Scheduling shared continuous resources on many-cores

Journal of Scheduling manuscript No.
(will be inserted by the editor)

Scheduling Shared Continuous Resources on Many-Cores

Ernst Althaus · André Brinkmann · Peter Kling · Friedhelm Meyer auf der Heide · Lars Nagel · Sören Riechers · Jiří Sgall · Tim Süß

Received: date / Accepted: date

Abstract We consider the problem of scheduling a number of jobs on m identical processors sharing a continuously divisible resource. Each job j comes with a resource requirement $r_j \in [0, 1]$. The job can be processed at full speed if granted its full resource requirement. If receiving only an x-portion of r_j, it is processed at an x-fraction of the full speed. Our goal is to find a resource assignment that minimizes the makespan (i.e., the latest completion time). Variants of such problems, relating the resource assignment of jobs to their *processing speeds*, have been studied under the term *discrete-continuous scheduling*. Known results are either very pessimistic or heuristic in nature.

In this article, we suggest and analyze a slightly simplified model. It focuses on the assignment of shared continuous *resources* to the processors. The *job* assignment to processors and the ordering of the jobs have already been fixed. It is shown that, even for unit size jobs, finding an optimal solution is NP-hard if the number of processors is part of the input. Positive results for unit size jobs

Supported by the Federal Ministry of Education and Research (BMBF) under Grant 01IH13004 (Project "FAST"), by the German Research Foundation (DFG) within the Collaborative Research Center "On-The-Fly Computing" (SFB 901), by the Center of Excellence – ITI (project P202/12/G061 of GA ČR), and by the Pacific Institute of Mathematical Sciences (PIMS).

Ernst Althaus (ernst.althaus@uni-mainz.de)
Computer Science Institute, Johannes Gutenberg-Universität Mainz, Germany.

André Brinkmann (brinkman@uni-mainz.de) · Lars Nagel (nagell@uni-mainz.de) · Tim Süß (suesst@uni-mainz.de)
Zentrum für Datenverarbeitung, Johannes Gutenberg-Universität Mainz, Germany.

Peter Kling (pkling@sfu.ca)
School of Computing Science, Simon Fraser University, Canada.

Friedhelm Meyer auf der Heide (fmadh@upb.de) · Sören Riechers (soeren.riechers@upb.de)
Heinz Nixdorf Institute & Computer Science Dept., Paderborn University, Germany.

Jiří Sgall (sgall@iuuk.mff.cuni.cz)
Computer Science Institute, Charles University, Prague, Czech Republic.

include a polynomial-time algorithm for any constant number of processors. Since the running time is infeasible for practical purposes, we also provide more efficient algorithm variants: an optimal algorithm for 2 processors and a $(2 - 1/m)$-approximation algorithm for m processors.

Keywords scheduling · approximation algorithms · resources

1 Introduction

The processor scheduling problem considered in this article is motivated by the observation that, in many cases, it is not a device's speed or energy consumption that limits the progress of a given computation but the fact that data cannot be provided at the necessary rate. In extreme cases, this may lead to situations where changing the available I/O rate (or bandwidth) by some factor x may directly affect the running time by (approximately) the same factor [19].

At first glance, this seems more a network issue than a problem of interest for processor scheduling. After all, bandwidth bottlenecks are typically imposed by the interconnection of devices (e.g., networks or data buses), and there is a huge body of literature concerned with such issues on the network layer. However, the analysis in this area typically concentrates on the *network's* performance. In contrast, our model focuses on how the distribution of the bandwidth shared by a fixed set of processing units can affect their *computational* performance. That is, given some information about the bandwidth requirement of a program (e.g., when does it need how much bandwidth to progress at full speed), the scheduler can speed up critical jobs by a suitable assignment of the available bandwidth to the different processors. Typical examples for such settings are many-core systems: They provide an immense computing power through the sheer number of processor cores. Yet, many (if not all) of the chip's cores share a single data bus to the outside world. If such a system has to process I/O-intensive tasks (as typical for scientific computing), the available bandwidth becomes the computational bottleneck, and the bandwidth distribution becomes the decisive scheduling factor. One can find similar effects, if at different scales, on many other levels. For example, consider virtual systems, where different virtual machines share a single, arbitrarily and dynamically divisible resource (like CPU or Memory) of a given host system.

A First Glimpse at the Model. From a more abstract point of view, the aforementioned bandwidth scheduling can be seen as a variant of resource constrained scheduling, the bandwidth being an example for the resource. Imagine a system consisting of several identical processors that run at a fixed speed and share a given resource. Assume that the resource is the system's performance bottleneck, in the sense that the running time of programs (tasks) depends directly (that is to say, linearly) on the share of the resource they are allowed to use. Each task provides information about its resource requirements by stating what share of the resource it needs at different phases of its processing to run at full speed. Thus, we can imagine a task i to consist of a number n_i of jobs

that must be processed sequentially, one after another. Each job represents a phase of the task's processing where the resource requirement is constant. The length of the phase (i.e., the job's processing time) is minimal at full speed and increases by a factor of $1/x$ if only a portion $x \in [0, 1]$ of the requested resource share is provided. We use the term CRSharing to refer to this problem of sharing continuous resources; see Section 3 for a more formal description.

We will see, especially in Section 2, that this type of resource assignment problem is comparatively complex. Most work considering similar problems seems to be of heuristic nature and analytical results are scarce (and, if surfaced, quite negative). Since we are interested in more analytical insights, we approach the problem by concentrating on the assignment of resources, removing the (classical) scheduling aspect almost completely. That is to say, we consider a scenario in which each processor has exactly one task, and each task consists of jobs of unit workload (but different resource requirements). Moreover, we assume discrete time steps, such that the scheduler can change the resource assignment only at the beginning of such a time step. As we will see, even this simple setting proves to be challenging.

Outline. Section 2 surveys the related work and describes our contribution in view of known results. A formal model description of the CRSharing problem is provided in Section 3. Section 4 equips the reader with basic definitions and results and discusses a first, simple result for a round robin algorithm. Our main results are given in Sections 5 to 8, where we study the complexity and achievable approximation ratio for the CRSharing problem. We conclude with a short outlook in Section 9.

2 Related Work & Contribution

The proposed CRSharing problem is a classical resource constrained scheduling problem. In such settings, the scheduler does not only manage the computational resources (e.g., the assignment of jobs to processors) but also the allocation of one or more additional resources to the currently processed jobs. In our context (processor scheduling), the most obvious examples for such resources are probably bandwidth and memory. However, note that models similar to ours are also used in project planning or for manufacturing systems.

The following discussion focuses on results for *cardinality constrained bin packing of splittable items* as well as so-called *discrete-continuous* models, where the computational resource is discrete (e.g., several processors) and the additional resources are continuous (e.g., bandwidth allotted in a continuous manner to the available processors). For a more general overview of resource constrained scheduling, the interested reader is referred to [13, Chs. 23-24] and [1, Ch. 12].

Packing splittable items with cardinality constraints. This problem setting was originally introduced by Chung et al. [3]. Here, we are given bins of size 1

and n items with size $s_1, s_2 \ldots, s_n$ (with $s_i > 0$ for all i). Items can be split arbitrarily (*splittable items*), but each bin is only allowed to hold at most $k \in \mathbb{N}$ many item parts (*cardinality constraints*). The objective is to minimize the number of bins used. This problem shows some resemblance to CRSharing by understanding the number of processors as cardinality constraints and the bins with a limited capacity as time steps. However, note that, unlike CRSharing, this problems allows for arbitrary preemption and migration, and jobs can be assigned to processors arbitrarily. This provides a much higher degree of freedom, making it easier for algorithms to avoid "leftovers" of the resource.

Chung et al. [3] show that this problem is NP-hard for any $k \geq 2$. For $k = 2$, they show an approximation factor of $\frac{3}{2} + \mathrm{o}(1)$ for a simple NextFit algorithm as well as a more general class of algorithms. Epstein and van Stee [5] later proved that the performance of the NextFit algorithm for arbitrary $k \geq 2$ yields an absolute approximation factor of $2 - \frac{1}{k}$. They also show that the general case for $k \geq 3$ is NP-hard and give an efficient $\frac{7}{5}$-approximation algorithm for $k = 2$. Finally, Epstein and van Stee [4] present polynomial-time approximation schemes for sublinear k.

Discrete-continuous Scheduling. The notion of *discrete-continuous scheduling* traces back to several papers by Józefowska and Weglarz, first and foremost [11]. While most results in this area study scenarios where the amount of allocated resources influences the processing time or release dates of jobs (see [6] for a survey), Józefowska and Weglarz [11] consider the case where the amount of allocated resources influences the processing *speed* of jobs. More precisely, if the function $R_j \colon \mathbb{R}_{\geq 0} \to [0, 1]$ models the share of the resource that job j gets assigned at some time $t \in \mathbb{R}_{\geq 0}$, its workload is processed at a speed of $f_j\big(R_j(t)\big)$. Here, f_j models how a job's processing speed is affected by the received resource amount and is assumed to be continuous and non-decreasing with $f_j(0) = 0$. Using this resource model, the authors consider the problem of scheduling n non-preemptable and independent jobs on m processors. They propose an analysis framework based on a mathematical programming formulation and demonstrate it for the objective of minimizing the schedule's makespan. For certain classes of f_j, this yields a simple analytical solution [11,7]. This holds especially for convex functions f_j, which encourage the scheduler to assign the full resource to a single processor. Finding an optimal solution for more realistic cases (especially concave f_j) remains infeasible. The results in [11] initiated several research efforts in this area, including a transfer of the methodology to other scheduling variants (e.g., average flow time instead of makespan [10]) as well as several heuristic approaches to obtain practical solutions in the general case [8,9,12,16]. A detailed and current survey about these results can be found in [17] (especially Section 7).

Our CRSharing problem shares several characteristics with discrete-continuous scheduling problems. In particular, the jobs' resource requirements can be modeled via concave functions f_j of the form $f_j(R) = \min(R/r_j, 1)$, where the value r_j denotes the resource requirement of job j (cf. Section 3). That is, the speed used to process a job depends linearly on the share of

the resource it receives, but is capped at one. Our model contains several other important differences, the most obvious being that the assignment of jobs to processors as well as the order of jobs on a given processor is fixed. This severely limits the possibilities of the scheduler, which can no longer try to distribute the jobs evenly among the available processors. Instead, it is compelled to use a sophisticated resource assignment in order to yield a schedule of low makespan. Still, this simplification allows us to focus on the inherent problem complexity of assigning the continuous resource such that the schedule's makespan is minimized, and to derive provably good algorithms. In contrast, most of the aforementioned results for the discrete-continuous setting are of heuristic nature and do not provide any provable quality guarantees with respect to the resulting schedules, and cases that can be analyzed analytically turn out to feature quite simple solution structures [11, 7].

Contribution. We introduce a new resource-constrained scheduling model for multiple processors, where job processing speeds depend on the assigned share of a common resource. Our focus lies on a variant with unit size jobs where the scheduler only has to manage the distribution of the resource among all processors. The objective is to minimize the total makespan (maximum completion time over all jobs). Even this simple variant turns out to be NP-hard in the number m of processors. For fixed m, we show that the problem is solvable in polynomial time. Since the respective algorithm is not practical, we also provide an exact quadratic-time algorithm for $m = 2$ and an approximation algorithm for any fixed m. The latter achieves a worst-case approximation ratio of exactly $2 - 1/m$. Our approach uses a hypergraph representation that allows us to capture non-trivial structural properties. To the best of our knowledge, this is the first strong analytical result for this type of problem.

3 Model & Notation

We start by defining the model for the general version of the CRSharing problem, which considers jobs of arbitrary sizes. Afterward, we discuss an alternative interpretation of our model that will ease our argumentation in the analysis part. Note that, while the model description considers jobs of arbitrary sizes, most of our analysis focuses on the case where all jobs are of unit size.

3.1 Formal Model Description

Consider a system of m identical fixed-speed processors sharing a common resource. At every time step $t \in \mathbb{N}$, the scheduler distributes the resource among the m processors. To this end, each processor i is assigned a share $R_i(t) \in [0, 1]$ of the resource, which it is allowed to use in time step t. It is the responsibility of the scheduler to ensure that the resource is not overused. That is, it must guarantee that $\sum_{i=1}^{m} R_i(t) \leq 1$ holds for all $t \in \mathbb{N}$. For each

processor i, there is a sequence of $n_i \in \mathbb{N}$ jobs that must be processed by the processor in the given order. We write (i,j) to refer to the j-th job on processor i. A processor is not allowed to process more than one job during any given time step. Each job (i,j) has a *processing volume* (size) $p_{ij} \in \mathbb{R}_{>0}$ and a *resource requirement* $r_{ij} \in [0,1]$. The resource requirement specifies what portion of the resource is needed to process one unit of the job's processing volume in one time step. In general, when a job is granted an x-portion of its resource requirement ($x \in [0,1]$), exactly x units of its processing volume are processed in that time step. There is no benefit in granting a job more than its requested share of the resource. That is, a job's processing cannot be sped up by granting it, for example, twice its resource requirement. A feasible schedule for an instance of the CRSharing problem consists of m resource assignment functions $R_i \colon \mathbb{N} \to [0,1]$ that specify the resource's distribution among the processors for all time steps without overusing the resource. At any time t, each processor i uses its assigned resource share $R_i(t)$ to process the job (i,j) with minimal j among all unfinished jobs. We measure a schedule's quality by its makespan (i.e., the time needed to finish all jobs). Our goal is to find a feasible schedule having minimal makespan. To simplify notation, we often identify a schedule S with its makespan (e.g., writing ${}^{S}/_{\mathrm{OPT}}$ for the makespan of schedule S divided by the makespan of an optimal schedule OPT).

Alternative Model Interpretation. An alternative interpretation of our scheduling problem can be obtained by the following observation: Consider a job (i,j) whose processing is started at time step t_1. It receives a share $R_i(t_1) \in [0,1]$ of the resource. By the previous model definition, exactly $\min({}^{R_i(t_1)}/_{r_{ij}}, 1)$ units of its processing volume are processed. Similarly, in the next time step $\min({}^{R_i(t_1+1)}/_{r_{ij}}, 1)$ units of its processing volume are processed. Consequently, the job is finished at the minimal time step $t_2 \geq t_1$ such that

$$\sum_{t=t_1}^{t_2} \min({}^{R_i(t)}/_{r_{ij}}, 1) \geq p_{ij} \tag{1}$$

or, equivalently if $r_{ij} > 0$, at the minimal time step $t_2 \geq t_1$ with

$$\sum_{t=t_1}^{t_2} \min(R_i(t), r_{ij}) \geq r_{ij} p_{ij} =: \tilde{p}_{ij}. \tag{2}$$

This observation allows us to get rid of the resource aspect by considering *variable speed* processors instead of fixed speed processors. The speed of such variable speed processors can be changed at runtime[1]. For our reinterpretation, think of a job (i,j) to have size $\tilde{p}_{ij}$ and of a processor i to be of variable speed. The value $R_i(t)$ denotes the speed processor i is set to during time step t. The scheduler is in control of these processor speeds, but it must ensure that the aggregated speed of all processors does never exceed one. Moreover, in addition

[1] This is also known as *speed scaling* (cf. [18]).

to the system's speed limit, each job (i, j) is annotated with the maximum speed r_{ij} it can utilize. In this light, our CRSharing problem becomes a speed scaling problem to minimize the makespan in which the scheduler is limited by both the system's maximum aggregated speed and a per-job speed limit. The unit size restriction for the CRSharing problem translates into the restriction that job sizes $\tilde{p}_{ij}$ equal the corresponding resource requirements r_{ij}. In other words, all jobs must be processable in one time step if run at maximum speed.

During the analysis, it will sometimes be more convenient to think of our problem in the way described above. For example, note that the total size (in the alternative model description) of all jobs in the system is $\sum_{i=1}^{m} \sum_{j=1}^{n_i} \tilde{p}_{ij}$. This load is processed at a maximal aggregated speed of 1. Thus, all processors together cannot process more than one unit of this total load per time step. This yields the following simple but useful observation:

Observation 1 *Any feasible schedule needs at least* $\sum_{i=1}^{m} \sum_{j=1}^{n_i} r_{ij} p_{ij}$ *time steps to finish a given set of jobs with resource requirements* r_{ij} *and sizes* p_{ij}.

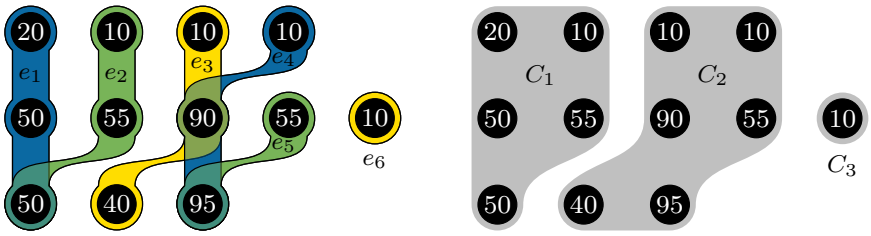
At times, we will use the notion *remaining resource requirement* to denote the remnants of a job's initial workload $\tilde{p}_{ij}$.

Additional Notation & Notions. The following additional notions and notation will turn out to be helpful in the analysis and discussion. For a processor i with n_i jobs, we define $n_i(t)$ as the number of unfinished jobs at the start of time step t. In particular, we have $n_i(1) = n_i$. The value $j_i(t) \coloneqq n_i - n_i(t)$ denotes the number of jobs completed on machine i at the start of step t. A processor i is said to be *active* at time step t if $n_i(t) > 0$. Similarly, we say that job (i, j) is *active* at time step t if $j_i(t) = n_i - n_i(t) = j - 1$ (i.e., if processor i has finished exactly $j - 1$ jobs at the start of time step t). We use $M_j \coloneqq \{\, i \mid n_i \geq j \,\}$ to denote the set of all processors having at least j jobs to process. Finally, we define $n \coloneqq \max_i n_i$ as the maximum number of jobs any processor has to process.

3.2 Graphical Representation

The remainder of this section introduces a hypergraph notation for CRSharing schedules and unit size jobs.

Given a problem instance of CRSharing with unit size jobs and a corresponding schedule S, we can define a weighted hypergraph $H_S = (V, E)$ as follows: The nodes of H_S and their weights correspond to the jobs and their resource requirements, respectively. That is, the node set is given by $V = \{\, (i, j) \mid i = 1, 2, \ldots, m \wedge j = 1, 2, \ldots, n_i \,\}$, and the weight of a node $(i, j) \in V$ is r_{ij}. The edges of H_S correspond to the schedule's time steps and contain the currently active jobs. More formally, the edge $e_t \subseteq V$ for time step t is defined as $e_t \coloneqq \{\, (i, j) \mid n_i(t) > 0 \wedge j = n_i - n_i(t) + 1 \,\}$. Thus, if we abuse S to also denote the makespan of schedule S, the edge set of H_S can be written as $E = \{\, e_1, e_2, \ldots, e_S \,\}$. We call H_S the *scheduling (hyper)graph* of S. See Figure 1a for an illustration.



(a) Scheduling graph H_S trying to greedily finish as many jobs as possible.

(b) Connected components of the scheduling graph, ordered from left to right.

Fig. 1: Hypergraph representation of a schedule for three processors. Resource requirements are given as node labels (in percent). Nodes are laid out such that each row corresponds to the job sequence of one processor (from left to right). Edges correspond to the schedule that prioritizes jobs in order of increasing remaining resource requirement.

Connected Components. In Section 4.1 and during the analysis in Section 8, we will see that the connected components formed by the edges of a scheduling graph H_S carry a lot of structural information about the schedule. To make use of this information, let us introduce some notation that allows us to directly argue via such components. We start with an observation that follows from the construction of H_S.

Observation 2 *Consider a connected component $C \subseteq V$ of H_S and two time steps $t_1 \leq t_2$ with $e_{t_1} \cup e_{t_2} \subseteq C$. Then, for all $t \in \{ t_1, t_1 + 1, \ldots, t_2 \}$ we have $e_t \subseteq C$.*

Let N denote the total number of connected components and let C_k denote the k-th connected component (for $k \in \{ 1, 2, \ldots, N \}$). Moreover, we use $\#_k$ to denote the number of edges of the k-th component. That is, we have $\#_k = |\{ e_t \in E \mid e_t \subseteq C_k \}|$. Observation 2 implies that a component C_k consists of $\#_k$ consecutive time steps. This allows us to order the components such that, for any two components k, k' and edges $e_t \subseteq C_k, e_{t'} \subseteq C_{k'}$ with $t \leq t'$, we have $k \leq k'$. That is, we can think of the components being processed by the processors from left to right. See Figure 1b for an illustration.

The maximal size of an edge in the k-th component, which equals the size of its first edge, gives us a rough estimate for the amount of potential parallelism available during the corresponding time steps. Note that while the size of edges e_t is monotonously decreasing in t, a schedule that tries to balance the number of remaining jobs on each processor will decrease the edge size only at the end of a component (for all components but the last one). We will make use of this fact in the proof of Lemma 6. For now, let us honor its foreshadowed importance by the following definition:

Definition 1 (Component Class) Given a component C_k, we define its *class* q_k as the size of its first edge, i.e., $q_k \coloneqq |e_t|$ with $t = \min \{ t' \mid e_{t'} \subseteq C_k \}$.

Besides being an upper bound on the size of a component's edges, the class q_k is also decreasing in k. Moreover, Lemma 2 will show that a component's class

allows us to formulate an important relation between its size and the total number of its edges.

4 Preliminaries

This section is intended to make the reader more comfortable with the introduced terms and notions and to equip her with the tools needed for the analysis in later sections. We start by discussing and proving some basic structural properties. Afterward, we analyze a simple round robin algorithm. Note that in this and all following sections, we only consider problem instances in which all jobs are of unit size.

4.1 Structural Properties

Let us use the introduced notions and notation to point out some structural properties of schedules for the CRSharing problem with unit size jobs. We start by defining three properties of schedules and show in Lemma 1 that we can restrict our analysis to schedules which have them.

Definition 2 (Non-wasting) We call a schedule *non-wasting* if it finishes all active jobs during every time step t with $\sum_{i=1}^{m} R_i(t) < 1$.

Definition 3 (Progressive) A schedule is *progressive* if, among all jobs that are assigned resources, at most one job is only partially processed during any time step t. More formally, we require that

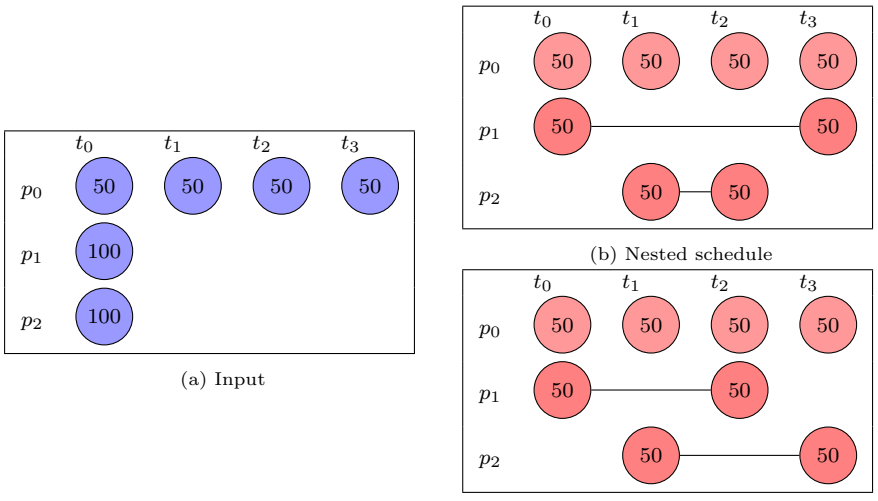
$$|\{\, i \mid n_i(t) = n_i(t+1) \wedge R_i(t) > 0 \,\}| \leq 1 \tag{3}$$

holds for all $t \in \mathbb{N}$.

Definition 4 (Nested) Let $S(i,j)$ and $C(i,j)$ denote the starting step and the completion step of job (i,j), respectively. A schedule is *nested* if, at no time t, there are two jobs (i,j) and (i',j') such that $S(i,j) < S(i',j') \leq t < C(i',j')$, $S(i',j') < C(i,j)$ and (i,j) is running during step t.

This last property intuitively means that among the partially processed jobs, we always prefer to run and complete the job that started at the latest step. Note that the condition of a nested schedule in particular implies that, for no jobs (i,j) and (i',j'), $S(i,j) < S(i',j') < C(i,j) < C(i',j')$. Otherwise we could choose $t = C(i,j)$ and job (i,j) would run in step $t = C(i,j)$. An example for a nested and an unnested schedule is given in Figure 2.

Lemma 1 *Every schedule S can be transformed into a schedule S' which is non-wasting, progressive and nested without increasing its makespan.*



(a) Input

(b) Nested schedule

(c) Unnested schedule

Fig. 2: The schedules in Figure 2b and 2c are based on the input in Figure 2a and observe a resource limit of 100. Both schedules are non-wasting and progressive, but only the schedule in Figure 2b is nested. In the other schedule, p_1's job is already running when p_2's job is started, and completed before p_2's job is completed.

Proof Making a given schedule non-wasting is trivial because, given a time step t with $\sum_{i=1}^{m} R_i(t) < 1$ and an active job (i', j'), we can increase $R_{i'}(t)$ until either the job is finished or $\sum_{i=1}^{m} R_i(t) = 1$ (and decrease the resource consumption of this job by the same amount in later steps). In both cases, the schedule's makespan does not increase. By doing this for each step t in ascending order, we will get a non-wasting schedule.

In the following we assume that we start with a non-wasting schedule. For each of the following modifications, it is easy to check that the schedule remains non-wasting.

First we guarantee by an exchange argument that for no two jobs (i, j) and (i', j') it holds that $S(i, j) < S(i', j') < C(i, j) < C(i', j')$. Suppose we have a pair of jobs (i, j) and (i', j') violating the condition. Consider all the resource the two jobs are using in steps $S(i', j'), \ldots, C(i, j)$ and redistribute it in each of these steps so that (i, j) is completed before or when (i', j') is started. This is done by first giving all resource assigned to (i', j') to (i, j) until (i, j) is finished and then giving all resource assigned to (i, j) to (i', j'). It follows that $C(i, j) \leq S(i', j')$ and that the condition is not longer violated for this pair of jobs. Furthermore, $C(i, j)$ is not increased, $S(i', j')$ is not decreased and all other start and completion times remain unchanged, so that no new violating pair is created. In this way we can eliminate the violating pairs one by one.

Now we modify the schedule for each time step $t = 1, 2, \ldots$ so that for this t the resulting schedule is nested and progressive. More precisely, we alter it in such a way that there is at most one job running in step t and active after step t; furthermore such a job has the smallest completion time among the jobs active after step t. This guarantees both properties.

Let (i, j) and (i'', j'') be two jobs that are running in step t and active after step t. Further let (i, j) have the smallest completion time among these jobs. Then at step t, give the maximal amount of resource assigned to job (i'', j'') to job (i, j), and balance this exchange by giving the same amount of resource from (i, j) to (i'', j'') at later time steps. Note that this exchange does not change $C(i'', j'')$. As a result of the exchange, either $C(i, j) = t$ or (i'', j'') does not run at time t. In both cases we have decreased the number of jobs that are partially processed at time t.

Decreasing $C(i, j)$ may create a new pair with $S(i, j) < S(i', j') < C(i, j) < C(i', j')$, however only for $S(i', j') > t$. We treat any such pair as in the previous paragraph, which changes the schedule only after time t. Now we repeat the process for the next pair of (i, j) and (i'', j'') as needed. □

Lemma 1 allows us to narrow our study to the subclass of non-wasting, progressive and nested schedules, and from now on we will assume any schedule to have these properties (if not stated otherwise).

Balanced Schedules. Intuitively, good schedules should try to balance the number of remaining jobs on each processor. This may provide the scheduler with more choices to prevent the underutilization of the resource later on (e.g., when only one processor with many jobs of low resource requirements remains). The better part of Section 8 serves the purpose of confirming this intuition. In the following, we formalize this balance property and, subsequently, work out further formal and concise properties of balanced schedules.

Definition 5 (Balanced) We say a schedule is *balanced* if, whenever a processor i finishes a job at a time step t, any processor i' with $n_{i'}(t) > n_i(t)$ does also finish a job.

Proposition 1 *Every balanced schedule features the following properties:*

(a) For all i_1, i_2 with $n_{i_1} \geq n_{i_2}$ and for all $t \in \mathbb{N}$, we have $n_{i_1}(t) \geq n_{i_2}(t) - 1$.
(b) For all i_1, i_2 with $n_{i_1} > n_{i_2}$ and for all $t \in \mathbb{N}$, we have $n_{i_1}(t) \leq n_{i_2}(t) + n_{i_1} - n_{i_2}$.

Proof Both statements follow easily from the definition of balanced schedules. To see this, first note that both properties hold for $t = 1$, since $n_i(1) = n_i$ for all processors i. Moreover, at any time step t, the number $n_i(t)$ of remaining jobs cannot increase, and decreases by at most one during the current time step. Thus, it is sufficient to show that if one of the statements holds at some time step t with equality, it still holds at time step $t + 1$. For statement (a), $n_{i_1}(t) = n_{i_2}(t) - 1$ and the balance property imply that if i_1 finishes its job, then so must i_2. Thus, we have $n_{i_1}(t + 1) \geq n_{i_2}(t + 1) - 1$. The very same argument works for statement (b). □

Proposition 2 *Consider a balanced schedule and the set M_j of processors having at least j jobs. Let (i,j) be a job that is active at time step t and assume $n_i(t) > 1$ (i.e., it is not the last job on processor i). Then all processors $i' \in M_j$ are active at time step t.*

Proof Let $i' \in M_j$ be a processor with at least j jobs and consider the case $n_{i'} \geq n_i$. By Proposition 1(a), we have $n_{i'}(t) \geq n_i(t) - 1 > 0$, so processor i is active at time t. If $n_{i'} < n_i$, we can apply Proposition 1(b) and get

$$n_{i'}(t) \geq n_{i'} - (n_i - n_i(t)) = n_{i'} - (j-1) \geq 1. \tag{4}$$

The equality uses the fact that job (i,j) is active at time step t, implying that the number $n_i - n_i(t)$ of jobs finished by processor i before time step t is exactly $j-1$. The last inequality comes from $i' \in M_j$. □

The final structural property of balanced schedules addresses, as indicated earlier, how a component's class allows us to relate its size (number of nodes) to the total number of its edges.

Lemma 2 *Consider a non-wasting, progressive, and balanced schedule. The number of nodes and edges in a component are related via the following properties:*

(a) The inequality $|C_k| \geq \#_k + q_k - 1$ holds for all $k \in \{1, 2, \ldots, N-1\}$.
(b) The last component satisfies $|C_N| \geq \#_N$.

Proof The second statement follows immediately from Lemma 1, which states that in each time step (i.e., for each edge) at least one job is finished.

For the first statement, fix a $k \in \{1, 2, \ldots, N-1\}$ and consider the first edge e_t of the component C_k. By definition, this edge consists of q_k different nodes. We now show that each of the remaining $\#_k - 1$ edges adds at least one new node to the component. So fix an edge $e_{t'} \subseteq C_k$ with $t' > t$ and consider the time step $t'-1$. Since we know that at least one job is finished in every time step (Lemma 1) and that S is balanced, at least one of the processors having the maximal number of remaining jobs finishes its current job. More formally, there is some processor $i' = \arg\max_i n_i(t'-1)$ that finishes its currently active job at time step $t'-1$. Because of $k \neq N$, we also know that $n_{i'}(t'-1) > 1$, such that there is a new active job for processor i' at time step t'. This yields the lemma's first statement. □

4.2 Warm-up: Round Robin Approximation

Consider the following simple round robin algorithm for the CRSharing problem (with unit size jobs): Given a problem instance where the maximal number of jobs on a processor is n, the algorithm operates in n phases. During phase j, it processes the j-th job on each processor, assigning the resource in an arbitrary way to any processors that have not yet finished their j-th

job. Note that this algorithm may waste resources (although only between two phases) and is possibly non-progressive. Still, the following theorem shows that it results in schedules that are not too bad.

Theorem 3 *The* RoundRobin *algorithm for the* CRSharing *problem with unit job sizes has a worst-case approximation ratio of exactly* 2*.*

Proof We start with the upper bound on the approximation ratio. The algorithm RoundRobin needs exactly $\lceil \sum_{i \in M_j} r_{ij} \rceil$ time steps to finish the j-th phase (cf. "Alternative Model Interpretation" in Section 3). Thus, the makespan of a RoundRobin schedule can be bounded by

$$\sum_{j=1}^{n} \left\lceil \sum_{i \in M_j} r_{ij} \right\rceil \leq n + \sum_{j=1}^{n} \sum_{i \in M_j} r_{ij}. \tag{5}$$

Since any processor can finish at most one job per time step, even an optimal schedule has a makespan of at least n. Observation 1 yields another lower bound on the optimal makespan, namely $\sum_{j=1}^{n} \sum_{i \in M_j} r_{ij}$. Together, we get that RoundRobin computes a 2-approximation.

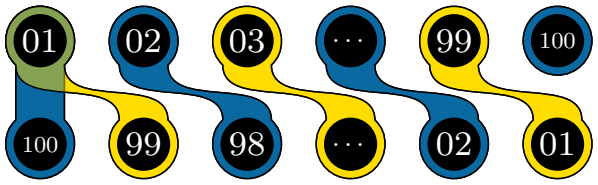
For the lower bound on the approximation ratio, consider the following CRSharing problem instance with unit size jobs on two processors: Let $n \in \mathbb{N}, \varepsilon \coloneqq 1/n > 0$ and define the resource requirements for the first processor as $r_{1j} \coloneqq j \cdot \varepsilon$ for $j \in \{1, 2, \ldots, n\}$. For the second processor, we define $r_{2j} \coloneqq (1 + \varepsilon) - r_{1j}$. Note that each processor has to process n jobs. Figure 3 illustrates the instance as well as the resulting optimal and RoundRobin schedules for $n = 100$. An optimal schedule, shown in Figure 3a, will waste no resource at all. In contrast, the RoundRobin schedule, as indicated in Figure 3b, wastes a share of $1 - \varepsilon$ of the resource in every second time step. As a result, the RoundRobin schedule needs $2n$ time steps, while an optimal schedule can finish the same workload in $n + 1$ time steps. Thus, for $n \to \infty$ we get an approximation ratio of 2. □

5 Problem Complexity

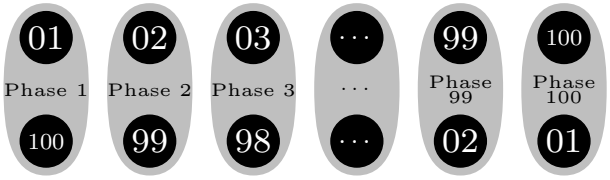
One of our first major results is the following theorem, showing that the CRSharing problem is (even in the case of unit size jobs) NP-hard in the number of processors.

Theorem 4 CRSharing *with unit size jobs is NP-hard if the number of processors is part of the input.*

Proof In the following, we prove the NP-hardness of the CRSharing problem with unit size jobs via a reduction from the Partition problem. Our reduction



(a) OPT schedule, wastes no resources and needs $n+1$ time steps.



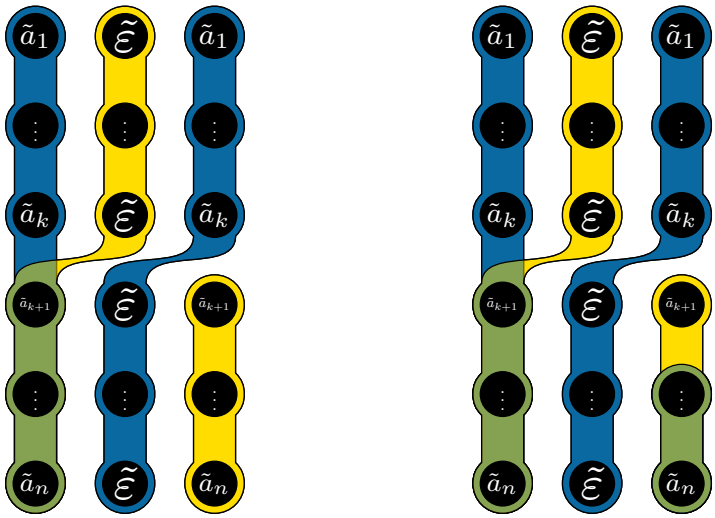
(b) RoundRobin, uses two time steps per phase and wastes 99% of the resource at the end of each phase.

Fig. 3: Worst-case example for RoundRobin schedule. Node labels give the jobs' resource requirements in percent.

transforms a Partition instance of n elements into a CRSharing instance on n processors, each having three jobs to process.

Let $a_1, a_2, \ldots, a_n \in \mathbb{N}$ and $A \in \mathbb{N}$ with $\sum_{i=1}^{n} a_i = 2A$ be the input of the Partition instance (w.l.o.g., $A \geq 2$). For our transformation, let $\varepsilon \in (0, 1/n)$ and set $\delta := n\varepsilon < 1$. We define the first and last job on any processor i to have resource requirements $r_{i1} = r_{i3} = \tilde{a}_i := \frac{a_i}{A+\delta}$. The second job on any processor i has a resource requirement of $r_{i2} = \tilde{\varepsilon} := \frac{\varepsilon}{A+\delta}$. Note that no schedule can finish the first job of all tasks in only one time step as we have $\sum_{i=1}^{n} r_{i1} = \frac{2A}{A+\delta} > 1$ by construction. Now, with each task containing three jobs, any schedule needs at least four time steps to finish all jobs. To finish our reduction, we show that there is an optimal schedule with makespan 4 if and only if the given Partition instance is a YES-instance (i.e., if it can be partitioned into two sets that sum up to exactly A).

Assume we are given a YES-instance of Partition and let, w.l.o.g., the first k elements form one partition. The schedule shown in Figure 4a is feasible and has makespan 4. Now, assume we are given a NO-instance and an optimal schedule for the corresponding CRSharing instance. W.l.o.g., exactly the first k processors finish their jobs in the first time step. This implies $\sum_{i=1}^{k} \tilde{a}_i \leq 1$, yielding the inequality $\sum_{i=1}^{k} a_i \leq A + \delta < A + 1$. Since the given Partition



(a) Optimum for YES-instances. (b) Optimum for NO-instances.

Fig. 4: Problem instance and schedules used for the reduction from PARTITION to CRSHARING with unit size jobs.

instance is a NO-instance, we also have $\sum_{i=1}^{k} a_i \neq A$. Together this implies $\sum_{i=1}^{k} a_i \leq A-1$, which, in turn, yields $\sum_{i=k+1}^{n} a_i \geq A+1$. Since we have not yet finished the jobs $(k+1,1),(k+2,1),\ldots,(n,1)$, we need at least two more time steps until we can start working on $(k+1,3),(k+2,3),\ldots,(n,3)$. Their total resource requirement is at least

$$\sum_{i=k+1}^{n} \tilde{a}_i \frac{\sum_{i=k+1}^{n} a_i}{A+\delta} \geq \frac{A+1}{A+\delta} > 1. \tag{6}$$

Thus, after the first three time steps, we need at least two more time steps to finish the remaining jobs, yielding a makespan of at least 5. □

Note that we also get the following lower bound from the proof of Theorem 4:

Corollary 1 *It is NP-hard to approximate* CRSHARING *with a factor better than* $5/4$.

While Theorem 4 proves NP-hardness of our problem, it leaves the question concerning the problem's complexity for *constant* m. In the next two sections we will show that in this case the problem is polynomial-time solvable.

6 Algorithm for Two Processors

While the previous section proves NP-hardness in the number of processors, there are exact polynomial-time algorithms for a fixed number of processors. Before we state and analyze the algorithm for arbitrary $m \geq 2$ in Section 7, we introduce a faster algorithm for two processors. Algorithm OptResAssignment traces out all reasonable scheduling decisions. To keep this approach feasible, we use Lemma 1 (implying the existence of an optimal schedule that finishes at least one job in each time step) and another structural property (see Lemma 3). These allow us to discard bad scheduling decisions early on.

Algorithm Description. The OptResAssignment algorithm uses a dynamic programming approach. To this end, it maintains a two-dimensional array B of size $n_1 \times n_2$. Each entry holds a tuple $B[i_1, i_2] = (r, t)$, which states that there is a schedule that, at time step t, has finished all jobs $(1, j_1)$ with $j_1 < i_1$ and $(2, j_2)$ with $j_2 < i_2$, and for which the remaining resource requirements of $(1, i_1)$ and $(2, i_2)$ sum up to r. OptResAssignment fills B in $n_1 + n_2 - 1$ phases, one phase for each diagonal of B. It maintains the invariant that, from the start of phase ℓ on, all entries on the $(\ell - 1)$-th diagonal (i.e., all $B[i_1, i_2]$ with $i_1 + i_2 = \ell$) are optimal. More precisely, such entries correspond to subschedules with minimal t (and, for this t, minimal r) reaching the jobs $(1, i_1)$ and $(2, i_2)$. See Algorithm 1 for the pseudocode. Note that, in our algorithm description, we compute only the makespan (and not a corresponding schedule) of an optimal solution. However, given the array B, one can easily trace back the final entry and derive an explicit schedule in linear time.

Correctness & Runtime. We start with a simple lemma, which will be used later on to show that the diagonal-wise processing of B is correct.

Lemma 3 *Consider two non-wasting and progressive schedules S and S' as well as a time step t such that $n_i(t) \leq n'_i(t)$ for $i \in \{1, 2\}$. Let $v_i(t)$ and $v'_i(t)$ be the remaining resource requirement of the job that is active at time t on processor $i \in \{1, 2\}$ in schedule S and S', respectively. If*

(a) $n_1(t) < n'_1(t)$ or $n_2(t) < n'_2(t)$, or
(b) $n_1(t) = n'_1(t)$ and $n_2(t) = n'_2(t)$ and, w.l.o.g., $v_1(t) + v_2(t) \leq v'_1(t) + v'_2(t)$,

then we can transform S without changing the first $t - 1$ time steps such that $S \leq S'$.

Proof First observe that we already have $S \leq S'$ if one of the properties applies at the end of S. Thus, it suffices to show that the properties can be maintained from t to $t + 1$.

(a) Without loss of generality, assume $n_1(t) < n'_1(t)$. If S' finishes only one job, S can complete a job on the same processor and hence maintains the inequalities. If S' finishes both jobs, this yields $n'_i(t + 1) = n'_i(t) - 1$ for $i \in \{1, 2\}$. Thus, if S finishes a job on processor 2 and assigns the remaining

Algorithm 1 OptResAssignment

```
1: // resource requirements are stored in A_1 and A_2
2: // subschedules are stored in two-dimensional array B
3: // extend A_1 as well as A_2 by an extra 0-entry
4: n_1 = length(A_1); n_2 = length(A_2);
5: initialize array B[1...n_1, 1...n_2] with null entries
6: B[1,1] = (A_1[1] + A_2[1], 0)
7: for ℓ = 2...n_1 + n_2 − 1 do
8:    for i_1 = max{1, ℓ − n_2}...min{ℓ − 1, n_1} do
9:       i_2 = ℓ − i_1
10:      (r, t) = B[i_1, i_2]
11:      if i_1 = n_1 then
12:         add(i_1, i_2 + 1, 0, A_2[i_2 + 1], t + 1)
13:      else if i_2 = n_2 then
14:         add(i_1 + 1, i_2, A_1[i_1 + 1], 0, t + 1)
15:      else if r ≤ 1 then
16:         add(i_1 + 1, i_2 + 1, A_1[i_1 + 1], A_2[i_2 + 1], t + 1)
17:         add(i_1, i_2 + 1, 0, A_2[i_2 + 1], t + 1)
18:         add(i_1 + 1, i_2, A_1[i_1 + 1], 0, t + 1)
19:      else
20:         add(i_1, i_2 + 1, A_1[i_1] + A_2[i_2] − 1, A_2[i_2 + 1], t + 1)
21:         add(i_1 + 1, i_2, A_1[i_1 + 1], A_1[i_1] + A_2[i_2] − 1, t + 1)
22: min = B[n_1, n_2]
23:
24: function add(i_1, i_2, v_1, v_2, t)
25: r = v_1 + v_2
26: (r_old, t_old) = B[i_1, i_2]
27: if (r_old, t_old) = null ∨ t < t_old ∨ (t = t_old ∧ r < r_old) then
28:    B[i_1, i_2] = (r, t)
```

bandwidth to the job on processor 1, this results in $n_1(t+1) = n_1(t) \leq n'_1(t+1)$ and $n_2(t+1) = n_2(t) - 1 \leq n'_2(t+1)$. If equality applies (otherwise (a) holds), then the same jobs are active at time $t+1$ in S' and S, say j_1 and j_2. This yields $v_1(t+1) + v_2(t+1) \leq r_{1j_1} + r_{2j_2} = v'_1(t+1) + v'_2(t+1)$, therefore (b) applies.

(b) Now, suppose $v_1(t) + v_2(t) \leq v'_1(t) + v'_2(t)$. If S' finishes both jobs, S can do the same and (b) holds with equality. If S' only finishes one job (w.l.o.g., job $j-1$ on processor 1), S can also finish that job. If $v_1(t) + v_2(t) \leq 1$, it also completes a second job and therefore (a) applies. On the other hand, if $v_1(t) + v_2(t) > 1$, this results in $v_1(t+1) + v_2(t+1) = r_{1j} + (v_1(t) + v_2(t) - 1) \leq r_{1j} + (v'_1(t) + v'_2(t) - 1) = v'_1(t+1) + v'_2(t+1)$, thus case (b) applies.

□

Theorem 5 *Consider a* CRSharing *instance with unit size jobs and two processors. The following statements hold:*

(a) OptResAssignment *computes an optimal solution.*
(b) OptResAssignment *has running time* $\mathrm{O}(n^2)$.

Proof The correctness of statement (b) is immediate, as OptResAssignment runs in $\mathrm{O}(n)$ phases and each phase considers the $\mathrm{O}(n)$ entries on the corresponding diagonal. It remains to prove the correctness of statement (a).

Remember the invariant from the algorithm description: At the beginning of phase ℓ, for each entry $B[i_1, i_2] = (r, t)$ on the $\ell - 1$-th diagonal the following holds: t is the earliest time at which all jobs preceding $(1, i_1)$ and $(2, i_2)$ can be finished and r is, for this t, the smallest possible sum of the remaining resource requirements of $(1, i_1)$ and $(2, i_2)$. If this invariant holds for phase $n_1 + n_2$, the correctness follows immediately (we use dummy jobs, so the last diagonal entry corresponds to all non-dummy jobs being fully processed). For the first phase, the invariant's correctness is obvious from the initialization, as there are no jobs preceding $(1, 1)$ and $(2, 1)$. Now assume the invariant holds for the first ℓ phases and consider an entry $B[i_1, i_2]$ processed in the $\ell + 1$-th phase. This entry corresponds to a subschedule that has processed all jobs preceding $(1, i_1)$ and $(2, i_2)$. Since each processor can finish at most one job in one time step, this subschedule must originate from one of the subschedules S_1, S_2, or S_3 that have finished all jobs preceding (i) $(1, i_1 - 1)$ and $(2, i_2)$, (ii) $(1, i_1)$ and $(2, i_2 - 1)$, and (iii) $(1, i_1 - 1)$ and $(2, i_2 - 1)$, respectively. By our induction hypothesis, the entries in $B[i_1 - 1, i_2]$, $B[i_1, i_2 - 1]$, and $B[i_1 - 1, i_2 - 1]$ correspond to the best possible such schedules. Since the algorithm uses these to compute $B[i_1, i_2]$ (lines 9-21) and the best of them is chosen as predecessor (line 27, correct by Lemma 3), the invariant is established for entry $B[i_1, i_2]$ (and, similarly, for all remaining entries on the same diagonal). □

An alternative implementation of the algorithm replaces the 2-dimensional array by a priority queue that orders intermediate schedules by their index sum $i_1 + i_2$. Although adding/retrieving such an entry has amortized costs $O(\log(n))$, this implementation runs faster for most of the instances, as it only considers index pairs that actually point to a schedule and many index pairs are usually not used. Consider, for instance, pair $(1, 1)$. If $A_1[1] + A_2[1] \leq 1$, the algorithm will proceed with $(2, 2)$ and all entries $(1, i_2)$ and $(i_1, 1)$ with $i_1, i_2 > 1$ will never be used.

7 Algorithm for m Processors

While in the previous section we discussed OptResAssignment, an exact algorithm for $m = 2$ having a worst case running time of $O(n^2)$, this section shows that there is even a polynomial-time algorithm for any fixed m; we call it OptResAssignment2. In the proof we will restrict the schedules to nested ones (see Definition 4) and use the new notion of an (extended) configuration representing the current state of a schedule. We argue that only a polynomial number of extended configurations has to be considered and show that this implies a polynomial running time.

Additional Notation. The configuration of a schedule S in round t can be described by the sequence $(j_1(t), \ldots, j_m(t))$ of jobs completed and the amounts $(v_1(t), \ldots, v_m(t))$ of resource spent for the active jobs before round t. In particular $v_i(t) = 0$ if the active job has not started yet.

Definition 6 ((Extended) configuration; core; support) A *configuration* γ is a vector $(t, j_1(t), \dots, j_m(t), v_1(t), \dots, v_m(t))$ where $j_i(t) \in \{0, \dots, n_i\}$ and $v_i(t) \in [0, 1]$. The *core* of γ is defined as $\text{core}(\gamma) = (j_1(t), \dots, j_m(t))$ and its *support* as $\text{supp}(\gamma) = \{ i \mid v_i(t) > 0 \}$. Further we define the *extended configuration* of γ as the tuple $E(\gamma) \coloneqq (\gamma, (i, \gamma_i)_{i \in \text{supp}(\gamma)})$, where γ_i is the configuration after the round in which processor i received resource for the last time.

We say, two configurations are *step-equal* if they are in the same time step and if their corresponding cores are equal. Two extended configurations $E(\gamma) = (\gamma, (i, \gamma_i)_{i \in \text{supp}(\gamma)})$ and $E(\gamma') = (\gamma', (i, \gamma'_i)_{i \in \text{supp}(\gamma')})$ are *step-equal* if (a) γ and γ' are step-equal, (b) they have the same support and (c) γ_i and γ'_i are step-equal for all $i \in \text{supp}(\gamma)$.

To obtain a polynomial-time algorithm, we reduce the number of relevant configurations to a polynomial number. If $(t, j_1(t), \dots, j_m(t), v_1(t), \dots, v_m(t))$ and $(t', j'_1(t'), \dots, j'_m(t'), v'_1(t'), \dots, v'_m(t'))$ were, for example, both feasible configurations with $t \leq t'$, $j_\ell(t) \geq j'_\ell(t')$ and $v_\ell(t) \geq v'_\ell(t')$ for all $1 \leq \ell \leq m$, we would not need the second configuration, as the first one is always to be preferred. We say that the first configuration *dominates* the second one. The following lemma proves a natural connection between this property of domination and step-equal configurations.

Lemma 4 *If two extended configurations are step-equal, then one dominates the other.*

Proof We proof the lemma by induction on $|\text{supp}(\gamma)|$.

First we discuss the cases $|\text{supp}(\gamma)| = 0$ and $|\text{supp}(\gamma)| = 1$. If $|\text{supp}(\gamma)| = 0$, then all $v_i(t) = 0$ and, hence, there cannot be another configuration with the same core. In the second case, any two configurations γ and γ' differ only in one value $v_i(t)$ so that either γ dominates γ' or vice versa.

Now consider two non-dominated and step-equal extended configurations $(\gamma, (i, \gamma_i)_{i \in \text{supp}(\gamma)})$ and $(\gamma', (i, \gamma'_i)_{i \in \text{supp}(\gamma')})$ with $|\text{supp}(\gamma)| = |\text{supp}(\gamma')| \geq 2$. For all $i \in \text{supp}(\gamma)$, denote by t_i the round of γ_i (and γ'_i), and let k such that $t_k = \max \{ t_i \mid i \in \text{supp}(\gamma) \}$. (Note that the t_i are pairwise distinct because there is at most one partly processed job in each round.)

As the extended configurations are step-equal, the extended configurations after round t_k, from which γ and γ' are derived, namely $(\gamma_k, (i, \gamma_i)_{i \in \text{supp}(\gamma) \setminus \{k\}})$ and $(\gamma'_k, (i, \gamma'_i)_{i \in \text{supp}(\gamma') \setminus \{k\}})$, are also step-equal. They must be the same because, due to the induction hypothesis, there are no two different non-dominated and step-equal extended configurations with a support smaller than $|\text{supp}(\gamma)|$.

After t_k, none of the tasks in $\text{supp}(\gamma)$ received resource in γ or γ' so that $v_i(t) = v'_i(t)$ for all $i \in \text{supp}(\gamma) \setminus \{k\}$. Furthermore, all of the resource was used in these rounds because there were unfinished jobs in each of them. And since the same set of jobs was completed in these rounds, it must hold that $\sum_{i \in \text{supp}(\gamma)} v_i(t) = \sum_{i \in \text{supp}(\gamma)} v'_i(t)$ and, thus, $v_k(t) = v'_k(t)$. Hence, $E(\gamma)$ and $E(\gamma')$ are the same. □

Algorithm. To find an optimal schedule, our algorithm OptResAssignment2 (Algortihm 2) enumerates all configurations that are not dominated by another configuration. Starting from the initial configuration $(1, 0, \ldots, 0, 0, \ldots, 0)$, it computes the configurations of the next round based on the configurations of the current round. While doing this, it makes sure that the respective schedules remain non-wasting, progressive, and nested. In each round, it additionally removes all dominated configurations by a pairwise comparison of the new configurations. When the algorithm hits an end configuration, it outputs the path to it and stops.

Algorithm 2 OptResAssignment2

```
1: C_1 = { (1,0,...,0,0,...,0) }
2: for t = 2,... do
3:    C_t = ∅
4:    for all γ ∈ C_{t-1} do
5:       succ(γ) = successors of γ
6:       store link between γ and each γ' ∈ succ(γ)
7:       C_t = C_t ∪ succ(γ)
8:    if (t, n_1 + 1, ..., n_m + 1, 0, ..., 0) ∈ C_t then
9:       output path to this configuration
10:      break
11:   for all γ ∈ C_t do
12:      for all γ' ∈ C_t \ { γ } do
13:         if γ dominates γ' then
14:            remove γ' from C_t
```

Theorem 6 OptResAssignment2 *computes an optimal schedule in time polynomial in n.*

Proof In each pass of the outer for loop, Algorithm 2 creates all subschedules of t steps which are non-wasting, progressive and nested and whose current configuration is not dominated by another one. As soon as a final configuration is reached, the algorithm outputs the results and stops. Therefore, the correctness of the algorithm follows from Lemma 1 which states that there is at least one optimal schedule among all non-wasting, progressive and nested schedules.

In order to show the running time, we will roughly bound the number of configurations that are computed by the algorithm: the non-dominated ones as well as the dominated ones (that are discarded right away). From Lemma 4 we know that there is exactly one configuration that dominates all the others.

Let ν_{ext} be the number of all possible non-dominated extended configurations which are pairwise not step-equal. Since the number of time steps is bounded by $\sum_{i=1}^{m} n_i \leq m \cdot n$ and the number of cores by $\prod_{i=1}^{m} n_i \leq n^m$, we can bound the number of configurations which are not step-equal by $m \cdot n \cdot n^m$. An extended configuration consists of up to $m + 1$ such configurations so that we obtain

$$\nu_{ext} \leq (m \cdot n \cdot n^m)^{m+1} = m^{m+1} \cdot n^{(m+1)^2}.$$

The number of (non-dominated and dominated) configurations that immediately succeed a given configuration is bounded by $m \cdot 2^m$ because there are at most 2^m possibilities to choose a subset of processors and at most m possibilities to choose the partly processed job. Since each non-dominated configuration is used only once as a base configuration (from which successive configurations are derived), we can bound the total number of computed configurations by $\nu_{ext} \cdot m \cdot 2^m$.

The time for each round is determined by the time for separating the dominated configurations which is quadratic in the number $\mathcal{C}_t$ of step-t configurations. Hence, very roughly, we can bound the total running time by

$$\mathrm{O}\Big(\Big(m^{m+1} \cdot n^{(m+1)^2} \cdot m \cdot 2^m\Big)^2\Big) = \mathrm{O}\Big(m^{2 \cdot m+4} \cdot n^{2 \cdot (m+1)^2} \cdot 2^{2 \cdot m}\Big).$$

□

8 Balanced Schedules

This section builds up to our last result, an approximation algorithm with a tight approximation ratio of $2 - 1/m$, in Theorem 7. While the quality of the result is obviously worse compared to OptResAssignment2, it can be achieved by running a simple linear-time algorithm called GreedyBalance. We start by providing two lower bounds for optimal schedules in terms of a given non-wasting and balanced schedule, respectively.

8.1 Lower Bounds for Optimal Schedules

The following lemma derives the first lower bound by exploiting the fact that, within a component, any non-wasting schedule always makes full use of the resource.

Lemma 5 *Let* OPT *denote the minimal makespan of a given problem instance and consider the scheduling graph H_S of a non-wasting schedule S. Then* OPT *can be bounded by*

$$\mathrm{OPT} \geq \sum_{k=1}^{N} (\#_k - 1). \tag{7}$$

Proof From Observation 1, we immediately get that $\mathrm{OPT} \geq \sum_{i=1}^{m} \sum_{j=1}^{n_i} r_{ij}$. Consider a connected component C_k of our schedule containing the edges $t_1, t_1 + 1, \ldots, t_2$. Since S is non-wasting, $\sum_{i=1}^{m} R_i(t) = 1$ holds for all time steps $t \in \{\, t_1, t_1 + 1, \ldots, t_2 - 1 \,\}$. If there were such a t with $\sum_{i=1}^{m} R_i(t) < 1$, the non-wasting property would imply that all active jobs are finished. But then the edge

e_{t+1} would not be part of C_k, yielding a contradiction. For the last time step t_2 of C_k we have $\sum_{i=1}^{m} R_i(t_2) \geq 0$. Since S is feasible and, w.l.o.g., does not use more of the resource than necessary, it follows that $\sum_{t=1}^{S}\sum_{i=1}^{m} R_i(t) = \sum_{i=1}^{m}\sum_{j=1}^{n_i} r_{ij}$. Let $e^{(k)}$ denote the last edge of C_k. Then we get:

$$\begin{aligned}\mathrm{OPT} &\geq \sum_{i=1}^{m}\sum_{j=1}^{n_i} r_{ij} = \sum_{t=1}^{S}\sum_{i=1}^{m} R_i(t) = \sum_{k=1}^{N}\sum_{e_t \subseteq C_k}\sum_{(i,j)\in e_t} R_i(t)\\ &\geq \sum_{k=1}^{N}\sum_{\substack{e_t \subseteq C_k\\ e_t \neq e^{(k)}}} 1 = \sum_{k=1}^{N}(\#_k - 1).\end{aligned}$$

□

The second lower bound centers around utilizing parallelism. In a problem instance where each processor has exactly n jobs, the maximum exploitable parallelism is m. On the other hand, in a schedule with components C_k of class q_k, the maximum parallelism that can be exploited in C_k is q_k. In a sense, the following lemma shows that, in the case of balanced schedules, this is not much worse than m.

Lemma 6 *Let* OPT *denote the minimal makespan of a given problem instance and remember that n denotes the maximum number of jobs any processor has to process. Given a balanced schedule S and its scheduling graph,* OPT *and n can be bounded by the inequalities*

$$\mathrm{OPT} \geq n \geq \sum_{k=1}^{N-1} \frac{|C_k|}{q_k} + \frac{|C_N|}{m}. \tag{8}$$

Proof Remember that M_j is the set of processors having at least j jobs to process. Since any schedule can process at most one job per processor in every time step, even an optimal schedule needs at least n time steps to finish all jobs. We can write n as $\sum_{(i,j)\in V} {}^{1}/_{|M_j|}$, yielding

$$\begin{aligned}\mathrm{OPT} \geq n &= \sum_{(i,j)\in V} \frac{1}{|M_j|} = \sum_{k=1}^{N}\sum_{(i,j)\in C_k} \frac{1}{|M_j|}\\ &\geq \sum_{k=1}^{N-1}\sum_{(i,j)\in C_k} \frac{1}{|M_j|} + \sum_{(i,j)\in C_N} \frac{1}{m}\\ &= \sum_{k=1}^{N-1}\sum_{(i,j)\in C_k} \frac{1}{|M_j|} + \frac{|C_N|}{m}.\end{aligned}$$

It remains to show that we have

$$\sum_{(i,j)\in C_k} \frac{1}{|M_j|} \geq \frac{|C_k|}{q_k} \tag{9}$$

for all but the last component. So fix $k \in \{1, 2, \dots N-1\}$ and let $(i_0, j_0) \in C_k$ be a job of the k-th component with minimal j_0. Let t_0 be the first time step when (i_0, j_0) is active. The minimality of j_0 implies that e_{t_0} is the first edge of C_k and, thus, $q_k = |e_{t_0}|$. We distinguish two cases:

Case 1: $n_{i_0}(t_0) > 1$
By applying Proposition 2, we get that all processors $i \in M_{j_0}$ are active at time step t_0. This yields $|M_{j_0}| \leq |e_{t_0}| = q_k$. Moreover, for a job $(i,j) \in C_k$, the minimality of j_0 gives us $|M_{j_0}| \geq |M_j|$. Combining both inequalities implies $|M_j| \leq q_k$. Applying this to the first part of Equation (9) eventually yields the desired inequality.

Case 2: $n_{i_0}(t_0) = 1$
In this case, (i_0, j_0) is the last job on processor i_0 at time step t_0. However, for any job $(i,j) \in C_k \setminus e_{t_0}$ we have $n_i(t_0) > 1$. Given such a job, let (i, j') be the job processed on i at time step t_0. Note that we have $j' < j$ and, thus, $M_j \subseteq M_{j'}$. By applying Proposition 2, we get that all $i' \in M_{j'}$ are active at time step t_0. Together with $M_j \subseteq M_{j'}$, this yields $|M_j| \leq q_k$. Thus, to prove Equation (9), it only remains to show $\sum_{(i,j)\in e_{t_0}} 1/|M_j| \geq \sum_{(i,j)\in e_{t_0}} 1/q_k (= 1)$.

To this end, note that, since C_k is not the last component, there exists at least one job $(i_1, j_1) \in e_{t_0}$ with $n_{i_1}(t_0) > 1$. Let this job be such that j_1 is minimal. Once more, by applying Proposition 2 we get that all $i \in M_{j_1}$ are active at time step t_0. Consider a job $(i,j) \in e_{t_0}$ with $i \in M_{j_1}$. If it is the last job on i (i.e., if $n_i(t_0) = 1$), we have $j = n_i$. Together with the definition of M_{j_1} we get $j = n_i \geq j_1$, yielding $|M_j| \leq |M_{j_1}|$. Similarly, if it is not the last job on i (i.e., if $n_i(t_0) > 1$), the minimality of j_1 gives us $|M_j| \leq |M_{j_1}|$. This yields the desired inequality as follows:

$$\sum_{(i,j)\in e_{t_0}} \frac{1}{|M_j|} \geq \sum_{\substack{(i,j)\in e_{t_0} \\ i\in M_{j_1}}} \frac{1}{|M_j|} \geq \sum_{\substack{(i,j)\in e_{t_0} \\ i\in M_{j_1}}} \frac{1}{|M_{j_1}|} = 1.$$

□

8.2 Deriving a $(2 - 1/m)$-Approximation

Finally, we have all the ingredients to prove our main result:

Theorem 7 *Consider a* CRSharing *instance with unit size jobs and a feasible schedule S for it that is non-wasting, progressive, and balanced. Then S is a $(2 - 1/m)$-approximation with respect to the optimal makespan.*

Proof In the following, let $\#_\varnothing \coloneqq \sum_{k=1}^{N} \#_k \,/N$ denote the average number of edges in a component. Our proof uses two bounds on the approximation ratio. The first one follows easily from Lemma 5 and leads to a better approximation for instances with large $\#_\varnothing$. The second bound is much more involved and mainly based on Lemma 6. It yields a better approximation for instances with small $\#_\varnothing$. To get the first bound, we simply apply Lemma 5 and get

$$\frac{S}{\mathrm{OPT}} \leq \frac{\sum_{k=1}^{N} \#_k}{\sum_{k=1}^{N} (\#_k - 1)} = \frac{\#_\varnothing}{\#_\varnothing - 1}. \tag{10}$$

Let us now consider the second bound, based on Lemma 6. Our goal is to show that the inequality

$$\frac{S}{\mathrm{OPT}} \leq \frac{m \cdot \#_\varnothing}{\#_\varnothing + m - 1} \tag{11}$$

holds. Once this is proven, we can combine both bounds by realizing that the bound from Equation (10) is monotonously decreasing in $\#_\varnothing$ and the bound from Equation (11) is monotonously increasing in $\#_\varnothing$. Equalizing yields that their minimum's maximum is obtained at $\#_\varnothing = \dfrac{2m-1}{m-1}$, which results in an approximation ratio of $2 - 1/m$.

The rest of this proof is geared towards proving Equation (11). We distinguish two cases. The first case covers the easier part, where we have $\mathrm{OPT} \geq n + 1$. That is, even an optimal solution cannot finish the jobs in n time steps. The second case, where we have $\mathrm{OPT} = n$, turns out to be more difficult to prove. While we can apply a similar analysis, we have to take more care when bounding our algorithm's progress in the first two time steps.

Case 1: $\mathrm{OPT} \geq n + 1$
Applying Lemma 6 to this case yields

$$\begin{aligned} \frac{S}{\mathrm{OPT}} &\leq \frac{\sum_{k=1}^{N} \#_k}{\sum_{k=1}^{N-1} \frac{|C_k|}{q_k} + \frac{|C_N|}{m} + 1} \\ &\leq \frac{N \cdot \#_\varnothing}{\sum_{k=1}^{N-1} \frac{\#_k + q_k - 1}{q_k} + \frac{\#_N + m - 1}{m}} \\ &\leq \frac{N \cdot \#_\varnothing}{\sum_{k=1}^{N} \frac{\#_k + m - 1}{m}} \leq \frac{m \cdot \#_\varnothing}{\#_\varnothing + m - 1} \end{aligned} \tag{12}$$

Case 2: $\mathrm{OPT} = n$
If we apply the same analysis as in the first case, we will fall short of our desired approximation ratio. Surprisingly, it turns out to be sufficient to bound only the first two time steps more carefully. The idea of the following analysis is to consider the first two time steps of S and the remaining part of S separately. To this end, first note that we can assume, w.l.o.g., that $\#_1 > 1$ (i.e., the first two time steps belong to the same component). If this is not the case,

our algorithm finishes all active jobs in the first time step and, thus, behaves optimally[2]. Consider the remaining jobs/workloads after the first two time steps. We can regard this as a subinstance of our original problem instance. Let S' denote the subschedule that results from restricting S to time steps $t \geq 3$. We use N', $\#'_k$, q'_k, and n' to refer to the corresponding properties of its scheduling graph $H_{S'}$. Note that we have $N' \geq N - 1$ (because of our assumption $\#_1 > 1$) as well as $N' \cdot \#'_\varnothing = N \cdot \#_\varnothing - 2$ (since exactly two time steps are missing in the subschedule). Moreover, we also have $n' = n - 2$. The inequality $n' \geq n-2$ is obvious. For $n' \leq n-2$, note that OPT must finish the jobs in the set $\{ (i,1) \mid n_i(1) \geq n-1 \} \cup \{ (i,2) \mid n_i(1) \geq n \}$ during the first two time steps. Thus, the total resource requirement of these jobs is at most two. Since S is balanced, it will prioritize and, thus, finish these jobs in the first two time steps. Finally, we can bound our approximation ratio as follows (the first inequality applies Lemma 6 to S'):

$$\begin{aligned}
\frac{S}{\mathrm{OPT}} = \frac{N \cdot \#_\varnothing}{2 + n'} &\leq \frac{N \cdot \#_\varnothing}{2 + \sum_{k=1}^{N'-1} \frac{|C'_k|}{q'_k} + \frac{|C'_{N'}|}{m}} \\
&\leq \frac{N \cdot \#_\varnothing}{1 + \frac{1}{m} + \sum_{k=1}^{N'-1} \frac{\#'_k + q'_k - 1}{q'_k} + \frac{\#'_{N'}}{m} + \frac{m-1}{m}} \\
&\leq \frac{N \cdot \#_\varnothing}{1 + \frac{1}{m} + \sum_{k=1}^{N'} \frac{\#'_k + m - 1}{m}} \\
&= \frac{N \cdot m \cdot \#_\varnothing}{m + 1 + N' \cdot \#'_\varnothing + N'(m-1)} \\
&\leq \frac{N \cdot m \cdot \#_\varnothing}{2 + (N \cdot \#_\varnothing - 2) + N(m-1)} = \frac{m \cdot \#_\varnothing}{\#_\varnothing + m - 1}.
\end{aligned}$$

This proves that Equation (11) also holds in this case.

□

8.3 Tight Approximation Algorithm

So far, we analyzed the quality of balanced schedules in general, but did not yet provide a concrete example of a corresponding algorithm. One of the most natural greedy algorithms schedules jobs by prioritizing processors with a higher number of remaining jobs and, in the case of a tie, by prioritizing jobs with larger remaining resource requirements. We name this algorithm GreedyBalance. In Section 8.2, we saw that balanced schedules and, as a consequence, the algorithm GreedyBalance yield a $(2 - 1/m)$-approximation for the CRSharing problem. Now, we show that this approximation ratio is tight for GreedyBalance.

Theorem 8 *The* GreedyBalance *algorithm for the* CRSharing *problem with jobs of unit size has a worst-case approximation ratio of exactly* $2 - 1/m$.

[2] This reduces our analysis to a smaller problem instance.

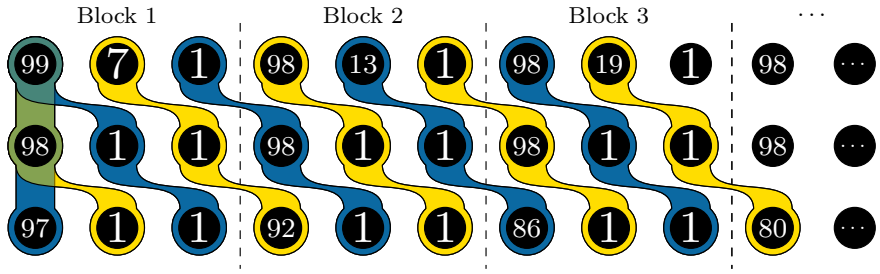
Proof Since GreedyBalance computes only balanced schedules, the upper bound follows immediately from Theorem 7. For the lower bound, consider a family of problem instances defined as follows: We define blocks of $m \times m$ jobs with resource requirements as described below. For the first block, let $r_{i1} \coloneqq 1 - i \cdot \varepsilon$ for $i \in \{1, 2, \ldots, m\}$, $r_{12} \coloneqq 1 - \sum_{i=1}^{m}(1 - r_{i1}) + \varepsilon$, and $r_{i2} \coloneqq \varepsilon$ for $i \in \{2, 3, \ldots, m\}$. Moreover, define $r_{ij} \coloneqq \varepsilon$ for all $i \in \{1, 2, \ldots, m\}$ and $j \in \{3, 4, \ldots, m\}$. This finishes the first $m \times m$-block of jobs. Having constructed the l-th block, we construct the next block, starting with its first column $j \coloneqq l \cdot m + 1$. We define $r_{ij} \coloneqq 1 - (m-1)\varepsilon$ for $i \in \{1, 2, \ldots, m-1\}$ and $r_{mj} \coloneqq 1 - \sum_{i'=1}^{m-1} r_{m-i',j-i'}$. For the second column of this block we set $r_{1,j+1} \coloneqq 1 - \sum_{i=1}^{m}(1 - r_{ij}) + \varepsilon$, and $r_{i,j+1} \coloneqq \varepsilon$ for $i \in \{2, 3, \ldots, m\}$. To finish the block, we set $r_{ij'} \coloneqq \varepsilon$ for all $i \in \{1, 2, \ldots, m\}$ and $j' \in \{j+2, j+3, \ldots, j+m-1\}$. We finish the construction once the next block would contain jobs with negative resource requirements. Note that by choosing ε small enough, we can make this construction arbitrarily long. See Figure 5 for an illustration of this construction and the schedules produced by GreedyBalance and an optimal algorithm. Our construction is such that GreedyBalance needs exactly $2m - 1$ time steps per block: By balancing the number of remaining jobs, it is forced to work m time steps on a block's first column (which contains a total resource requirement of roughly m) before it can finish the remaining $m - 1$ columns of a block. In contrast, the optimal algorithm ignores any balancing issues, which allows it to exploit that all diagonals have a total resource requirement of 1. □

9 Conclusion & Outlook

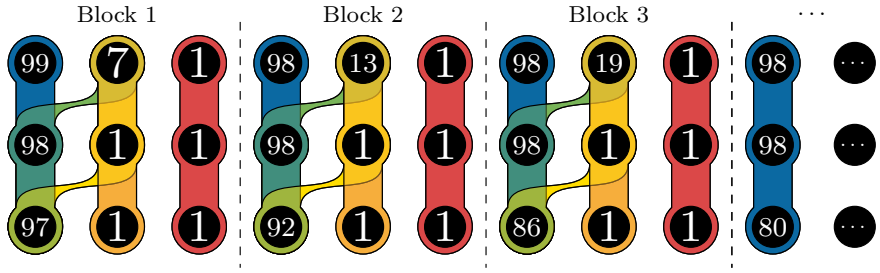
We introduced a new resource-constrained scheduling problem where job processing speeds depend on the share of the resource a job is assigned. Even for unit size jobs, this problem turned out to be NP-hard in the number of processors. However, we were able to derive an algorithm that computes an optimal solution in polynomial time. This algorithm merely proves that such solutions exist, but is by no means practical. While for $m = 2$ processors we presented an exact quadratic-time algorithm, it remains an open question whether there are efficient polynomial-time algorithms for $m \geq 3$. For these cases we provided a linear-time approximation algorithm with a worst-case approximation ratio of $2 - 1/m$.

Restricting the analysis to unit size jobs, we did not give analytical results for jobs of arbitrary sizes[3], but we conjecture that almost all results should be

[3] One could also consider resource requirements > 1. However, the most natural extension of our model can easily be shown to reduce to non-unit size jobs with resource requirements



(a) An optimal schedule.



(b) Schedule computed by GreedyBalance.

Fig. 5: Construction and schedules used in the proof of Theorem 8 for $m = 3$ and $\varepsilon = 0.01$. Node labels show the corresponding job's resource requirement in percent (e.g., $r_{12} = 0.07$). Note that the optimal schedule needs (essentially) m time steps to finish a block, while S needs $2m - 1$ time steps per block.

transferable. However, extending our analysis turns out to be non-trivial. In particular our scheduling (hyper-) graphs cannot, with their current definition, capture such problem instances. And yet, intuition suggests that one should be able to extend our definitions and find similar structural properties for arbitrary job sizes.

Besides extending our results to jobs of arbitrary sizes, it seems worthwhile to extend the model to other, possibly more realistic scenarios. What analytical results are possible if we re-introduce the classical scheduling aspect, where jobs of a task are not a priori fixed to a specific processor? It may also be possible to use our insights to get analytical results in special cases of discrete-continuous models as proposed by Józefowska and Weglarz [11]. Another interesting direction are models that consider energy as a continuously divisible resource. One might imagine a multiprocessor model in the spirit of the original speed scaling model by Yao et al. [18], but with a shared energy source (cf. [14, 15]). Finally, we opted for a discrete time model, both because it facilitates the analysis and because it fits well in typical implementations of real-world schedulers (which are usually called at regular time intervals [2]). Nevertheless,

≤ 1 (rescale jobs with resource requirement $r > 1$ and workload p such that it has resource requirement $1/r \cdot r = 1$ and workload $r \cdot p$).

it seems an intriguing question to consider this problem in a more sophisticated, continuous setting where the scheduler can act at arbitrary times.

References

1. Błażewicz, J., Ecker, K.H., Pesch, E.: Handbook on Scheduling: From Theory to Applications. Springer (2007)
2. Błażewicz, J., Ecker, K.H., Pesch, E., Schmidt, G., Weglarz, J.: Handbook on Scheduling - From Theory to Applications. International Handbooks on Information Systems. Springer-Verlag Berlin Heidelberg (2007)
3. Chung, F., Graham, R., Mao, J., Varghese, G.: Parallelism versus memory allocation in pipelined router forwarding engines. Theory of Computing Systems **39**(6), 829–849 (2006)
4. Epstein, L., Levin, A., van Stee, R.: Approximation schemes for packing splittable items with cardinality constraints. Algorithmica **62**(1-2), 102–129 (2012). DOI 10.1007/s00453-010-9445-6. URL `http://dx.doi.org/10.1007/s00453-010-9445-6`
5. Epstein, L., van Stee, R.: Improved results for a memory allocation problem. Theory Comput. Syst. **48**(1), 79–92 (2011). DOI 10.1007/s00224-009-9226-2. URL `http://dx.doi.org/10.1007/s00224-009-9226-2`
6. Janiak, A., Janiak, W., Lichtenstein, M.: Resource management in machine scheduling problems: A survey. Decision Making in Manufacturing and Services **1**(12), 59–89 (2007)
7. Józefowska, J., Mika, M., Różycki, R., Waligóra, G., Weglarz, J.: Discrete-continuous scheduling to minimize the makespan for power processing rates of jobs. Discrete Applied Mathematics **94**(1), 263–285 (1999)
8. Józefowska, J., Mika, M., Różycki, R., Waligóra, G., Weglarz, J.: Solving the discrete-continuous project scheduling problem via its discretization. Mathematical Methods of Operations Research **52**(3), 489–499 (2000)
9. Józefowska, J., Mika, M., Różycki, R., Waligóra, G., Weglarz, J.: A heuristic approach to allocating the continuous resource in discrete-continuous scheduling problems to minimize the makespan. Journal of Scheduling **5**(6), 487–499 (2002)
10. Józefowska, J., Weglarz, J.: Discrete-continuous scheduling problems – mean completion time results. European Journal of Operational Research **94**(2), 302–309 (1996)
11. Józefowska, J., Weglarz, J.: On a methodology for discrete-continuous scheduling. European Journal of Operational Research **107**(2), 338–353 (1998)
12. Kis, T.: A branch-and-cut algorithm for scheduling of projects with variable-intensity activities. Mathematical Programming **103**(3), 515–539 (2005)
13. Leung, J.Y.T.: Handbook of Scheduling: Algorithms, Models, and Performance Analysis. Chapman & Hall/CRC (2004)
14. Różycki, R., Weglarz, J.: Power-aware acheduling of preemptable jobs on identical parallel processors to minimize makespan. Annals of Operations Research pp. 1–18 (2011). DOI 10.1007/s10479-011-0957-5
15. Różycki, R., Weglarz, J.: Power-aware scheduling of preemptable jobs on identical parallel processors to meet deadlines. European Journal of Operational Research **218**(1), 68–75 (2012). DOI 10.1016/j.ejor.2011.10.017
16. Waligóra, G.: Heuristic approaches to discrete-continuous project scheduling problems to minimize the makespan. Computational Optimization and Applications **48**(2), 399–421 (2011)
17. Weglarz, J., Józefowska, J., Mika, M., Waligóra, G.: Project scheduling with finite or infinite number of activity processing modes – a survey. European Journal of Operational Research **208**(3), 177–205 (2011)
18. Yao, F.F., Demers, A.J., Shenker, S.: A scheduling model for reduced cpu energy. In: Proceedings of the 36th Annual Symposium on Foundations of Computer Science (FOCS), pp. 374–382 (1995)
19. Zhuravlev, S., Saez, J.C., Blagodurov, S., Fedorova, A., Prieto, M.: Survey of scheduling techniques for addressing shared resources in multicore processors. ACM Computing Surveys (CSUR) **45**(1), 4:1–4:28 (2012). DOI 10.1145/2379776.2379780. URL `http://doi.acm.org/10.1145/2379776.2379780`