Sharing is Caring: Multiprocessor Scheduling with a Sharable Resource

Peter Kling
University of Hamburg
Vogt-Kölln-Str. 30
Hamburg, Germany
peter.kling@uni-hamburg.de

Alexander Mäcker
Paderborn University
Heinz Nixdorf Institute
Fürstenallee 11
Paderborn, Germany
alexander.maecker@upb.de

Sören Riechers
Paderborn University
Heinz Nixdorf Institute
Fürstenallee 11
Paderborn, Germany
soeren.riechers@upb.de

Alexander Skopalik
Paderborn University
Heinz Nixdorf Institute
Fürstenallee 11
Paderborn, Germany
alexander.skopalik@upb.de

ABSTRACT

We consider a scheduling problem on m identical processors sharing an arbitrarily divisible resource. In addition to assigning jobs to processors, the scheduler must distribute the resource among the processors (e.g., for three processors in shares of 20%, 15%, and 65%) and adjust this distribution over time. Each job j comes with a size $p_j \in \mathbb{R}$ and a resource requirement $r_j > 0$. Jobs do not benefit when receiving a share larger than r_j of the resource. But providing them with a fraction of the resource requirement causes a linear decrease in the processing efficiency. We seek a (non-preemptive) job and resource assignment minimizing the makespan.

Our main result is an efficient approximation algorithm which achieves an approximation ratio of $2+1/(m-2)$. It can be improved to an (asymptotic) ratio of $1+1/(m-1)$ if all jobs have unit size. Our algorithms also imply new results for a well-known bin packing problem with splittable items and a restricted number of allowed item parts per bin.

Based upon the above solution, we also derive an approximation algorithm with similar guarantees for a setting in which we introduce so-called tasks each containing several jobs and where we are interested in the average completion time of tasks (a task is completed when all its jobs are completed).

CCS CONCEPTS

• **Theory of computation → Scheduling algorithms**;

This work is partially supported by the German Research Foundation (DFG) within the Collaborative Research Center "On-The-Fly Computing" (SFB 901) and the German Academic Exchange Service (DAAD).

SPAA '17, July 24–26, 2017, Washington, DC, USA
© 2017 Copyright held by the owner/author(s). Publication rights licensed to Association for Computing Machinery.
ACM ISBN 978-1-4503-4593-4/17/07...$15.00
https://doi.org/10.1145/3087556.3087578

KEYWORDS

multiprocessor scheduling; approximation algorithm; resource constraints; shared resources; bin packing with cardinality constraints and splittable items

1 INTRODUCTION

Multiprocessor scheduling is a classical resource allocation problem. In its simplest version, a computing system consisting of m identical processors has to execute n independent jobs of possibly different workloads. The objective is to find an assignment of jobs to processors that minimizes some quality of service measure like the makespan (latest completion time of any job) or average completion time (average time a job has to wait for its completion). Specific results differ widely depending on additional model parameters: Is preemption (pausing and resuming jobs) allowed? Can jobs be migrated from one to another processor? Is there any additional knowledge about the jobs (like size, priority, or dependencies)? Leung [14] gives a good overview of these and many more.

This work considers the following multiprocessor model: In addition to the processors and (non-preemptive) jobs, there is a *common finite resource* (think of bandwidth or power supply) that is to be shared by the processors. The scheduler controls the resource assignment, which can be adjusted over time. We assume that the resource can be divided arbitrarily between the processors. For example, the scheduler might distribute the total available bandwidth for a few processor cycles in portions of 20%, 35%, and 45% among three available processors and change it later to 10%, 85%, and 5%, depending on how communication intensive the currently processed jobs are.

The dependence of different jobs on the resource might vary a lot. In the bandwidth example, some jobs might be very data-intensive and require a lot of communication while others do not communicate at all. We model this aspect via a job's *resource requirement*. This is a positive value that indicates what portion of the resource is needed to finish one unit of the job's workload. Providing the job with a higher share of the resource does not speed it up (it cannot use the excess bandwidth). But assigning it a significantly smaller share might slow the job down drastically. As a first step towards such a *scalable resource* model in job scheduling, we consider a performance decrease that depends linearly on the resource: for example, if a job of unit size receives $1/k$-th ($k > 1$) of its resource requirement during each time step it is executed, its processing takes $\lceil k \rceil$ steps. Note that this model gives insights on scenarios where resource requirement is the bottleneck of the system, which is often the case in today's big data applications. In contrast, the aspect of processing power is disregarded by assuming that sufficient processing power is available at any time.

The first part of this article studies the above model for the objective of minimizing the makespan. We refer to this problem as ***S**hared Resource **J**ob-**S**cheduling* (SoS) (see Section 1.1 for the full, formal specification). In the second part, we extend this model to the setting of *composed services*, where the processors have to finish a set of *tasks* and each task consists itself of a set of jobs (each of which has its own resource requirement). A task is finished when all its jobs are finished. We aim at minimizing the average completion time of all tasks. This is a typical setting in cloud computing, where users submit applications (tasks) composed of many smaller parts (jobs) and require the output of all these parts. We refer to this setting as ***S**hared Resource **T**ask-**S**cheduling* (SaS).

1.1 Model & Notation

Consider a system of $m \in \mathbb{N}$ processors from the set $M := [m] = \{1,2,\ldots,m\}$ and $n \in \mathbb{N}$ *jobs* from the set $J := [n]$. There is a *resource* that is to be shared by the processors. In each time step $t \in \mathbb{N}$, each processor i is assigned a share $R_i(t) \in [0,1]$ of the resource. The resource may not be overused, such that we require $\sum_{i \in [m]} R_i(t) \leq 1$. Each processor can process at most one job per time step and each job can be processed by at most one processor. A job j has a *processing volume (size)* $p_j \in \mathbb{R}_{>0}$ and a *resource requirement* $r_j \in \mathbb{R}_{>0}$. Note that we will assume $p_j \in \mathbb{N}$ for convenience throughout this paper, but all our results carry over to $p_j \in \mathbb{R}_{>0}$ (also see explanation below Equation (1)). Without loss of generality, we assume $r_1 \leq r_2 \leq \cdots \leq r_n$. The resource requirement specifies what portion of the resource is needed to finish one unit of a job's processing volume. More exactly, assume job j is processed by processor i during time step t. Then exactly $\min(R_i(t)/r_j, 1)$ units of j's processing volume are finished during that time step. A job is *finished* once all p_j units of its processing volume have been finished. Preemption and migration of jobs is not allowed. The objective is to find a *schedule* S (i.e., a resource and job assignment) having minimal *makespan* $|S|$ (the number of time steps until all jobs are finished). We refer to this problem as ***S**hared Resource **J**ob-**S**cheduling* (SoS). As a special case, we sometimes consider $p_j = 1$ for all $j \in J$. We refer to this as the setting of jobs with unit size.

During our analysis, it will be convenient to adopt the following perspective on SoS: Given a schedule S, consider job j processed on processor i during time step t. Without loss of generality, we assume $R_i(t) \leq r_j$ (setting $R_i(t)$ to $\min\{R_i(t), r_j\}$ yields a valid schedule with the same makespan). Let t_1 and t_2 denote the time steps when j was started and finished, respectively. Since j is finished, we have $\sum_{t=t_1}^{t_2} R_i(t)/r_j \geq p_j$. Rearranging yields $\sum_{t=t_1}^{t_2} R_i(t) \geq r_j \cdot p_j$. Thus, if we define $s_j := r_j \cdot p_j$ as the *total resource requirement* of job j, we can think of j as being finished once the total resource shares it received over time equal (at least) s_j. We define $s_j(t) := s_j - \sum_{t'=t_1}^{t} R_i(t')$ as the total resource requirement remaining after time step t. Note that job j is finished in the first time step t for which $s_j(t) = 0$. We use $J(t) := \{j \in J \mid s_j(t) > 0\}$ to denote the set of jobs that are not finished after time step t. Throughout the paper, for any set S, we write $\max S := \max_{s \in S} s$.

A polynomial time algorithm A is called to have an *(absolute) approximation ratio* of α if, on any instance I, the schedule S produced by A satisfies $|S|/|\textsc{Opt}| \leq \alpha$, where Opt denotes an optimal solution for I. A has an *asymptotic ratio* of α if, on any instance, $|S| = \alpha \cdot |\textsc{Opt}| + o(|\textsc{Opt}|)$.

Lower Bounds. Let Opt denote an optimal schedule. Two simple lower bounds for any schedule, including Opt, are $\lceil s_0(J) \rceil$ and $\frac{1}{m} \cdot \sum_{j \in J} \lceil s_j/r_j \rceil$. The former holds since each job needs to receive a total of s_j resource shares over time. The latter holds since each job must be split in at least $\lceil s_j/r_j \rceil$ parts, and each such part needs a dedicated machine in one time step to be processed. Thus, we have

$$|\textsc{Opt}| \geq \max\left\{ \lceil s_0(J) \rceil, \frac{1}{m} \cdot \sum_{j \in J} \left\lceil \frac{s_j}{r_j} \right\rceil \right\}. \tag{1}$$

Note that these lower bounds on Opt remain valid if allowing $p_j \in \mathbb{R}$ and rescaling $p'_j := \lceil p_j \rceil$ and $r'_j := s_j/p'_j$, as this modification maintains the s_j and by $\lceil p'_j \rceil = \lceil p_j \rceil$ the bound in Equation (1) remains the same. Also, the lower bounds remain valid for the preemptive setting as they are only based on observations of the overall workload.

1.2 Related Work

In the following we survey related problems, focusing on *resource constrained scheduling* and *bin packing with cardinality constraints*. As it turns out, the latter is among the most relevant related work, as it can be seen as a special case of our problem (except for the difference that preemption is allowed).

Bin Packing. The problem supposedly closest related to SoS is *bin packing with cardinality constraints and splittable items* as introduced in [4]. In this problem, a set of n items needs to be packed into as few bins of capacity one as possible. In contrast to standard bin packing, items can have an arbitrary size in $(0,\infty)$ and may be split and distributed among different bins. However, there is a constraint on the maximum number of (parts of) different items that may be packed into a single bin given by some predefined value k. Chung et al. [4] proved this problem to be strongly NP-hard for $k = 2$ and provided a simple approximation algorithm with asymptotic approximation ratio of $3/2$ (also for $k = 2$). Epstein and van Stee [7] extended the NP-hardness to any fixed $k \geq 2$. They also gave efficient algorithms with asymptotic approximation ratio $7/5$ for $k = 2$ and an absolute approximation ratio of $2 - 1/k$ for $k \geq 2$, respectively. Finally, Epstein et al. [5] presented an efficient polynomial-time approximation scheme (EPTAS) for the case $k = o(n)$. They also proved that for $k = \Theta(n)$ a polynomial-time approximation algorithm with a ratio smaller than $3/2$ cannot exist unless P = NP.

Note that bin packing with cardinality constraints and splittable items is, except for the lack of the notion "preemption", equivalent to SoS with unit size jobs: If items correspond to jobs of size one and each bin is identified with one time step, the packing of a bin describes the jobs executed in this time step and the part size of an item corresponds to the share of the resource the respective job gets. The cardinality constraint k corresponds to having k processors.

A variant of bin packing with cardinality constraints and splittable items, which is motivated by a scheduling problem similar to SoS, was studied by König et al. [12]. This work assumes the items to be edges of a tree and the goal is to find a packing into as few bins as possible such that no bin contains (parts of) items incident to more than k nodes. They proved this problem to be NP-hard in the strong sense for any constant value k and constant degree of the underlying tree. Concerning approximations, two algorithms are presented: For paths NextFit achieves an asymptotic approximation ratio of $2-1/(2(k-1))$ and for general trees an algorithm with approximation ratio $5/2$ was given.

Scheduling. Another strain of closely related works is resource constrained scheduling. Garey and Graham [8] considered a model in which a set of jobs needs to be processed on m parallel processors. Additionally, they assume ℓ resources to be part of the system and, in addition to a workload, each job also has a requirement on each of the resources. In contrast to our model, however, each job needs to be assigned its full resource requirement at each step during its execution. They studied list-scheduling algorithms for this problem and proved an approximation ratio of at most $\min\left\{\frac{m+1}{2}, \ell+2-\frac{2\ell+1}{m}\right\}$.

For the restriction to a single resource, the results discussed above directly imply that their list scheduling algorithm achieves an approximation factor of $3-\frac{3}{m}$. Niemeier and Wiese [15] improved these results by presenting a $(2+\varepsilon)$-approximation algorithm for this problem using techniques such as grouping and linear programming. They also proved that this problem cannot be approximated within an (absolute) approximation ratio less than $\frac{3}{2}$ by a straightforward reduction from the Partition problem. Finally, Jansen et al. [10] very recently presented an asymptotic fully polynomial-time approximation scheme (AFPTAS) for this problem.

A simplified variant of our scheduling problem was studied by Brinkmann et al. [3]. They focused solely on the resource assignment aspect of the problem: Each job was already assigned to one of the m processors, and even the order in which the jobs on each processor had to be processed was fixed. Further restricting the problem to jobs of the same computational size, they proved NP-completeness for variable m and proposed a natural combinatorial approximation algorithm that achieved an approximation ratio of $2-1/m$. They also provided an optimal algorithm for two processors based on dynamical programming. The case for non-unit size jobs as well as incorporating the job assignment (instead of assuming it given) were left as the central open issues. In [2], an exact algorithm was given that runs in polynomial time for any fixed $m \geq 2$. Another related, older scheduling problem that allows an arbitrary (continuous) assignment of a given resource to multiple processors was considered under the name *discrete-continuous* scheduling, but provided mostly heuristic results (cf. [11, 16]).

With respect to Section 4, where we consider a model generalization for tasks that are composed of multiple jobs, [13] should be mentioned. Here, a production model is considered, where tasks represent orders and each job of an order must be processed on a subset of specific machines. In particular they consider a similar objective function for this setting (see Section 4). However, note that these *order scheduling models* do not consider resource sharing in our sense, but instead only the allocation to the (non-identical) machines.

Apart from these resource constrained scheduling results, one should at least briefly mention the related classical multiprocessor makespan scheduling problem. Here, a set of jobs needs to be scheduled on m identical machines so as to minimize the makespan. Note that in the special case of SoS in which all jobs have negligible resource requirements, both problems become equivalent. For scheduling identical machines, a PTAS is known [1] if m is part of the input and for fixed m even an FPTAS is possible [9].

1.3 Our Contribution

We study a new scheduling model for a setting of parallel processors sharing a common scarce resource in terms of its complexity and approximations. Our model is an extension of a simpler variant studied in [3] and is closely related to a well-known bin packing problem [4]. Precisely, our results are as follows:

- We prove SoS and SaS to be strongly NP-hard (Section 2).
- For SoS, we design and analyze a polynomial time algorithm with an approximation ratio of $2+1/(m-2)$ for jobs of arbitrary size and (asymptotically) $1+1/(m-1)$ for unit size jobs (Section 3). Our algorithm is based on the idea of a *maximal sliding window*: We order jobs by non-decreasing resource requirement and (for each time step) create a sliding window trying to find a subset of consecutive jobs such that $m-1$ of these jobs can be finished and the full resource can be used.
- Our algorithm implies the same (asymptotic) guarantee of $1+1/(k-1)$ for bin packing with splittable items and cardinality constraint k. Besides a known PTAS, which has a quite high running time, the best known fast algorithm for this problem has an approximation ratio of $2-1/(k-1)$. For computing centers typically containing a huge amount of processors, this ratio approaches 2 whereas our algorithm is almost optimal.
- We generalize our algorithm to obtain an asymptotic approximation ratio of $2+4/(m-3)$ for SaS where jobs are grouped into tasks and where we aim at minimizing the average completion time of all tasks (Section 4).

2 COMPLEXITY

In the following, we explore the complexity of the SoS problem.

Theorem 2.1. *The SoS problem with jobs of unit size is strongly NP-hard for $m=2$.*

Proof. Bin packing with cardinality constraints and splittable items is equal to our setting with preemption. The NP-hardness of SoS, even for unit size jobs, can hence be shown similarly to the reduction found in [4]. For completeness sake and to show its adaptivity to our setting, it can be found in the full version of this paper. □

Note that the hardness of the general SoS problem directly follows. This also holds for the SaS problem, as it contains the SoS problem as a special case.

As stated before, there is a PTAS [6] for bin packing with cardinality constraints and splittable items if the cardinality constraint (corresponding to the number of processors in our model) is in $o(n)$. This bin packing variant is similar to the unit size version of our problem, but with preemption. However, this PTAS can be adapted easily to the setting without preemption by restricting the set of solutions to non-preemptive schedules. For unit size jobs, this implies a better approximation ratio than our algorithm in Section 3, but at the cost of very high running time.

3 APPROXIMATION ALGORITHM

We provide some additional notation for this section: Let $j \in J, U \subseteq J$ and $t \in \mathbb{N}_0$. We define $r(U) := \sum_{j \in U} r_j$ and $s_t(U) := \sum_{j \in U} s_j(t)$. We say job j is *fractured* at time t if $s_j(t) = k \cdot r_j + q_j(t)$ for some $k \in \mathbb{N}_0$ and $q_j(t) \in (0, r_j)$ (i.e., $s_j(t)$ is not an integer multiple of r_j). Note that, since $s_j(0) = s_j = p_j \cdot r_j$ and $p_j \in \mathbb{N}$, initially no job is fractured. We also define $L_t(U) := \{j \in J(t-1) \mid j < \min U\}$ as the set of jobs remaining at the beginning of time step t that have a resource requirement smaller than any job in U ("left of U"). Similarly, $R_t(U) := \{j \in J(t-1) \mid j > \max U\}$. For convenience, we define $L_t(\emptyset) := \emptyset$ and $R_t(\emptyset) := J(t-1)$.

We continue with the central definition of *maximal (job) windows*, a subset of remaining jobs that can be processed efficiently (see algorithmic intuition below). Our algorithm will ensure that it always processes jobs from such a window. The bulk of the analysis goes towards proving that we can always find a maximal window.

Definition 3.1 (Job Window). A subset of unfinished jobs $W \subseteq J(t-1)$ is called a *job window* for time step t if

(a) $j_1, j_2 \in W \Rightarrow J(t-1) \cap \{j_1, j_1+1, \ldots, j_2\} \subseteq W$,
(b) $r(W \setminus \{\max W\}) < 1$,
(c) $|\{j \in W \mid q_j(t-1) > 0\}| \leq 1$, and
(d) $j \in J(t-1) \setminus W \Rightarrow s_j(t-1) = s_j$.

We say W is k-*maximal* if, additionally, it has size $|W| \leq k$ and the following properties hold:

(e) $|W| < k \Rightarrow L_t(W) = \emptyset$ and
(f) $r(W) < 1 \Rightarrow R_t(W) = \emptyset$.

In other words a window W (of size $\leq m$) is a set of consecutive jobs (Property (a)) such that we can assign all but the rightmost job their full resource requirements (Property (b)). Moreover, W contains all started jobs and at most one of these is fractured (Properties (c) and (d)). To be k-maximal, a window of size at most k must contain either exactly k jobs or lie at the left border, and either utilize the full resource or lie at the right border (Properties (e) and (f)).

Algorithmic Intuition. We design our algorithm such that it has three key properties:

- During any time step t, it processes jobs from an $(m-1)$-maximal window $W_t \subseteq J(t-1)$ (Lemma 3.7).
- If the window W_t is at the left border of the remaining jobs (i.e., $L_t(W_t) = \emptyset$), then this remains true for all $W_{t'}$ with $t' > t$ (Lemma 3.8(a)).
- If the window W_t is at the right border of the remaining jobs (i.e., $R_t(W_t = \emptyset)$), then this remains true for all $t' > t$ (Lemma 3.8(b)).

Note that if W_t is *not* at the left border of the remaining jobs, Properties (b) and (e) of Definition 3.1 imply that we can assign the resource such that at least $m-2$ jobs (all of W except for $\max W_t$) receive their full resource requirement r_j during time step t. Similarly, if W_t is not at the right border of the remaining jobs, Property (f) implies that we can utilize the full resource during time step t.

```
1   for $(t, W) \leftarrow (1, \emptyset)$; $J(t-1) \neq \emptyset$; $t \leftarrow t+1$:
2     $W \leftarrow W \cap J(t-1)$
3     $W \leftarrow$ GrowWindowLeft($W, t, m-1, 1$)
4     $W \leftarrow$ GrowWindowRight($W, t, m-1, 1$)
5     $W \leftarrow$ MoveWindowRight($W, t, 1$)
6
7     if $\exists$ fractured job $\iota \in W$: $F \leftarrow \{\iota\}$
8     else:                              $F \leftarrow \emptyset$
9     if $r(W \setminus F) \geq 1$:
10      process each job $j \in W \setminus (F \cup \{\max W\})$ with resource $r_j$
11      if $F = \{\iota\}$:
12        process job $\iota$ with resource $q_\iota(t)$
13      process job max $W$ with the remaining resource
14    else:
15      process each job $j \in W \setminus F$ with resource $r_j$
16      if $F = \{\iota\}$:
17        process job $\iota$ with resource $\min\{1 - r(W \setminus F), s_\iota(t-1)\}$
18      if resource left and $R_t(W) \neq \emptyset$:
19        assign remaining resource to job $\min R_t(W)$
20        $W \leftarrow W \cup \min R_t(W)$
```

Listing 1: Approximation algorithm for SoS.

```
GrowWindowLeft($W, t, size, R$)
  while $(|W| < size$ and $L_t(W) \neq \emptyset)$ and $r(W) < R$:
    $W \leftarrow W \cup \{\max L_t(W)\}$
  return $W$

GrowWindowRight($W, t, size, R$)
  while $(r(W) < R$ and $R_t(W) \neq \emptyset)$ and $|W| < size$:
    $W \leftarrow W \cup \{\min R_t(W)\}$
  return $W$

MoveWindowRight($W, t, R$)
  while $(r(W) < R$ and $R_t(W) \neq \emptyset)$ and $s_{\min W} = s_{\min W}(t-1)$:
    $W \leftarrow (W \setminus \{\min W\}) \cup \{\min R_t(W)\}$
  return $W$
```

Listing 2: Auxiliary procedures. The parameters *size* and *R* are only to facilitate the algorithm from Section 4. In this section, we call these only with $size = m-1$ and $R = 1$.

Consider the first time step T such that $L_T(W_T) \cup R_T(W_T) = \emptyset$. In particular, W_T contains all remaining jobs. It is not hard to see that these can be finished by our algorithm in $|\text{OPT}|$ time steps. On the other hand, up to time step T the three key properties and the above observations imply that in each time step either at least $m-2$ jobs receive their full resource requirement or the full resource is utilized. In the former case, the lower bound from Equation (1) implies $T \leq \frac{m}{m-2} \cdot |\text{OPT}|$. In the latter case, the same bound implies $T \leq |\text{OPT}|$. Together, this yields an approximation ratio of at most $\frac{m}{m-2} + 1 = 2 + \frac{m-2}{m}$.

A slightly more careful but similar analysis yields Theorem 3.3. We proceed to describe our algorithm. Afterward we show that the three key properties hold and formalize the above argument.

3.1 Algorithm Description

In the following we describe our algorithm. The corresponding pseudocode can be found in Listing 1 (with some auxiliary procedures outsourced to Listing 2). If not explicitly stated otherwise, references to lines refer to Listing 1. Note that the implementation as shown in Listing 1 has only pseudo-polynomial running time. It is not hard to adapt it such that it yields polynomial running time; we describe how to do that in the proof of Theorem 3.3.

Lines 2 to 5 compute an $(m-1)$-maximal window W for this time step. Lines 7 to 20 compute the resource assignment of this time step. The computation of the maximal window W starts by removing any jobs that were finished in the last time step (Line 2). Lines 3 to 5 take the resulting window and greedily grow it first left, then right, and finally move it as far to the right as possible. This way, W becomes $(m-1)$-maximal for this time step.

To compute the resource assignment, let $F := \{\iota\}$ be the set containing the only fractured job of W (or $F := \emptyset$ if there is no fractured job). We distinguish two cases:

Case 1: $r(W \setminus F) \geq 1$
Note that $\iota \neq \max W$, as otherwise Property (b) of Definition 3.1 violates the case assumption. Each job $j \in W$ except for ι and $\max W$ receives its full resource requirement r_j. Job ι receives resource $q_\iota(t-1)$. Any remaining resource is assigned to $\max W$.

Case 2: $r(W \setminus F) < 1$
In this case, each job $j \in W$ except for ι receives its full resource requirement r_j. Job ι receives resource $\min\{1-r(W\setminus F), s_\iota(t-1), r_\iota\}$. If there is resource left, we use it to process $\min R_t(W)$ (this is the only case where we use all m instead of only $m-1$ processors). In that case, we add $\min R_t(W)$ to W.

Our analysis requires that there is always at most one fractured job[1]. The case distinction above is chosen with this goal in mind: If there is no fractured job, all $j \in W \setminus \{\max W\}$ receive their full resource requirement. The remaining resource goes to $\max W$, possibly fracturing it. If there is already a fractured job ι, doing the same might fracture a second job ($\max W$). Instead, we distinguish whether $r(W \setminus \{\iota\}) \geq 1$ or not. If so, we "unfracture" ι and instead fracture $\max W$; $r(W \setminus \{\iota\}) \geq 1$ guarantees that we can still use the full resource, even if $s_\iota(t-1) = \varepsilon \ll r_\iota$. Otherwise, $r(W \setminus \{\iota\}) < 1$ allows us to assign all $j \in W \setminus \{\iota\}$ their full resource requirement and keep only ι fractured (it gets the remaining resource). This case might leave us with some unecessarily wasted resource (if $s_\iota(t-1) = \varepsilon \ll r_\iota$ and $R_t(W) \neq \emptyset$). If so, we finish ι and use the (so far unused) m-th processor to start a new job. We gather this discussion in the following observation.

Observation 3.2. Given an $(m-1)$-maximal window W for the current time step, Lines 7 to 20 compute a resource assignment for jobs in W such that at least $|W|-1$ jobs $j \in W$ receive their full resource requirement r_j, at most one job is fractured after this time step, and at most $|W|$ jobs are started (and not finished) after this time step.

3.2 Analysis

The goal of this section is to prove the following theorem.

Theorem 3.3. *The algorithm from Listing 1 generates a schedule S with approximation ratio $2 + \frac{1}{m-2}$. If jobs have unit size, we get the stronger guarantee $|S| \leq (1 + \frac{2}{m-2}) \cdot |\text{Opt}| + 1$. The algorithm can be implemented with a running time of $O((m+n) \cdot n)$.*

It is not hard to see that for jobs of unit size, a minor algorithm modification avoids to reserve the m-th processor: If jobs have unit size, we have $s_j = r_j$ for all $j \in J$. Note that there will be always at most one started (and, thus, at most one fractured) job: Indeed, by the while loops of the auxiliary procedures, the window can contain at most one job with $s_j = r_j > 1$ (this will be $\max W$). Since for all jobs $j \in W \setminus \{\max W\}$ we have $s_j = r_j \leq 1$ and $r(W \setminus \{\max W\}) < 1$ (Property (b) of Definition 3.1), such j will be finished in the current time step. We can treat the only started job ι in step t as a job with resource requirement $s_\iota(t-1)$ and reorder the jobs accordingly. The next time step will either finish ι or it will once more be the only started job. This modification does not need the reserved processor, so we can use m-maximal instead of $(m-1)$-maximal windows, improving the approximation factor for unit size jobs from $\frac{m}{m-2} = 1 + \frac{2}{m-2}$ to $\frac{m}{m-1} = 1 + \frac{1}{m-1}$. The analysis is analogous to the one given below for the unmodified algorithm.

We now start to provide tools for the proof of Theorem 3.3. We start with some auxiliary claims and then prove the above mentioned key properties in Lemmas 3.7 and 3.8

Claim 3.4. *If Properties (a) to (d) from Definition 3.1 hold for W right before we call the auxiliary procedures, then they also hold at any later point in this time step.*

Proof. Property (a) holds since jobs are added one by one at the left/right borders (Lines 3 and 8 in Listing 2) or one job is removed at the left border and another added at the right border (Line 13). Property (b) is enforced by the while loops' conditions. Property (c) holds since only unstarted (and, thus, unfractured) jobs are added to W. Finally, Property (d) holds since the while loop in Line 12 of Listing 2 ensures that no started jobs are removed. □

Claim 3.5. *If $W = \emptyset$ after Line 2 of Listing 1 in time step t and no job in $J(t-1)$ is started, then W is an $(m-1)$-maximal window when* `MoveWindowRight` *exits.*

Proof. We have $W = \emptyset$ right before the auxiliary procedures are called. In particular, W is a (trivial) window for time step t. We apply Claim 3.4 to get that Properties (a) to (d) of Definition 3.1 hold when `MoveWindowRight` exits. Since the while loops ensure that the window size is at most $m-1$, it remains to show that Properties (e) and (f) hold after the auxiliary procedures.

For Property (e), note that $L_t(W) = L_t(\emptyset) = \emptyset$. Thus, procedure `GrowWindowLeft` exits immediately, leaving $W = \emptyset$. Now, if `GrowWindowRight` exits because of $|W| = m-1$, Property (e) holds (and remains true since `MoveWindowRight` does not change the size of W). Otherwise, if `GrowWindowRight` exits because the condition "$r(W) < 1 \land R_t(W) \neq \emptyset$" is violated, `MoveWindowRight` exits immediately for the same reason. But then, we still have $\min J(t-1) \in W$ (impplying $L_t(W) = \emptyset$) and Property (e) holds.

For Property (f), note that `MoveWindowRight` cannot exit because of the condition "$s_{\min W} = s_{\min W}(t-1)$" (there are no started jobs). Thus, it can only exit because one of the other two conditions is violated, which immediately implies Property (f). □

Claim 3.6. *If $W \neq \emptyset$ after Line 2 of Listing 1 in time step t and the window $\tilde{W}$ computed in the previous time step was $(m-1)$-maximal, then W is a $(m-1)$-maximal window when* `MoveWindowRight` *exits.*

Proof. We have $W = \tilde{W} \cap J(t-1)$ right before the auxiliary procedures are called. $\tilde{W}$ was a maximal window, and removing finished jobs cannot violate Properties (a) to (d) of Definition 3.1. We apply Claim 3.4 to get that Properties (a) to (d) hold when procedure `MoveWindowRight` exits. It remains to show that Properties (e) and (f) hold after the auxiliary procedures.

For Property (e), we first show that it holds after `GrowWindowLeft`. When we call `GrowWindowLeft` for window W, note that $L_t(W) = L_{t-1}(\tilde{W})$. Thus, if $L_{t-1}(\tilde{W}) = \emptyset$, Property (e) holds trivially after

[1]Otherwise, we could end up with $m-1$ fractured jobs $j \in W$, each with $s_j(t-1) = \varepsilon \ll r_j$. This may cause almost the full resource to be wasted during that step.

GrowWindowLeft (the while loop exits immediately because of the condition "$L_t(W)\neq\emptyset$"). If $L_{t-1}(\tilde{W})\neq\emptyset$, since Property (e) holds for window $\tilde{W}$, we have $|\tilde{W}|=m-1$. Note that for all $j\in L_t(W)=L_{t-1}(\tilde{W})$ and $j'\in\tilde{W}$ we have $r_j\leq r_{j'}$ (by the job ordering). This implies that we cannot violate condition "$r(W)<1$" of the while loop of GrowWindowLeft before adding $|\tilde{W}|-|W|$ jobs. Moreover, we cannot violate "$|W|\leq m-1$" before adding $|\tilde{W}|-|W|$ jobs (since $|W|+(|\tilde{W}|-|W|)=|\tilde{W}|\leq m-1$). Thus, GrowWindowLeft adds at least min $\{|L_t(W)|,|\tilde{W}|-|W|\}$ jobs to W. If the minimum equals $|L_t(W)|$ we added all jobs left of W and Property (e) holds. If the minimum equals $|\tilde{W}|-|W|$, Property (e) holds since the resulting window has size at least $|W|+(|\tilde{W}|-|W|)=|\tilde{W}|=m-1$.

So Property (e) holds for W right before GrowWindowRight. We show that it still holds after procedure MoveWindowRight. The statement is trivial if $|W|=m-1$ (both procedures do not decrease W). Otherwise, we use that W has Property (e) to get $L_t(W)=\emptyset$. If GrowWindowRight exits because the condition "$r(W)<1\wedge R_t(W)\neq\emptyset$" got violated, MoveWindowRight exits immediately for the same reason, leaving $L_t(W)=\emptyset$. Otherwise, if GrowWindowRight exits because of $|W|=m-1$, this is maintained by MoveWindowRight and, thus, Property (e) holds after MoveWindowRight.

All that remains is to prove that Property (f) holds after procedure MoveWindowRight. Consider the conditions of the while loop in Line 12 of Listing 2. Property (f) holds if the while loop exits because the condition "$r(W)<1\wedge R_t(W)\neq\emptyset$" got violated. So assume it exits only because of the condition "$s_{\min W}=s_{\min W}(t-1)$". At that moment, we have a window W with $r(W)<1$, $R_t(W)\neq\emptyset$, and $s_{\min W}>s_{\min W}(t-1)$. The first two imply that $|W|=m-1$, since otherwise GrowWindowRight would not have exited. The inequality $s_{\min W}>s_{\min W}(t-1)$ implies that job min W is already started, so it must have been in the last time step's window $\tilde{W}$. Now, since W has maximal size $m-1$ and its leftmost job was also in $\tilde{W}$, we get $r(\tilde{W})\leq r(W)<1$ as well as $R_t(W)\subseteq R_{t-1}(\tilde{W})$. But since $\tilde{W}$ had Property (f), we know $R_{t-1}(\tilde{W})=\emptyset$. Together, $R_t(W)=\emptyset$, a contradiction. □

Lemma 3.7. *Fix $t\in\mathbb{N}_0$ and consider the job window W processed during time step t. Then W is an $(m-1)$-maximal window for time step t.*

Proof. We prove the statement inductively. In the first time step $t=1$, we start with $W=\emptyset$ (initialization by the for loop) and no job has been started. We apply Claim 3.5 to get that W is an $(m-1)$-maximal window after the auxiliary procedures. For $t>1$ we either have $W=\emptyset$ or $W\neq\emptyset$ after Line 2 of Listing 1. In the former case, we once more apply Claim 3.5. In the latter case, we apply Claim 3.6. In both cases, we get that W is an $(m-1)$-maximal window after the auxiliary procedures, proving the desired statement. □

Lemma 3.8. *Let $\tilde{W}\subseteq J(t-2)$ and $W\subseteq J(t-1)$ be the $(m-1)$-maximal windows processed during time step $t-1$ and t, respectively. Then*

(a) $L_{t-1}(\tilde{W})=\emptyset\Rightarrow L_t(W)=\emptyset$ and
(b) $R_{t-1}(\tilde{W})=\emptyset\Rightarrow R_t(W)=\emptyset\wedge r(W)\leq r(\tilde{W})$.

Proof. For (a), note that W starts out as $\tilde{W}\cap J(n-1)$ in time step t. Since $L_t(W)=L_{t-1}(\tilde{W})=\emptyset$, we only add jobs from $R_t(W)=R_{t-1}(\tilde{W})$. All these jobs have a larger resource requirement than any job in $\tilde{W}$. As a consequence, after GrowWindowRight we have $|W|\leq|\tilde{W}|$. If $|W|<|\tilde{W}|\leq m-1$, MoveWindowRight exits immediately and we have $L_t(W)=\emptyset$. Otherwise, if $|W|=|\tilde{W}|$ after GrowWindowRight, we must have $r(W)\geq r(\tilde{W})$ and $R_t(W)\subseteq R_{t-1}(\tilde{W})$. Since $\tilde{W}$ is $(m-1)$-maximal in time step $t-1$, this implies either $r(W)\geq 1$ or $R_t(W)=\emptyset$, such that MoveWindowRight exits immediately and leaves $L_t(W)=\emptyset$. This proves (a). The first part of Statement (b) follows analogously. The second part holds either since $|W|=|\tilde{W}|=m-1$ and jobs that were finished in $\tilde{W}$ are exchanged for jobs with at most the same resource requirement, or since $W\subseteq\tilde{W}$ (if $L_{t-1}(\tilde{W})=\emptyset$). □

With these lemmas, we are ready to prove Theorem 3.3.

Proof of Theorem 3.3. We consider the schedule S produced by our algorithm from Listing 1. By Lemma 3.7, the jobs processed during each time step $t\in\mathbb{N}$ are contained in a maximal window W_t for time step t. We define $T_L:=\min\{t\in\mathbb{N}\mid|W_t|<m-1\}$ and, similarly, $T_R:=\min\{t\in\mathbb{N}\mid r(W_t)<1\}$. By Properties (e) and (f) of Definition 3.1 and Lemma 3.8 we have $L_t(W_t)=R_t(W_t)=\emptyset$ and $r_t(W_t)<1$ for all $t\geq\max\{T_L,T_R\}=:T$. In particular, the former implies $W_t=J(t-1)$ for all $t\geq T$. Combining these insights we get that for each $t\geq T$, each of the at most $|W_t|\leq|W_T|<m-1$ remaining jobs gets its full resource requirement. Thus, each $j\in W_T$ is finished after exactly $\lceil s_j(T-1)/r_j\rceil$ additional time steps. Let $p:=\max\{s_j(T-1)/r_j\mid j\in W_T\}$. Note that $|S|=T-1+\lceil p\rceil$. We distinguish two cases:

Case 1: $T=T_L$

For each $t<T$ we have $|W_t|=m-1$. Thus, by Observation 3.2, at least $|W_t|-1=m-2$ jobs $j\in W_t$ receive their full resource requirement r_j. Remember that $p_j=s_j/r_j$. An average argument gives

$$T-1\leq\frac{\sum_{j\in J}p_j-\lceil p\rceil}{m-2}\leq|\textsc{Opt}|\cdot\frac{m}{m-2}-\frac{\lceil p\rceil}{m-2}.$$

Combining everything with the lower bound $|\textsc{Opt}|\geq\lceil p\rceil$ we compute

$$|S|=T-1+\lceil p\rceil\leq|\textsc{Opt}|\cdot\frac{m}{m-2}-\frac{\lceil p\rceil}{m-2}+\lceil p\rceil$$
$$\leq|\textsc{Opt}|\cdot\left(\frac{m}{m-2}+1-\frac{1}{m-2}\right)=|\textsc{Opt}|\cdot\left(2+\frac{1}{m-2}\right).$$

Case 2: $T=T_R$

For each $t<T$ we have $r(W_t)\geq 1$. Using that Opt cannot overuse the resource, we see $T-1\leq r(J)\leq|\textsc{Opt}|$. Similar to the first case, we compute $|S|=T-1+\lceil p\rceil\leq 2\cdot|\textsc{Opt}|$.

The result for jobs of unit size follows by realizing that $|S|=T-1+1=T$. Thus, the bounds above give $|S|\leq|\textsc{Opt}|\cdot\left(1+\frac{2}{m-2}\right)+1$ (Case 1) and $|\textsc{Opt}|+1$ (Case 2).

For the the running time, first note that the implementation given in Listing 1 has actually pseudo-polynomial running time (it depends on the sum $\sum_{j\in J}p_j$, since each job j needs a dedicated processor for at least p_j time steps). However, note that if no job is finished in the current time step, the maximal window in the next step will be identical to the current maximal window. With this observation, we can calculate via a simple linear equation after how many step with the current maximal window the first job in the window will be finished. This allows us to "skip" time steps where no job is finished. Thus, given the maximal window in a time step t, we go over the $O(m)$ jobs in the window and find the first one(s) that will be finished under the current resource assignment. To compute the next maximal window, we remove the finished jobs and grow the window left/right. This can be computed in time $O(|W|) = O(m)$ (each adding/removal can be implemented trivially in constant time using doubly linked lists). Then we move the window up to n steps to the right, which can be done in time $O(n)$. Since this always eliminates at least one job from the old maximal window, this repeats at most $O(n)$ times, yielding a total running time of $O(n \cdot (m+n))$. □

As the lower bounds on Opt are still valid for the preemptive setting (see description below Equation (1)), and the upper bounds of the algorithm obviously do not increase by allowing preemption, our results for unit size jobs carry over to bin packing with cardinality constraints and splittable items. Our algorithm scales well with the number of processors in contrast to existing simple (i.e. fast) algorithms, but (obviously) does not reach the approximation ratio of the existing EPTAS [5]. Note that in the following corollary, k denotes the cardinality constraint as this is common notion in the related literature.

Corollary 3.9. *Our results give an algorithm for bin packing with cardinality constraints and splittable items [4] with asymptotic approximation ratio* $1+1/(k-1)$ *and running time* $O((k+n)n)$.

Proof. The lower bounds on the optimum remain valid for the preemptive setting as they only use a notion of overall workload. Also, our algorithm still computes a valid solution, as the preemptive setting removes a constraint. The claim follows. □

4 THE SHARED RESOURCE TASK-SCHEDULING PROBLEM

Computational tasks often consist of multiple parts that may be executed in parallel and independently of each other. Such situations often arise in the context of composed cloud services that consist of several smaller services that can be executed in parallel in a computing center.

We now consider the model where a set of tasks needs to be executed and where each task consists of multiple unit size jobs, i.e. given a task set $\mathcal{T} = \{T_1,\ldots,T_k\}$ each containing a set of jobs $T_i = \{j_{i1},\ldots,j_{in_i}\}$ with $p_{ik} = 1$ for all i,k. The objective is to minimize the average completion time, where the completion time f_i of a task T_i denotes the time the last job of this task is finished, i.e. $f_i := \max\{t : s_j(t-1) > 0 \text{ for some } j \in T_i\}$. This is equivalent to minimizing the sum of completion times, which we consider throughout this section. Note that this model uses a composed objective: for a single task we aim to minimize the latest completion time of the involved jobs, while over all tasks we aim to minimize the average completion time. Similar measures were considered, for example, in [13] (for *order scheduling models*, where a task consists of jobs and each job needs to be processed on a subset of specific machines).

We denote the set of unfinished tasks after time t by $T(t)$, the set of unfinished jobs of task i after time t by $J_i(t)$, and the remaining resource requirement for set U by $\tilde{r}(U)$.

4.1 Prerequisites

Our algorithm for this setting partitions the set of tasks into two sets $\mathcal{T}_1$ and $\mathcal{T}_2$. For each task, we consider the average resource requirement of its jobs. The tasks with jobs that have a high resource requirement belong to $\mathcal{T}_1$, those with jobs that have a low resource requirement belong to $\mathcal{T}_2$. The algorithm schedules both sets of tasks independently in parallel, each on (roughly) half the processors with half the resource.

```
1  for $(t,S,W,i) \leftarrow (1,\emptyset,\emptyset,1); T(t-1) \neq \emptyset; t \leftarrow t+1$:
2      $m' \leftarrow m$
3      while $(\tilde{r}(S)+\tilde{r}(J_i(t-1)) \leq 1)$:
4          $S \leftarrow S \cup T_i; i \leftarrow i+1; m' \leftarrow m' - |J_i(t-1)|$
5          process all jobs in $T_i$ with their full resource requirement
6      $W \leftarrow W \cap J_i(t-1)$
7      $W \leftarrow$ GrowWindowLeft$(W, t, m', 1-\tilde{r}(S))$
8      $W \leftarrow$ GrowWindowRight$(W, t, m', 1-\tilde{r}(S))$
9      $W \leftarrow$ MoveWindowRight$(W, t, 1-\tilde{r}(S))$
10     if $\exists$ fractured job $\iota \in W$: $F \leftarrow \{\iota\}$
11     else: $F \leftarrow \emptyset$
12     process each job $j \in W \setminus (F \cup \{\max W\})$ with resource $r_j$
13     if $F = \{\iota\}$:
14         process job $\iota$ with resource $q_\iota(t)$
15     process job $\max W$ with the remaining resource
```

Listing 3: Algorithm for task set $\mathcal{T}_1$.

We begin with the tasks that have high resource requirements. Here, the available resource is R instead of 1 as in the previous section. Note that the auxiliary procedures called in the algorithms in Listing 3 (Lines 7 to 9) and Listing 4 (Lines 8 to 10) are applied only to the currently considered task instead of the whole set of jobs.

Lemma 4.1. *For a set of tasks* $\mathcal{T} = \{T_1,\ldots,T_k\}$ *with* $\frac{r(T)}{|T|} > \frac{R}{(m-1)}$ *for all* $T \in \mathcal{T}$*, the algorithm in Listing 3 computes a schedule such that the completion time* f_i *of task* T_i *is bounded by* $f_i \leq \left\lceil \frac{\sum_{l=1}^{i} r(T_l)}{R} \right\rceil$.

Proof. The algorithm in Listing 3 processes tasks by increasing index and proceeds according to the algorithm in Listing 1 and Section 3 separately for each task.

Note that for a task $T_i = \{J_{i1},\ldots,J_{in_i}\}$ the average size of the jobs is more than $R/(m-1)$. We inductively prove that $\frac{\tilde{r}(J_i(t))}{|J_i(t) \setminus F|} \geq \frac{R}{m-1}$ remains true for each unfinished task T_i after any time step t. This would imply that after time step t there is a sliding window using the full resource R in that time step (except in the last time step of the schedule) and hence the lemma would follow. We distinguish two cases.

Case 1: First consider the case in which there is no transition between tasks in the current time step $t+1$. As we have $\frac{\tilde{r}(J_i(t))}{|J_i(t) \setminus F|} \geq \frac{R}{m-1}$ and $|F| \leq 1$, the algorithm always finds an m-maximal window using the full resource R in time step $t+1$. By Property (e) from Section 3, we have that (i) the windows has size m or (ii) $L_t(W) = \emptyset$. In case (i), $\tilde{r}(J_i(t))$ is reduced by R and $|J_i(t) \setminus F|$ by at least $m-1$, thus $\frac{\tilde{r}(J_i(t+1))}{R} = \frac{\tilde{r}(J_i(t))-R}{R} \geq \frac{|J_i(t) \setminus F|-(m-1)}{m-1} \geq \frac{|J_i(t+1) \setminus F|}{m-1}$. In case (ii), the jobs from $J_i(t)$ with the smallest resource requirement are finished, thus the ratio $\frac{\tilde{r}(J_i(t))}{|J_i(t) \setminus F|} \geq \frac{R}{m-1}$ can only increase. The claim follows.

Case 2: Now consider the case that there is a transition between tasks. That is, there is an arbitrary number of tasks that is finished in Line 3 of Listing 3. Those tasks used $m-m'$ processors, hence at least $m-m'-1$ processors were occupied with full jobs. By induction hypothesis and by the average size of jobs in task set $\mathcal{T}$, at least a resource of $\frac{m-m'-1}{m-1} \cdot R$ was used. Hence, the resource available to the sliding window determined in Lines 7 to 9 is at most $\frac{m'}{m-1} \cdot R$. By Lemma 3.7, we

```
1   for $(t,S,W,i) \leftarrow (1,\emptyset,\emptyset,1)$; $T(t-1) \neq \emptyset$; $t \leftarrow t+1$:
2       $m' \leftarrow m$
3       while $(\tilde{r}(S)+\tilde{r}(J_i(t-1)) \leq 1)$ and $(|S|+|J_i(t-1)| \leq m)$:
4           $S \leftarrow S \cup T_i$; $i \leftarrow i+1$; $m' \leftarrow m' - |J_i(t-1)|$
5           process all jobs in $T_i$ with their full resource requirement
6       $m' \leftarrow \min\{m', \lfloor (1-\tilde{r}(S)) \cdot \frac{m-1}{R} \rfloor + 1\}$;$R \leftarrow (m'-1) \cdot \frac{R}{m-1}$
7       $W \leftarrow W \cap J_i(t-1)$
8       $W \leftarrow$ GrowWindowLeft($W, t, m', 1-\tilde{r}(S)$)
9       $W \leftarrow$ GrowWindowRight($W, t, m', 1-\tilde{r}(S)$)
10      $W \leftarrow$ MoveWindowRight($W, t, 1-\tilde{r}(S)$)
11      if $\exists$ fractured job $\iota \in W$: $F \leftarrow \{\iota\}$
12      else: $F \leftarrow \emptyset$
13      process each job $j \in W \setminus (F \cup \{\max W\})$ with resource $r_j$
14      if $F = \{\iota\}$:
15          process job $\iota$ with resource $q_\iota(t)$
16      process job max $W$ with the remaining resource
```

Listing 4: Algorithm for task set $\mathcal{T}_2$.

conclude that we computed an m'-maximal window. Now, we either have (a) $|W| = m'$ or (b) $|W| < m'$. In case (a), $\tilde{r}(W)$ was reduced by at most $\frac{m'}{m-1} \cdot R$, whereas $|T_i \setminus F|$ was reduced by exactly $|W| = m'$. Hence $\frac{\tilde{r}(J_i(t+1))}{R} \geq \frac{\tilde{r}(J_i(t)) - m' R/(m-1)}{R} \geq \frac{|J_i(t) \setminus F| - m'}{m-1} = \frac{|J_i(t+1) \setminus F|}{m-1}$ in case (b) with $|W| < m'$, we know $L_t(W) = \emptyset$ by Property (e) from Section 3, implying that the smallest jobs of the new task were executed. The claim follows, as the average size of jobs in each task is at least $R/m-1$.

□

We now consider tasks with jobs that have low resource requirements on average.

Lemma 4.2. *For a set of tasks $\mathcal{T} = \{T_1,\ldots,T_k\}$ with $\frac{r(T)}{|T|} \leq \frac{R}{(m-1)}$ for all $T \in \mathcal{T}$, the algorithm in Listing 4 computes a schedule such that the completion time f_i of task T_i is bounded by $f_i \leq \left\lceil \frac{\sum_{l=1}^{i} |T_i|}{m-1} \right\rceil$.*

Proof. Let $t_i := \frac{\sum_{l=1}^{i} |T_i|}{m-1}$. We show that the following properties hold for every task T_i

(i) T_i is finished at $f_i \leq \lceil t_i \rceil$.
(ii) The number of processors occupied by tasks $T_1 \ldots, T_i$ in time step $\lceil t_i \rceil$ is at most $m_i := (t_i - (\lceil t_i \rceil - 1))(m-1)$.
(iii) Tasks $T_1 \ldots, T_i$ occupy at most a resource of $m_i \cdot \frac{R}{m-1}$ in time step $\lceil t_i \rceil$.

These properties obviously hold for T_1. For the sake of induction, assume they are true for the tasks $T_1,\ldots,T_i$. We distinguish the case two cases whether task T_{i+1} is finished in step f_i or not.

Case 1: If T_{i+1} is finished in time step f_i, it is among those tasks added during the loop in Line 3. Then $f_{i+1} = f_i \leq \lceil t_i \rceil \leq \lceil t_{i+1} \rceil$ and Statement (i) directly follows. For (ii), T_{i+1} uses $|T_{i+1}|$ processors, hence the number of processors used by tasks $T_1, \ldots, T_{i+1}$ in time step f_i is at most $m_i + |T_{i+1}| = (t_{i+1} - (\lceil t_i \rceil - 1))(m-1) = m_{i+1}$. Finally, by $\frac{r(T_{i+1})}{|T_{i+1}|} \leq \frac{R}{(m-1)}$, the resource occupied by tasks $T_1,\ldots,T_{i+1}$ in time step f_i is at most $m_i \cdot \frac{R}{m-1} + r(T_{i+1}) \leq m_i \cdot \frac{R}{m-1} + |T_{i+1}| \cdot \frac{R}{m-1} = m_{i+1} \cdot \frac{R}{m-1}$, which shows (iii).

Case 2: In the case that T_{i+1} is not finished in time step f_i, we will start task T_{i+1} with $m' \geq m - m_i$ processors and allow a resource of at most $(m'-1) \cdot \frac{R}{m-1}$ in this time step (and the full resource in any following non-transitional time step). Since the average resource per full non-fractured job in our sliding window (Lines 8 to 10) is $\frac{R}{m-1}$, we get an analogue statement to Lemma 3.8 from Section 3. That is, we will (a) finish $m'-1$ jobs in f_i and $m-1$ jobs in any time step $t \in (f_i, f_{i+1})$ or (b) use the full resource in any time step $t \in (f_i, f_{i+1})$.

In case (a), we have

$$\begin{aligned} f_{i+1} &\leq f_i + \left\lceil \frac{|T_{i+1}| - (m'-1)}{m-1} \right\rceil \\ &\leq t_i + \frac{m'-1}{m-1} + \left\lceil \frac{|T_{i+1}| - (m'-1)}{m-1} \right\rceil \\ &= \left\lceil \frac{\sum_{k=1}^{i} |T_k| + |T_{i+1}|}{m-1} \right\rceil = \lceil t_{i+1} \rceil, \end{aligned}$$

which yields (i). For (ii), the number of occupied processors in time step $\lceil t_{i+1} \rceil$ is at most $\sum_{k=1}^{i+1} |T_k| - (\lceil t_{i+1} \rceil - 1) \cdot (m-1) = m_{i+1}$. For (iii), observe that the average resource of the window is non-increasing by Lemma 3.8 (b). In particular, the resource used at time $\lceil t_{i+1} \rceil$ is at most $m_{i+1} \cdot \frac{R}{m-1}$.

For case (b), by using $\frac{r(T)}{|T|} \leq \frac{R}{(m-1)}$ in the first inequality, we will finish the task at time

$$\begin{aligned} f_{i+1} &\leq \lceil t_i \rceil + \left\lceil \frac{r(T_{i+1}) - (m'-1) \cdot R/(m-1)}{R} \right\rceil \\ &\leq t_i + \frac{m'-1}{m-1} + \left\lceil \frac{|T_{i+1}| - (m'-1)}{m-1} \right\rceil = \lceil t_{i+1} \rceil, \end{aligned}$$

which yields (i). By using the same reasoning without rounding, the resource used in time step $\lceil t_{i+1} \rceil$ by tasks $T_1,\ldots,T_{i+1}$ can be upper bounded by

$$\begin{aligned} &\left(t_i + \frac{r(T_{i+1}) - (m'-1) \cdot \frac{R}{(m-1)}}{R} - (\lceil t_{i+1} \rceil - 1) \right) \cdot R \\ &\leq (t_{i+1} - (\lceil t_{i+1} \rceil - 1)) \cdot R = m_{i+1} \cdot \frac{R}{m-1}, \end{aligned}$$

which yields (iii). For (ii), it remains to be shown that the number of processors occupied at time $\lceil t_{i+1} \rceil$ by jobs from tasks $T_1 \ldots, T_{i+1}$ is at most m_{i+1}. Since (a) did not hold, less than m processors were occupied at some time step prior to t_{i+1}. This implies that the remaining full jobs must have an average size of more than $\frac{R}{m-1}$. The claim follows.

□

We now give bounds for the optimal algorithm.

Lemma 4.3. *The sum of completion times of the optimal solution can be bounded as follows.*

(a) Given a set of tasks $\mathcal{T} = \{T_1,\ldots,T_k\}$ with $R_l \leq R_{l+1}$ for all l, we have $\text{Opt}_{\mathcal{T}} \geq \sum_{i=1}^{k} \left\lceil \sum_{l=1}^{i} R_l \right\rceil$.
(b) Given a set of tasks $\mathcal{T} = \{T_1,\ldots,T_k\}$ with $|T_l| \leq |T_{l+1}|$ for all l, we have $\text{Opt}_{\mathcal{T}} \geq \sum_{i=1}^{k} \left\lceil \sum_{l=1}^{i} \frac{|T_k|}{m} \right\rceil$.

Proof. We first prove (a). As the optimal solution cannot overuse the resource, there is obviously an order $\mathcal{T} = \{T_{\sigma_1},\ldots,T_{\sigma_l}\}$ such that $\text{Opt}_{\mathcal{T}} \geq \sum_{i=1}^{k} \left\lceil \sum_{l=1}^{i} R_{\sigma_l} \right\rceil$. We denote the bounds on the completion times as $f_i := \left\lceil \sum_{l=1}^{i} R_l \right\rceil$ and $f_i' := \left\lceil \sum_{l=1}^{i} R_{\sigma_l} \right\rceil$. We prove $f_i \leq f_i'$ for all i which directly implies (a). Let i be arbitrary. Assume $f_i > f'_i$. Then $\left\lceil \sum_{l=1}^{i} R_l \right\rceil > \left\lceil \sum_{l=1}^{i} R_{\sigma_l} \right\rceil$, hence $\sum_{l=1}^{i} R_l > \sum_{l=1}^{i} R_{\sigma_l}$. This is a contradiction to $R_l \leq R_{l+1}$ for all l.

For (b), as the optimal solution cannot finish more than m jobs per time step, there is an order $\mathcal{T} = \{T_{\sigma_1}, \dots, T_{\sigma_l}\}$ such that $\text{OPT}_{\mathcal{T}} \geq \left\lceil \sum_{l=1}^{i} \frac{|T_{\sigma_l}|}{m} \right\rceil$. Now, denote $f_i := \left\lceil \sum_{l=1}^{i} \frac{|T_l|}{m} \right\rceil$, $f'_i := \left\lceil \sum_{l=1}^{i} \frac{|T_{\sigma_l}|}{m} \right\rceil$. Assume $f_i > f'_i$ for some i. Hence $\sum_{l=1}^{i} |T_l| > \sum_{l=1}^{i} |T_{\sigma_l}|$. A contradiction. □

To upper bound rounding errors later, the following lemma proves to be useful.

Lemma 4.4. *Given $z \in \mathbb{N}_{\geq 3}$ and $\{x_1, \dots, x_k\} \in \mathbb{R}_{\geq 1/z}$ such that $x_i + 1/z \leq x_{i+1}$ for all $i \in \{1, \dots, k-1\}$, there is a $q \in \mathbb{N}_0$ such that*

$$\sum_{i=1}^{k} \left(\left\lceil \frac{z}{\lfloor (z-1)/2 \rfloor} \cdot x_i \right\rceil - \frac{z}{\lfloor (z-1)/2 \rfloor} \cdot \lceil x_i \rceil \right) \leq q, \text{ and} \tag{2}$$

$$\sum_{i=1}^{k} \lceil x_i \rceil \geq \frac{2}{3} (\sqrt{q} - 2)^3 + (k - q). \tag{3}$$

Proof. First, denote $err_i := \left\lceil \frac{z}{\lfloor (z-1)/2 \rfloor} \cdot x_i \right\rceil - \frac{z}{\lfloor (z-1)/2 \rfloor} \cdot \lceil x_i \rceil$. Also, let $E_{>0} := \{i \in \{1, \dots, k\} : err_i > 0\}$ and $E_{\leq 0}$ analogously. Clearly, we have

$$err_i \leq \left(\frac{z}{\lfloor (z-1)/2 \rfloor} \cdot x_i + 1 \right) - \frac{z}{\lfloor (z-1)/2 \rfloor} \cdot x_i = 1$$

for all $i \in \{1, \dots, k\}$. We choose $q = |E_{>0}|$, so (2) follows. We further show that

$$x_i \in \left[l, \frac{\lfloor (z-1)/2 \rfloor}{z} \cdot \left(\left\lceil \frac{(l+1) \cdot z}{\lfloor (z-1)/2 \rfloor} \right\rceil - 1 \right) \right] \text{ for } l \in \mathbb{N}_0 \tag{4}$$

implies $err_i \leq 0$. Note that (4) implies $\lceil x_i \rceil \leq l+1$. We upper bound

$$\left\lceil \frac{z}{\lfloor (z-1)/2 \rfloor} \cdot x_i \right\rceil \leq \left\lceil \frac{(l+1) \cdot z}{\lfloor (z-1)/2 \rfloor} \right\rceil - 1 \leq \frac{(l+1) \cdot z}{\lfloor (z-1)/2 \rfloor} = \frac{z}{\lfloor (z-1)/2 \rfloor} \cdot \lceil x_i \rceil. \tag{5}$$

Now, each x_i, $i \in E_{>0}$ has to be in an open interval of the form

$$\left(\frac{\lfloor (z-1)/2 \rfloor}{z} \cdot \left(\left\lceil \frac{l \cdot z}{\lfloor (z-1)/2 \rfloor} \right\rceil - 1 \right), l \right) \tag{6}$$

for some $l \in \mathbb{N}$ since otherwise $err_i \leq 0$ by Inequality (5). The length of each such interval can be upper bounded by

$$\begin{aligned}
& l - \frac{\lfloor (z-1)/2 \rfloor}{z} \cdot \left(\left\lceil \frac{l \cdot z}{\lfloor (z-1)/2 \rfloor} \right\rceil - 1 \right) \\
&= \frac{\left\lfloor \frac{z-1}{2} \right\rfloor}{z} \cdot \left(\frac{l\left(z - 2\left(\left\lfloor \frac{z-1}{2} \right\rfloor\right)\right)}{\lfloor (z-1)/2 \rfloor} - \left\lceil \frac{l\left(z - 2\left(\left\lfloor \frac{z-1}{2} \right\rfloor\right)\right)}{\lfloor (z-1)/2 \rfloor} \right\rceil + 1 \right) \\
&\leq \frac{\lfloor (z-1)/2 \rfloor}{z} \cdot \left(\frac{l(z - 2 \cdot (z-2)/2)}{\lfloor (z-1)/2 \rfloor} - \left\lceil \frac{l(z - 2 \cdot (z-1)/2)}{\lfloor (z-1)/2 \rfloor} \right\rceil + 1 \right) \\
&\leq \frac{\lfloor (z-1)/2 \rfloor}{z} \cdot \left(\frac{2l}{\lfloor (z-1)/2 \rfloor} - 1 + 1 \right) = \frac{2l}{z},
\end{aligned}$$

where the first equality is just a transformation, the second inequality bounds the floor and ceiling functions and the last inequality follows from $\frac{l(z-2 \cdot (z-1)/2)}{\lfloor (z-1)/2 \rfloor} = \frac{l}{\lfloor (z-1)/2 \rfloor} > 0$. Hence, at most $2l$ different x_i can be in the lth such interval. Considering the first p such intervals, they can contain at most $\sum_{l=1}^{p} 2l = p(p+1)$ different x_i. This leads to $x_i > p$ for all $i > p(p+1)$. We conclude

$$\begin{aligned}
\sum_{i=1}^{k} \lceil x_i \rceil &= \sum_{i \in E_{>0}} \lceil x_i \rceil + \sum_{i \in E_{\leq 0}} \lceil x_i \rceil \\
&\geq \left(\sum_{p=1}^{\lfloor \sqrt{q} \rfloor - 1} \sum_{i=(p-1)p+1}^{p(p+1)} p \right) + (k - q) \\
&\geq \sum_{p=1}^{\lfloor \sqrt{q} \rfloor - 1} 2p^2 + (k - q) \\
&= \frac{2 \cdot (\lfloor \sqrt{q} \rfloor - 1) \cdot (\lfloor \sqrt{q} \rfloor - 1/2) \cdot \lfloor \sqrt{q} \rfloor}{3} + (k - q) \\
&\geq \frac{2}{3} (\sqrt{q} - 2)^3 + (k - q),
\end{aligned}$$

where the first inequality is by rearranging the sum and omitting some summands as well as $\lceil x_i \rceil \geq 1$ for all i and the last equality is by a well-known formula for summing up squares. □

4.2 Approximation Algorithm

We are now ready to describe our algorithm. The algorithm divides the tasks into task sets

$$\mathcal{T}_1 = \left\{ T \in \mathcal{T} \;\middle|\; \frac{|T|}{\sum_{J_i \in T} r_i} < m - 1 \right\} \text{ and } \mathcal{T}_2 = \left\{ T \in \mathcal{T} \;\middle|\; \frac{|T|}{\sum_{J_i \in T} r_i} \geq m - 1 \right\}.$$

We assign $\lfloor m/2 \rfloor$ processors to task set $\mathcal{T}_1$ and $\lceil m/2 \rceil$ processors to task set $\mathcal{T}_2$. Denote the sum of completion times of the task using our algorithm by S and the optimal sum of completion times by OPT. The partial sum of completion times of tasks from $\mathcal{T}_1$ and $\mathcal{T}_2$ are called $\text{OPT}_{\mathcal{T}_1}$ and $\text{OPT}_{\mathcal{T}_2}$, respectively. Also, let $k_1 = |\mathcal{T}_1|$, $k_2 = |\mathcal{T}_2|$ (implying $k = k_1 + k_2$). We prove the following lemmata.

Lemma 4.5. *Scheduling $\mathcal{T}_1$ using the algorithm in Listing 3 with $\left\lfloor \frac{m}{2} \right\rfloor$ processors and a resource of $R = \frac{\lfloor m/2 \rfloor - 1}{m-1} < \frac{1}{2}$, there is a $q_1 \in \mathbb{N}_0$ such that the sum of completion times is at most $\left(2 + \frac{4}{m-3}\right) \text{OPT}_{\mathcal{T}_1} + q_1$ as well as*

$$\text{OPT}_{\mathcal{T}_1} \geq \frac{2}{3} (\sqrt{q_1} - 2)^3 + (k_1 - q_1). \tag{7}$$

Proof. For all $T \in \mathcal{T}_1$, we have

$$\frac{r(T)}{|T|} > \frac{1}{m-1} = \frac{(\lfloor m/2 \rfloor - 1)/(m-1)}{\lfloor m/2 \rfloor - 1}$$

by construction of $\mathcal{T}_1$. Assume the tasks $\mathcal{T}_1 = \{T_1, \dots, T_{k_1}\}$ are ordered by non-decreasing overall resource requirement (i.e., $r(T_1) \leq r(T_2) \leq \dots \leq r(T_{k_1})$). Applying Lemma 4.1, we know that the full resource of $\frac{\lfloor m/2 \rfloor - 1}{m-1}$ is used in every time step. Hence, the tasks are scheduled such that the sum of their completion times is

$$S_{\mathcal{T}_1} = \sum_{i=1}^{k_1} \left\lceil \frac{\sum_{l=1}^{i} r(T_l)}{(\lfloor m/2 \rfloor - 1)/(m-1)} \right\rceil = \sum_{i=1}^{k_1} \left\lceil \frac{m-1}{(\lfloor m/2 \rfloor - 1)} \sum_{l=1}^{i} r(T_j) \right\rceil.$$

From Lemma 4.3 (a), we have $\text{OPT}_{\mathcal{T}_1} \geq \sum_{i=1}^{k_1} \left\lceil \sum_{l=1}^{i} r(T_j) \right\rceil$. Now, using Lemma 4.4 with $x_i := \sum_{l=1}^{i} r(T_j)$ and $z := m-1$, we conclude that there is a $q_1 \in \mathbb{N}_0$ such that

$$S_{\mathcal{T}_1} \leq \frac{m-1}{(\lfloor m/2 \rfloor - 1)} \text{OPT}_{\mathcal{T}_1} + q_1 \leq \left(2 + \frac{4}{m-3}\right) \text{OPT}_{\mathcal{T}_1} + q_1$$

together with Property (7), which proves the claim. □

Lemma 4.6. *Scheduling $\mathcal{T}_2$ using the algorithm in Listing 4 with $\lceil \frac{m}{2} \rceil$ processors and a resource of $R = \frac{1}{2}$, there is a $q_2 \in \mathbb{N}_0$ such that the sum of completion times is at most $\left(2 + \frac{4}{m-2}\right) \textsc{Opt}_{\mathcal{T}_2} + q_2$ and*

$$\textsc{Opt}_{\mathcal{T}_2} \geq \frac{2}{3}(\sqrt{q_2} - 2)^3 + (k_2 - q_2). \tag{8}$$

Proof. For all $T \in \mathcal{T}_2$, we have

$$\frac{r(T)}{|T|} \leq \frac{1}{m-1} = \frac{1/2}{(m+1)/2 - 1} \leq \frac{1/2}{\lceil m/2 \rceil - 1}$$

by construction of $\mathcal{T}_2$. Assume the tasks $\mathcal{T}_2 = \{T_1, \ldots, T_{k_2}\}$ are ordered by non-decreasing number of jobs (i.e., $|T_1| \leq |T_2| \leq \cdots \leq |T_{k_2}|$). By Lemma 4.2, we have $S_{\mathcal{T}_2} = \sum_{i=1}^{k_2} \left\lceil \frac{|T_i|}{\lceil m/2 \rceil - 1} \right\rceil$, and from Lemma 4.3 (b) we have $\textsc{Opt}_{\mathcal{T}_2} \geq \sum_{i=1}^{k_2} \left\lceil \frac{|T_i|}{m} \right\rceil$. Now, observing $\lceil m/2 \rceil = \lfloor (m+1)/2 \rfloor$ and using Lemma 4.4 with $x_i := \frac{|T_i|}{m}$ and $z := m$, we conclude that there is a $q_2 \in \mathbb{N}_0$ with

$$S_{\mathcal{T}_1} \leq \frac{m}{\lfloor \frac{(m+1)}{2} \rfloor - 1} \cdot \textsc{Opt}_{\mathcal{T}_2} + q_2 \leq \left(2 + \frac{4}{m-2}\right) \textsc{Opt}_{\mathcal{T}_2} + q_2$$

together with Property (8). □

For our final result, we need the following technical lemma.

Lemma 4.7. *Given $q_1, q_2, k_1, k_2 \in \mathbb{N}_0$ and $k \in \mathbb{N}$ such that $q_1 + q_2 \leq k$. Then*

$$\frac{q_1 + q_2}{\frac{2}{3}(\sqrt{q_1} - 2)^3 + \frac{2}{3}(\sqrt{q_2} - 2)^3 + k - (q_1 + q_2)} = \mathrm{O}\left(k^{-1/5}\right)$$

with respect to k.

Proof. If $q_1 + q_2 \leq k^{4/5}$,

$$\frac{q_1 + q_2}{\frac{2}{3}(\sqrt{q_1} - 2)^3 + \frac{2}{3}(\sqrt{q_2} - 2)^3 + k - (q_1 + q_2)}$$
$$\leq \frac{q_1 + q_2}{-2 \cdot \frac{16}{3} + k - (q_1 + q_2)} < \frac{k^{4/5}}{k - k^{4/5} - 11} \leq \frac{1}{k^{1/5} - 12}.$$

On the other hand, if $k^{4/5} < q_1 + q_2 \leq k$, then $q_1 > 1/2 k^{4/5} > 1/4 k^{4/5}$ or $q_2 > 1/4 k^{4/5}$. Hence

$$\frac{q_1 + q_2}{\frac{2}{3}(\sqrt{q_1} - 2)^3 + \frac{2}{3}(\sqrt{q_2} - 2)^3 + k - (q_1 + q_2)}$$
$$\leq \frac{q_1 + q_2}{\frac{2}{3}(\sqrt{q_1} - 2)^3 + \frac{2}{3}(\sqrt{q_2} - 2)^3} \leq \frac{k}{\frac{2}{3}(\frac{1}{2}k^{2/5} - 2)^3}.$$

The claim follows. □

We are now ready to state the main results of this section. Note that we use o(1) with respect to the number of tasks.

Theorem 4.8. *Splitting up $\mathcal{T}$ into task sets $\mathcal{T}_1$ and $\mathcal{T}_2$ and scheduling them separately with the algorithms from Listing 3 and Listing 4 results in a sum of completion times of*

$$\left(\left(2 + \frac{4}{(m-3)}\right) + \mathrm{o}(1)\right) \cdot \textsc{Opt}.$$

Proof. By $S = S_{\mathcal{T}_1} + S_{\mathcal{T}_2}$ and $\textsc{Opt} = \textsc{Opt}_{\mathcal{T}_1} + \textsc{Opt}_{\mathcal{T}_2}$ as well as Lemma 4.5 and Lemma 4.6, there are $q_1, q_2 \in \mathbb{N}_0$ such that

$$S \leq \left(2 + \frac{4}{m-3}\right) \textsc{Opt}_{\mathcal{T}_1} + q_1 + \left(2 + \frac{4}{m-2}\right) \textsc{Opt}_{\mathcal{T}_2} + q_2$$
$$\leq \left(2 + \frac{4}{m-3}\right)(\textsc{Opt}_{\mathcal{T}_1} + \textsc{Opt}_{\mathcal{T}_2}) + q_1 + q_2$$

and

$$\textsc{Opt} \geq \frac{2}{3}(\sqrt{q_1} - 2)^3 + (k_1 - q_1) + \frac{2}{3}(\sqrt{q_2} - 2)^3 + (k_2 - q_2).$$

Dividing S by Opt, using these inequalities, and applying Lemma 4.7 completes the proof. □

If we denote the total number of jobs by $n = \sum_{i=1}^{k} n_i$ and by applying the same arguments as in the proof of Theorem 3.3, we get a bound on the running time.

Corollary 4.9. *The algorithms can be implemented with a running time of* $\mathrm{O}((m+n) \cdot n)$.

REFERENCES

[1] Noga Alon, Yossi Azar, Gerhard J Woeginger, and Tal Yadid. 1998. Approximation schemes for scheduling on parallel machines. *Journal of Scheduling* 1, 1 (1998), 55–66.

[2] Ernst Althaus, André Brinkmann, Peter Kling, Friedhelm Meyer auf der Heide, Lars Nagel, Sören Riechers, Jiří Sgall, and Tim Süß. 2017. Scheduling shared continuous resources on many-cores. *Journal of Scheduling* (2017).

[3] André Brinkmann, Peter Kling, Friedhelm Meyer auf der Heide, Lars Nagel, Sören Riechers, and Tim Süß. 2014. Scheduling Shared Continuous Resources on Many-Cores. In *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '14)*. ACM, 128–137.

[4] Fan Chung, Ronald Graham, Jia Mao, and George Varghese. 2006. Parallelism versus Memory Allocation in Pipelined Router Forwarding Engines. *Theory of Computing Systems* 39, 6 (2006), 829–849.

[5] Leah Epstein, Asaf Levin, and Rob van Stee. 2012. Approximation Schemes for Packing Splittable Items with Cardinality Constraints. In *Algorithmica*. Vol. 62. Springer, 102–129.

[6] Leah Epstein and Rob van Stee. 2007. Approximation schemes for packing splittable items with cardinality constraints. In *Proceedings of the 5th International Workshop on Approximation and Online Algorithms (WAOA '07)*. Springer, 232–245.

[7] Leah Epstein and Rob van Stee. 2011. Improved results for a memory allocation problem. *Theory of Computing Systems* 48, 1 (2011), 79–92.

[8] Michael R Garey and Ronald L. Graham. 1975. Bounds for multiprocessor scheduling with resource constraints. *SIAM J. Comput.* 4, 2 (1975), 187–200.

[9] Ellis Horowitz and Sartaj Sahni. 1976. Exact and approximate algorithms for scheduling nonidentical processors. *Journal of the ACM (JACM)* 23, 2 (1976), 317–327.

[10] Klaus Jansen, Marten Maack, and Malin Rau. 2016. Approximation schemes for machine scheduling with resource (in-) dependent processing times. In *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms*. Society for Industrial and Applied Mathematics, 1526–1542.

[11] Joanna Józefowska and Jan Weglarz. 1998. On a Methodology for Discrete-continuous Scheduling. *European Journal of Operational Research* 107, 2 (1998), 338–353.

[12] Jürgen König, Alexander Mäcker, Friedhelm Meyer auf der Heide, and Sören Riechers. 2016. Scheduling with Interjob Communication on Parallel Processors. In *Proceedings of the 10th Annual International Conference on Combinatorial Optimization and Applications (COCOA '16)*. Springer, 563–577.

[13] Joseph YT Leung, Haibing Li, and Michael Pinedo. 2005. Order Scheduling Models: An Overview. In *Multidisciplinary Scheduling: Theory and Applications*. Springer, 37–53.

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

[15] Martin Niemeier and Andreas Wiese. 2015. Scheduling with an Orthogonal Resource Constraint. *Algorithmica* 71, 4 (2015), 837–858.

[16] Jan Weglarz, Joanna Józefowska, Marek Mika, and Grzegorz Waligóra. 2011. Project Scheduling with Finite or Infinite Number of Activity Processing Modes – A Survey. *European Journal of Operational Research* 208, 3 (2011), 177–205.