} + 1$ possible values of $\beta \leq \beta_{\text{max}}$, set $\alpha = C - \beta$, and pick the best obtained solution. A standard dynamic programming algorithm can be used to solve the instances $(I \setminus I^c, \emptyset, \beta, 0)$ in time $O(n \beta_{\text{max}})$ each.

While the problem for the compressible items could be solved the same way, we are looking for an algorithm that is polynomial in $\log C$. We propose several
techniques, exploiting the compressibility of the items. First, we treat the size of compressible items approximately. Second, we further bound the number of knapsack problems to solve by $O\left(\frac{\log C}{\rho}\right)$ by using an approximate, possibly larger value for $\alpha$. We also solve all knapsack problems in one pass.

4.2.3 Solving the Knapsack Problem for Compressible Items

One way to implement the dynamic program for solving the knapsack problem exactly is given by Lawler [17]: assume there are $n_C \leq n$ compressible items $i_1, \ldots, i_{n_C}$, and the capacity is $\alpha$. A list $L$ of pairs $(p, s)$ is initialized with the single pair $(0, 0)$. In the $k$-th iteration, for each pair $(p, s)$ in $L$, a new pair $(p + p(i_k), s + s(i_k))$ is added to $L$, unless $s + s(i_k) > \alpha$. Thus, at the end of the $k$-th iteration, a pair $(p, s)$ indicates that $p$ is the highest profit that can be obtained with the items $i_1, \ldots, i_k$ and total size at most $s$. A pair $(p, s)$ is said to dominate $(p', s')$ if $p \geq p'$ and $s \leq s'$. After each modification, we remove dominated pairs from the list. The optimum value then is $\max\{p \mid (p, w) \in L\}$.

By storing additional backtracking information, an optimal solution can be found in time $O(n_C \alpha) = O(n_C m)$.

Let $\bar{n}$ be an upper bound to the number of compressible items in any solution. In our scheduling setting, such a bound is imposed by the fact that wide jobs are compressible. A common approach is to round the sizes $s(i)$ and the capacity $\alpha$ down to the next multiple of $U = \rho \left(1 - \frac{\rho}{\bar{n}}\right)\alpha$, such that all sizes in pairs stored in $L$ are multiples of $U$. This can be equivalently achieved as follows: cover the range $0, \ldots, \alpha$ with disjoint intervals of length $U$, i.e. $[0, \alpha] \subseteq [0, U] \cup [U, 2U] \cup [2U, 3U] \cup \ldots$. This requires $\left\lfloor \frac{\alpha}{U} \right\rfloor = O(\bar{n})$ intervals. On creation, the pairs $(p, s)$ are normalized such that $s \in [\ell U, (\ell + 1)U)$ implies $s = \ell U$, i.e. the value of $s$ is reduced by at most $U$ such that it is a multiple of $U$. The profit of an optimal solution to the rounded instance can be calculated as $\max\{p \mid (p, s) \in L\}$. Since the actual width of the items does not decrease, the solution may be up to $\bar{n} U$ units larger than the capacity, but is small enough when we take into account that all items are compressible:

$$(1 - \rho) \times (\alpha + \bar{n} U) = (1 - \rho) \times \left(1 + \frac{\rho}{(1 - \rho)}\right)\alpha = \alpha. \quad (14)$$

The required running time of the algorithm is $O(n_C\bar{n})$.

4.2.4 Solving the Knapsack Problems in One Pass

We first demonstrate how to solve the knapsack problems $(I \setminus I^c, \emptyset, \beta, 0)$ for each $\beta$ in some set $B$ in one pass. For this we modify the dynamic programming approach by Lawler outlined in Section 4.2.3 to solve the knapsack problem for several capacities. Similar to the original algorithm, we build the list $L$ of pairs $(p, s)$, but up to the largest capacity max $B$. This requires time $O(n_I \max B)$, where
\[ n_I = n - n_C \]

is the number of incompressible items. A pair \((p, s)\) in the list means that \(p\) is the best obtainable profit with capacity \(s\). For each \(\beta \in B\) we now find the largest \(s \leq \beta\) such that a pair \((p, s)\) is in the list. Then \(p\) is the optimal profit for capacity \(\beta\). Thus all knapsack problems can be solved in time \(O(n_I \max B)\).

For compressible items, we additionally use the normalization technique from Section 4.2.3. Let \(\alpha_{\text{min}} > 0\) be a lower bound any non-zero \(\alpha\), e.g. the minimum size of a compressible item. When solving a knapsack problem for several capacities, we cannot normalize all sizes in the same way. Instead, we use adaptive normalization. The idea is illustrated in Fig. 4. To bound the running time, we also need to impose a requirement on the set of capacities.

**Lemma 12.** Let \(A = \{\alpha_1 < \cdots < \alpha_k\}\) be a set of \(k\) capacities such that for each \(i \in \{1, \ldots, k\}\) we have

\[ \alpha_i - \alpha_{i-1} \leq \rho \alpha_i, \quad (15) \]

where \(\alpha_0 = \alpha_{\text{min}}\). Then all knapsack problems \((I^c, I^c, \alpha, \rho)\) with \(\alpha \in A\) can be solved with profit at least \(\text{OPT}(I^c, \emptyset, \alpha, 0)\) in time \(O(n_C n |A|)\).

**Proof.** Partition the interval \([\alpha_{\text{min}}, \alpha_k]\) into intervals \(I^{(1)}, \ldots, I^{(k)}\). For this, define \(I^{(i)} = [\alpha_{i-1}, \alpha_i]\) for each \(i \in \{1, \ldots, k\}\). Further partition each interval \(I^{(i)}\) into subintervals, similar as in Section 4.2.3: for \(i \in \{1, \ldots, k\}\) set \(U_i = \frac{\rho}{(1 - \rho)n} \alpha_i\), and let \(I^{(i)}_\ell = [\ell U_i, (\ell+1)U_i] \cap I^{(i)}\) for \(\ell \in \{\ell^{(i)}_{\text{min}}, \ldots, \ell^{(i)}_{\text{max}}\}\). We choose \(\ell^{(i)}_{\text{min}} = \lfloor \frac{\alpha_{i-1}}{U_i} \rfloor\) and \(\ell^{(i)}_{\text{max}} = \lceil \frac{\alpha_i}{U_i} \rceil\). Then \(I^{(i)}_{\ell^{(i)}_{\text{min}}}\) and \(I^{(i)}_{\ell^{(i)}_{\text{max}}}\) are the subintervals that start respectively end at \(\alpha_{i-1}\) and \(\alpha_i\). See Fig. 4 for a visualization of the interval structure. We again use a dynamic program to calculate the list \(L\) of pairs \((p, s)\), and normalize the widths such that \(s \in I^{(i)}_\ell \implies s = \min I^{(i)}_\ell = \max(\ell U_i, \alpha_{i-1})\). Then we can, for each \(i \in \{1, \ldots, k\}\), compute the profit of a solution to \((I^c, I^c, \alpha_i, \rho)\) as \(\max \{p \mid (p, s) \in L \text{ and } s \leq \alpha_i\}\): Since \(U_1 \leq \cdots \leq U_k\), the subintervals in \([0, \alpha_i]\) have width at most \(U_i\). Therefore, the width of a solution for a capacity in
this range is underestimated by at most $\bar{n}U_i$, which the compression compensates for, see also Eq. (14).

To determine the running time, we need to count the number of subintervals. For $i \in \{1, \ldots, k\}$, the interval $I^{(i)}$ has $\ell^{(i)}_{\text{max}} - \ell^{(i)}_{\text{min}} + 1$ many subintervals. Therefore,

$$f^{(i)}_{\text{max}} - f^{(i)}_{\text{min}} = \left\lfloor \frac{\alpha_i}{U_i} \right\rfloor - \left\lfloor \frac{\alpha_{i-1}}{U_i} \right\rfloor \leq \frac{\alpha_i}{U_i} - \left( \frac{\alpha_{i-1}}{U_i} - 1 \right) \leq \frac{\alpha_i - \alpha_{i-1}}{\rho \alpha_i} \bar{n} + 1 \leq \frac{\rho \alpha_i (1 - \rho)}{\rho \alpha_i} \bar{n} + 1 = (1 - \rho) \bar{n} + 1 = O(\bar{n}).$$

(16)

In total, we have $O(\bar{n}|A|)$ many intervals. The dynamic program therefore has running time $O(nC\bar{n}|A|)$.

4.2.5 Reducing the Number of Knapsack Problems

We propose to approximate the space $\alpha$ available for compressible items with a value $\tilde{\alpha} \geq \alpha$, and use half of the compressibility for this. Assume that $\alpha > 0$, the case $\alpha = 0$ will be handled separately. The uncompressed items in a solution of $(I^c, I^c, \hat{\alpha}, \rho)$ have size at most $\frac{\hat{\alpha}}{1 - \rho}$. Compressed with factor $\rho' = 2\rho - \rho^2$, they have a size of at most $\frac{1 - 2\rho + \rho^2}{1 - \rho} \tilde{\alpha} = (1 - \rho)\hat{\alpha}$. We therefore require $\tilde{\alpha}$ to satisfy

$$\alpha \leq \tilde{\alpha} \leq \frac{1}{1 - \rho} \alpha,$$

(17)

and will construct a set of such values by using a geometric progression.

**Definition 13.** For any positive reals $L$, $U$, and $x > 1$ we define

$$\text{geom}(L, U, x) = \{Lx^i \mid i \in 0, \ldots, \lceil \log_x \frac{U}{L} \rceil \}.$$  

(18)

**Lemma 14.** For any $L \leq U$, and $1 < x < 2$ we have $|\text{geom}(L, U, x)| = O(\frac{1}{x-1} \log \frac{U}{L})$.

**Proof.** Define $y = x - 1 > 0$ and

$$i_{\text{max}} = \left\lfloor 2 \frac{\log \frac{U}{L}}{y} \right\rfloor + 1.$$  

(19)

We claim that $Lx^{i_{\text{max}}} > U$, or equivalently, $(1+y)^{i_{\text{max}}} > \frac{U}{L}$.  

18
Since \( x < 2 \) implies \( y < 1 \), we have
\[
i_{\text{max}} > 2 \frac{\log \frac{U}{y}}{1 + y} \geq \frac{\log U}{y 1 + y}.
\] (20)

Now consider that \( \frac{y}{1 + y} \leq \log(1 + y) \) for \( y \geq 0 \): It holds for \( y = 0 \), and we have
\[
\frac{d}{dy} \frac{y}{1 + y} = \frac{1}{(1 + y)^2} \leq \frac{1}{1 + y} = \frac{d}{dy} \log(1 + y)
\] (21)

for \( y \geq 0 \). Therefore
\[
i_{\text{max}} > \frac{\log U}{\frac{y}{1 + y}} \geq \frac{\log U}{\log(1 + y)} = \log_{1+y} \frac{U}{L},
\] (22)

which proves the claim. Hence \( \text{geom}(L, U, x) \) has cardinality at most \( i_{\text{max}} = O\left(\frac{1}{y} \log \frac{U}{T}\right) \).

Recall that \( \alpha_{\text{min}} \) is a lower bound on any \( \alpha > 0 \). Let \( A = \text{geom}(\alpha_{\text{min}} \frac{1}{1 - \rho}, C, \frac{1}{1 - \rho}) \), then \( A \) contains an \( \tilde{\alpha} \) with property (17) for each \( \alpha \in \{\alpha_{\text{min}}, \ldots, C\} \). Furthermore, Lemma [14] yields \( |A| = O\left(\frac{1}{\rho} \log \frac{C}{\alpha_{\text{min}}}\right) \).

Putting everything together, we obtain Algorithm 2, a fully polynomial algorithm for the knapsack problem with compressible items.

---

**Algorithm 2:** Solve knapsack with compressible items

**Input:** \( I, I^c, C, \rho, \alpha_{\text{min}}, \beta_{\text{max}}, n \)

1. \( \alpha_{\text{min}} \leftarrow \max(\alpha_{\text{min}}, C - \beta_{\text{max}}) \)
2. \( A \leftarrow \text{geom}(\alpha_{\text{min}} \frac{1}{1 - \rho}, C, \frac{1}{1 - \rho}) \), \( A_0 \leftarrow A \cup \{0\} \)
3. for \( \tilde{\alpha} \in A \) do \( \beta(\tilde{\alpha}) \leftarrow C - (1 - \rho)\tilde{\alpha} \)
4. \( \beta(0) \leftarrow \beta_{\text{max}} \), \( B \leftarrow \{\beta(\tilde{\alpha}) \mid \tilde{\alpha} \in A_0\} \)
5. Solve \( (I \setminus I^c, \emptyset, \beta, 0) \) for \( \beta \in B \)
6. Solve \( (I^c, I^c, \tilde{\alpha}, \rho) \) for \( \tilde{\alpha} \in A_0 \), solution for \( \tilde{\alpha} = 0 \) is \( \emptyset \)
7. for \( \tilde{\alpha} \in A_0 \) do
8. Combine solutions for \( (I \setminus I^c, \emptyset, \beta(\tilde{\alpha}), 0) \) and \( (I^c, I^c, \tilde{\alpha}, \rho) \)
9. return the best combined solution

---

**Theorem 15.** Algorithm 2 finds a solution to the instance \( (I, I^c, C, \rho') \) with profit at least \( \text{OPT}(I, \emptyset, C, 0) \) in time \( O(n_1 \beta_{\text{max}} + n C \bar{n} \rho \log \frac{C}{\alpha_{\text{min}}}) \).

**Proof.** First note that the choice of \( B \) is reasonable, since \( \tilde{\alpha} \leq \frac{1}{1 - \rho} C \) for \( \tilde{\alpha} \in A \), implying \( \beta(\tilde{\alpha}) = C - (1 - \rho)\tilde{\alpha} \geq 0 \). We now show that each combined solution is a feasible solution for \( (I, I^c, C, \rho') \). Let \( I' \subseteq I \) be a solution created in line 2 for some fixed \( \tilde{\alpha} \). Then \( s(I' \cap I^c) \leq \frac{1}{1 - \rho} \tilde{\alpha} \) and \( s(I' \setminus I^c) \leq \beta(\tilde{\alpha}) = C - (1 - \rho)\tilde{\alpha} \). Compressing items from \( I^c \) with factor \( \rho' \) leads to a total size of at most
\[
(1 - 2\rho + \rho^2) \frac{1}{1 - \rho} \tilde{\alpha} + C - (1 - \rho)\tilde{\alpha} = C.
\] (23)
For the profit, consider an optimal solution \( I^* \) of \((I, \emptyset, C, 0)\). Now define \( \alpha = \sum_{i \in I^* \cap J^c} s(i) \) and \( \beta = \sum_{i \in I^* \setminus J^c} s(i) \). We claim that there is an \( \tilde{\alpha} \in A_0 \) such that \( \tilde{\alpha} \geq \alpha \) and \( \beta(\tilde{\alpha}) \geq \beta \). Recall that we enforced \( \alpha_{\text{min}} \geq C - \beta_{\text{max}} \) in line 2 but this is not a restriction, since there always must be \( C - \beta_{\text{max}} \) space available for compressible jobs. If \( \alpha = 0 \), \( \tilde{\alpha} = 0 \) clearly satisfies the claim. If \( 0 < \alpha < \alpha_{\text{min}} \), then \( \tilde{\alpha} = \min A = \frac{1}{\rho} \alpha_{\text{min}} \) is larger than \( \alpha \), and since there is \( \alpha_{\text{min}} \) space available for compressible jobs, \( \beta(\tilde{\alpha}) = C - \alpha_{\text{min}} \) space must suffice for the incompressible jobs. Otherwise we have \( \alpha \geq \alpha_{\text{min}} \), so there is one \( \tilde{\alpha} \in A \) which satisfies Eq. (17). Furthermore, \( \beta(\tilde{\alpha}) = C - (1 - \rho) \tilde{\alpha} \geq C - \alpha \geq \beta \). According to Theorem 12, the profit of the found solution is at least \( \text{OPT}(I^*, \emptyset, \tilde{\alpha}, 0) + \text{OPT}(I \setminus I^*, \emptyset, \beta(\tilde{\alpha}), 0) \). Lemma 11 now proves that the profit is at least \( \text{OPT}(I, \emptyset, C, 0) \).

Regarding the running time, the definition of \( A \) clearly satisfies Eq. (15). Therefore, we can apply the methods described in Section 4.2.4 for lines 2 and 2. Because we ensured \( \alpha_{\text{min}} \geq C - \beta_{\text{max}} \) and \( \tilde{\alpha} \geq \frac{1}{\rho} \alpha_{\text{min}} \) for each \( \tilde{\alpha} \in A \), \( \beta(\tilde{\alpha}) = C - (1 - \rho) \tilde{\alpha} \leq C - \alpha_{\text{min}} \leq \beta_{\text{max}} \). Therefore, \( \max B \leq \beta_{\text{max}} \) and line 2 requires time \( O(n \beta_{\text{max}}) \). Line 2 requires time \( O(n \rho^{-1} \log \frac{C}{\alpha_{\text{min}}}) \). These steps clearly dominate the running time of the algorithm, and \( \rho = \Theta(\rho^2) \).

Now that we have an efficient means to solve the knapsack problem with compressible items, we can create a simple fully polynomial algorithm based on Algorithm 1. To use Algorithm 2, we need to provide good bounds on \( \alpha_{\text{min}}, \beta_{\text{max}} \) and \( \bar{n} \). Therefore, we only use Algorithm 1 if \( m < 16n \). Otherwise, if \( m \geq 16n \), we can use our FPTAS with \( \varepsilon = \frac{1}{2} \) to obtain a schedule with makespan \( \frac{1}{2}d \). Then choose \( \alpha_{\text{min}} = \frac{1}{\rho}, \beta_{\text{max}} = m = O(n) \), and \( \bar{n} = m \rho = O(\varepsilon n) \). Solving the knapsack problem thus requires \( O(n^2 \log \varepsilon m) \) operations. Using the PTAS or precomputing the required values of \( \gamma_j \) requires \( O(n \log m) \) time. Finding the schedule in line 1 requires \( O(n \log n) \) time using Lemma 7. The dual algorithm has running time \( O(n(\log m + n \log \varepsilon m)) \) in total.

### 4.3 The Improved Algorithm

In the bounded knapsack problem, the input defines \( k \) item types with sizes and profits, and an item count \( c_t \) for each item type \( t \in [k] \). The number of items can be much larger than the number of item types. An instance of the bounded knapsack instance with \( k \) item types and capacity \( C \) can be transformed into an instance of the regular knapsack problem with \( O(\log m) \) items per type \([16] \). Each of these items serves as a container for an integer number of items of the same type.

To further speed up the solution of the knapsack problem \((J_B(d), J^C, m, \rho)\), we transform it into a bounded knapsack problem. Beforehand, we reduce the number of item types by rounding the jobs. But first, we define the threshold \( b \) for compressible jobs. We also introduce an accuracy parameter \( \delta \), which we later choose depending on \( \varepsilon \).
Lemma 16. Let $\delta \in (0, 1]$, $\rho = \frac{1}{4}(\sqrt{1 + \delta} - 1)$, and $b = \frac{1}{2\rho - \rho^2}$. Any job that uses at least $b$ processors can be compressed with factor $2\rho - \rho^2$, decreasing its processor count by a factor $(1 - \rho)^2$ and increasing its processing time by a factor of less than $1 + \delta$. Furthermore we have $\rho = \Theta(\delta)$ and $b = \Theta(\frac{1}{\delta})$.

Proof. Since $\delta \leq \frac{4}{5}$ we have $2\rho \leq \frac{1}{4}$, so $2\rho - \rho^2$ is a valid compression factor. Compression reduces the processor count by a factor $(1 - 2\rho + \rho^2 = (1 - \rho)^2$. The processing time increases by a factor of

$$1 + 4(2\rho - \rho^2) = 1 + 8\rho - 4\rho^2 < (1 + 4\rho)^2 = 1 + \delta.$$  \hfill (24)

The identity $(1 + 4\rho)^2 = 1 + \delta$ implies $\rho = \frac{\delta}{8(1 + 2\rho)} \geq \frac{\delta}{12}$, since $\rho \leq \frac{1}{4}$. Also, since $\sqrt{1 + \delta} \leq 1 + \delta$ we have $\rho \leq \frac{1}{4}\delta$, hence $\rho = \Theta(\delta)$.

Finally, $b = \frac{1}{2\rho - \rho^2} = \Theta(\frac{1}{\rho}) = \Theta(\frac{1}{\delta})$, because $0 < \rho^2 \leq \frac{1}{4}\rho$, so $\frac{1}{\rho} \leq 2\rho - \rho^2 < 2\rho$. \hfill \qedsymbol

To define rounded sizes and profits for the items (jobs in $J_B(d)$), we introduce a notation to round values geometrically; $\tilde{gr}(a, L, U, x) = \max\{a' \in \text{geom}(L, U, x) \mid a' \leq a\}$ for rounding down, and $\hat{gr}$ analogously for rounding up. For $s \in \{\frac{d}{2}, d\}$, round the processor counts down

$$\hat{\gamma}_j(s) = \begin{cases} \gamma_j(s) & \text{if } \gamma_j(s) \leq b \\ \hat{gr}(\gamma_j(s), b, m, 1 + \rho) & \text{otherwise} \end{cases} \hfill (25)$$

and let $s(j) = \hat{\gamma}_j(d)$. If $\gamma_j(d) < b$, round the original profit $v_j(d)$ to

$$p(j) = \begin{cases} 0 & \text{if } v_j(d) < \frac{d}{2}d \\ \hat{gr}(v_j(d), \frac{d}{2}d, \frac{d}{2}d, 1 + \frac{d}{2}) & \text{otherwise}. \hfill (26) \end{cases}$$

Otherwise, when $\gamma_j(d) \geq b$, we consider rounded processing times for $s \in \{\frac{d}{2}, d\}$, $\hat{t}_j(s) = \hat{gr}(t_j(\gamma_j(s)), \frac{d}{2}, s, 1 + 4\rho)$ and set the profit as the saved work according to the rounded values, i.e. $p(j) = \hat{t}_j(\frac{d}{2})\hat{\gamma}_j(\frac{d}{2}) - \hat{t}_j(d)\hat{\gamma}_j(d)$.

4.3.1 Bounding the Number of Item Types

In the first step, we round the processor counts $\gamma_j(d)$ and $\gamma_j(\frac{d}{2})$. Counts larger than $b$ are rounded down to the next value from $\text{geom}(b, m, 1 + \rho)$. According to Lemma [14], this leaves us with $O(\frac{\log(bm)}{\delta})$ many different counts.

First, consider the types of compressible jobs, i.e. jobs that have processor count $\gamma_j(d) \geq b$.

Lemma 17. If we round the processing times $\tilde{t}_j(\gamma_j(s))$, where $s = d$ or $s = \frac{d}{2}$, down to the next value in $\text{geom}(\frac{d}{2}, s, 1 + 4\rho)$ then there are at most $O(\frac{1}{\delta})$ many different rounded processing times (heights).
Proof. Consider that the height of each job is more than half of the shelf height $s$: Assume job $j$ has $t_j(\gamma_j(s)) \leq \frac{s}{2}$ for the sake of contradiction. Then $\gamma_j(s) \geq 1$, otherwise it would be a small job, which we removed. We claim that reducing the processor count by 1 at most doubles the processing time, because we at most halve the processor count. More formally, we have $\gamma_j(s) \leq 2(\gamma_j(s) - 1)$, because $\gamma_j(s) \geq 2$, and
\[
(\gamma_j(s) - 1) t_j(\gamma_j(s) - 1) = w_j(\gamma_j(s) - 1) \\
\leq w_j(\gamma_j(s)) = \gamma_j(s) t_j(\gamma_j(s))
\] due to monotony. Therefore
\[
t_j(\gamma_j(s) - 1) \leq \frac{\gamma_j(s)}{\gamma_j(s) - 1} t_j(\gamma_j(s)) \leq 2 t_j(\gamma_j(s)).
\] It follows that $t_j(\gamma_j(s) - 1) \leq s$, a contradiction to the definition of $\gamma_j(s)$.

The distance between two rounded processing times then must be at least $\frac{s}{2} \times 4\rho$, thus there are at most $\frac{1}{4\rho}$ rounded processing times between $\frac{s}{2}$ and $s$. \qed

In the case of compressible jobs, the rounding step can be simplified:

**Lemma 18.** Compressible jobs must have one of the two largest rounded processing times.

*Proof.* Let $s$ be again the shelf height, and $j$ be a job that is wide in the shelf, i.e. $\gamma_j(s) \geq b$. Assume $t_j(\gamma_j(s)) \leq \frac{1}{1 + 4\rho} s$. Then $t_j([((1 - \rho) \gamma_j(s)]) \leq s$ by Lemma 4 contradicting the definition of $\gamma_j(s)$. Therefore, $t_j(\gamma_j(s)) > \frac{1}{1 + 4\rho} s$. Since $\log(\frac{s}{2}, s, 1 + 4\rho)$ contains at most one value in the interval $(\frac{1}{1 + 4\rho} s, s]$, $t_j(\gamma_j(s))$ is either rounded down to that value, or the one below. \qed

Jobs that are wide in shelf $S_1$ must also be wide in shelf $S_2$, so there are $O(\frac{\log(\delta m)}{\delta})$ different processor counts and two processing times each for shelf $S_1$ and shelf $S_2$ to consider. This leaves us with no more than $k_C = O(\frac{1}{\delta} \log^2(\delta m))$ types of compressible items.

Now consider the types of incompressible jobs. These jobs have $b - 1 = O(\frac{1}{\delta})$ many different processor counts in shelf $S_1$. They can be narrow or wide in shelf $S_2$. If a job is wide in shelf $S_2$, we round the processing times according to Lemma 17. Then there are $O(\frac{1}{\delta})$ many different processing times in shelf $S_1$, and $O(\frac{\log(\delta m)}{\delta})$ different processor counts, but only two processing times possible in shelf $S_2$. Therefore, there are at most $k = O(\frac{1}{\delta^2} \log(\delta m))$ item types for such jobs.

If a job is narrow in shelf $S_2$, we directly rounded the profit down to 0 or up to the next value in $\log(\frac{s}{2}, d, \frac{s}{2} d, 1 + \frac{s}{2})$. This leaves us with $O(\frac{1}{\delta^2} \log \frac{1}{\delta})$ many different profits, or $O(\frac{1}{\delta^2} \log \frac{1}{\delta})$ item types.

In total, there are at most $k_I = O(\frac{1}{\delta^2}(\log \frac{1}{\delta} + \log(\delta m)) = O(\frac{1}{\delta^2} \log m)$ types of incompressible items.
4.3.2 Putting it Together

Our algorithm, Algorithm 3, is very similar to Algorithm 1. The main difference is how we solve the knapsack instance in line 3. We interpret it as an instance

Algorithm 3: Scheduling of monotone moldable jobs using bounded knapsack with compressible items

Input: \( J, m, d, \varepsilon \)

1. \( \delta \leftarrow \frac{1}{5} \varepsilon \), \( \rho \leftarrow \frac{1}{4} (\sqrt{1 + \delta} - 1) \), \( b \leftarrow \frac{1}{2\rho - \rho^2} \), \( d' \leftarrow (1 + \delta)^2 d \), \( J^C \leftarrow \{ j \in J_B(d) \mid \gamma_j(d) \geq \frac{1}{2} \} \)

2. for \( j \in J_B(d) \) do
   3. Precompute \( \gamma_j\left(\frac{d}{2}\right), \gamma_j(d), \gamma_j\left(\frac{d}{2}\right), \gamma_j(d'), \gamma_j\left(\frac{3}{2}d'\right), s(j), p(j) \)
   4. Find solution \( J' \) to \((J_B(d), J^C, m, \rho)\) via bounded knapsack
   5. Apply Lemma 4 to \( J' \cap J_B(d') \), obtain schedule for jobs \( J \) with makespan \( \frac{2}{3}d' \)
   6. if schedule is infeasible then reject \( d \)
   7. else return the schedule

of the bounded knapsack problem, transform it into a regular knapsack instance \((I, I^C, m, \rho)\) with \( O(k_C \log m) \) compressible and \( O(k_I \log m) \) incompressible items, and solve it using Algorithm 2. We proceed to replace the container items in the solution with the appropriate number of items (jobs) from the corresponding type. Note that we used only part of the compressibility for the solution of the knapsack problem in line 3. The rest is required to compensate for the rounding of the processor counts.

Lemma 19. When the original processor counts and processing times are considered, \( J' \) is a solution of the knapsack problem \( \text{KP}(J_B(d), m, d') \), and, unless there is no schedule with makespan \( d \), \( W(J', d') \leq md' - W_S(d) \).

Proof. We have to carefully consider the implications of our rounding of the jobs on \( J' \). We underestimated the processor count (size) of wide jobs by a factor of at most \( 1 - \rho \). The total size of all selected containers does not exceed \( m \) when the size of containers for wide jobs is multiplied by a factor of \( 1 - \rho \). According to Lemma 15, one compression with factor \( 2\rho - \rho^2 \) reduces the processor count by a factor of \((1 - \rho)^2\), and increases the processing time by a factor of less than \( 1 + \delta \). The jobs then fit into \( m \) machines. Even before the rounding, the processing time was at most \( d \), so after the compression they have processing time less than \((1 + \delta)d < d'\), so \( J' \) is a solution to \( \text{KP}(J_B(d), m, d') \).

We further have to bound the work \( W(J', d') \), assuming there is a schedule with makespan \( d \). By Lemma 6 there is a two-shelf-schedule with work at most \( md - W_S(d) \). This must also be true when the rounded processor counts and processing times are considered.

But our solution to the knapsack problem may not be optimal, since we modified the profits of the items corresponding jobs that are narrow in both shelves.
Because a solution can contain at most \( m \) jobs, omitting jobs with profit below \( \delta \frac{d}{2} \) reduces the profit by at most \( \delta \frac{m}{2}d \). Rounding the profits of the other items up lead us to overestimate the profit of the solution by at most \( m b \frac{d}{2} \delta = \delta \frac{md}{2} \), since each of the at most \( m \) items has original profit at most \( b \frac{d}{2} \), which we increased by a factor of at most \( 1 + \frac{\delta}{2} \). Since the profit equates to the saved work, the work of our schedule with rounding may be up to \( md - W_S(d) + \delta md \).

Undoing the rounding can also increase the work of our schedule, namely by a factor of \( 1 + 4 \rho \) for the rounding of the processing times, and a factor of \( 1 + \rho \) for the rounding of the processor counts. So with the original, unrounded numbers we must have

\[
W(J', d') \leq (md - W_S(d) + \delta md)(1 + 4\rho)(1 + \rho) \\
\leq (m(1 + \delta)d - W_S(d))(1 + 4\rho)^2 \\
\leq md(1 + \delta)(1 + 4\rho)^2 - W_S(d) \\
\leq md' - W_S(d).
\]

According to Corollary 10, Algorithm 3 yields a schedule with makespan \( \delta \frac{d}{2} \) \( \leq (\frac{3}{2} + \epsilon) d \).

The running time of line 3 includes the transformations of the instances and the solution, which can be done in \( O(n + k_I + k_C) \). For the solution of the knapsack instance itself, Theorem 15 states that time \( O(n_I \beta_{\text{max}} + n_C \tilde{n} \frac{\beta}{\rho} \log \frac{C}{\alpha_{\text{min}}}) \) is required, where \( n_I = O(k_I \log m) \) and \( n_C = O(k_C \log m) \). Using the same bounds \( \alpha_{\text{min}} \), \( \beta_{\text{max}} \), and \( \tilde{n} \) as before, the running time is \( O(n_I \frac{1}{\epsilon^2} \log m(\frac{\log m}{\epsilon} + \log^3(\epsilon m))) \). Again, this is the dominating step.

However, while the solution of the knapsack problem is now linear in the number \( n \) of jobs, applying the transformation rules (see Section 4.1.1) to create a feasible schedule still requires \( O(n \log n) \) operations. The total running time thus is \( O(n \frac{1}{\epsilon^2} \log m(\frac{\log m}{\epsilon} + \log^3(\epsilon m)) + n \log n) \).

### 4.3.3 Obtaining a Linear Algorithm

The super-linear running time when applying the transformations stems from organizing the jobs in \( S_1 \) in a heap. Instead of using the exact processing time \( t_j(\gamma_j(d)) \), we can use the rounded processing time \( t_j(d) \) we introduced in the last section. Then the jobs can be organized in \( O(\frac{1}{\delta}) \) lists. The running time for the transformation rules then is \( O(\frac{1}{\delta}) \). We could even organize the lists of jobs in a heap, resulting in a running time of \( O(\log \frac{1}{\delta}) \).

Since we underestimate the processing time of the jobs by at most \( \delta d \), the makespan of the final schedule is at most \( \frac{3}{2}(1 + \delta)^2 + \delta) \). Using \( \delta = \frac{4}{3} \epsilon \) will result in a \( (\frac{3}{2} + \epsilon) \)-dual approximation algorithm. Since the other steps of the dual algorithm remain the same, the total running time is \( O(n \frac{1}{\epsilon^2} \log m(\frac{\log m}{\epsilon} + \)
Conclusion

We have presented several techniques to exploit monotony of jobs, mostly based on the ability to compress wide jobs. For two different algorithms we demonstrated that these techniques can help to reduce the running time from polynomial in $m$ to polynomial in $\log m$. We also showed that arbitrarily good approximation guarantees can be achieved in polynomial time. On the negative side, we proved the NP-hardness of scheduling monotonic jobs. It remains open whether a better approximation guarantee than $\frac{3}{2}$ can be achieved efficiently, e.g. in the form of an EPTAS.

References

[1] Krishna P. Belkhale and Prithviraj Banerjee. An approximate algorithm for the partitionable independent task scheduling problem. In Proceedings of the 1990 International Conference on Parallel Processing (ICPP 1990), pages 72–75. Pennsylvania State University Press, 1990.

[2] Chandra Chekuri, Ashish Goel, Sanjeev Khanna, and Amit Kumar. Multi-processor scheduling to minimize flow time with $\varepsilon$ resource augmentation. In Proceedings of the Thirty-sixth Annual ACM Symposium on Theory of Computing, STOC ’04, pages 363–372, New York, NY, USA, 2004. ACM.

[3] Maciej Drozdowski. On the complexity of multiprocessor task scheduling. In Bulletin of the Polish Academy of Sciences, Technical Sciences, volume 43, pages 381–392. 1995.

[4] Jianzhong Du and Joseph Y.-T. Leung. Complexity of scheduling parallel task systems. SIAM Journal on Discrete Mathematics, 2:473–487, 1989.

[5] M. R. Garey and R. L. Graham. Bounds for multiprocessor scheduling with resource constraints. SIAM Journal on Computing, 4:187–200, 1975.

[6] Michael R. Garey and David S. Johnson. Computers and Intractability: A Guide to the Theory of NP-Completeness. Series of Books in the Mathematical Sciences. W. H. Freeman & Co., 1979.

[7] Alexander Grigoriev and Marc Uetz. Scheduling parallel jobs with linear speedup. In Workshop on Approximation and Online Algorithms (WAOA 2006), pages 203–215. Springer, 2006.
[8] Dorit S. Hochbaum and David B. Shmoys. Using dual approximation algorithms for scheduling problems: Theoretical and practical results. *Journal of the ACM*, 34(1):144–162, 1987.

[9] Klaus Jansen. A \((3/2 + \varepsilon)\) approximation algorithm for scheduling moldable and non-moldable parallel tasks. In *Proceedings of the 24th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA 2012)*, pages 224–235. ACM Press, 2012.

[10] Klaus Jansen, Felix Land, and Kati Land. Bounding the running time of algorithms for scheduling and packing problems. Technical Report 1302, Institute of Computer Science, University of Kiel, 2013.

[11] Klaus Jansen, Felix Land, and Kati Land. Bounding the running time of algorithms for scheduling and packing problems. *SIAM Journal on Discrete Mathematics*, 30(1):343–366, 2016.

[12] Klaus Jansen and Lorant Porkolab. Computing optimal preemptive schedules for parallel tasks: linear programming approaches. *Mathematical Programming*, 95(3):617–630, 2003.

[13] Klaus Jansen and Lorant Porkolab. On preemptive resource constrained scheduling: Polynomial-time approximation schemes. *SIAM Journal on Discrete Mathematics*, 20:545–563, 2006.

[14] Klaus Jansen and Ralf Thöle. Approximation algorithms for scheduling parallel jobs. *SIAM Journal on Computing*, 39(8):3571–3615, 2010.

[15] Klaus Jansen and Hu Zhang. Scheduling malleable tasks with precedence constraints. *Journal of Computer and System Sciences*, 78:245–259, 2012.

[16] Hans Kellerer, Ulrich Pferschy, and David Pisinger. *Knapsack Problems*. Springer, 2004.

[17] Eugene L. Lawler. Fast approximation algorithms for knapsack problems. *Mathematics of Operations Research*, 4(4):339–356, 1979.

[18] Walter Ludwig and Prasoon Tiwari. Scheduling malleable and nonmalleable parallel tasks. In *Proceedings of the Fifth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 167–176. Society for Industrial and Applied Mathematics, 1994.

[19] Konstantin Makarychev and Debmalya Panigrahi. Precedence-constrained scheduling of malleable jobs with preemption. In *Automata, Languages, and Programming (ICALP 2014)*, volume 8572 of *Lecture Notes in Computer Science*, pages 823–834. Springer, 2014.
[20] Grégory Mounié, Christophe Rapine, and Denis Trystram. Efficient approximation algorithms for scheduling malleable tasks. In Gary Miller and Vijaya Ramachandran, editors, Proceedings of the Eleventh annual ACM Symposium on Parallel Algorithms and Architectures (SPAA 1999), pages 23–32. ACM, 1999.

[21] Grégory Mounié, Christophe Rapine, and Denis Trystram. A $\frac{3}{2}$-dual approximation algorithm for scheduling independent monotonic malleable tasks, 2004.

[22] Grégory Mounié, Christophe Rapine, and Denis Trystram. A $\frac{3}{2}$-approximation algorithm for scheduling independent monotonic malleable tasks. SIAM Journal on Computing, 37:401–412, 2007.

[23] Peter Sanders and Jochen Speck. Efficient parallel scheduling of malleable tasks. In 2011 IEEE International Parallel & Distributed Processing Symposium (IPDPS), pages 1156–1166. IEEE Computer Society, 2011.

[24] Denis Trystram, Mikhail Y. Kovalyov, Jacek Blazewicz, Maciej Machowiak, and Jan Weglarz. Preemptable malleable task scheduling problem. IEEE Transactions on Computers, 55:486–490, 2006.

[25] John Turek, Joel L. Wolf, and Philip S. Yu. Approximate algorithms scheduling parallelizable tasks. In Proceedings of the fourth annual ACM symposium on Parallel algorithms and architectures (SPAA '92), pages 323–332. ACM Press, 1992.