INSTITUT FÜR INFORMATIK

Approximation Algorithms for Scheduling Parallel Jobs: Breaking the Approximation Ratio of 2

Klaus Jansen and Ralf Thöle

Bericht Nr. 0808
September 2008

CHRISTIAN-ALBRECHTS-UNIVERSITÄT

ZU KIEL

Institut für Informatik der
Christian-Albrechts-Universität zu Kiel
Olshausenstr. 40
D – 24098 Kiel

Approximation Algorithms for Scheduling Parallel Jobs: Breaking the Approximation Ratio of 2

Klaus Jansen and Ralf Thöle

Bericht Nr. 0808
September 2008

e-mail: kj@informatik.uni-kiel.de, rth@informatik.uni-kiel.de

Approximation Algorithms for Scheduling Parallel Jobs: Breaking the Approximation Ratio of 2*

Klaus Jansen Ralf Thöle

Institut für Informatik

Universität zu Kiel, Olshausenstr. 40, 24098 Kiel, Germany

Email: {kj, rth}@informatik.uni-kiel.de

In this paper we study variants of the non-preemptive parallel job scheduling problem in which the number of machines is polynomially bounded in the number of jobs. For this problem we show that a schedule with length at most $(1+\varepsilon)\,\mathrm{OPT}$ can be calculated in polynomial time. Unless $P = NP$, this is the best possible result (in the sense of approximation ratio), since the problem is strongly NP-hard.

For the case, where all jobs must be allotted to a subset of consecutive machines, a schedule with length at most $(1.5+\varepsilon)\,\mathrm{OPT}$ can be calculated in polynomial time. The previously best known results are algorithms with absolute approximation ratio 2.

Furthermore, we extend both algorithms to the case of malleable jobs with the same approximation ratios.

1 Introduction

In classical scheduling theory, each job is executed by only one processor at a time. In the last years however, due to the rapid development of parallel computer systems, new theoretical approaches have emerged to model scheduling on parallel architectures (for

*An extended abstract of this paper appeared in proceedings of the 35th International Colloquium on Automata, Languages and Programming (ICALP 2008), LNCS 5125, 234–245. Springer, 2008.
Research was supported by PPP funding "Approximation algorithms for d-dimensional packing problems" 315/ab D/05/50457 granted by the DAAD, and by EU research project AEOLUS, Algorithmic Principles for Building Efficient Overlay Computers, EU contract number 015964.

an overview about scheduling of multiprocessor jobs on such parallel architectures see for example [4, 7, 23]).

In this paper, we study variants of the non-preemptive parallel job scheduling problem. An instance of this problem is given by a list $L := \{J_1, \ldots, J_n\}$ of jobs and for each job J_j an execution/processing time p_j and the number of required machines q_j is given. A schedule $S = ((s_1, r_1), \ldots, (s_n, r_n))$ is a sequence of starting times $s_j \geq 0$ together with the set of assigned machines $r_j \subseteq \{1, \ldots, m\}$ ($|r_j| = q_j$) for $j \in \{1, \ldots, n\}$. A schedule is feasible if each machine executes at most one job at the same time. The length of a schedule is defined as its latest job completion time $C_{\max} = \max\{s_j + p_j \mid j \in \{1, \ldots, n\}\}$. The objective is to find a feasible schedule of minimal length. This problem is denoted by $P|\text{size}_j|C_{\max}$ (for more information on this three field notation see for example [7]).

1.1 Known Results

$P|\text{size}_j|C_{\max}$ is strongly NP-hard, since the problem $P5|\text{size}_j|C_{\max}$, where the number of available processors is 5, is NP-hard in the strong sense [8]. Furthermore, there is no approximation algorithm with a performance ratio better than 1.5 for $P|\text{size}_j|C_{\max}$ [20], unless P = NP.

The best known algorithm with polynomial running time for this problem was implicitly given by Garey and Graham [11]. They proposed a list-based algorithm with approximation ratio 2 for a resource-constrained scheduling problem. In this scheduling problem one or more resources are given and each job requires a certain amount of each resource for the duration of its execution time. As pointed out by Ludwig & Tiwari [24] this resource-constrained scheduling problem can be used to model $P|\text{size}_j|C_{\max}$ by using the available processors as the single resource. The existence of a polynomial time approximation scheme (PTAS) for the case that the number of available processors is a constant, $Pm|\text{size}_j|C_{\max}$, was presented in [1, 15].

A problem closely related to $P|\text{size}_j|C_{\max}$ is the *strippacking* problem (i.e. packing of rectangles in a strip of width 1 while minimizing the packing height). The main difference is that machines assigned to a job need to be contiguous in a solution of the strippacking problem. Turek et al. [30] pointed out that using contiguous machine assignments is desirable in some settings; for example to maintain a physical proximity of processors allotted to a job. This contiguous case is known under different names in the literature, amongst others: scheduling on a line, $P|\text{line}_j|C_{\max}$, or non-fragmentable multiprocessor system. In the literature, even further models are considered to model the underlying network topology, such as *meshes* or hypercubes. Note that the one-dimensional mesh corresponds to the line model, whereas the two-dimensional mesh corresponds to three-dimensional strippacking (see for example [9, 10, 3, 31]). One of the first results for the strippacking problem was given by Coffman et al. [5]. They proved that the level-based algorithms NFDH (Next Fit Decreasing Height) and FFDH (First Fit Decreasing Height) algorithms have an approximation ratio of 3 and 2.7 respectively. The currently best known algorithms with absolute approximation ratio 2 were given independently by Schiermeyer [27] and Steinberg [29]. Coffman et al. [5] also analyzed the asymptotic performance of NFDH and FFDH, which is $2 \cdot \text{OPT} + h_{\max}$ and $1.7 \cdot \text{OPT} + h_{\max}$ respectively, where OPT is the height of an optimal solu-

tion and $h_{\max}$ denotes the height of the tallest rectangle. An AFPTAS for the strippacking-problem was presented by Kenyon & Rémila [21]. Only recently, Jansen and Solis-Oba [17] presented an asymptotic polynomial time approximation scheme (APTAS) with additive term 1 at the cost of a higher running time.

A similar problem is the scheduling of so-called *malleable* jobs, where the number of required machines for each job is not known a priori; the execution time of each job depends on the number of allotted machines. That is, instead of p_j, q_j each job J_j has an associated function $p_j : \{1,\dots,m\} \to \mathbb{Q}^+$ that gives the execution time $p_j(\ell)$ of that job in terms of the number ℓ of processors that are assigned to J_j. For this scheduling problem Ludwig & Tiwari [24] presented an algorithm with approximation ratio 2. Jansen & Porkolab [15] developed an approximation scheme with linear running time for both malleable jobs and non-malleable jobs as long as the number of machines is constant. For the case that preemptions are allowed (jobs can be interrupted at any time at no cost and restarted later on a possibly different set of processors) Jansen & Porkolab [16] provide an optimal algorithm with running time polynomial in m and linear in n. Mounié et al. [25] present an $(1.5+\varepsilon)$ approximation algorithm for scheduling a set of independent monotonic malleable jobs, where the machines allotted to each job have consecutive addresses. Implicitly, this algorithm requires the number of machines to be polynomially bounded in the number of jobs, since the running time of the algorithm depends on the number of machines. Decker et al. [6] presented a 1.25-approximation for scheduling n independent identical malleable jobs on p identical processors (the jobs are called identical if the execution time on any number of processors is the same for all jobs). In [13], Jansen presented an asymptotic fully polynomial time approximation scheme (AFPTAS) for scheduling malleable jobs on an arbitrary number of machines.

In the literature, a lot of scheduling problems with additional constraints are studied. Numerous publications deal with so called *online scheduling of parallel jobs* (for a survey see [28]). In *online scheduling*, not all informations about the instance are known a priori, e.g. unknown release dates, unknown running times (see for example [26, 31]). Another often studied type of constraint are so-called *precedence constraints*, where a job can only be scheduled for execution if all of its predecessors have already completed their execution (see for example [2, 18, 19, 22]). In many papers also combinations of different constraints are studied; for example in [3, 9, 10]) *online* scheduling with *precedence constraints* is studied. Note that in [9, 10] results for different network topologies such as *PRAM*, *line*, *meshes*, *hypercubes* are presented; in [3] *hypercubes* and *arrays* are considered as underlying network topology; in [31] *hypercubes* are considered.

1.2 New Results

In this paper, we focus on the natural case where the number of machines is polynomially bounded in the number of jobs (in most scenarios the number of machines will be even smaller than the number of jobs). We will denote this problem by $P\text{poly}|\text{size}_j|C_{\max}$ or by $P\text{poly}|\text{line}_j|C_{\max}$ in the contiguous case. Using a reduction from 3-PARTITION [12, SP15] , it is easy to see that these problems are strongly NP-hard.

3-PARTITION: Given an integer value B and a list of $n = 3k$ integers s_i with $\frac{B}{4} < s_i < \frac{B}{2}$

and $\sum_{i=1}^{n} s_i = kB$. The objective is to find a partition into k subsets where each subset has sum B. Note that the above constraints on the values imply that every subset must contain exactly three integers.

The reduction works as follows. We set the number of machines to B, and introduce n jobs J_i, each with processing time $p_i := 1$ and number of required machines $q_i := s_i$. If there is a schedule of length k, we have a solution for 3-PARTITION. On the other hand, if there is a solution to 3-PARTITION there exists a schedule of length k. Since 3-PARTITION is NP-hard in the strong sense, the problem is also NP-hard for instances where all numbers are polynomially bounded in n. In particular, all instances in which B is polynomially bounded in n are NP-hard.

For the case that all machines assigned to a job have contiguous addresses, we show the existence of an algorithm with approximation ratio arbitrarily close to 1.5.

Theorem 1. *For every $\varepsilon > 0$ there exists an algorithm A such that for every instance I of $Ppoly|line_j|C_{\max}$*

$$A(I) \leq (1.5 + \varepsilon)\,\mathrm{OPT}(I)$$

holds and the running time is polynomial in n, where $A(I)$ is the length of the schedule for instance I generated by algorithm A and $\mathrm{OPT}(I)$ is the length of an optimal schedule for instance I.

The previous best known result for this problem is a 2-approximation algorithm by Ludwig & Tiwari [24]. The algorithm for scheduling monotonic malleable jobs presented by Mounié et al. [25] with approximation ratio $(1.5 + \varepsilon)$ makes use of the monotonic character and thus is not applicable to the non-malleable case. Interestingly, the result is otherwise very similar. They also generate a schedule with contiguous machine addresses and they also (implicitly) assume that the number of machines is polynomially bounded in the number of jobs since the running time depends on the number of machines.

This result holds also for the strippacking problem if we restrict the instances such that the width of each rectangle is a multiple of $1/m$ for some integer value m that is polynomially bounded in the number of rectangles.

In the general case (non-contiguous addresses) we show the existence of a polynomial time approximation scheme (PTAS).

Theorem 2. *For every $\varepsilon > 0$ there exists an algorithm A such that for every instance I of $Ppoly|size_j|C_{\max}$*

$$A(I) \leq (1 + \varepsilon)\,\mathrm{OPT}(I)$$

holds and the running time is polynomial in n, where $A(I)$ is the length of the schedule for instance I generated by algorithm A and $\mathrm{OPT}(I)$ is the length of an optimal schedule for instance I.

The previous best known result for this problem is the above mentioned 2-approximation algorithm for the resource-constrained scheduling problem by Garey & Graham [11]. Other algorithms with absolute approximation ratio lower than 2 presume that the number of machines is a constant.

Furthermore, we show in section 6, how these results can be extended to the malleable case (with the same approximation ratios), even if the execution time of the jobs is not monotone. This result cannot be obtained by applying the framework described by Ludwig & Tiwari [24], since the analysis of the approximation ratio of our algorithm is not based on lower bounds that are required for their framework.

For the non-contiguous case these are the best possible results (in the sense of approximation ratio), since the problem is NP-hard in the strong sense for both the malleable and the non-malleable case.

1.3 Structure

We start this paper with a short outline of the algorithms in section 2. In section 3, we present an algorithm to pack rectangles into a constant number of bins. This algorithm will be used as *subroutine* in the following scheduling algorithms. In section 4, we present the algorithm for scheduling jobs on machines with contiguous addresses. We present the scheduling algorithm for the case that the machines allotted to each job are not required to have contiguous addresses in section 5. In section 6, we show how the algorithms for the non-malleable cases can be extended to solve the corresponding malleable versions with the same approximation ratio. We conclude with open problems in section 7.

2 Outline of the Algorithms

Before we go into the details, we give a short outline of the algorithms.

A crucial part of the algorithms is to schedule *critical* jobs (jobs with long execution time or large number of required processors) nearly optimal. This is done by enumerating a polynomial number of schedules for the critical jobs. To ensure that there is at least one schedule among these that allows a nearly optimal solution, we have to reduce the search space. Therefore, we show that it is possible to modify an optimal schedule such that the resulting schedule is nearly optimal and has a simpler structure.

To be more specific, the first step in our algorithms is to guess (enumerate) the approximate value of an optimal solution (sections 4.1.1, 5.1.1, 6.1.1). This allows us to divide the solution into a constant number of *slots* with height depending on the accuracy. After that, we partition the set of jobs. The purpose of this step is to create a gap in size (processing time / number of required machines) between *big* and *small* jobs. This is done by discarding *middle-sized* jobs (sections 4.1.2, 5.1.2, 6.1.2). We schedule all discarded jobs using a greedy algorithm in a post-processing step. Then, we round the long jobs (sections 4.1.3, 5.1.3, 6.1.3) and define containers into which we place (some of) the short jobs. From here on the algorithms for the non-contiguous and the contiguous case differ significantly. For the contiguous case we guess (enumerate) a set of containers. Since the actual packing algorithm cannot guarantee to schedule all long jobs, a crucial step is to take care of jobs with running time $> 1/2$ (section 4.2.3). We then solve a linear program (section 4.3) and use its solution to create the actual schedule for the containers and for a subset of the long

jobs (section 4.4). The scheduling of short jobs inside the container is done by a modified version of the algorithm by Kenyon & Rémila [21] (section 4.4.6).

For the non-contiguous case we show that the long jobs can be scheduled in a canonical way (section 5.3) and use a dynamic program to assign the long jobs to *slots* (section 6.2). The scheduling of the short jobs is again done by using the modified version of the algorithm by Kenyon & Rémila (section 5.4).

The extension to the malleable cases is done by choosing an assignment of jobs to a number of machines in a first phase (section 6). After that the solution can be found by applying the algorithms used for the non-malleable cases.

3 Packing into a Constant Number of Bins

In the following, we present a modification of the algorithm by Kenyon and Rémila in [21], which we will call mKR. Instead of packing into one target strip, we want to pack into a constant number of bins with different sizes. We show that under certain assumptions (see (A1)–(A6)) almost all rectangles can be packed into the bins, i.e. the rectangles that are not packed have small total area.

Let $\mathscr{C} = \{C_1, \dots, C_k\}$ be a set of k bins. We assume that all bins have width and height bounded by 1. Let $L = \{R_1, \dots, R_n\}$ denote the set of rectangles and let $L = L_{\text{sm}} \cup L_{\text{wi}}$ be a partition of the set of rectangles into wide and small rectangles and define

- $h^w_{\max} := \max_{R \in L_{\text{wi}}} \mathrm{h}(R)$, $w^w_{\min} := \min_{R \in L_{\text{wi}}} \mathrm{w}(R)$, $w^w_{\max} := \max_{R \in L_{\text{wi}}} \mathrm{w}(R)$,
- $h^s_{\max} := \max_{R \in L_{\text{sm}}} \mathrm{h}(R)$, $w^s_{\max} := \max_{R \in L_{\text{sm}}} \mathrm{w}(R)$.

With $\mathrm{h}(Q), \mathrm{w}(Q)$ we denote the height and the width of a rectangle Q or a bin Q, respectively. Furthermore, we denote with $\mathrm{A}(Q) := \mathrm{h}(Q)\,\mathrm{w}(Q)$ the area of Q. We extend the notations to sets in the straight-forward manner (for example $\mathrm{A}(\mathscr{C}) = \sum_{C \in \mathscr{C}} \mathrm{A}(C)$).

Let $c, \delta > 0$. With the following assumptions we can formulate the theorem.

$$\text{There exists a packing of } L_{\text{wi}} \text{ into the bins } \mathscr{C}, \tag{A1}$$

$$\mathrm{A}(L) \le \mathrm{A}(\mathscr{C}), \tag{A2}$$

$$w^w_{\min} \ge \delta, \tag{A3}$$

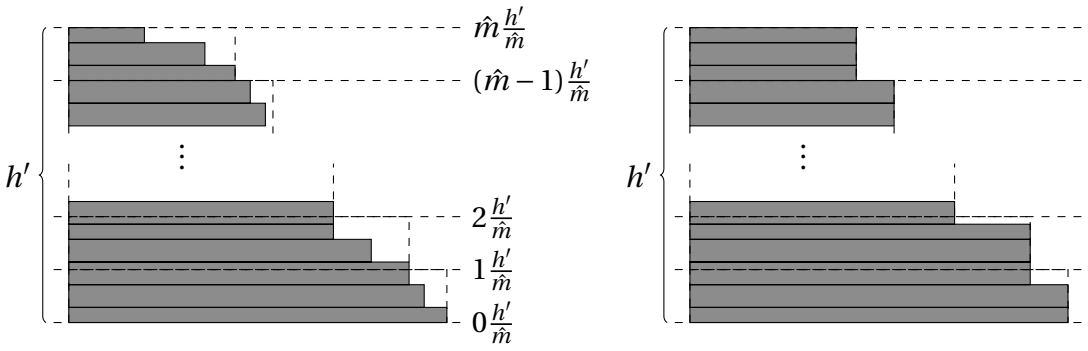
$$w^s_{\max} \le \frac{1}{k}\frac{\delta}{7}, \tag{A4}$$

$$h^w_{\max} \le \frac{\delta}{7} \min\left\{\frac{1}{k}, w^w_{\min}\frac{\delta}{7}\right\}, \tag{A5}$$

$$h^s_{\max} \le \frac{\delta}{7} \min\left\{\frac{1}{4k}, w^w_{\min}\frac{\delta}{4 \cdot 7}\right\}. \tag{A6}$$

Theorem 3. *Under the assumptions (A1)–(A6) there exists an algorithm with running time polynomial in n, k and $1/\delta^2$ that packs almost all rectangles into the bins, i.e. the unpacked rectangles have total area at most $\delta\,\mathrm{A}(L)$ if $\mathrm{A}(L) \ge 1$ or δ otherwise.*

In the following sections, we briefly describe the algorithm.



(a) Grouping

(b) Rounding rectangles, L_{sup}

Figure 1: Grouping of wide rectangles

3.1 Grouping and Rounding

In order to simplify the problem, we transform the rectangles from L_{wi} into a set L_{sup} that consists only of rectangles with $\hat{m}$ (a constant depending on $w^w_{\min}$) different widths. This transformation is similar to the grouping technique used by Kenyon and Rémila in [21]. Note that this simplification is feasible, since it is not necessary to pack the rectangles optimally.

First we order all rectangles from L_{wi} by non-increasing width. Then we stack them left-aligned on top of each other, resulting in a stack of height $h' = \text{h}(L_{\text{wi}})$ (see figure 1a). Next we draw horizontal cutting lines at heights $i(h'/\hat{m})$ for $i \in \{0,\dots,\hat{m}\}$ across the stack. We say that rectangle $R \in L_{\text{wi}}$ belongs to group i, if its upper side u_R satisfies $i(h'/\hat{m}) < u_R \leq (i+1)(h'/\hat{m})$.

We generate L_{sup} by rounding up the width of each rectangle $R_i \in L_{\text{wi}}$ such that its width is the same as the width of the widest rectangle belonging to the same group (see figure 1b). Since all wide rectangles are packable (Assumption (A1)) we can ensure the existence of a feasible fractional packing for L_{sup} by removing all rectangles intersecting $[0, w^w_{\max}] \times [0, h'/\hat{m}]$. These intersecting rectangles have total area bounded by

$$V_1 := \left(\frac{h'}{\hat{m}} + h^w_{\max}\right) w^w_{\max} \leq \frac{h'}{\hat{m}} + h^w_{\max}.$$

The feasibility follows bascially by area arguments; since the width of each rectangle in group i is smaller than the width of each rectangle in group $(i-1)$, all rectangles can be packed (at least fractionally) into the space occupied by rectangles from the next lower group.

3.2 Fractional Binpacking

To find a fractional packing for L_{sup} we use basically the same linear program as in [21]. The main difference is that

- we have a set of configurations for each bin instead of one set of configurations, and
- we need k constraints in addition to the $\hat{m}$ constraints in [21].

The k additional constraints ensure that the total height of the configurations for each bin does not exceed the height of the bin. The $\hat{m}$ *original* constraints ensure that all rectangles are covered (one constraint for each group of rectangles). Thus, instead of $\hat{m}$ constraints we have $\hat{m}+k$ constraints and solving the linear program results in at most $\hat{m}+k$ variables with non-zero value instead of $\hat{m}$ non-zero variables.

Since we assume that all wide rectangles are packable (Assumption (A1)) and since we discarded all rectangles intersecting $[0, w^w_{\max}] \times [0, h'/\hat{m}]$, there exists a feasible (fractional) solution for the rounded instance.

3.3 Packing the Rectangles

Each of the $\hat{m}+k$ non-zero variables corresponds to one of the configurations. In order to generate a packing, we define layers inside each bin with height corresponding to the value of the variables. In contrast to the algorithm by Kenyon and Rémila we do not increase the height of the layers.

The space for each layer can now be divided into the *left* side with width equal to the width of the corresponding configuration and the *right* side, which will be used for packing the rectangles from L_{sm}.

The packing of the rectangles from L_{sup}, or rather the packing of the rectangles from L_{wi}, is done in the same way as in [21], but since we have not increased the height of the layers, the topmost rectangles might be overlapping into the next layer or over the upper border of the bin. In order to make the packing feasible, we simply remove all overlapping rectangles. The rectangles removed in this step have total area bounded by

$$V_2 := (\hat{m}+k)(1 \cdot h^w_{\max}),$$

since the height of each of these rectangles is bounded by $h^w_{\max}$ and the width of each layer is bounded by 1 and the number of layers is bounded by $\hat{m}+k$.

The rectangles from L_{sm} will be added by a modified version of the Next Fit Decreasing Height (mNFDH) algorithm. After ordering L_{sm} by non-increasing height, we use NFDH layer by layer. Note that we add an additional layer for each bin, if the upper border of the last layer is below the upper border of this bin. Furthermore, we add an additional layer if the space reserved by a configuration is not completely packed. Since this can only happen if there are not *enough* rectangles belonging to a specific group, we get at most $\hat{m}$ additional layers. Thus, in total we pack the low rectangles into at most $2\hat{m}+k$ layers.

If all of L_{sm} is packed by the mNFDH algorithm the total area of all unpacked rectangles is bounded by V_1+V_2. Otherwise, we have to calculate how much of the total area of all bins is covered by the (fractionally) packed rectangles from L_{wi} and by the packed rectangles from L_{sm} in order to get an upper bound for the total area of the remaining rectangles from L_{sm}.

Obviously, the wasted space on the right side of each bin and layer is bounded by the width of the small rectangles (since we use NFDH); this bound holds even for the last layer, since not all rectangles could be packed. Additionally, we can guarantee for each layer (including the additional layers) that an area with height corresponding to the height of the layer minus two times the height of the tallest rectangles from L_{sm} is covered, i.e. the

uncovered space is bounded in total by

$$\begin{aligned} V_3 &:= \mathrm{h}(\mathscr{C})\, w^s_{\max} && \text{right side} \\ &\quad + (2\hat{m}+k+k)2h^s_{\max} && \text{upper bound for uncovered area for each layer} \\ &\le k w^s_{\max} + (2\hat{m}+2k)2h^s_{\max}. \end{aligned}$$

In total, we obtain as upper bound for the total area of all unpacked rectangles

$$\begin{aligned} V_1+V_2+V_3 &= \frac{h'}{\hat{m}} + h^w_{\max} + (\hat{m}+k)(1\cdot h^w_{\max}) + k w^s_{\max} + (2\hat{m}+2k)2h^s_{\max} \\ &\le \frac{\mathrm{A}(L)}{w^w_{\min}\hat{m}} + h^w_{\max} + (\hat{m}+k)(1\cdot h^w_{\max}) + k w^s_{\max} + (2\hat{m}+2k)2h^s_{\max} \\ &\overset{(A5)}{\le} \frac{\mathrm{A}(L)}{w^w_{\min}\hat{m}} + \frac{\delta}{7} + \hat{m}h^w_{\max} + \frac{\delta}{7} + k w^s_{\max} + 4\hat{m}h^s_{\max} + 4k h^s_{\max} \\ &\overset{(A4),(A6)}{\le} \frac{\mathrm{A}(L)}{w^w_{\min}\hat{m}} + \frac{\delta}{7} + \hat{m}h^w_{\max} + \frac{\delta}{7} + \frac{\delta}{7} + 4\hat{m}h^s_{\max} + \frac{\delta}{7}. \end{aligned}$$

Choosing $\hat{m} := \frac{7}{\delta w^w_{\min}} \overset{(A3)}{\le} \frac{7}{\delta^2}$, this equals

$$\begin{aligned} &= \mathrm{A}(L)\frac{\delta}{7} + \frac{4\delta}{7} + \frac{7}{\delta w^w_{\min}}h^w_{\max} + \frac{4\cdot 7}{\delta w^w_{\min}}h^s_{\max} \\ &\overset{(A5),(A6)}{\le} \mathrm{A}(L)\frac{\delta}{7} + \frac{4\delta}{7} + \frac{7}{\delta w^w_{\min}}\frac{\delta}{7}w^w_{\min}\frac{\delta}{7} + \frac{4\cdot 7}{\delta w^w_{\min}}\frac{\delta}{7}w^w_{\min}\frac{\delta}{4\cdot 7} \\ &= \mathrm{A}(L)\frac{\delta}{7} + \frac{4\delta}{7} + \frac{\delta}{7} + \frac{\delta}{7} \\ &= \mathrm{A}(L)\frac{\delta}{7} + \frac{6\delta}{7} \\ &\le \begin{cases} \delta & \text{if } \mathrm{A}(L) \le 1 \\ \delta\,\mathrm{A}(L) & \text{if } \mathrm{A}(L) \ge 1. \end{cases} \end{aligned}$$

The running time of our algorithm is polynomial in n, k and $1/\delta^2$. For a detailed analysis of the running time we refer the reader to the analysis used in [21]. The main difference is the number of configurations and the additional k constraints.

This proves Theorem 3.

4 Contiguous Parallel Job Scheduling

In this section, we present the algorithm for scheduling parallel jobs such that the machines assigned to a job have contiguous indices, $P\mathrm{poly}|\mathrm{line}_j|C_{\max}$. Note that the algorithm presented in this section uses some of the techniques presented in [17]. The main difference concerns jobs with processing time $> 1/2$. Our modifications ensure that no job with processing time $> 1/2$ is discarded.

Since each job is required to be executed on contiguous machines, an instance of this problem can be translated directly into a strippacking instance; for each job J_i create a rectangle $R_i = (w_i, h_i)$ of width $w_i = q_i/m$ and height $h_i = p_i$ (where q_i is the number of required machines and p_i is the execution time of J_i). We scale the width of R_i by $1/m$, such that the target strip in the resulting strippacking instance has width 1. The objective is then to find an orthogonal, axis parallel arrangement of all rectangles into a strip of width 1 and minimal height (without rotations). Thus, the strippacking and the scheduling notation can be used synonymously in the contiguous case. In this paper (especially in this section), we will use the strippacking notation since this notation is more descriptive. The non-contiguous case can also be viewed as strippacking problem if *fragmentation* of rectangles is allowed in one dimension (width).

Obviously, a solution for the contiguous case is a feasible solution for the non-contiguous case. However, an optimal solution for the contiguous case is in general not optimal for the non-contiguous case. Turek et al. [30] presented an example instance that shows that the length of an optimal schedule for the non-contiguous case can be shorter than for the contiguous case.

4.1 Near-Optimal Schedule with Simple Structure

In the following, we describe the construction of a nearly optimal solution with simple structure based on a given optimal solution. This simply-structured solution is similar to the solution constructed in [17]. The main difference is that here we make sure that the total area of the discarded jobs is small, while the total profit of the discarded rectangles is small in the construction in [17].

In the following, let $0 < \varepsilon \leq 1$ be the required accuracy and let $L = \{R_1, \ldots, R_n\}$ be an instance of $P\text{poly}|\text{line}_j|C_{\max}$. For each rectangle (job) R_i let w_i be its width (number of processors q_i/m) and let h_i be its height (execution time p_i). A packing (schedule) P for instance L is given as a set of pairs $P = \{(x_1, y_1), \ldots, (x_n, y_m)\}$, where each pair $(x_i, y_i) \in \mathbb{R}^2_{\geq 0}$ denotes the position of the lower left corner of rectangle R_i in the strip. Note that in this case the representation of the schedule by a packing is sufficient, since the subset of assigned processors is well-defined by the first assigned processor. We assume that the lower left corner of the strip coincides with the origin of a Cartesian system of coordinates. A packing P is valid if the rectangles do not overlap and $x_i + w_i \leq 1$ for all $i \in \{1, \ldots, n\}$. The height of packing P is given by

$$\mathrm{h}(P) := \max_{i \in \{1,\ldots,n\}} (y_i + h_i).$$

4.1.1 Bounded Height

Since we want to divide the solution into a constant number of *slots*, we need to know the height of an optimal solution, at least up to the required accuracy ε.

By using the strippacking algorithm of Steinberg [29], we can find a solution for the strippacking instance with height $\nu \leq 2 \cdot \mathrm{OPT}$, where OPT is the height of an optimal solution.

Obviously, there exists a value

$$v^* \in \left\{(1+0\varepsilon)\frac{v}{2}, (1+1\varepsilon)\frac{v}{2}, \ldots, (1+\left\lceil\frac{1}{\varepsilon}\right\rceil\varepsilon)\frac{v}{2}\right\}$$

such that $\mathrm{OPT} \leq v^* \leq (1+\varepsilon)\,\mathrm{OPT}$; we only have to consider $\lceil 1/\varepsilon \rceil + 1$ different candidates to find the right one. For simplicity we divide the height of each rectangle by v^*, such that the height $\mathrm{OPT}' := \frac{\mathrm{OPT}}{v^*}$ of an optimal solution for the scaled instance satisfies

$$1-\varepsilon < \frac{1-\varepsilon}{1-\varepsilon^2} = \frac{1}{1+\varepsilon} = \frac{\mathrm{OPT}}{(1+\varepsilon)\,\mathrm{OPT}} \leq \frac{\mathrm{OPT}}{v^*} \leq 1.$$

In the following, we show the existence of an algorithm that packs all rectangles of a scaled instance into a strip of height at most $(1+\varepsilon+1/2)$ (see section 4.5). This height bound is sufficient to prove Theorem 1, since rescaling yields

$$\begin{aligned} v^*\left(1+\varepsilon+\frac{1}{2}\right) &\leq (1+\varepsilon)\,\mathrm{OPT}\left(\frac{3}{2}+\varepsilon\right) = \left(\frac{3}{2}+\frac{5\varepsilon}{2}+\varepsilon^2\right)\mathrm{OPT} \\ &\leq (1.5+4\varepsilon)\,\mathrm{OPT}. \end{aligned} \tag{1}$$

In the following, we assume that the instance is already scaled such that an optimal packing P^* has height $\mathrm{h}(P^*)$, where $(1-\varepsilon) < \mathrm{h}(P^*) \leq 1$.

4.1.2 Partitioning the Set of Rectangles/Creating a Gap

In this section, we create a gap between *tall* and *low* rectangles and between *wide* and *narrow* rectangles. To create this gap we need to remove some of the rectangles. The following lemma proves that the rectangles we remove have small total area.

Let ε' be the largest value of the form $\varepsilon' = 1/(2a)$ for an integer a such that $\varepsilon' \leq \varepsilon/15$. Let $\sigma_0 := 1, \sigma_1 := \varepsilon'$, and $\sigma_k := (\sigma_{k-1})^{8/\sigma_{k-1}^3}$ for all $k \geq 2$. Define

$$L^{>1/2} := \left\{R_i \in L \mid h_i > \frac{1+2\varepsilon'}{2}\right\},$$

and

$$L_k := \left\{R_i \in L \setminus L^{>1/2} \mid w_i \in (\sigma_k, \sigma_{k-1}] \text{ or } h_i \in (\sigma_k, \sigma_{k-1}]\right\}.$$

Define for each subset $L' \subseteq L$ the total area of L' by $\mathrm{A}(L') = \sum_{R_i \in L'}(w_i \cdot h_i)$.

Lemma 4. *There exists $k \in \{2, \ldots, 2/\varepsilon'+1\}$ such that*

$$A(L_k) \leq \varepsilon' A(L).$$

Proof. Since each rectangle belongs to at most two sets L_k,

$$\sum_{j \in \{2,\ldots,\frac{2}{\varepsilon'}+1\}} A(L_j) \leq 2A(L).$$

Obviously, there exists $k \in \{2, \ldots, 2/\varepsilon'+1\}$ with $A(L_k) \leq \varepsilon'/2 \cdot 2A(L) = \varepsilon' A(L)$, since otherwise

$$\sum_{j \in \{2,\ldots,\frac{2}{\varepsilon'}+1\}} A(L_j) > \frac{2}{\varepsilon'}\varepsilon' A(L) = 2A(L).$$

□

Choose the smallest value k satisfying the conditions of Lemma 4 and define $\delta := \sigma_{k-1}$ and $s := 8/\delta^3$ and $\gamma := \delta^s = \sigma_k$.

Note 5. *Since $\frac{1}{\varepsilon'}$ is integral, $\frac{1}{\sigma_i}$ is integral for all $i \in \mathbb{N}$ and thus δ and γ, δ^s are integral.*

For simplicity, we define the following sets and call rectangles belonging to each set accordingly

$$\begin{aligned} L_{\text{ta}} &:= \{R_i \in L \mid h_i > \delta\} && \text{tall rectangles} \\ L_{\text{lo}} &:= \left\{R_i \in L \mid h_i \leq \delta^s\right\} && \text{low rectangles} \\ L_{\text{wi}} &:= \{R_i \in L \mid w_i > \delta\} && \text{wide rectangles} \\ L_{\text{na}} &:= \left\{R_i \in L \mid w_i \leq \delta^s\right\} && \text{narrow rectangles.} \end{aligned}$$

Note that $L = (L_{\text{ta}} \cup L_{\text{lo}} \cup L_{\text{wi}} \cup L_{\text{na}}) \cup L_k$ and $(L_{\text{ta}} \cup L_{\text{lo}} \cup L_{\text{wi}} \cup L_{\text{na}}) \cap L_k = \emptyset$ and $L^{>1/2} \subseteq L_{\text{ta}}$. We will denote the subset of low-wide rectangles in the following with $L_{\text{lo-wi}} := L_{\text{lo}} \cap L_{\text{wi}}$. For the following steps, we discard the *middle-sized* rectangles L_k. They will be packed in a post-processing step by a simple greedy algorithm (see section 4.5).

4.1.3 Rounding and Shifting Tall Rectangles

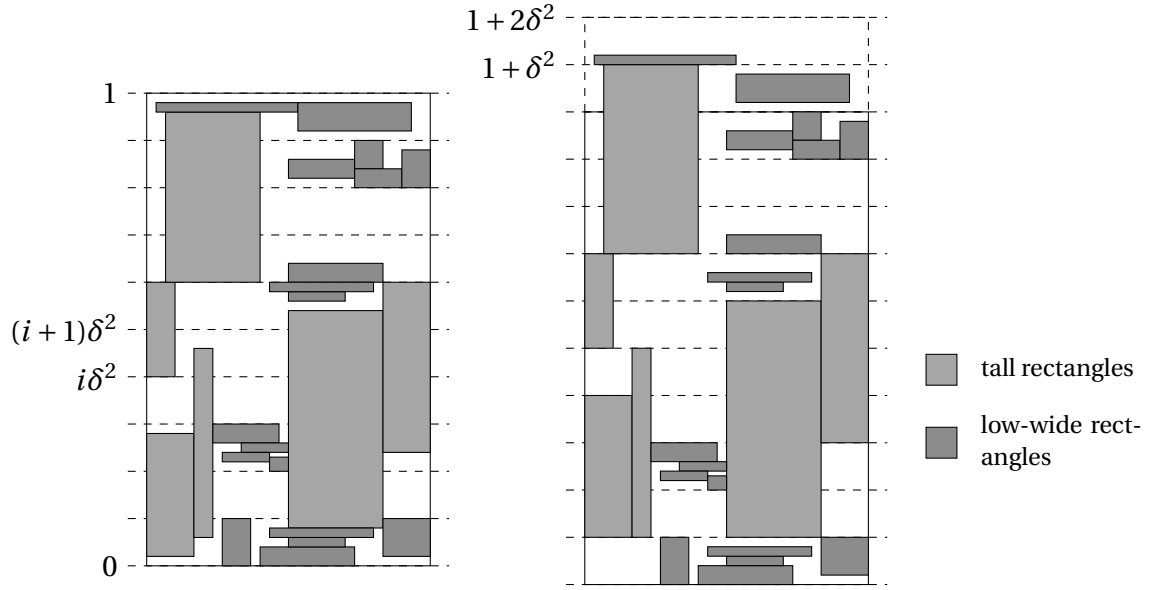
A crucial part of our simple structure are the positions and heights of the tall rectangles. Let P be an optimal packing for all rectangles L. First we increase the height of each tall rectangle $R_i \in L_{\text{ta}}$ to the nearest (integral) multiple of δ^2. Then, we shift the rectangles up such that all rectangles $R_i \in L_{\text{ta}}$ have their corners placed at points (x'_i, y'_i), such that there exist integral values k_i with $x'_i = x_i$ and $y'_i = k_i\delta^2$ (see figure 2). These modifications increase the height of the solution by at most 2δ.

Lemma 6. *Let P be a packing for all rectangles L with $\mathrm{h}(P) \leq 1$. At the cost of an increase in height of at most 2δ we can round up all tall rectangles to the nearest multiple of δ^2 and we can shift the rectangles such that the lower left corner of all tall rectangles is a multiple of δ^2.*

Proof. Let P be a packing for all rectangles L with $\mathrm{h}(P) \leq 1$. Let (x_i, y_i) be the position that P assigns to each rectangle R_i and let $z_i := y_i + h_i$ be the upper bound of R_i in P. Multiply each z_i by $1+2\delta$; that is, we shift up all rectangles depending on their upper bound without changing their size or the feasibility of the packing. Obviously, this modification increases the height of the packing by at most $2\delta\,\mathrm{h}(P) \leq 2\delta$. Since tall rectangles have height at least δ this shifting creates a gap with height at least

$$z_i(1+2\delta) - z_i = z_i 2\delta \geq 2\delta^2$$

below each tall rectangle. Thus, rounding up the size of each tall rectangle to the next multiple of δ^2 (without changing z_i) and shifting down each tall rectangle to a multiple of δ^2 does not change the feasibility of the packing. □



(a) Original packing (b) Packing with simpler structure

Figure 2: Rounding and shifting rectangles

After scaling and rounding the set of tall rectangles can be partitioned into a constant number of subsets. Define

$$I_{\text{ta}} := \left\{ \frac{1}{\delta} + i \mid i \in \mathbb{N} : 1 \leq i \leq \frac{1-\delta}{\delta^2} \right\} \quad \text{and} \tag{2}$$

$$I^{>1/2} := \left\{ \frac{1+\varepsilon'}{2\delta^2} + i \mid i \in \mathbb{N} : 1 \leq i \leq \frac{1-2\varepsilon'}{2\delta^2} \right\} \quad \text{and} \tag{3}$$

$$L(i) := \left\{ R_i \in L \mid h_i = i \cdot \delta^2 \right\}. \tag{4}$$

Lemma 7. *The set of all tall rectangles can be partitioned into a constant number of subsets:*

$$L_{ta} = \dot{\bigcup_{i \in I_{ta}}} L(i) \quad \textit{and} \tag{5}$$

$$L^{>1/2} = \dot{\bigcup_{i \in I^{>1/2}}} L(i). \tag{6}$$

In particular, the number of partitions of L_{ta} and $L^{>1/2}$ are bounded by $|I_{ta}| = \frac{1-\delta}{\delta^2} \leq \frac{1}{\delta^2}$ and $|I^{>1/2}| = \frac{1-2\varepsilon'}{2\delta^2} \leq \frac{1}{2\delta^2}$, respectively.

Proof. Obviously for all $i, j \in I_{\text{ta}}, i \neq j : L(i) \cap L(j) = \emptyset$. Due to the scaling, the height of each rectangle is at most 1 and since $1/\delta$ is integral (see Note 5), this bound still holds after rounding. After rounding, each tall rectangle has height $a\delta^2$ for an integer value a. Let $R_i \in L_{\text{ta}}$. Then $\delta < \text{h}(R_i) \leq 1$ and since δ is a multiple of δ^2, the smallest possible value for $\text{h}(R_i)$ is

$$\delta + \delta^2 = \left(\frac{1}{\delta} + 1 \right) \delta^2.$$

The biggest possible value for $h(R_i)$ is

$$1 = \delta + 1 - \delta = \frac{\delta^2}{\delta} + \frac{(1-\delta)\delta^2}{\delta^2} = \left(\frac{1}{\delta} + \frac{1-\delta}{\delta^2}\right)\delta^2.$$

Since all multiples of δ^2 between these bounds are contained in I_{ta},

$$L_{\text{ta}} = \dot{\bigcup_{i \in I_{\text{ta}}}} L(i).$$

If $R_i \in L^{>1/2}$ then $\frac{1+2\varepsilon'}{2} < h(R_i) \leq 1$. Since $\frac{1}{\varepsilon'}$ is an even integer and δ^2 is a multiple of ε', $\frac{1+2\varepsilon'}{2}$ is a multiple of δ^2. The smallest possible value for $h(R_i)$ is

$$\frac{1+2\varepsilon'}{2} + \delta^2 = \left(\frac{1+2\varepsilon'}{2\delta^2} + 1\right)\delta^2.$$

The biggest possible value for $h(R_i)$ is

$$1 = \frac{1+2\varepsilon'+1-2\varepsilon'}{2} = \left(\frac{1+2\varepsilon'}{2\delta^2} + \frac{1-2\varepsilon'}{2\delta^2}\right)\delta^2.$$

Again, since all multiples of δ^2 between these bounds are contained in $I^{>1/2}$,

$$L^{>1/2} = \dot{\bigcup_{i \in I^{>1/2}}} L(i). \qquad \square$$

4.1.4 Containers for Low Rectangles

Since we want to increase only the height but not the width of the packing, we cannot round up the widths of the wide rectangles in order to reduce the complexity. Instead we introduce *containers*, into which all low-wide ($L_{\text{lo-wi}}$) and a subset of the low-narrow rectangles will be packed.

Consider a scaled and shifted packing P. Draw horizontal lines spaced by a distance δ^2 across the strip (due to the rounding and shifting, the lower and upper sides of the tall rectangles coincides with two of these lines). These lines split the strip into at most $(1+2\delta)/\delta^2$ horizontal rectangular regions that we call *slots* (see figure 2). A container is a rectangular region inside a slot whose left boundary is either the right side of a tall rectangle or the left side of the strip, and whose right boundary is either the left side of a tall rectangle or the right side of the strip. In the following, we consider only containers that contain at least one low-wide rectangle. In figure 3 for example, we have two containers that contain low-wide rectangles.

Lemma 8. *Let P be a scaled and shifted packing for all rectangles L. The number of containers which contain at least one low-wide rectangle is bounded by $\frac{2}{\delta^3}$.*

In particular, the number of all possible sets of containers (containing at least one low-wide rectangle) is polynomial in n.

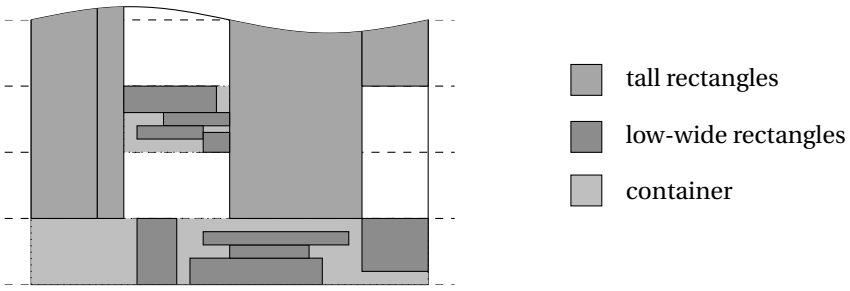


Figure 3: Container for low rectangles

Proof. The height of each container is δ^2 by definition. Since each container in consideration contains at least one low-wide rectangle the width of each container is at least δ. Thus, the total number of containers (containing at least one low-wide rectangle) is bounded by

$$(1+2\delta)\frac{1}{\delta\cdot\delta^2} \leq 2\frac{1}{\delta^3}.$$

Furthermore, the width of each container is a multiple of $1/m$, since in each packing (schedule) x_i is a multiple of $1/m$ for each rectangle R_i and the width w_i is a multiple of $1/m$. Thus, the width of each container is in

$$\left\{\frac{1}{m},\ldots,\frac{m}{m}\right\}.$$

Therefore, a rough upper bound for the number of different sets of containers is $(m+1)^{\frac{2}{\delta^3}}$ (encode each set as a $\frac{2}{\delta^3}$-tupel, where each entry denotes the width of the corresponding container or 0 if it is not in the set). Note that in general this number will be (much) smaller, since the ordering is not relevant and the width of each container containing a low-wide rectangle is $> 1/\delta$. Since we assume that m is polynomial in n, the number of all possible sets of containers (containing at least one low-wide rectangle) is polynomial in n. □

Since the number of different sets of containers is polynomial, we can find a set corresponding to the set induced by an optimal packing by enumerating all possible sets of containers in polynomial time.

4.1.5 Properties / Summary

In summary, we have shown in this section that an optimal packing P for rectangle set L with height bounded by 1 can be transformed into a packing $\hat{P}$ for rectangle set $\hat{L}$ with height at most $1+2\delta$ and simple structure, as follows

(a) every tall rectangle R_i in $\hat{L}$ has its height rounded up to the nearest multiple of δ^2 and its lower border is at a position y_i that is a multiple of δ^2 (see Lemma 6),

(b) each container C containing at least one low-wide rectangle has height δ^2 and width $i\cdot 1/m \geq \delta$ where $i \leq m$ is a non-negative integer (see section 4.1.4),

(c) there is in gap in size between tall and low rectangles and between wide and narrow rectangles (see Lemma 4),

(d) the total area of the discarded rectangles is bounded, $A(L \setminus \hat{L}) \leq \varepsilon'/2$ and the height of each discarded rectangle is bounded by $(1+2\varepsilon')/2$, since we did not discard any rectangles belonging to $L^{>1/2}$ (see section 4.1.2).

4.2 Pre-Positioning

The next step is to determine the positions of the containers and a subset of the tall rectangles. On the one hand we have to make sure that all rectangles we are discarding have height bounded by $1/2$; otherwise, the NFDH algorithm used to pack all discarded rectangles during post-processing produces a packing of height $> 1/2$, leading to an overall approximation ratio greater than $1.5+\varepsilon'$. On the other hand we have to make sure that the pre-positioning has a polynomial running time. In particular, we can only enumerate the positions of a constant number of tall rectangles and containers.

From here on let $\mathscr{C}$ be the (current) set of containers and let $L'_{\text{ta}} \subseteq L_{\text{ta}} \setminus L^{>1/2}$ be the subset of K tall rectangles with largest area for some constant K, which we will define in section 4.4.4; we set $L'_{\text{ta}} := L_{\text{ta}} \setminus L^{>1/2}$ if $|L_{\text{ta}} \setminus L^{>1/2}| \leq K$. Furthermore, let

$$L' = \mathscr{C} \cup L'_{\text{ta}}$$

be the union of the set of containers and the chosen subset of at most K tall rectangles. Note that since $|\mathscr{C}| \leq 2\delta^{-3}$ (see Lemma 8),

$$|L'| \leq K + 2\delta^{-3}. \tag{7}$$

In order to determine the positions of the rectangles from L', first we guess (enumerate) assignments of the K tall rectangles L'_{ta} and of the containers $\mathscr{C}$ to *slots* and *snapshots*. Then we describe a dynamic program that assigns the tall rectangles from $L^{>1/2}$ to snapshots without enumerating all possibilities, since there might be too many of them. Using these assignments we set up a linear program (LP). If this LP has a solution, we have found a fractional solution for the packing problem. Furthermore, if almost all low-wide rectangles fit into the containers, we show that the fractional solution can be transformed into a feasible integral solution by discarding some rectangles with small total area.

4.2.1 Slot Assignment

We split again the strip into horizontal slots of height δ^2. A slot assignment for L' is a mapping $f : L' \to M$ where $M = \{1, \ldots, (1+2\delta)/\delta^2\}$ corresponds to the set of slots. For a given slot assignment f the set of slots that will be used for packing a rectangle $R_j \in L'$ is given by $\{f(R_j), \ldots, f(R_j)+\gamma_j - 1\}$, where $\gamma_j \delta^2 = h_j$ is the height of rectangle R_j (in particular $\gamma_j \geq 1/\delta$ for each $R_i \in L_{\text{ta}}$, since $h_j > \delta$, and $\gamma_j = 1$ if R_j is a container). Since the number of different mappings f is bounded by

$$|M|^{|L'|} \leq \left(\frac{1+2\delta}{\delta^2}\right)^{K+2\delta^{-3}} \tag{8}$$

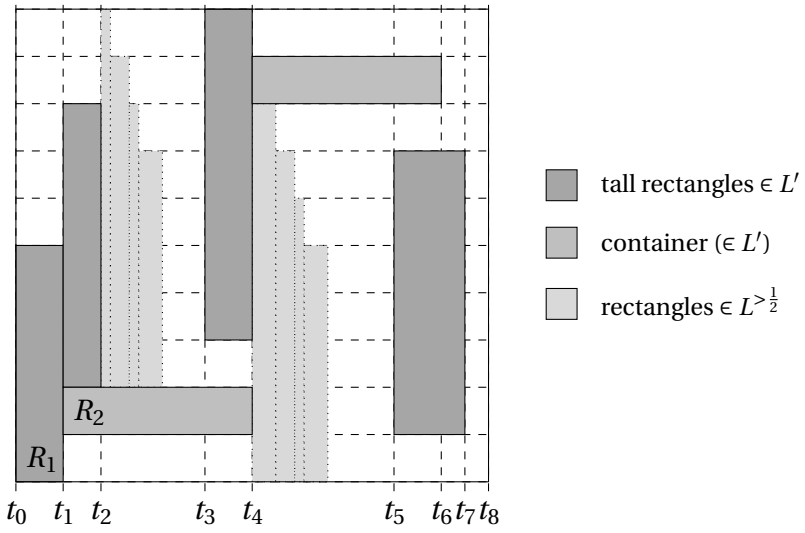


Figure 4: Packing of rectangles and containers and induced snapshots

and thus constant, we can consider all mappings f in polynomial time and try to find a packing for L that is consistent with f for each mapping.

4.2.2 Snapshots

In order to handle the x position of the rectangles, we introduce snapshots. We use the snapshots to model the relative horizontal positions of all rectangles in L'.

Consider a packing for L'. Trace vertical lines extending the sides of the rectangles in L' (see figure 4). The region between two adjacent lines is called a snapshot. If we index all snapshots from left to right, every rectangle $R_j \in L'$ appears in a sequence of consecutive snapshots $S_{\alpha_j}, \ldots, S_{\beta_j}$, where α_j denotes the index of the first snapshot in which rectangle R_j occurs and β_j denotes the index of the last snapshot. In figure 4 for example rectangle R_1 is contained in snapshot S_1, while R_2 is contained in snapshots S_2, S_3, S_4, thus $\alpha_1 = 1, \beta_1 = 1, \alpha_2 = 2$ and $\beta_2 = 4$. More formal, an assignment of all rectangles in L' to snapshots is given by two functions $\alpha, \beta : L' \to \{1, \ldots, g\}$, where g denotes the number of snapshots.

Since $|L'| \leq K + 2\delta^{-3}$ (Equation (7)) the maximum number of snapshots g in any packing for L' is at most

$$g \leq 2|L'| \leq 2(K + 2\delta^{-3}), \tag{9}$$

and thus the number of different assignments of L' to snapshots is polynomial, $O(g^{2|L'|})$.

4.2.3 Dynamic Program for $L^{>1/2}$-rectangles

In general, we cannot consider all assignments of rectangles in $L^{>1/2}$ to snapshots, because there might be up to n rectangles in $L^{>1/2}$. In the following, we introduce an algorithm that allows us to enumerate a subset of all snapshot assignments for $L^{>1/2}$ such that the size of

the subset is polynomially bounded in n and there exists one snapshot assignment in this subset that is *equivalent* to a snapshot assignment induced by an optimal packing.

Rectangles in $L^{>1/2}$ that intersect more than one snapshot are handled separately (see end of section), since our packing algorithm can only be used for tall rectangles that do not intersect more than one snapshot.

As already mentioned in section 4.1.3, $L^{>1/2}$ can be partitioned into sets $L(i)$ for all $i \in I^{>1/2}$ such that $L^{>1/2} = \dot{\bigcup}_{i \in I^{>1/2}} L(i)$ (see Equation (3) and (6)) and

$$|I^{>1/2}| = \frac{1-2\varepsilon'}{2\delta^2} < \frac{1}{2\delta^2}. \tag{10}$$

Consider a packing of all rectangles and snapshots as defined above. Then we can define a vector $v^i = (v^i_1, \ldots, v^i_g)$ for each height $i\delta^2, i \in I^{>1/2}$, where $v^i_j \in \{0, \ldots, m\}$ is chosen such that $v^i_j \cdot 1/m$ is the sum of widths of all rectangles of height $i\delta^2$ contained in snapshot S_j. Obviously,

$$\sum_{i \in I^{>1/2}} \sum_{j=1}^{g} v^i_j \frac{1}{m} = \sum_{R_i \in L^{>1/2}} w_i \leq 1 \tag{11}$$

(otherwise the packing is not feasible) and thus, we have

$$v^i_j \leq m \tag{12}$$

for each $i \in I^{>1/2}$, $j \in \{1, \ldots, g\}$.

With a dynamic programming approach we can compute a list of all feasible vectors satisfying (11) and (12). A rough upper bound for the number of feasible vectors for each height $i\delta^2$ is given by $(m+1)^g$ since there are g components and every component $v^i_j \in \{0, \ldots, m\}$. The algorithm to calculate all feasible vectors for a given height $i\delta^2$ works as follows.

Assume that $L(i) = \{R_1, \ldots, R_{k_i}\}$. Starting with a set $V := \{(0, \ldots, 0)\}$ containing only the null vector we replace in step $l \in \{1, \ldots, k_i\}$ each vector $v \in V$ with all vectors that can be generated by adding $\gamma_l := w_l \cdot m$ to one of its components. To ensure that the number of vectors is bounded by $(m+1)^g$, we discard any vector that equals (componentwise) an already added vector. This can be done efficiently by keeping the list sorted (for example in lexicographical order). Since in each step at most $g(m+1)^g$ vectors are generated and the number of operations used for the insertion (insertion sort with binary search) is bounded by $\log((m+1)^g) = g\log(m+1)$, the number of operations for each step is bounded by

$$g(m+1)^g \cdot g\log(m+1) \leq g^2(m+1)^{g+1}.$$

Thus, the number of operations for each height $i \in I^{>1/2}$ is bounded by

$$k_i \cdot g^2(m+1)^{g+1} \leq m \cdot g^2(m+1)^{g+1} \leq g^2(m+1)^{g+2},$$

since

$$k_i = |L(i)| \leq |L^{>1/2}| \leq m.$$

Let V^i denote the set of vectors generated for this height class $L(i)$. Obviously, the vector induced by a given packing can be found among the generated vectors.

Repeating this computation for every $i \in I^{>1/2}$ leads to $|I^{>1/2}| \leq 1/(2\delta^2)$ sets of at most $(m+1)^g$ vectors. We build the direct product $V := \bigtimes_{i \in I^{>1/2}} V^i$ of these sets. V contains at most

$$|V| \leq ((m+1)^g)^{|I|} \leq ((m+1)^g)^{\frac{1}{2\delta^2}} \tag{13}$$

elements and each of these elements consists of one vector for each height class. One element $v \in V$ corresponds to the vectors induced by the given packing. In our packing algorithm we guess an element $v \in V$ consisting of components $v^i, i \in I^{>1/2}$ and use these vectors v^i to pack the tall rectangles into the snapshots. Note that in practice we do not need the direct product, we can simply enumerate all elements in an arbitrary order. We use this notation only for convenience.

Using the dynamic program results in (many) vectors of widths only. However, for our packing algorithm we need to know what combination of rectangles leads to the given width per snapshot. This can be achieved by extending the dynamic program such that for each vector a component consists not only of the current width, but also of a set of rectangles. During the vector generation step, a rectangle is added to this set if its width is added to the corresponding width component. Due to this modification the space needed to store the vectors increases but is still polynomial in n; the running time of the dynamic program is not affected significantly.

In order to handle rectangles intersecting snapshot boundaries, we simply guess the subset $\hat{L} \subseteq L^{>1/2}$ of rectangles intersecting snapshot boundaries, which can be done in polynomial time since there are at most g of these rectangles. In order to pack these rectangles we add them to L' (the set of K tall rectangles and containers). This modification increases the size of L' such that

$$|L'| \leq 3 \cdot (K + 2\delta^{-3}), \tag{14}$$

and of g such that

$$g \leq 6(K + 2\delta^{-3}). \tag{15}$$

Note that this modification does not increase the dimension of the vectors we defined above, since the snapshots introduced by these added rectangles obviously do not allow further $L^{>1/2}$ rectangles to be packed in them (height $> 1/2$).

As stated above, among all generated vectors there is one that is equivalent to the vector induced by a nearly optimal schedule with simpler structure. They are equivalent in the sense that the total width in each component is in both vectors the same. For our algorithm this is sufficient, since our packing algorithm ensures that all rectangles assigned to one component are packed next to each other (see sections 4.4.1, 4.4.2).

4.3 Linear Program

In this subsection, we present a linear program (LP), which allows us to calculate the width of all snapshots, and thus determine the positions of all rectangles in $L' = \mathscr{C} \cup L'_{\text{ta}}$. We

now assume that we have chosen a slot assignment f (see section 4.2.1), functions α, β (see section 4.2.2), and $v \in V$ consisting of vectors v^i of widths for each height class as described in section 4.2.3.

Since all low-wide rectangles and a subset of the low-narrow rectangles get packed into the containers, we do not need to consider them in the LP. For convenience we call the subset of the low-narrow rectangles packed into the containers $L^C_{\text{lo-na}}$, and the remaining low-narrow rectangles $L_{\text{lo-na}}$. We construct $L^C_{\text{lo-na}}$ by greedily adding low-narrow rectangles as long as

$$\mathrm{A}(L^C_{\text{lo-na}}) + \mathrm{A}(L_{\text{lo-wi}}) \leq \mathrm{A}(\mathscr{C}). \tag{16}$$

We discard the first low-narrow rectangle that exceeds the total area in order to ensure that enough space can be reserved for the remaining rectangles in the following LP. This discarded rectangle has an area of at most δ^{2s}. In fact, we will show that this discarded rectangle can be packed along with the rectangles from $L_{\text{lo-na}}$ (see section 4.4.5).

Since f, α, β are fixed, we can calculate the set of *free* slots (i.e. the slots not occupied by L' rectangles) for each snapshot. These free slots will be used for the remaining tall rectangles and for the small rectangles from $L_{\text{lo-na}}$. In order to formulate constraints to ensure that enough space is reserved for these rectangles, we introduce configurations. We define a configuration as a pair (SN, Π) where SN is a subset of the free slots reserved for rectangles from $L_{\text{lo-na}}$ and Π is a partition of the remaining free slots into sets of consecutive slots reserved for rectangles from $L_{\text{ta}} \setminus L'$; every subset $F \in \Pi$ of cardinality $l = |F|$ is reserved to pack rectangles from L_{ta} of height $l\delta^2$. Let n_j denote the number of different configurations for each snapshot S_j and let $c^j_i := (\mathrm{SN}^j_i, \Pi^j_i)$ denote the different configurations for snapshot S_j, $i \in \{1, \ldots, n_j\}$ and let $n^j_i(\ell) := |\{F \in \Pi^j_i : |F| = \ell\}|$ denote the number of sets of cardinality ℓ in Π^j_i for each $\ell \in I_{\text{ta}}$. The total width of all rectangles in $L_{\text{ta}} \setminus L'$ of height ℓ is denoted as W_ℓ. The variables $x^j_i, j \in \{1, \ldots, g\}, i \in \{1, \ldots, n_j\}$ are used to determine the width of each configuration c^j_i. Additional variables $t_j, j \in \{1, \ldots, g\}$ are used to determine the width of each snapshot S_j.

$$
\begin{aligned}
\mathrm{LP}(f, \alpha, \beta, v): \quad & t_0 = 0, t_g \leq 1 && && \\
& t_j \geq t_{j-1} && \forall j \in \{1, \ldots, g\} && \\
& t_{\beta_j} - t_{\alpha_j} = w_j && \forall R_j \in L' && (17) \\
& \sum_{i=1}^{n_j} n^j_i(\ell) x^j_i \geq \frac{1}{m} v^\ell_j && \forall j \in \{1, \ldots, g\}, \ell \in I^{>1/2} && (18) \\
& \sum_{j=1}^{g} \sum_{i=1}^{n_j} n^j_i(\ell) x^j_i \geq W_\ell && \forall \ell \in I_{\text{ta}} \setminus I^{>1/2} && (19) \\
& \sum_{j=1}^{g} \sum_{i=1}^{n_j} x^j_i |\mathrm{SN}^j_i| \delta^2 \geq \mathrm{A}(L_{\text{lo-na}}) && && (20) \\
& \sum_{i=1}^{n_j} x^j_i \leq t_j - t_{j-1} && \forall j \in \{1, \ldots, g\} && (21) \\
& x^j_1, \ldots, x^j_{n_j} \geq 0 && \forall j \in \{1, \ldots, g\} &&
\end{aligned}
$$

Constraint (17) ensures that the width of the snapshots corresponds to the width of the assigned pre-positioned rectangles or containers. Constraint (18) makes sure that the total width of all configurations in each snapshot is greater or equal than the width needed for packing the rectangles from $L^{>1/2}$ as given by the vector v. Similarly, constraint (19) ensures that the chosen configurations reserve enough space to pack all rectangles from $L_{\text{ta}} \setminus L'$ (at least fractionally). Constraint (20) ensures that enough space is reserved for (fractionally) packing the rectangles from $L_{\text{lo-na}}$. Constraint (21) makes sure that the width of all configurations for a snapshot does not exceed the width of that snapshot.

Since $g, n_j, |L'|, |J|, |I|$ are independent of n, this linear program can be solved in polynomial time. If LP(f, α, β, v) has no feasible solution, we construct a new LP with a new combination of $\mathscr{C}, f, \alpha, \beta, v$.

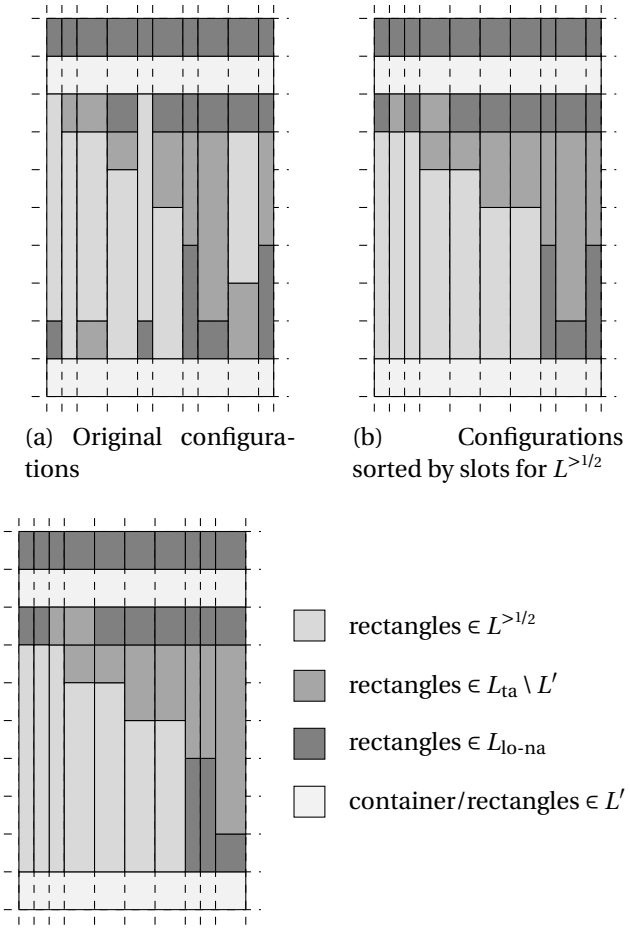
4.4 Packing the Rectangles

Let (t^*, x^*) be a feasible solution for LP(f, α, β, v). For simplicity, we remove all snapshots $[t_j^*, t_{j+1}^*)$ of zero width and combine all snapshots that do not contain any L' rectangles, i.e. the set of free slots is $F = M$ (remember that M corresponds to the set of all slots), as the last snapshot. Obviously the modified solution is still feasible. Let g^* denote the number of resulting snapshots.

Since we solve the LP fractionally, the solution might contain configurations with widths that are not multiples of $1/m$. Nevertheless, in the following we pack the rectangles using this fractional solution. If the resulting packing is not feasible, we can add a simple post-processing step in which we shift all rectangles beginning with the leftmost, bottommost infeasible rectangle R_i (position (x_i, y_i)), such that x_i is a multiple of $1/m$. This shifting is possible since all rectangles which are positioned left of R_i start at a feasible position and have a width that is a multiple of $1/m$. Repeating this shifting step for each infeasible rectangle leads to a feasible packing.

4.4.1 Adapting and Sorting the Configurations

Before we start packing the rectangles, we sort and modify the configurations inside each snapshot (see figure 5). The objective is to make sure that on the one hand no rectangles from $L^{>1/2}$ get split and, on the other hand that the fragmentation of the slots reserved for the low-narrow rectangles is *limited*. In each snapshot, we first sort all configurations based on the number of slots reserved for rectangles from $L^{>1/2}$. After this sorting, all configurations reserved for rectangles from $L^{>1/2}$ of the same height appear next to each other. Note that the sorting is well-defined, since in each configuration at most one subset of contiguous slots is reserved for rectangles from $L^{>1/2}$, and they cannot appear on top of each other (height $> (1+2\varepsilon')/2$). Furthermore, we need to modify the configurations such that the slots reserved for rectangles from $L^{>1/2}$ of the same height are the same. In each snapshot all configurations cover the same subset of slots (all slots save the slots occupied by the pre-positioned rectangles from L') and obviously, there is at most one contiguous subset Π' of these covered slots that can contain slots reserved for rectangles from $L^{>1/2}$. Now we modify each configuration that contains a subset $\Pi'' \subseteq \Pi'$ reserved for rectangles from $L^{>1/2}$, by



(a) Original configurations

(b) Configurations sorted by slots for $L^{>1/2}$

(c) Configurations sorted by slots for $L^{>1/2}$ and $L_{\text{lo-na}}$

Figure 5: Adapting configurations

shifting Π'' *inside* Π' as far down as possible. The resulting configuration does not change the solution of our LP, since no assumptions about the locations of the reserved slots are made. But now packing the $L^{>1/2}$ rectangles next to each other will not lead to splittings (see figure 5).

To limit the fragmentation of the slots reserved for low-narrow rectangles, we place configurations with the same set SN_i^j of slots reserved for low-narrow rectangles next to each other, but without disturbing the previous sorting.

4.4.2 Packing Pre-Positioned Rectangles

Each rectangle $R_i \in L'$ is placed in the slots assigned by function f such that its left side is at distance $t^*_{\alpha_i}$ from the left side of the strip, i.e. the position (lower left corner) for R_i is given by $(t^*_{\alpha_i}, f(R_i)\delta^2)$. In particular, no rectangle from L' is split in this process.

The next step is to pack the $L^{>1/2}$ rectangles. Since the solution is feasible, in each snapshot S_j the widths of the configurations that are reserved for the $L^{>1/2}$ rectangles are at least

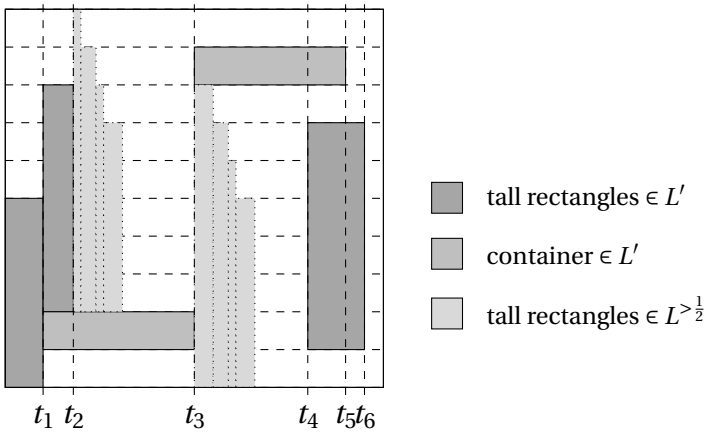


Figure 6: Packing of pre-positioned rectangles

as large as the widths given by vector v^j (see constraint (18)). Furthermore, due to the ordering of the configurations within each snapshot, we can simply pack all these rectangles next to one other according to the configurations. Due to the ordering described in the previous subsection (section 4.4.1), the packing of the tall rectangles and the containers has a structure as in figure 6.

4.4.3 Tall Rectangles

The next step is to pack the remaining tall rectangles. Let $\mathscr{R}_l = \{R_{l,1}, \ldots, R_{l,n_l}\} = L(l) \setminus L' \subseteq L_{\text{ta}} \setminus L'$ be the rectangles of height $l\delta^2$ for every $l \in I_{\text{ta}} \setminus I^{>1/2}$. Take the first configuration $c_i^j = (\text{SN}_i^j, \Pi_i^j)$ in the above ordering with $x_i^j > 0$ and select for each set $X \in \Pi_i^j$ with $l = |X|$ successively the first not yet completely packed rectangle $R \in \mathscr{R}_l$. These rectangles are packed within the slots X starting at position $x(c_i^j)$ until their total width is at least x_i^j or all rectangles in $\mathscr{R}_l$ are packed. If the total width is greater than x_i^j the last rectangle is split such that the width is exactly x_i^j. Repeating the packing process for each configuration in each snapshot leads to a fractional packing of all tall rectangles, since (19) ensures that there is sufficient space reserved for them.

This fractional packing of tall rectangles allows a certain number of tall rectangles to get split. In the following, we show that the number of these split rectangles is bounded and that the total area of these rectangles is at most δ.

The splitting of rectangles is caused by the transition from one subset X of slots to the next as described above. Thus, the number of split rectangles is bounded by the number of subsets of slots in each configuration times the number of configurations per snapshot times the number of snapshots. The number of subsets of slots in each configuration is bounded by $|M|$ (M corresponds to the set of all slots). For each snapshot there are at most $(2|M|)^{|M|}$ different configurations, since the number of all subsets of the set of slots is $2^{|M|}$ and the number of partitions of M is bounded by $|M|^{|M|}$. In total, this leads to at most $g^*|M|(2|M|)^{|M|}$ divided rectangles. Note that the number of chosen configurations ($x_i^j \neq 0$)

is bounded by the number of constraints in the LP. Thus, the number of split rectangles is possibly much lower. Since $g^* \leq g \leq 6(K + 2\delta^{-3})$ (see Equation (15)), the number of split rectangles is bounded by

$$g^*|M|(2|M|)^{|M|} \leq 6(K + 2\delta^{-3})|M|(2|M|)^{|M|}. \tag{22}$$

With this bound for the number of split tall rectangles, we only have to choose the constant K such that the total area of split tall rectangles is bounded by 2δ, which will be done in the next subsection.

4.4.4 Choosing Constant K

In order to choose a constant K such that the total area of the split tall rectangles is bounded, we use a slightly modified version of a result by Jansen and Porkolab [14, Lemma 2.5].

Lemma 9 ([14])**.** *Suppose $d_1 \geq d_2 \geq \cdots \geq d_n \geq 0$ is a sequence of real numbers and $D = \sum_{j=1}^{n} d_j$. Let p, q be nonnegative integers, $\alpha > 0$, and assume that $n > (\lceil \frac{1}{\alpha} \rceil p + 1)(q+1)^{\lceil \frac{1}{\alpha} \rceil}$. Then, there exists an integer $k = k(p, q, \alpha)$ such that*

$$d_k + \cdots + d_{k+p+qk-1} \leq \alpha D$$

and

$$k \leq (q+1)^{\lceil \frac{1}{\alpha} \rceil - 1} + p\left(1 + (q+1) + \cdots + (q+1)^{\lceil \frac{1}{\alpha} \rceil - 2}\right). \tag{23}$$

Proof. Decompose the sum $d_1 + \cdots + d_n$ into blocks $B_0 = d_1 + \cdots + d_{f(1)-1}, B_1 = d_{f(1)} + \cdots + d_{f(2)-1}, \ldots B_i = d_{f(i)} + \cdots + d_{f(i+1)-1}$, where the function f is defined recursively by the following equation:

$$f(0) = 1, \quad f(i+1) = f(i) + p + q \cdot f(i). \tag{24}$$

Since $\sum_{j=1}^{n} d_j = D$, at most $\lceil \frac{1}{\alpha} \rceil - 1$ blocks are larger in size than $\alpha \cdot D$. Now let i be the smallest integer for which $B_i \leq \alpha D$. Then $i \leq \lceil \frac{1}{\alpha} \rceil - 1$, and $B_i = d_{f(i)} + \cdots + d_{f(i+1)-1} \leq \alpha \cdot D$. This implies that there is an index $k \leq f(i)$ such that $d_k + \cdots + d_{k+p+qk-1} \leq \alpha \cdot D$. It follows from (24) that

$$f(i) = (q+1)^i + p\left(1 + (q+1) + \cdots + (q+1)^{i-1}\right), \tag{25}$$

which along with the bound on i implies (23). □

We choose $d_j = w_j \cdot h_j$ for each $R_j \in L_{\text{ta}}$ (sorted by non-increasing area), and define $\alpha := \delta$, and $p := 6|M|2\delta^{-3}(2|M|)^{|M|}$, and $q := 6|M|(2|M|)^{|M|}$. Then $D \leq 1 + 2\delta$, since $\mathrm{A}(L) \leq 1 + 2\delta$ (see section 4.1.5). If $|L_{\text{ta}}| \leq (\lceil \frac{1}{\alpha} \rceil p + 1)(q+1)^{\lceil \frac{1}{\alpha} \rceil}$, we can add *dummy* rectangles with area 0. Note that the algorithm would also work without this modification, since the number of tall rectangles would be constant in this case and thus all tall rectangles could be prepositioned. However, this would make the following proofs more complicated.

Lemma 9 yields that there exists a constant K such that for each set $\hat{L} \subseteq L_{\text{ta}} \setminus \tilde{L}$ with $|\hat{L}| \leq p + qK$, the total area of $\hat{L}$ is bounded by $|\hat{L}| \leq \alpha \cdot D \leq \delta(1 + 2\delta) = \delta + 2\delta^2 \leq 2\delta$, if $\tilde{L} \subseteq L_{\text{ta}}$ contains the K rectangles with largest area. Furthermore,

$$K \leq (1 + 2\delta^{-3})(6|M|(2|M|)^{|M|})^{\frac{1}{\delta}-1}, \tag{26}$$

since

$$\begin{aligned}
k &\leq (q+1)^{\lceil \frac{1}{\alpha} \rceil - 1} + p\left(1 + (q+1) + \cdots + (q+1)^{\lceil \frac{1}{\alpha} \rceil - 2}\right) \\
&= (q+1)^{\lceil \frac{1}{\alpha} \rceil - 1} + p\frac{(q+1)^{\lceil \frac{1}{\alpha} \rceil - 1} - 1}{(q+1) - 1} \\
&\leq (q+1)^{\lceil \frac{1}{\alpha} \rceil - 1} + \frac{p}{q}(q+1)^{\lceil \frac{1}{\alpha} \rceil - 1} \\
&= (1 + \frac{p}{q})(q+1)^{\lceil \frac{1}{\alpha} \rceil - 1} \\
&= (1 + \frac{6|M|2\delta^{-3}(2|M|)^{|M|}}{6|M|(2|M|)^{|M|}})(6|M|(2|M|)^{|M|})^{\frac{1}{\delta}-1} \\
&= (1 + 2\delta^{-3})(6|M|(2|M|)^{|M|})^{\frac{1}{\delta}-1}.
\end{aligned}$$

Since the number of split rectangles is at most

$$\begin{aligned}
g|M|(2|M|)^{|M|} &\leq 6|M|(K + 2\delta^{-3})(2|M|)^{|M|} \\
&= 6|M|2\delta^{-3}(2|M|)^{|M|} + 6|M|(2|M|)^{|M|}K \\
&= p + qK
\end{aligned}$$

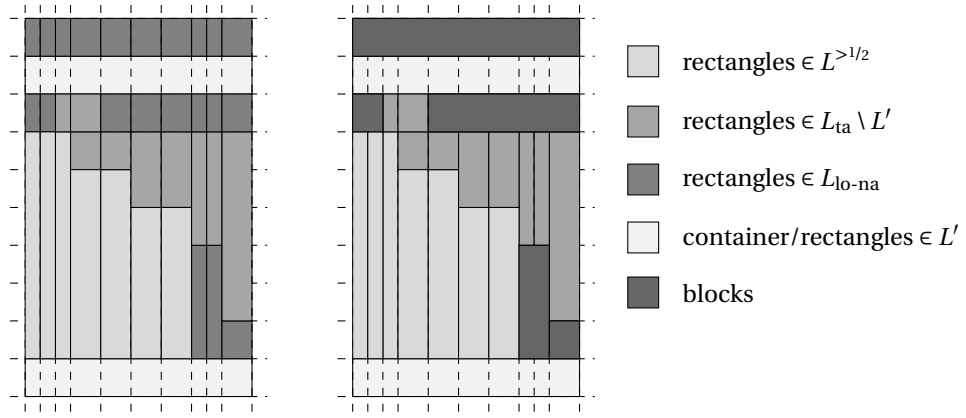
and the K rectangles with largest profit (L'_{ta}) are not split (see section 4.4.2), Lemma 9 yields that the total area of split rectangles is at most 2δ.

4.4.5 Packing the Low-Narrow Rectangles

The next step is to pack the subset $L_{\text{lo-na}}$ of the low-narrow rectangles that are not assigned to the containers. Due to the ordering, configurations c_i^j with the same set SN_i^j of slots reserved for low-narrow rectangles are adjacent (if possible). In the following, we combine adjacent reserved slots into blocks. Then, if we pack the $L_{\text{lo-na}}$ rectangles only into blocks of width at least $4\delta^{s-3}$, we can pack almost all rectangles, i.e. the remaining rectangles have a total area of at most δ. To be more specific, in each snapshot we define blocks $B_1, \ldots, B_l$ by combining all adjacent subsets $Y \subseteq M$ reserved for low-narrow rectangles $L_{\text{lo-na}}$ that occur in adjacent configurations (see figure 7). For example assume that a set of adjacent slots $Y \subseteq M$ occurs in adjacent configurations $c_i^j, c_i^{j+1}, c_i^{j+2}$, that is $Y \in \text{SN}_i^j = \text{SN}_i^{j+1} = \text{SN}_i^{j+2}$. Then we combine these reserved regions into a block B_k with width $x_i^j + x_i^{j+1} + x_i^{j+2}$ and height $|Y|\delta^2$. Then each block is a rectangular region and the height of each block is a multiple of δ^2.

Let B be a block with height $d\delta^2$ and width b. We select low-narrow rectangles to be packed into this block by adding rectangles to a set S until the total area of S is at least $d\delta^2 b$. Since each small rectangle has area at most δ^{2s} the total area of S is bounded by $d\delta^2 b + \delta^{2s}$. We pack the small rectangles into the block using the NFDH (Next Fit Decreasing Height) algorithm introduced by Coffman et al. [5]. We pack in each block with width at least $4\delta^{s-3}$ a subset $S' \subseteq S$ with $\text{A}(S') \geq \text{A}(S) - \delta\,\text{A}(S)$ (see figure 8), since

$$\text{A}(S') \geq \sum_{i=2}^{n^{(S)}} h_i(b - \delta^s)$$



(a) Sorted configurations

(b) Reserved space combined to blocks

Figure 7: Blocks

$$
\begin{aligned}
&\geq (b-\delta^s) \underbrace{\sum_{i=1}^{n^{(S)}} h_i}_{\geq d\delta^2-\delta^s} - \underbrace{h_1}_{\leq \delta^s} \\
&\geq (b-\delta^s)\big((d\delta^2-\delta^s)-\delta^s\big) \\
&= (b-\delta^s)\big(d\delta^2-2\delta^s\big) \\
&= d\delta^2 b - b2\delta^s - d\delta^2\delta^s + 2\delta^{2s} \\
&= \underbrace{d\delta^2 b + \delta^{2s}}_{=\mathrm{A}(S)} + \delta^{2s} - b2\delta^s - d\delta^{s+2} \\
&= \mathrm{A}(S) + \delta^{2s} - (b2\delta^s + d\delta^{s+2}) \qquad (27) \\
&\geq \mathrm{A}(S) - (2 + \underbrace{d\delta^2}_{\leq 1+2\delta})\delta^s \\
&\geq \mathrm{A}(S) - (\underbrace{2+1+2\delta}_{\leq 4})\delta^s \\
&\geq \mathrm{A}(S) - \delta(\underbrace{\delta^2}_{\leq d\delta^2} \underbrace{4\delta^{s-3}}_{\leq b}) \\
&\geq \mathrm{A}(S) - \delta\,\mathrm{A}(S),
\end{aligned}
$$

where h_i denotes the height of ith level generated by NFDH and $n^{(S)}$ denotes the total number of levels generated by NFDH.

To take care of the discarded small rectangle (while partitioning the low-narrow rectangles, see section 4.3), we add this rectangle to one set S without changing the bound $\delta\,\mathrm{A}(S)$ for the total area of the unpacked rectangles (see Equation (27)).

Thus, after packing all blocks, the total area of the unpacked low-narrow rectangles is bounded by $\delta\,\mathrm{A}(L_{\text{lo-na}}) \leq \delta$.

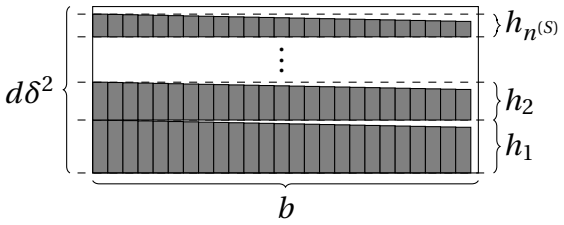


Figure 8: NFDH for packing blocks

Area lost by discarding small blocks. While packing the small boxes we discarded all blocks with width smaller than $4\delta^{s-3}$. However, the area lost by discarding these blocks is bounded by δ.

Lemma 10. *The total area of all blocks with width* $< 4\delta^{s-3}$ *is bounded by* δ.

Proof. Let us first note that there are at most $g^*(\frac{1}{2\delta^2}+1)2^{|M|}|M|$ blocks; this bound holds because for a fixed subset of free slots, there are at most $\frac{|M|}{2} \leq |M|$ blocks. Furthermore, there are at most $2^{|M|}$ different subsets of the free slots. (Remember that we combined subsets of free slots if the sets SN_i^j are equal for adjacent configurations.)

Due to the sorting of the configurations (see section 4.4.1), configurations with equal sets SN_i^j are adjacent for each snapshot and each height class. Thus, the above bound for the number of blocks holds for each snapshot ($\leq g^*$) and for each height class ($\leq \frac{1}{2\delta^2}+1$). In total the number of blocks is bounded by

$$g^*(\frac{1}{2\delta^2}+1)2^{|M|}|M|.$$

If we assume that all blocks have width smaller than $4\delta^{s-3}$ (get discarded) and that each block has height 1, we can bound the total area of the discarded blocks by $(4\delta^{s-3})g^*(\frac{1}{2\delta^2}+1)2^{|M|}|M| \leq \delta$.

It holds that

$$\begin{aligned}
K &\leq (1+2\delta^{-3})(6|M|(2|M|)^{|M|})^{\frac{1}{\delta}-1} \\
&\leq 3\delta^{-3}(6|M|(2|M|)^{|M|})^{\frac{1}{\delta}-1} \\
&\leq 3\delta^{-3}\left(6\frac{1+2\delta}{\delta^2}\left(2\frac{1+2\delta}{\delta^2}\right)^{\frac{1+2\delta}{\delta^2}}\right)^{\frac{1}{\delta}-1} \\
&\leq 3\delta^{-3}\left(6\frac{2}{\delta^2}\left(2\frac{2}{\delta^2}\right)^{\frac{2}{\delta^2}}\right)^{\frac{1}{\delta}-1} \\
&= 3\delta^{-3}\left(\frac{12}{\delta^2}\left(\frac{4}{\delta^2}\right)^{\frac{2}{\delta^2}}\right)^{\frac{1}{\delta}}\left(\frac{12}{\delta^2}\left(\frac{4}{\delta^2}\right)^{\frac{2}{\delta^2}}\right)^{-1} \\
&\leq 3\delta^{-3}2^{\frac{4}{\delta}}\delta^{-\frac{2}{\delta}}\left(\frac{2^2}{\delta^2}\right)^{\frac{2}{\delta^3}}2^{-3}\delta^2\left(\frac{2^2}{\delta^2}\right)^{-\frac{2}{\delta^2}}
\end{aligned}$$

$$
\begin{aligned}
&= 3\delta^{2-3-\frac{2}{\delta}} 2^{\frac{4}{\delta}-3} \left(\frac{2}{\delta}\right)^{\frac{4}{\delta^3}-\frac{4}{\delta^2}} \\
&= 3\delta^{-1-\frac{2}{\delta}-\frac{4}{\delta^3}+\frac{4}{\delta^2}} 2^{\frac{4}{\delta}-3+\frac{4}{\delta^3}-\frac{4}{\delta^2}} \\
&= 3\delta^{-(\overbrace{\frac{\delta^3+2\delta^2+4-4\delta}{\delta^3}}^{\leq 4, \text{ since } 4\delta \geq \delta^3+2\delta^2})} 2^{\overbrace{\frac{4\delta^2-3\delta^3+4-4\delta}{\delta^3}}^{\leq 4}} \\
&\leq 3\delta^{-\frac{4}{\delta^3}} 2^{\frac{4}{\delta^3}}
\end{aligned}
$$

and thus

$$
\begin{aligned}
4(K+2\delta^{-3}) &\leq 4(3\delta^{-\frac{4}{\delta^3}} 2^{\frac{4}{\delta^3}} + 2\delta^{-3}) \\
&\leq 4(4\delta^{-\frac{4}{\delta^3}} 2^{\frac{4}{\delta^3}}) \\
&= \delta^{-\frac{4}{\delta^3}} 2^{\frac{4}{\delta^3}+4}.
\end{aligned}
$$

Using these inequalities we conclude

$$
\begin{aligned}
4\delta^{s-3} g^* (\frac{1}{2\delta^2}+1) 2^{|M|} |M| &\overset{\text{def } M, g^*}{\leq} 4\delta^{s-3}(4(K+2\delta^{-3}))(\frac{1}{2\delta^2}+1) 2^{\frac{1+2\delta}{\delta^2}} \frac{1+2\delta}{\delta^2} \\
&\overset{\delta \leq \frac{1}{4}}{\leq} 4\delta^{s-3}(\delta^{-\frac{4}{\delta^3}} 2^{\frac{4}{\delta^3}+4})(\frac{1}{\delta^2}) 2^{\frac{2}{\delta^2}} \frac{2}{\delta^2} \\
&= 2^3 \delta^{s-3-\frac{4}{\delta^3}-2-2} 2^{\frac{4}{\delta^3}+\frac{2}{\delta^2}+4} \\
&= \delta^{s-(7+\frac{4}{\delta^3})} 2^{7+\frac{4}{\delta^3}+\frac{2}{\delta^2}} \\
&= \delta^{s-(\overbrace{\frac{7\delta^3+4}{\delta^3}}^{\leq 5})} 2^{\overbrace{\frac{7\delta^3+4+2\delta}{\delta^3}}^{\leq 5}} \\
&\leq \delta^{s-\frac{5}{\delta^3}} 2^{\frac{5}{\delta^3}} \\
&\overset{\text{def } s}{=} \delta^{\frac{8}{\delta^3}-\frac{5}{\delta^3}} 2^{\frac{5}{\delta^3}} \\
&= \delta^{\frac{3}{\delta^3}} 2^{\frac{5}{\delta^3}} \\
&= \delta^{\frac{1}{\delta^3}} \delta^{\frac{2}{\delta^3}} 2^{\frac{5}{\delta^3}} \\
&\overset{\delta \leq \frac{1}{14} \leq 2^{-3}}{\leq} \delta^{\frac{1}{\delta^3}} 2^{-\frac{6}{\delta^3}} 2^{\frac{5}{\delta^3}} \\
&\leq \underbrace{\delta^{\frac{1}{\delta^3}}}_{\leq \delta} \underbrace{2^{-\frac{1}{\delta^3}}}_{\leq 1} \\
&\leq \delta.
\end{aligned}
$$

□

4.4.6 Packing Containers

In the following, we describe how to pack the remaining low-narrow rectangles $L^C_{\text{lo-na}}$ and the low-wide rectangles $L_{\text{lo-wi}}$ into the containers.

Assume that we have chosen the *right* set of containers $\mathscr{C}$, that is, the set of containers corresponds to the set induced by an optimal packing. If we have not chosen the *right* set, packing the remaining low-narrow rectangles $L^C_{\text{lo-na}}$ and the low-wide rectangles $L_{\text{lo-wi}}$ into the containers might not be possible. In this case, we restart with another set of containers.

Unfortunately, some low-wide rectangles might intersect two containers. To ensure that all low-wide rectangles are packable into containers, we increase the height of each container by δ^s. This is sufficient, since the height of all low rectangles is bounded by δ^s. In the following lemma, we prove that the mKR algorithm (see section 3) can be used to pack nearly all rectangles into the containers.

Lemma 11. *Nearly all rectangles from $L_{lo\text{-}wi}$ and $L^C_{lo\text{-}na}$ can be packed into the containers, i.e. the total area of unpacked rectangles is bounded by δ.*

Proof. It is sufficient to prove that the assumptions (A1) – (A6) of Theorem 3 are fullfilled.

Let $L_{\text{wi}} := L_{\text{lo-wi}}, L_{\text{sm}} := L^C_{\text{lo-na}}$. Since we increased the height of all containers by δ^s, all rectangles from $L_{\text{lo-wi}}$ are packable inside the containers; consequently, (A1) holds. Furthermore, $L^C_{\text{lo-na}}$ was chosen such that

$$\mathrm{A}(L^C_{\text{lo-na}}) + \mathrm{A}(L_{\text{lo-wi}}) \leq \mathrm{A}(\mathscr{C}) \qquad \text{(see Equation (16))}.$$

Thus, Assumption (A2) is fulfilled. Assumption (A3) is fullfilled since the width of each rectangle in $L_{\text{lo-wi}}$ is at least δ.

To show (A4) – (A6), it is sufficient to show (A6), since $h^s_{\max}, h^w_{\max}, w^s_{\max} \leq \gamma = \delta^s$ and the right hand side of (A6) is the strictest. It holds that

$$\begin{aligned}
h^s_{\max} \leq \delta^s = \delta^{\frac{8}{\delta^3}} &= \delta^2 \cdot \delta^2 \delta^3 \underbrace{\delta^{\frac{1}{\delta^3}}}_{\leq 1} \\
&\overset{\delta \leq \frac{1}{7}}{\leq} \frac{\delta}{7} \cdot \frac{\delta^3}{7 \cdot 7} \\
&\leq \frac{\delta}{7} \min\left\{ \frac{\delta^3}{4 \cdot 2}, \frac{\delta}{4 \cdot 7} \delta \right\} \\
&\overset{|\mathscr{C}| \leq \frac{2}{\delta^3}, w^w_{\min} \geq \delta}{\leq} \frac{\delta}{7} \min\left\{ \frac{1}{4|\mathscr{C}|}, \frac{\delta}{4 \cdot 7} w^w_{\min} \right\}.
\end{aligned}$$

□

Hence, the mKR algorithm allows us to pack almost all of the low-narrow rectangles from $L^C_{\text{lo-na}}$ and $L_{\text{lo-wi}}$, in particular the total area of the unpacked rectangles is bounded by δ. But since we increased the height of the containers, some of the low rectangles might intersect with other rectangles. However, we can move all intersecting rectangles to the top of the strip at the cost of an increase in height by at most $2\delta^s \cdot (1+2\delta)/\delta^2 \leq \delta$.

4.5 Analysis of the Algorithm

In the following, we summarize the approximation algorithm for the non-malleable, contiguous case, $P\text{poly}|\text{line}_j|C_{\max}$ (see Algorithm 1 for pseudo-code).

Algorithm 1: Algorithm for $P\text{poly}|\text{size}_j|C_{\max}$

Input: Set of jobs $L = \{R_i \mid i \in \{1,\dots,n\}\}$, and precision ε
Output: A schedule S with $h(S) \leq (1.5+\varepsilon)\,\text{OPT}$

```
/* see section 4.1.1 */
```
Let v be the height of the solution generated by 2-approximation
foreach $v^* \in \{(1+0\varepsilon)\frac{v}{2}, (1+1\varepsilon)\frac{v}{2}, \dots, (1+\lceil\frac{1}{\varepsilon}\rceil\varepsilon)\frac{v}{2}\}$ **do**
  `/* see section 4.1.2 */`
  Set ε' such that $\varepsilon' = \frac{1}{2a}$ for an integer a and such that $\varepsilon' \leq \frac{\varepsilon}{14}$
  Find δ
  Set $\gamma \leftarrow \delta^s$
  Partition L into $L^{>1/2}, L_{\text{ta}}, L_{\text{lo}}, L_{\text{wi}}, L_{\text{na}}$
  `/* see section 4.1.3 */`
  Round up h_i to the next multiple of δ^2 for all $R_i \in L_{\text{ta}}$
  Set $M \leftarrow \frac{1+2\delta}{\delta^2}$
  Set $K \leftarrow (1+2\delta^{-3})(6|M|(2|M|)^{|M|})^{\frac{1}{\delta}-1}$
  Let $L_K \subseteq L_{\text{ta}} \setminus L^{>1/2}$ be the subset of K tall rectangles with largest area
  `/* see section 4.2 */`
  foreach *choice of containers* $\mathscr{C}$ **do**
    if $L_{lo\text{-}wi}$ *are nearly packable into* $\mathscr{C}$ **then**
      Set $L' \leftarrow \mathscr{C} \cup L_K$
      foreach *slot assignment* f **do**
        foreach *snapshot assignment* α, β **do**
          Calculate V by dynamic program `/* see section 4.2.3 */`
          foreach $v \in V$ **do**
            Solve LP `/* see section 4.3 */`
            if *LP has a solution* **then**
              `/* see section 4.4 */`
              Adapt and Sort configurations `/* 4.4.1 */`
              Pack pre-positioned rectangles `/* 4.4.2 */`
              Pack low-narrow rectangles `/* 4.4.5 */`
              Pack containers (if possible) `/* 4.4.6 */`
              Save solution (if it exists)
Choose schedule with minimal length

We first guess (enumerate) the length of the optimal schedule. The next step in our algorithm is to create a gap between tall and low rectangles and between wide and narrow rectangles. In this process, we discard all *middle-sized* rectangles. The total area of these rectangles is bounded by $\varepsilon' =: A_1$ (see section 4.1.2). In the following, we accept a slightly increased height of the solution by rounding and shifting all tall rectangles. This additional height is bounded by $2\delta =: h_1$, as was shown in section 4.1.3.

Then we guess (enumerate) the set of containers and we guess/enumerate a slot assignment and a snapshot assignment for L', where L' contains all containers and a subset of the tall rectangles (or all tall rectangles if $|L_{\text{ta}}| \leq K$). With a dynamic program we construct a set V of all distinguishable assignments of rectangles from $L^{>1/2}$ to snapshots. After choosing one assignment $v \in V$ we set up a LP. If the LP has no solution, we try the next combination of containers, slot assignment, snapshot assignment and vector $v \in V$. Otherwise, we start the actual packing of the rectangles beginning with the subset of tall rectangles and containers L' and the rectangles from $L^{>1/2}$. All of these rectangles can be packed (see sections 4.4.2, 4.4.3). The remaining tall rectangles are packed fractionally according to the solution of the LP. By removing all split tall rectangles, we discard rectangles with total area bounded by $2\delta =: A_2$ (see section 4.4.4).

For packing low-narrow rectangles into the space reserved for them by the LP, we use an approximation algorithm. The low-narrow rectangles that are not packed by this algorithm and the discarded blocks have total area bounded by $2\delta =: A_3$ (see section 4.4.5).

The next step is to pack the rectangles assigned to the containers (see section 4.4.6). Note that this step is not always successful if we have chosen the *wrong* set of containers. In case of failure, we try another set of containers. We increased the height of each container such that all rectangles fit into the containers. Shifting all intersecting rectangles to the top of the strip increases its height by at most $\delta =: h_2$, since there are at most $(1+2\delta)/\delta^2$ slots and the height of each intersected rectangle is bounded by δ^s. Furthermore, rectangles with total area bounded by $\delta := A_4$ are not packed by mKR.

In total, we packed almost all rectangles into a strip of height $1 + h_1 + h_2 = 1 + 3\delta$. We add the discarded rectangles using the NFDH algorithm by Coffman et al. [5]. This leads to an additional strip with height bounded by

$$2 \cdot (A_1 + A_2 + A_3 + A_4) + h_3 \leq 2\varepsilon' + 9\delta + \frac{(1+2\varepsilon')}{2},$$

where $h_3 \leq (1+2\varepsilon')/2$ is the height of the tallest rectangle among all discarded rectangles. Thus, all rectangles can be packed into a strip with height bounded by

$$\begin{aligned}(1+3\delta) + (2\varepsilon' + 9\delta + (1+2\varepsilon')/2) &\leq 1 + \frac{1}{2} + 3\varepsilon' + 12\delta \\ &\leq 1 + \frac{1}{2} + 15\varepsilon' \leq 1 + \frac{1}{2} + \varepsilon.\end{aligned}$$

As already mentioned in section 4.1.1 (see Equation (1)), rescaling yields Theorem 1.

The running time of the algorithm is in $O(n^{f(\frac{1}{\varepsilon})})$ for some (super-exponential) function f.

5 Non-Contiguous Parallel Job Scheduling

In this section, we study the problem $P\text{poly}|\text{size}_j|C_{\max}$. In this problem, the indices of the machines allotted to a job are not required to be contiguous. In the following, we construct a polynomial time approximation scheme (PTAS) for this case. First we show the existence of a nearly optimal schedule with simpler structure. Therefore, we guess the height of an optimal schedule and scale the instance such that the height of an optimal solution for the scaled instance is bounded by 1 (see section 4.1.1). Instead of the 2-approximation algorithm for the strippacking problem, we use a 2-approximation algorithm presented by Garey & Graham [11] for resource-constrained scheduling. We partition the jobs into tall, low-narrow, and low-wide jobs. Again, we reduce the search space by rounding and shifting the tall jobs in the same manner as before (see section 4.1.3).

The actual algorithm for this case works as follows. We use a dynamic program to find a distribution of the tall jobs among the slots. Then we schedule the tall jobs according to the distribution in a canonical way. The remaining space is merged into one container per slot. We schedule the low jobs by packing them into these containers using the mKR algorithm. Then, creating a feasible schedule can be done by a simple greedy algorithm. In contrast to the previous, case we do not guess the structure of the containers. The structure is given automatically after assigning the tall jobs.

Again we use the notations job/rectangle and schedule/packing synonymously, although rectangles might be misleading in this case, since horizontal fragmentation is allowed; the height of a rectangle h_i corresponds to the length (processing time) p_i of a job (i.e. $p_i = h_i$) and the width w_i of a rectangle corresponds to the number of required machines q_i of a job divided by m (i.e. $w_i = q_i/m$).

5.1 Simple Structure

Again, the first step is to show the existence of a nearly optimal solution with simple structure.

5.1.1 Bounded Height

Since we want to divide the solution into a constant number of *slots*, we need to know the height of an optimal solution, at least up to the required accuracy ε. By using the 2-approximation algorithm by Garey & Graham [11], we can find a solution with height $\nu \leq 2 \cdot \text{OPT}$, where OPT is the height of an optimal solution. Again, there exists a value

$$\nu^* \in \{(1+0\varepsilon)\,\nu/2, (1+1\varepsilon)\,\nu/2, \ldots, (1+\lceil 1/\varepsilon \rceil\,\varepsilon)\,\nu/2\}$$

such that $\text{OPT} \leq \nu^* \leq (1+\varepsilon)\,\text{OPT}$. Therefore, we only have to consider $\lceil 1/\varepsilon \rceil + 1$ different candidates to find the right one. For simplicity, we divide the height of each job by ν^* such that the height $\text{OPT}' := \frac{\text{OPT}}{\nu^*}$ of an optimal solution for the scaled instance satisfies

$$1-\varepsilon < \frac{1-\varepsilon}{1-\varepsilon^2} = \frac{1}{1+\varepsilon} = \frac{\text{OPT}}{(1+\varepsilon)\,\text{OPT}} \leq \frac{\text{OPT}}{\nu^*} \leq 1.$$

5.1.2 Creating a Gap

Again we create a gap in size between tall and low and between wide and narrow jobs by discarding *middle-sized* jobs. These discarded jobs will be scheduled in a post-processing step.

Let ε denote the requested accuracy and let $\varepsilon' \leq \varepsilon/9$ be the largest value of the form $\varepsilon' = 1/a$ for some integer value a. Let $\sigma_0 := 1, \sigma_1 := \varepsilon'$, and $\sigma_k := \sigma_{k-1}^3/(4 \cdot 7^2)$ for all $k \geq 2$. Define

$$L_k := \{R_i \in L \mid h_i \in (\sigma_k, \sigma_{k-1}] \text{ or } w_i \in (\sigma_k, \sigma_{k-1}]\}.$$

By Lemma 4 there exists $k \in \{2, \ldots, 2/\varepsilon' + 1\}$ such that $A(L_k) \leq \varepsilon' A(L)$. Choose the smallest value $k \in \{2, \ldots, 2/\varepsilon' + 1\}$ with this property, and let $\delta := \sigma_{k-1}$, and $\gamma := \sigma_k = \delta^3/(4 \cdot 7^2)$. Define

$$\begin{aligned} L_{\text{ta}} &:= \{R_i \in L \mid h_i > \delta\} && \text{tall jobs} \\ L_{\text{lo-wi}} &:= \{R_i \in L \mid h_i \leq \gamma, w_i > \delta\} && \text{low-wide jobs} \\ L_{\text{lo-na}} &:= \{R_i \in L \mid h_i \leq \gamma, w_i \leq \gamma\} && \text{low-narrow jobs.} \end{aligned}$$

In the following, we present an algorithm to schedule $L_{\text{ta}}, L_{\text{lo-wi}}$, and $L_{\text{lo-na}}$. The remaining rectangles will be scheduled in a post-processing step using a greedy algorithm. This is possible since

$$\mathrm{A}(L \setminus (L_{\text{ta}} \cup L_{\text{lo-wi}} \cup L_{\text{lo-na}})) = \mathrm{A}(L_k) \leq \varepsilon'.$$

5.1.3 Shifting and Rounding

Analogous to the previous case, we can round up the height of the jobs to the next multiple of δ^2 and shift the positions of the tall jobs in an optimal schedule up to the next multiple of δ^2. Again this modifications increases the height of the schedule by at most 2δ.

Lemma 12. *Let S be an optimal schedule, $\mathrm{h}(S) \leq 1$, for all jobs L. At the cost of an increase in height of at most 2δ we can round up all tall jobs to the nearest multiple of δ^2 and we can shift the jobs such that the start time of all jobs is a multiple of δ^2.*

Proof. Analogous to Lemma 6 (section 4.1.3). □

5.2 Dynamic Program for Tall Jobs

Draw horizontal lines spaced by a distance δ^2 across the schedule starting with the x-axis as first such line. Note that, due to the rounding and shifting the lower side of each tall job corresponds to one of these horizontal lines. We say that job R_i is in slot i if its starting time (in a given schedule) corresponds to the ith horizontal line. Let

$$I_{\mathrm{S}} := \left\{1, \ldots, \frac{(1+2\delta)}{\delta^2}\right\}$$

denote the set of slots and let

$$q := |I_{\mathrm{S}}| = \frac{(1+2\delta)}{\delta^2} \leq \frac{2}{\delta^2}. \tag{28}$$

Since we cannot enumerate all possible assignments of tall jobs to slots in polynomial time, we use a dynamic programming approach similar to the approach in section 4.2.3. Again, due to the height bound of 1 and the rounding, we can partition the set of tall jobs L_{ta} into a constant number of height classes $L(i)$ with

$$L(i) := \{R_j \in L_{\text{ta}} \mid h_j = i \cdot \delta^2\} \qquad \text{for all } i \in I_{\text{ta}} := \left\{\frac{1}{\delta} + i \mid i \in \mathbb{N} : 1 \leq i \leq \frac{1-\delta}{\delta^2}\right\}.$$

Lemma 13. *The set of all tall rectangles can be partitioned into a constant number of subsets:*

$$L_{ta} = \dot{\bigcup_{i \in I_{ta}}} L(i). \tag{29}$$

In particular, the number of partitions of L_{ta} is $|I_{ta}| = \frac{1-\delta}{\delta^2} \leq \frac{1}{\delta^2}$.

Proof. Analogous to Lemma 7 (section 4.1.3). □

Consider an optimal schedule for a scaled, shifted and rounded instance. We can define a vector $v^i = (v^i_1, \ldots, v^i_q)$ for each height $i \in I_{\text{ta}}$ such that each entry $v^i_j \cdot m$ denotes the total width of all tall jobs with height $i \cdot \delta^2$ in slot $j \in I_{\text{S}}$. Obviously,

$$\sum_{i \in I_{\text{ta}}} \sum_{j \in I_{\text{S}}} v^i_j \frac{1}{m} = \sum_{R_i \in L_{\text{ta}}} w_i \leq 1 \tag{30}$$

(otherwise the schedule is not feasible) and thus we have

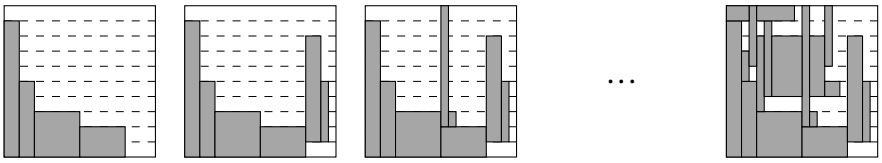
$$v^i_j \leq m \tag{31}$$

for each $i \in I_{\text{ta}}$, $j \in I_{\text{S}}$.

Furthermore, the value of every entry v^i_j is integral, since the width of each job is a multiple of $1/m$. Thus, a rough upper bound for the number of feasible vectors for each height $i\delta^2$ ($i \in I_{\text{ta}}$) is given by $(m+1)^q$ since there are $q = |I_{\text{S}}|$ components (one for each slot) and for every component we have $v^i_j \in \{0, \ldots, m\}$. This number is polynomial in n, since m is polynomial in n and q is a constant (see Equation (28)).

With a dynamic programming approach we can compute a list of all feasible vectors satisfying (30) and (31). The algorithm to calculate all feasible vectors for a given height class $i \in I_{\text{ta}}$ works as follows. Assume that $L(i) = \{R_1, \ldots, R_{k_i}\}$. Starting with a set $V^i := \{(0, \ldots, 0)\}$ containing only the null vector, in step $l \in \{1, \ldots, k_i\}$ we replace each vector $v \in V^i$ with all vectors that can be generated by adding $\gamma_l := w_l \cdot m$ to one of its components. After each step remove all duplicate vectors. Note that for height class i only the slots $\{1, \ldots, \frac{(1+2\delta)}{\delta^2} - i + 1\}$ have to be considered, since jobs belonging to $L(i)$ can be only in these slots without exceeding the height bound of $1+2\delta$. Since the number of different vectors is bounded by $(m+1)^q$ and $q \leq 2/\delta^2$ (see Equation 28), we can show that the running time of the algorithm is polynomially bounded in m with similar arguments as in section 4.2.3.

After repeating this computation for each height class $L(i)$, again we can build the direct product of all sets of vectors $V := \bigtimes_{i \in I_{\text{ta}}} V^i$ and again there is an element $v \in V$ that corresponds to the vectors of widths induced by an optimal solution for the scaled instance.



(a) First slot (b) Second slot (c) Third slot (d) (e) All slots

Figure 9: Canonical packing for tall rectangles

Again we use the direct product notation only for convenience. Analogous to the first case, we extend our dynamic program such that for each component of each vector a set of associated jobs is stored. Let $L(v_j^i)$ be this set of jobs associated with v_j^i.

Define $\varphi_j(i) := \max\{1, j-i+2\}$. Then, $\varphi_j(i)$ is the index of the lowest slot, such that a rectangle of height $i\delta^2$ starting in slot $\varphi_j(i)$ intersectsslot j, since

$$(\varphi_j(i)-1)\delta^2 + i\delta^2 \geq ((j-i+2)-1)\delta^2 + i\delta^2 = (i+1)\delta^2.$$

We call an element $v \in V$ *feasible*, if for each slot the total width of all tall jobs intersecting this slot (including all tall jobs in this slot) is not greater than 1. That is,

$$\sum_{i \in L_{\text{ta}}} \sum_{k=\varphi_j(i)}^{k \leq j} v_k^i \leq m \qquad \text{for all } j \in I_{\text{S}}. \tag{32}$$

Obviously, the vector induced by a scaled, rounded, and shifted optimal solution is feasible.

5.3 Canonical Packing for Tall Jobs

In the following, we present a canonical way to schedule all tall jobs (see figure 9).

Lemma 14. *Let $v = (v^1, \ldots, v^{|I_{ta}|}) \in V$ be a feasible vector. There exists a canonical schedule for all tall jobs.*

Proof. Let $v = (v^1, \ldots, v^{|I_{\text{ta}}|}) \in V$ be a feasible vector. The algorithm starts with the first slot ($j = 1$) and schedules/packs left aligned all tall jobs $L(v_1^i)$, $i \in I_{\text{ta}}$ into this slot (see figure 9a). Obviously, this is possible, since the vector is feasible, i.e.

$$\sum_{i \in L_{\text{ta}}} \sum_{k=\varphi_j(i)}^{k \leq 1} v_k^i = \sum_{i \in I_{\text{ta}}} v_1^i \leq m.$$

Now assume that we have scheduled all slots prior to slot j. Since v is feasible, the free space in slot j is sufficient to (fractionally) pack all jobs assigned by $v_j^i, i \in I_{\text{ta}}$. Note that all jobs scheduled in previous slots and intersecting the current slot are accounted for in Equation (32). Furthermore, the free space in this slot is also free in all following slots (see figure 9c). This allows us to (fractionally) pack all jobs $L(v_j^i)$ ($i \in I_{\text{ta}}$) left aligned into slot j. □

5.4 Packing Low Jobs

Given a feasible vector $v \in V$, the total width for each slot that is not occupied by tall jobs can be computed. Let

$$w_j^f := 1 - \left(\sum_{i \in L_{\text{ta}}} \sum_{k=\varphi_j(i)}^{k \leq j} \frac{v_k^i}{m} \right) \qquad \text{for each } j \in I_S$$

denote the total width of the free space for slot j and define for each slot j a container C_j of width w_j^f and height δ^2.

Consider an optimal schedule of all jobs (after scaling, rounding and shifting). In this optimal schedule some low jobs might intersect the horizontal lines that form the borders of the slots. Since the height of all low jobs is bounded by γ, increasing the height of the containers to $\delta^2 + \gamma$ ensures that all low jobs are packable inside the containers. Analogous to Lemma 11, almost all low jobs can be packed into the containers using the mKR algorithm. The total area of the discarded jobs is bounded by δ. Note that the discarded jobs will be packed in a post-processing step.

The next step is to *schedule* the containers. Since we increased the height of the containers in order to ensure that all low-wide rectangles can be packed, the first step is to decrease the height again and remove all overlapping jobs. Since the height of each low job is bounded by γ, the total area of all overlapping jobs is at most

$$2\gamma \frac{1+2\delta}{\delta^2} \leq 2 \frac{\delta^3}{4 \cdot 7^2} \frac{1+2\delta}{\delta^2} \leq \frac{\delta + 2\delta^2}{3} \leq \frac{\delta + 2\delta}{3} = \delta.$$

Thus, the total area of discarded jobs is at most 2δ. Altough now the height of all containers corresponds to the height of the free space in each slot, in general it is still not possible to schedule the container without fragmentation. Therefore, we split the containers into slices of width $1/m$. This is feasible since the jobs need not be schedule on contiguous machines. Due to the definition of the containers the width of each container corresponds exactly to the free space of each slot. This allows us to add the slices successively, left-aligned to the free space of each corresponding slot (see figure 10). Note that packing the low rectangles into the containers might not be possible, if we have chosen the *wrong* vector v or scaling factor v^*.

5.5 Analysis

In the following, we summarize the algorithm for $P\text{poly}|\text{size}_j|C_{\max}$ (see Algorithm 2 for pseudo-code).

While creating the gap, we discard jobs with total area bounded by $\varepsilon' := A_1$ (see section 5.1.2). Due to the shifting and rounding of the tall jobs we increased the height of the resulting schedule to $1 + 2\delta =: h_1$ (see section 5.1.3). In order to ensure that all low-wide jobs fit into the containers, we increased the height of all containers by γ. This led to overlapping jobs with total area bounded by $\delta =: A_2$ (see section 5.4). Furthermore, we discarded all jobs that were not packed by the modified Kenyon and Rémila algorithm, these

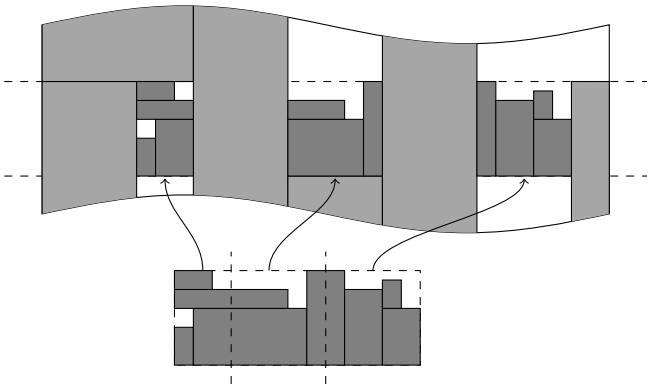


Figure 10: Split container to pack slots

Algorithm 2: Algorithm for $P\text{poly}|\text{size}_j|C_{\max}$

Input: Set of jobs $L = \{R_i \mid i \in \{1,\dots,n\}\}$, and precision ε
Output: A schedule S with $h(S) \leq (1+\varepsilon)\,\text{OPT}$

`/* see section 5.1.1 */`
Let v be the height of the solution generated by 2-approximation
foreach $v^* \in \{(1+0\varepsilon)\frac{v}{2}, (1+1\varepsilon)\frac{v}{2}, \dots, (1+\lceil\frac{1}{\varepsilon}\rceil\,\varepsilon)\frac{v}{2}\}$ **do**
  `/* see section 5.1.2 */`
  Set ε' such that $\varepsilon' = \frac{1}{a}$ for an integer a and such that $\varepsilon' \leq \frac{\varepsilon}{8}$
  Find δ
  Set $\gamma \leftarrow \frac{\delta^3}{4\cdot 7^2}$
  Partition L into $L_{\text{ta}}, L_{\text{lo-wi}}, L_{\text{lo-na}}$
  `/* see section 5.1.3 */`
  Round up h_i to the next multiple of δ^2 for all $R_i \in L_{\text{ta}}$
  Calculate V by dynamic program `/* see section 5.2 */`
  foreach $v \in V$ **do**
    if v *is feasible* **then**
      Pack L_{ta} in a canonical way `/* see section 5.3 */`
      Pack $L_{\text{lo-wi}} \cup L_{\text{lo-na}}$ (if possible) `/* see section 5.4 */`
      Save solution (if it exists)
Choose schedule with minimal length

discarded jobs have total area $\delta =: A_3$ (see section 5.4). Note that this step is not always successful if we have chosen the *wrong* vector v or scaling factor v^*. In case of failure, we try another combination. Overall the resulting schedule has height

$$h_1 = 1 + 2\delta$$

and we discarded jobs with total area bounded by

$$A_1 + A_2 + A_3 = \varepsilon' + \delta + \delta.$$

In a post-processing step we pack all discarded jobs on top of the schedule. For this step we use the NFDH algorithm. Due to the fact that all discarded jobs have height bounded by δ, this results in an additional height of at most

$$2(\varepsilon' + 2\delta) + \delta = 2\varepsilon' + 5\delta.$$

Thus, the height of the resulting schedule is bounded by

$$(1 + 2\delta) + (2\varepsilon' + 5\delta) \overset{\delta \le \varepsilon'}{\le} 1 + 9\varepsilon' \overset{\varepsilon' \le \frac{\varepsilon}{9}}{\le} 1 + \varepsilon.$$

Since we scaled the instance in section 5.1.1, the last step is to multiply the length of the schedule by v^*:

$$\begin{aligned} v^*(1+\varepsilon) &\le (1+\varepsilon)\,\mathrm{OPT}(1+\varepsilon) = (1 + 2\varepsilon + \varepsilon^2)\,\mathrm{OPT} \\ &\le (1 + 3\varepsilon)\,\mathrm{OPT}. \end{aligned}$$

This proves Theorem 2. The running time of the algorithm is in $O(n^{f(\frac{1}{\varepsilon})})$ for some exponential function f.

6 Malleable Parallel Job Scheduling

In the following, we extend both algorithms for scheduling malleable jobs. We denote the malleable versions of Ppoly|line$_j$|$C_{\max}$ and Ppoly|size$_j$|$C_{\max}$ by Ppoly|fnct_line$_j$|$C_{\max}$ and Ppoly|fnct$_j$|$C_{\max}$, respectively. Instead of a fixed pair consisting of the number of required processors and the execution time, in this setting each job J_j is associated with a function $p_j : \{1, \ldots, m\} \to \mathbb{Q}^+$ that gives the execution time $p_j(\ell)$ of J_j in terms of the number ℓ of processors assigned to J_j.

We present a dynamic program that generates a polynomial number of assignments of jobs to the number of processors they use. If we have chosen an assignment, we use the corresponding non-malleable algorithm to find a nearly optimal solution. Iterating over all assignments generated by the dynamic program allows us to find a nearly optimal solution. In the following, we assume that $\varepsilon \le 1/2$, where ε is the required accuracy.

6.1 Simple Structure

Before we present the dynamic program, we show again that an optimal solution can be transformed into a nearly optimal solution with simpler structure.

6.1.1 Bounded Height

Analogous to the non-malleable cases, we can find a schedule with length $\nu \leq 2 \cdot \mathrm{OPT}$ by using the 2-approximation algorithm by Ludwig & Tiwari [24] (for both cases contiguous and non-contiguous), where OPT is the length of an optimal solution.

Let $C := \{(1+0\varepsilon)\nu/2, (1+1\varepsilon)\nu/2, \ldots, (1+\lceil 1/\varepsilon \rceil\, \varepsilon)\nu/2\}$. Then there exists a value $\nu^* \in C$ such that $\mathrm{OPT} \leq \nu^* \leq (1+\varepsilon)\,\mathrm{OPT}$. Obviously, we only have to consider $\lceil 1/\varepsilon \rceil + 1$ different candidates to find this value.

Again, we scale the execution times of all jobs by ν^*. But since we do not know the number of processors assigned to each job, the scaling of the execution time of each job is done by components, that is, for each job J_j and for each number of processors l we scale the value of $p_j(\ell)$.

6.1.2 Partitioning

In order to simplify a given schedule, we are going to partition the set of jobs into tall jobs, middle-sized jobs and small jobs as before. But since we have no knowledge about the *size* of the jobs in the optimal solution, we cannot calculate δ, γ in advance. We have to enumerate all possible values for δ, γ. This is feasible since there is only a constant number of candidates (see sections 4.1.2 and 5.1.2).

- In the contiguous case we have $2/\varepsilon'$ candidates σ_k with $\sigma_0 := 1, \sigma_1 := \varepsilon'$, and $\sigma_k := (\sigma_{k-1})^{8/\sigma_{k-1}^3}$ for all $k \geq 2$; we enumerate all values $k \in \{2, \ldots, 2/\varepsilon' + 1\}$ and set $\delta := \sigma_{k-1}$ and $\gamma := \sigma_k = \delta^{8/\delta^3}$.
- In the non-contiguous case we have $2/\varepsilon'$ candidates σ_k with $\sigma_0 := 1, \sigma_1 := \varepsilon'$, and $\sigma_k := \sigma_{k-1}^3/(4 \cdot 7^2)$ for all $k \geq 2$; we enumerate all values $k \in \{2, \ldots, 2/\varepsilon' + 1\}$ and set $\delta := \sigma_{k-1}$ and $\gamma := \sigma_k = \delta^3/(4 \cdot 7^2)$.

Even knowing δ and γ we cannot partition the set of rectangles at this point, since we do not know how many processors will be assigned to each job. However, if we have fixed the number of processors ℓ assigned to a job R_j, we will call R_j

tall	if $p_j(\ell) > \delta$
low	if $p_j(\ell) \leq \gamma$,
wide	if $\ell > \delta m$, and
narrow	if $\ell \leq \gamma m$.

The following lemma shows that among these $2/\varepsilon'$ candidates there is a least one allowing the gap creation as before, i.e. the set of discarded jobs has small total area.

Lemma 15. *Let S be an arbitrary schedule for L. Then there exists a pair δ, γ among all candidates such that the area*

$$\mathrm{A}(L_k) \leq \varepsilon' \mathrm{A}(L)$$

with $L_k := \{R_i \in L \mid \gamma < r_i < \delta \quad \text{or} \quad \gamma < p_i(r_i) < \delta\}$.

Proof. Analogous to the proof of Lemma 4. □

6.1.3 Rounding and Shifting

Let S be an optimal schedule for a given instance. In particular in this schedule for each job, the number of assigned processors is fixed and thus, the processing time is known.

Lemma 16. *There exists a schedule S with nearly optimal length and simpler structure. That is:*

(a) $\mathrm{h}(S) \leq (1+3\delta)\,\mathrm{OPT}$, *where* $\mathrm{h}(S)$ *denotes the length of schedule S.*

(b) For each tall job $R_i \in L_{ta}$ the start time is a multiple of δ^2 and the processing time / height of each tall job can be rounded up to the next multiple of δ^2.

(c) For all other rectangles, $R_i \in L \setminus L_{ta}$, the processing time can be rounded up to the next multiple of γ/n.

Proof. Let S^* be an optimal schedule. We can modify this schedule basically in the same way as in the non-malleable cases, since in schedule S^* the number of processors assigned to each job is fixed and thus the processing time is also fixed.

Rounding up the processing time of all non-tall jobs R_i (i.e. jobs with processing time $\leq \delta$) to the next multiple of γ/n increases the height by at most

$$n \cdot \frac{\gamma}{n} = \gamma.$$

Analogous to the proof of Lemma 6, we can shift the starting time and round the processing time of each tall job after scaling the schedule by $(1+2\delta)$. Since $\gamma \leq \varepsilon \leq 1/2$, this leads to an increased height of at most

$$(1+2\delta)(1+\gamma) = 1+\gamma+2\delta+2\delta\gamma \leq 1+3\delta.$$

□

Thus, we can round up the running time of all jobs at the cost of at most 3δ. Again, since we do not know the number of processors assigned to each job, the rounding of the execution time of each job is done by components, that is, for each job R_i and for each number of processors ℓ we round up the value of $p_i(\ell)$; the value is rounded up to the next multiple of δ^2 if $p_i(\ell) > \delta$ and to the next multiple of γ/n otherwise. In the following, we denote the scaled and rounded execution times by $\tilde{p}_j(\ell)$. Note that $\tilde{p}_j(\ell) \leq \delta$ iff $p_j(\ell) \leq \delta$, since δ is a multiple of γ and thus δ is a multiple of γ/n. Furthermore, $\tilde{p}_j(\ell) \leq \gamma$ iff $p_j(\ell) \leq \gamma$, since γ is a multiple of γ/n.

6.1.4 Container

In the following, we show that there exists a constant number of widths such that the width of all low-wide rectangles can be rounded up to one of these widths.

Lemma 17. *Let S be a nearly optimal schedule (scaled, rounded, shifted). There exists a packing $\hat{S}$ and a vector of width $(b_1, \ldots, b_{\hat{m}})$ such that*

(a) $h(\hat{S}) \leq h(S) + \delta$

(b) the total area of discarded rectangles is bounded by 2δ*, and*

(c) the width of each low-wide rectangle can be rounded up to a width b_i *for some* $i \in \{1, \ldots, \hat{m}\}$.

Proof. Consider a nearly optimal schedule (scaled, rounded, shifted) S. We define slots as in section 4.1.4 or 5.4. Draw horizontal lines spaced by a distance δ^2 across the schedule. Due to the rounding and shifting, the lower and upper sides of the tall jobs lie along two of these lines. These lines split the schedule into at most $(1+3\delta)/\delta^2$ horizontal rectangular regions that we call *slots*. The definition of containers depends on the scheduling problem we are considering.

In the contiguous case, we define a container as a rectangular region inside a slot whose left boundary is either the right side of a tall rectangle or the left side of the strip, and whose right boundary is either the left side of a tall rectangle or the right side of the schedule. We consider only containers that contain at least one low-wide rectangle (see section 4.1.4).

In the non-contiguous case, we define a container a little bit differently. Let w_j^f denote the total width for each slot j that is not occupied by tall rectangles. We define for each slot j a container C_j of width w_j^f and height δ^2 (see section 5.4).

As we have shown in section 4.4.6 (contiguous case) and in section 5.4 (non-contiguous case) we can pack almost all low-wide rectangles using the mKR algorithm. The mKR algorithm stacks all low-wide rectangles on top of each other and divides the stack into $\hat{m}$ groups (see section 3.1, figure 1). The width of each wide rectangle is rounded to the width of the widest rectangle of the corresponding group. During repacking, rectangles with total area at most δ are discarded. Furthermore, we have to discard overlapping rectangles with total area bounded by δ (see sections 4.4.6 and 5.4).

Thus,

- the total area of all discarded rectangles is bounded by 2δ, and
- there is only a constant number of different widths among the low-wide rectangles.

□

6.1.5 Induced Vector

Assume that we have a scaled, rounded, and shifted solution according to sections 6.1.1, 6.1.2, 6.1.3, 6.1.4. Let $g := 1/\delta^2$, $q := (1+3\delta)/\delta^2$ and let $\hat{m}$ denote the number of groups constructed in the mKR algorithm (see section 3.1). For this schedule with simpler structure we can define a vector

$$v = (v_1^t, \ldots, v_g^t, v_1^w, \ldots, v_{\hat{m}}^w, v^s, v^d)$$

with the following semantics:

- $v_i^t = (v_{i,1}^t, \ldots, v_{i,q}^t)$ is a vector and $v_{i,j}^t \cdot \frac{1}{m}$ denotes the total width of all tall jobs with height $i\delta^2$ in slot j for each $i \in \{1, \ldots, g\}, j \in \{1, \ldots, q\}$; e.g. $v_{\frac{1}{\delta}+1,2}^t = 5$ means that the

sum of the widths of all jobs with height $(\frac{1}{\delta}+1)\cdot\delta^2 = \delta+\delta^2$ in slot 2 is $\frac{5}{m}$. Note that $v^t_{i,j} = 0$ for all $i \le \frac{1}{\delta}$, since all tall jobs have height $> \delta = \frac{1}{\delta}\cdot\delta^2$. Furthermore, $v^t_{i,j} = 0$ for all $j > \frac{(1+3\delta)}{\delta^2} - i + 1$, since jobs of height $i\delta^2$ would exceed the height $(1+3\delta)$ if placed in such slots j.

- $v^w_j \cdot \frac{\gamma}{n}$ denotes the total height of all wide jobs belonging to group j as constructed by the mKR algorithm for each $j \in \{1,\dots,\hat{m}\}$; e.g. $v^w_2 = 5$ means that the total height of all wide jobs R_i with width $b_2 \le w_i < b_3$ (or $b_2 = w_i$ if $b_2 = b_3$) is $5 \cdot \frac{\gamma}{n}$.
- $v^s \cdot \frac{\gamma}{nm}$ denotes total area of all small jobs; e.g. $v^s = 5$ means that the total area of all small jobs is $5 \cdot \frac{\gamma}{nm}$.
- $v^d \cdot \frac{\gamma}{nm}$ denotes total area of all discarded jobs; e.g. $v^d = 5$ means that the total area of all discarded jobs is $5 \cdot \frac{\gamma}{nm}$.

Due to the rounding and normalization, all components must have discrete values and the value of each component is bounded. We have:

- $v^t_{i,j} \in \{0,\dots,\frac{(1+3\delta)}{\delta}m\}$ for each $i \in \{1,\dots,g\}, j \in \{1,\dots,q\}$; the value is integral, since the width of each job is a multiple of $\frac{1}{m}$; the value is bounded by $\frac{(1+3\delta)}{\delta}m$, since otherwise the total area of all jobs with height $i\delta^2$ is

$$\underbrace{i\delta^2}_{>\delta} \cdot \underbrace{v^t_{i,j}}_{>\frac{(1+3\delta)}{\delta}m} \cdot \frac{1}{m} > \delta\frac{(1+3\delta)}{\delta}m\frac{1}{m} = 1+3\delta.$$

- $v^w_i \in \{0,\dots,\frac{(1+3\delta)}{\delta}\frac{n}{\gamma}\}$ for each $i \in \{1,\dots,\hat{m}\}$; the value is integral, since the height of all non-tall jobs is a multiple of $\frac{\gamma}{n}$; the value is bounded by $\frac{(1+3\delta)}{\delta}\frac{n}{\gamma}$, since otherwise the total area of all jobs in group i is at least

$$\underbrace{b_{i+1}}_{>\delta}\frac{(1+3\delta)}{\delta}\frac{n}{\gamma}\frac{\gamma}{n} > 1+3\delta.$$

- $v^s \in \{0,\dots,(1+3\delta)\frac{(mn)}{\gamma}\}$; the value is integral, since the height of all non-tall jobs is a multiple of $\frac{\gamma}{n}$ and the width of all jobs is a multiple of $\frac{1}{m}$; the value is bounded by $(1+3\delta)\frac{(mn)}{\gamma}$, since otherwise the total area of all small jobs exceeds

$$(1+3\delta)\frac{(mn)}{\gamma}\cdot\frac{\gamma}{nm} = 1+3\delta.$$

- $v^d \in \{0,\dots,3\varepsilon'\frac{nm}{\gamma}\}$; the value is integral, since the height of all discarded (non-tall) jobs is a multiple of $\frac{\gamma}{n}$ and the width of all jobs is a multiple of $\frac{1}{m}$; the value is bounded by $3\varepsilon'\frac{nm}{\gamma}$, since otherwise the total area of all discarded jobs exceeds

$$3\varepsilon'\frac{nm}{\gamma}\cdot\frac{\gamma}{nm} = 3\varepsilon'.$$

However, the total area of all discarded jobs is at most $3\varepsilon'$.

Thus in total, the number of different vectors is bounded by

$$\left(\frac{(1+3\delta)}{\delta}m+1\right)^{gq}\left(\frac{(1+3\delta)}{\delta}\frac{n}{\gamma}+1\right)^{\hat{m}}\left((1+3\delta)\frac{nm}{\gamma}+1\right)\left(\frac{3\varepsilon'\cdot nm}{\gamma}+1\right)\in O(m^{gq+2}n^{\hat{m}+2}) \quad (33)$$

and

$$gq=\frac{1}{\delta^2}\frac{(1+3\delta)}{\delta^2}=\frac{1+3\delta}{\delta^4},$$
$$\hat{m}\in O(\frac{1}{\delta^2}).$$

6.2 Dynamic Program

In the following, we assume that we have chosen a vector $b := (b_1,\dots,b_{\hat{m}})$ such that b_i denotes the width of group i (as introduced by the mKR algorithm). Let $b_{\hat{m}+1} := \delta$.

The dynamic program works as follows. We start with a set $V^0 := \{(0,\dots,0)\}$ containing only the null vector. Then we iterate over the set of jobs $L = \{J_1,\dots,J_n\}$ and generate in each step i a new set V^i of vectors by replacing each vector from V^{i-1} with all vectors that can be generated by *adding* J_i to each component, if feasible. To be more specific let $v = (v_1^t,\dots,v_g^t,v_1^w,\dots,v_{\hat{m}}^w,v^s,v^d) \in V^{i-1}$. To generate new vectors we try to *add* J_i to each component in turn. This *adding* is done as follows.

v_k^t For each $j \in \{1,\dots,q\}$ and for each $k \in \{1,\dots,g\}$ with $k\delta^2 > \delta$ let ℓ be the minimal number of processors such that $\tilde{p}_i(\ell) = k\delta^2$.

If ℓ exists, define $\hat{v}_k^t := (v_{k,1}^t,\dots,v_{k,j-1}^t,v_{k,j}^t+\ell,v_{k,j+1}^t,v_{k,q}^t)$ and $v' := (v_1^t,\dots,v_{k-1}^t,\hat{v}_k^t,v_{k+1}^t,\dots,v_g^t,v_1^w,\dots,v_{\hat{m}}^w,v^s,v^d)$ and add v' to V^i. If ℓ is not existing, we continue with the next component.

v_k^w For each $k \in \{1,\dots,\hat{m}\}$ with $b_k \neq b_{k+1}$ we choose $\ell \in (m\cdot b_{k+1}, m\cdot b_k]$ such that $\tilde{p}_i(\ell)$ is minimal; if $b_k = b_{k+1}$ we choose $\ell := b_k m$.

If $\tilde{p}_i(\ell) \leq \gamma$ (i.e. J_i is a low job, furthermore J_i is a wide job, since $b_k \geq b_{\hat{m}+1} = \delta$), define $v' := (v_1^t,\dots,v_g^t,v_1^w,\dots,v_{k-1}^w,v_k^w+\frac{\tilde{p}_i(\ell)n}{\gamma},v_{k+1}^w,\dots,v_{\hat{m}}^w,v^s,v^d)$ and add v' to V^i. If $\tilde{p}_i(\ell) > \gamma$, we continue with the next component.

v^s Choose $\ell \in \{1,\dots,\gamma m\}$ such that $\tilde{p}_i(\ell) \leq \gamma$ and such the area of J_i, $\mathrm{A}_\ell(J_i) = \ell/m\cdot\tilde{p}_i(\ell)$, is minimal.

If such ℓ exists, define $v' := (v_1^t,\dots,v_g^t,v_1^w,\dots,v_{\hat{m}}^w,v^s+\mathrm{A}_\ell(J_i)\cdot\frac{nm}{\gamma},v^d)$ and add v' to V^i.

v^d Choose $\ell \in \{(\gamma m)+1,\dots,\delta m\}$ such that $\gamma < \tilde{p}_i(\ell) \leq \delta$ and such the area of J_i, $\mathrm{A}_\ell(J_i) = \ell/m\cdot\tilde{p}_i(\ell)$, is minimal.

If such ℓ exists, define $v' := (v_1^t,\dots,v_g^t,v_1^w,\dots,v_{\hat{m}}^w,v^s,v^d+\mathrm{A}_\ell(J_i)\cdot\frac{nm}{\gamma})$ and add v' to V^i.

If during an *adding step* a vector is generated that contains a component with value exceeding the above mentioned bounds, this vector is discarded. Since V^i is a set, no duplicate vectors (vectors that have the same components) occur. Consequently, the number of all vectors in V^i is polynomially bounded,

$$|V^i| \in O(m^{gq+2} n^{\hat{m}+2}), \qquad \text{see Equation (33).}$$

Again, we extend the dynamic program such that for each component of each vector a set of associated jobs with fixed number of processors is stored. This increases the space needed to store the vectors, but it is still polynomial in n. Due to this extension, we can create a list of *non-malleable* jobs based on a vector $v \in V$. We denote this distinct list by $L(v)$.

In contrast to the previous dynamic programs, V might not contain a vector corresponding to the vector induced by a nearly optimal solution. However, we show in the following lemma that there is a vector $v \in V$ that is nearly optimal.

Lemma 18. *Let S be an optimal schedule for the scaled instance (i.e. $\mathrm{h}(S) \leq 1$). Then there exists $v \in V$ such that*

$$\mathrm{OPT}(L(v)) \leq (1+\tilde{\varepsilon}),$$

where $\mathrm{OPT}(L(v))$ denotes an optimal solution for $L(v)$ and $\tilde{\varepsilon}$ is some constant depending on ε'.

Proof. Let S be an optimal schedule and let S' be the nearly optimal schedule with simpler structure (i.e. rounded and repacked) and let L^d be the set of discarded jobs. Then $\mathrm{h}(S') \leq (1+3\delta)\,\mathrm{h}(S)$ and $\mathrm{A}(L^d) \leq 2\delta + \varepsilon'$ (see sections 6.1.3, 6.1.4). Note that in S' the number of processors for each job is fixed and for each tall job the slot it is scheduled in is given.

Create v' as follows. We add each job basically in the same way as in the dynamic program. In distinction to the dynamic program, the component a job is added to is determined by S'. The resulting vector v' might differ from the induced vector, since in each adding step the number of assigned processors might be different to the number assigned by S'. The following discrepancies might occur. For each case we argue why the upper bound for an optimal solution for $L(v')$ still holds.

tall jobs The number of assigned processors might be smaller in v' than in S'. Obviously, this does not affect the bound.

low-wide jobs The number of processors assigned is chosen in the same interval, but the height is minimized. If we repack the container with the mKR algorithm, we round up the widths of these jobs to the upper bound of the interval. Consequently, only the height of the job is affecting the solution, but the height is equal or even lower.

small/discarded jobs The number of processors might be different, but the area of the job is minimized. Since only area arguments are used for small or discarded jobs, the bound is not affected.

The dynamic program includes the same adding steps as we used for generating v'. Thus, V contains a vector v that is equivalent to v', that is, the values of all components are equal.

For the problem $P\text{poly}|\text{fnct}_j|C_{\max}$ this is sufficient, since it is possible to schedule the tall jobs fractionally and the other jobs are scheduled using only area arguments (small or discarded jobs) or height arguments (low-wide jobs). Thus, in this case

$$S(v) \leq (1+3\delta)\,\mathrm{h}(S)$$

and

$$L^d(v) \leq 2\delta + \varepsilon',$$

where $S(v)$ is the schedule with simpler structure for $L(v)$ and $L^d(v)$ is the set of discarded jobs. We add the discarded jobs using the NFDH algorithm. This leads to an additional strip of height bounded by

$$2(2\delta+\varepsilon') + \delta,$$

since the height of all discarded jobs is bounded by δ. Thus in total, we have

$$\begin{aligned}\mathrm{OPT}(L(v)) &\leq (1+3\delta)\,\mathrm{h}(S) + 5\delta + 2\varepsilon' \\ &\leq 1 + 2\varepsilon' + 8\delta \\ &\leq 1+\tilde{\varepsilon}\end{aligned}$$

for $\tilde{\varepsilon} := 2\varepsilon' + 8\delta$.

Unfortunately, for $P\text{poly}|\text{fnct\_line}_j|C_{\max}$ this is not sufficient. Since in v might be jobs which are wider than the jobs in an optimal solution, there might not be a feasible solution for v although it is equivalent to a nearly optimal solution.

Therefore, we guess a subset $L_{(K)}$ with K jobs (where K is the same constant as in section 4.4.4). For each of these K jobs, we try all numbers of processors such that this job is tall. In the following, we consider only vectors $v \in V$ where all other tall jobs have area smaller than the jobs in $L_{(K)}$. This ensures that the jobs $L_{(K)}$ are chosen in the corresponding non-malleable algorithm for pre-positioning (see section 4.2). Since we try all possibilities, we find the set corresponding to the nearly optimal solution eventually. For the remaining jobs we have the same arguments as for the $P\text{poly}|\text{fnct}_j|C_{\max}$ case. The (remaining) tall jobs are packed fractionally. The upper bound for the low-wide jobs depends only on the height of the jobs and the bound for the small and the discarded jobs depends only on area arguments. If we annotate the jobs from $L_{(K)}$ in v with the guessed number of processors, we have the same result as for the previous case, i.e.

$$\mathrm{OPT}(L(v)) \leq 1+\tilde{\varepsilon}. \qquad \square$$

6.3 The Algorithm

Using Lemma 18 we get the following theorem.

Theorem 19.

(a) For every $\varepsilon > 0$ there exists an algorithm A for every instance I of $P\text{poly}|\text{fnct\_line}_j|C_{\max}$ such that

$$A(I) \leq (1.5+\varepsilon)\,\mathrm{OPT}(I)$$

holds and the running time is polynomial in n, where $A(I)$ is the length of the schedule for instance I generated by algorithm A and $\mathrm{OPT}(I)$ is the length of an optimal schedule for instance I.

(b) For every $\varepsilon > 0$ there exists an algorithm A for every instance I of $P\mathrm{poly}|fnct_j|C_{\max}$ such that

$$A(I) \leq (1+\varepsilon)\,\mathrm{OPT}(I)$$

holds and the running time is polynomial in n, where $A(I)$ is the length of the schedule for instance I generated by algorithm A and $\mathrm{OPT}(I)$ is the length of an optimal schedule for instance I.

Proof. Since we use the algorithms for $P\mathrm{poly}|\mathrm{size}_j|C_{\max}$ (Algorithm 2) and $P\mathrm{poly}|\mathrm{line}_j|C_{\max}$ (Algorithm 1), respectively, it remains to show that there exists an algorithm to find a proper assignment of a number of required processors to each job.

Lemma 18 shows that among all vectors generated by the dynamic program is at least one which allows a nearly optimal solution. Thus, after enumerating all possibilities (for pseudo-code see Algorithm 3), the algorithm for $P\mathrm{poly}|\mathrm{line}_j|C_{\max}$ or $P\mathrm{poly}|\mathrm{size}_j|C_{\max}$ finds a nearly optimal solution. □

7 Conclusion

In this paper, we have shown that the problem of scheduling parallel jobs can be solved within $(1+\varepsilon')$ of the optimum, if we restrict the instances such that the number of machines is polynomially bounded in the number of jobs, $P\mathrm{poly}|\mathrm{size}_j|C_{\max}$. Furthermore, we presented an extension to the problem of scheduling malleable jobs, $P\mathrm{poly}|\mathrm{fnct}_j|C_{\max}$. These are in a sense the best results possible, since the problems are NP-hard in the strong sense. However, the running times of the presented algorithms are far from practical, so the obvious question is if the running times can be improved or whether there exists an efficient PTAS (EPTAS), i.e. an algorithm with running time $O(f(\varepsilon^{-1})n^c)$ for a constant c.

For $P\mathrm{poly}|\mathrm{line}_j|C_{\max}$ and $P\mathrm{poly}|\mathrm{fnct\_line}_j|C_{\max}$ we presented $(1.5+\varepsilon)$ approximation algorithms. The existence of a PTAS is still open for these problems. Thus, it is an interesting question if a lower bound for the approximation ratio can be shown or if algorithms with better approximation ratio exist. Of course, an improvement of the running time would also be interesting.

We assume that the described approach can also be applied to other scheduling problems like resource constrained scheduling.

Acknowledgements. The authors thank Roberto Solis-Oba for his support, time, and hospitality at the University of Western Ontario and for many helpful discussions.

Algorithm 3: Algorithm for the malleable case

Input: Set of jobs $L = \{R_i \mid i \in \{1,\dots,n\}\}$, and precision ε

Output: A schedule S with

$$\mathrm{h}(S) \leq \begin{cases} (1.5+\varepsilon)\,\mathrm{OPT} & \text{contiguous case, } P\text{poly}|\text{fnct\_line}_j|C_{\max} \\ (1+\varepsilon)\,\mathrm{OPT} & \text{non-contiguous case, } P\text{poly}|\text{fnct}_j|C_{\max} \end{cases}$$

`/* see section 6.1.1 */`

Let v be the height of the solution generated by 2-approximation

foreach $v^* \in \{(1+0\varepsilon)\frac{v}{2}, (1+1\varepsilon)\frac{v}{2}, \dots, (1+\lceil\frac{1}{\varepsilon}\rceil\,\varepsilon)\frac{v}{2}\}$ **do**

- `/* see section 6.1.2 */`
- **foreach** $k \in \{2,\dots,\frac{2}{\varepsilon'}\}$ **do**
  - Set $\delta \leftarrow \sigma_{k-1}$
  - Set $\gamma \leftarrow \sigma_k$
- `/* see section 6.1.3 */`
- Define $\tilde{p}_i(\ell)$ for each $R_i \in L$
- **foreach** *Vector of widths* $(b_1,\dots,b_{\hat{m}})$ **do**
  - Calculate V by dynamic program `/* see section 6.2 */`
  - **foreach** $L_{(K)}$ `/* Problem` $P\text{poly}|\text{fnct\_line}_j|C_{\max}$ `only */` **do**
    - **foreach** $v \in V$ **do**
      - Use Algorithm 1 or 2 to find a solution
      - Save solution

Choose schedule with minimal length

References

[1] Abdel Krim Amoura, Evripidis Bampis, Claire Kenyon, and Yannis Manoussakis. Scheduling independent multiprocessor tasks. *Algorithmica*, 32(2):247–261, 2007.

[2] Krishna P. Belkhale and Prithviraj Banerjee. A scheduling algorithm for parallelizable dependent tasks. In *Proceedings of the 5th International Parallel Processing Symposium (IPPS 1991)*, pages 500–506, 1991.

[3] Stefan Bischof and Ernst W. Mayr. On-line scheduling of parallel jobs with runtime restrictions. *Theoretical Computer Science*, 268(1):67–90, 2001.

[4] Jacek Błażewicz, Klaus H. Ecker, Erwin Pesch, Günter Schmidt, and Jan Węglarz. *Handbook on Scheduling: From Theory to Applications (International Handbooks on Information Systems)*. Springer, 2007.

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

[6] Thomas Decker, Thomas Lücking, and Burkhard Monien. A 5/4-approximation algorithm for scheduling identical malleable tasks. *Theoretical Computer Science*, 361(2):226–240, 2006.

[7] Maciej Drozdowski. Scheduling multiprocessor tasks – an overview. *European Journal of Operational Research*, 94(2):215–230, 1996.

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

[9] Anja Feldmann, Ming-Yang Kao, Jiří Sgall, and Shang-Hua Teng. Optimal on-line scheduling of parallel jobs with dependencies. *Journal of Combinatorial Optimization*, 1(4):393–411, 1998.

[10] Anja Feldmann, Jiří Sgall, and Shang-Hua Teng. Dynamic scheduling on parallel machines. *Theoretical Computer Science (Special Issue on Dynamic and On-line Algorithms)*, 130(1):49–72, 1994.

[11] Michael R. Garey and Ronald L. Graham. Complexity results for multiprocessor scheduling under resource constraints. *SIAM Journal on Computing*, 4(4):397–411, 1975.

[12] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, New York, 1979.

[13] Klaus Jansen. Scheduling malleable parallel tasks: An asymptotic fully polynomial time approximation scheme. *Algorithmica*, 39(1):59–81, 2004.

[14] Klaus Jansen and Lorant Porkolab. Improved approximation schemes for scheduling unrelated parallel machines. *Mathematics of Operations Research*, 26(2):324–338, 2001.

[15] Klaus Jansen and Lorant Porkolab. Linear-time approximation schemes for scheduling malleable parallel tasks. *Algorithmica*, 32(3):507–520, 2002.

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

[17] Klaus Jansen and Roberto Solis-Oba. New approximability results for 2-dimensional packing problems. In *Symposium on Mathematical Foundations of Computer Science (MFCS 2007)*, volume 4708 of *Lecture Note in Computer Science*, pages 103–114, 2007.

[18] Klaus Jansen and Hu Zhang. Scheduling malleable tasks with precedence constraints. In *Proceedings of the seventeenth annual ACM symposium on Parallelism in Algorithms and Architectures (SPAA 2005)*, pages 86–95, 2005.

[19] Klaus Jansen and Hu Zhang. An approximation algorithm for scheduling malleable tasks under general precedence constraints. *ACM Transactions on Algorithms (TALG)*, 2(3):416–434, 2006.

[20] Berit Johannes. Scheduling parallel jobs to minimize the makespan. *Journal of Scheduling*, 9(5):433–452, 2006.

[21] Claire Kenyon and Eric Rémila. A near optimal solution to a two-dimensional cutting stock problem. *Mathematics of Operations Research*, 25:645–656, 2000.

[22] Renaud Lepère, Denis Trystram, and Gerhard J. Woeginger. Approximation algorithms for scheduling malleable tasks under precedence constraints. *International Journal of Foundations of Computer Science (IJFCS)*, 13(4):613–627, 2002.

[23] Joseph Y-T. Leung, editor. *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*. Chapman and Hall/CRC, 2004.

[24] Walter Ludwig and Prasoon Tiwari. Scheduling malleable and nonmalleable parallel tasks. In *Proceedings of the Fifth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 1994)*, pages 167–176. ACM Press, 1994.

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

[26] Edwin Naroska and Uwe Schwiegelshohn. On an on-line scheduling problem for parallel jobs. *Information Processing Letters*, 81(6):297–304, 2002.

[27] Ingo Schiermeyer. Reverse-fit: A 2-optimal algorithm for packing rectangles. In *Proceedings of the Second Annual European Symposium on Algorithms (ESA 1994)*, volume 855 of *Lecture Note in Computer Science*, pages 290–299. Springer-Verlag, 1994.

[28] Jiří Sgall. On-line scheduling. In Amos Fiat and Gerhard J. Woeginger, editors, *Online Algorithms — The State of the Art*, volume 1442 of *Lecture Notes in Computer Science*, pages 196–231. Springer-Verlag, 1998.

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

[30] John Turek, Joel L. Wolf, and Philip S. Yu. Approximate algorithms for scheduling parallelizable tasks. In *Proceedings of the 4th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA 1992)*, pages 323–332, 1992.

[31] Deshi Ye and Guochuan Zhang. On-line scheduling mesh jobs with dependencies. *Theoretical Computer Science*, 372(1):94–102, 2007.