PADERBORN UNIVERSITY

# On Cloud Assisted, Restricted, and Resource Constrained Scheduling

Dissertation

In partial fulfillment of the requirements for the degree of

Doctor rerum naturalium (Dr. rer. nat.)

at the Faculty of Computer Science,
Electrical Engineering and Mathematics
at Paderborn University

submitted by

SIMON PUKROP

**Reviewers**

- Prof. Dr. Friedhelm Meyer auf der Heide,
  Paderborn University

- Prof. Dr. Klaus Jansen,
  Kiel University

*Ja, mach nur einen Plan!*
*Sei nur ein großes Licht!*
*Und mach dann noch'nen zweiten Plan*
*Gehn tun sie beide nicht.*

Bertolt Brecht
*Lied von der Unzulänglichkeit menschlichen Strebens*

# Zusammenfassung

Scheduling ist eines der grundlegendsten Probleme in der Informatik. Eine Menge von Jobs muss Maschinen zugewiesen werden, um eine gegebene Gütefunktion, oft die Gesamtlaufzeit des resultierenden Schedules, zu optimieren. In dieser Arbeit untersuchen wir drei verschiedene Scheduling-Problemfamilien, diskutieren ihre komplexitätstheoretische Härte und zeigen, wie verschiedene algorithmische Ansätze für diese Probleme geeignet sind.

Zuerst stellen wir ein neues Problem vor, das wir entwickelt haben, um die Interaktion zwischen einem kleineren Rechner und mietbarer Cloud-Rechenleistung zu modellieren, motiviert durch den wachsenden Trend des Cloud Computing. In diesem Modell, das wir *Server Cloud Scheduling* nennen, werden große Abläufe als eine Sammlung von kleineren Jobs mit bestimmten Abhängigkeiten zwischen ihnen gegeben. Die Jobs können auf einem Server ausgeführt werden, der auf sequentielle Ausführung beschränkt, aber kostenlos ist, oder in die Cloud ausgelagert werden, die unendliche Parallelität erlaubt, aber Kosten verursacht. Das Ziel ist entweder die Minimierung der Gesamtlaufzeit unter Einhaltung eines vorgegebenen Budgets oder die Minimierung der Kosten unter Einhaltung einer vorgegebenen Frist. Unsere Ergebnisse beinhalten unter anderem zwei FPTAS (Fully Polynomial Approximation Scheme), die auf dynamischer Programmierung basieren, für recht allgemeine Spezialfälle des Modells, und ein starkes NP-härte Ergebnis, das sogar dann gilt, wenn alle Joblängen gleich 1 sind.

Zweitens untersuchen wir Probleme aus der Familie des *Unrelated Machine Scheduling*. Insbesondere betrachten wir das *Restricted Assignment Interval* Problem, bei dem die Maschinen in einer bestimmten Reihenfolge gegeben sind und jeder Job nur auf einer konsekutiven Teilmenge der Maschinen ausgeführt werden kann. Ziel ist dabei die Minimierung der Gesamtlaufzeit. Obwohl es sich um einen eingeschränkten Spezialfall des Unrelated Machine Scheduling handelt, waren die besten bekannten algorithmischen Ergebnisse für beide Probleme im Wesentlichen von gleicher Qualität. Es ist uns gelungen, diese Ergebnisse zu übertreffen und den ersten Approximationsalgorithmus mit einem konstanten Approximationsfaktor kleiner als 2 für das Restricted Assignment Interval Problem zu liefern. Der Algorithmus erweitert frühere Ansätze der linearen Programmierung.

Zuletzt betrachten wir *Many Shared Resources Scheduling*. Zusätzlich zu den Maschinen und Jobs erhalten wir eine Menge von Ressourcen. Jeder Job kann nun während seiner Laufzeit Zugriff auf eine der Ressourcen anfordern, und Jobs, die dieselbe Ressource benötigen, können nicht parallel ausgeführt werden. Das Ziel ist erneut, die Gesamtlaufzeit zu minimieren. Wir stellen zwei kombinatorische Approximationsalgorithmen vor, eine einfache und elegante $5/3$-Approximation und eine technisch aufwendigere $3/2$-Approximation. Unser erster Algorithmus schlägt bereits den bisherigen Stand der Technik in Form einer 2-Approximation. Wir schließen mit einigen Inapproximierbarkeitsergebnissen für eine Generalisierung des Problems, bei der Jobs mehr als eine Ressource benötigen können.

# Abstract

Scheduling is one of the most fundamental problems in computer science. A set of jobs has to be assigned to a set of machines in order to optimize some utility function, often the makespan of the resulting schedule. In this thesis, we study three different families of scheduling problems, discuss their complexity theoretical hardness and show how the problems lend themselves to various algorithmic approaches.

First, we introduce a new problem that we created to model interactions between smaller computing devices and rentable cloud computing power, following the growing trend of cloud computing. In this model, called *server cloud scheduling*, large tasks are given as a collection of smaller jobs with some precedence relation between themselves. The jobs can be processed on a purely sequential but free server or offloaded to the cloud, which allows infinite parallelization but incurs costs. The objective is either to minimize the makespan while adhering to a budget or to minimize the cost while adhering to a deadline. Some of our main results are two dynamic programming based FPTAS (fully polynomial approximation scheme) for two fairly general special cases of the model; and a strong NP-hardness result that holds even when all processing times are equal to 1.

Second, we study problems in the family of *unrelated machine scheduling*. Specifically, we consider the *restricted assignment interval* problem, where the machines are given in some order, and each job is only eligible on a consecutive subset of the machines. The objective is to minimize the makespan. Although this can be modeled as a quite limited special case of unrelated scheduling, the best-known algorithmic results for both were essentially of the same quality. We managed to beat these results and give the first approximation algorithm with a constant approximation ratio less than 2 for restricted assignment interval. The algorithm refines and expands previous linear programming approaches.

Last, we consider *many shared resources scheduling*. In addition to the machines and jobs, we are also given a set of resources. Each job may now require exclusive access to one of the resources during its processing time, or in other words, two jobs needing the same resource may not be scheduled in parallel. The goal is again to minimize the makespan. We present two combinatorial approximation algorithms, a simple and elegant $5/3$-approximation and a technically more involved $3/2$-approximation. Both of these beat the previous best-known algorithm, which was a 2-approximation. We conclude with some inapproximability results for a generalization of the problem, where jobs may require more than one resource.

# Acknowledgments

I would like to sincerely thank my advisor Friedhelm Meyer auf der Heide. Both for his support and advice on this thesis and for giving me the opportunity to work in his group for over three and a half years with my lovely (ex-)colleagues: Marten Maack, Jannik Castenow, Till Knollmann, Jonas Harbig, Matthias Fischer, Christian Soltenborn, André Graute, Gleb Polevoy, Alex Mäcker, Sascha Brandt, Manuel Malatyali and Björn Feldkord. I had a brilliant time working, traveling, laughing, and arguing with all of you.

I would like to extend my thanks to all my co-authors, namely, Friedhelm Meyer auf der Heide, Marten Maack, Anna Rodriguez Rasmussen, Klaus Jansen, Max Deppert, and Malin Rau. I am proud of the work we have done together! A special thanks to Marten Maack and Till Knollmann for proofreading parts of this thesis.

Last but not least, I would like to thank my friends, my mother Gisa, my brother Thomas, my sister Sarah and especially my soon-to-be wife Maren for supporting me through my many ups and downs and for being the marvelous people I have the fortune to be around.

*Simon Pukrop*
*March 2023*

# Contents

# 1. Introduction

Scheduling is one of the foundational problems in computer science. At its core, scheduling is about allocating finite resources to optimize for a chosen quality. A simple example is distributing the processing of tasks onto a set of machines such that the last task is done as fast as possible. With the steady increase in complexity in computing systems, good scheduling algorithms are crucial for the efficiency and performance of modern systems. Inevitably, starting in the mid-1950s, there has been ever-growing attention to scheduling problems in research and development, both from a theoretical and a practical perspective. This thesis is firmly located on the theoretical side of the research spectrum. It seeks to deepen the understanding of classical scheduling problems and to establish and explore new models that apply to different properties of real-world applications.

## 1.1 What is Scheduling, and Why Bother?

Scheduling problems are a class of optimization problems that involve allocating resources, such as time or machines, to tasks efficiently and effectively. These problems are prevalent in computer science but also in many other industries, including manufacturing, transportation, healthcare, and logistics. The impact of optimized scheduling on overall performance and productivity in each of those areas can be immense; therefore, its importance is hard to overstate.

Naturally, there are a lot of motivating examples to be found in computer science. One can think about resources like CPU time, memory, and communication channels assigned to various processes to minimize a chosen objective. Examples of such an objective include the overall running time, energy consumption, and the maximum starvation of a single task. The increase in complexity in computing systems poses new challenges for the development of scheduling approaches to guarantee efficient and performant algorithms.

In manufacturing, scheduling problems involve determining the most efficient order to produce products using limited machines and resources. For example, a factory may have several production lines that can produce different products, but the factory only has a limited number of machines and workers. In this case, the scheduling problem might involve determining the optimal production order to maximize efficiency while minimizing working hours. In transportation, a company may have a fleet of trains that need to make

deliveries to different locations. The scheduling problem entails determining the most efficient train routes to minimize travel time and cost while considering the rail network's congestion. In health care, scheduling problems involve allocating scarce resources, such as operating rooms and medical staff, to patients who need surgery or other procedures. A hospital may have limited operating rooms and surgical teams, but many patients require surgery at the same time. The scheduling problem might be finding the optimal surgery schedule to minimize patient waiting time and maximize resource utilization. Efficient scheduling can lead to significant cost savings, improved productivity, and better utilization of resources. However, finding the optimal schedule can be a complex and challenging task due to the many variables and constraints involved. The number of jobs, available resources, time frames, and specific requirements of each task, as well as the various constraints such as resource availability, time constraints, and deadlines, often make it a computationally hard problem. As a result, scheduling problems pose major challenges in both computer science and operations research.

Models completely faithful to reality are often both too vague and too complex to be studied from the theoretical side. Therefore, we will abstract problems to a certain degree for the analysis, which could look like the following: Imagine a problem where jobs have to be scheduled, one after another, on a single machine. Each job has a certain processing time, a due date, and a weight (or penalty) that has to be paid when the job does not finish in time. This problem is called the *single machine weighted number of tardy jobs* problem. For such abstracted problems, research is done on the constructive side, developing new algorithms and techniques to solve these problems, and on the complexity theoretical side, proving bounds on the capabilities of algorithms. Various techniques for scheduling problems have been developed for the constructive side, including heuristics, (mixed-)integer programming, artificial intelligence, dynamic programming, linear programming rounding, and combinatorial approaches. These techniques have been applied to many scheduling problems to efficiently find optimal or near-optimal solutions and in this thesis, we will focus on the last three mentioned approaches. The complexity theoretical research is mostly done via reductions, which can be used to show that some problem is computationally at least as hard as another problem. In the following, we introduce the formal concepts of problems (languages) and computational hardness as a foundation to describe how to mathematically abstract scheduling problems.

## 1.2 On Problems, Languages, and Efficient Encodings

We give a short introduction to problems and languages, roughly following the discussion at the beginning of an influential paper by Garey and Johnson from 1978 [30]. In theoretical computer science, we depend on precise definitions and formal arguments. A foundation for this is the notion of problems, problem instances, and languages.

A problem contains a model definition together with a question. For example, the partition problem is defined as a set of values $A \in \mathbb{N}^*$ and the question "is there a partition of $A$ into $A_1$ and $A_2$, with $A_1 \cup A_2 = A$ and $A_1 \cap A_2 = \emptyset$, such that $\sum_{a \in A_1} a = \sum_{a' \in A_2} a'$ ?". A problem instance is a concrete instance of the model at hand, in this case, a set of numbers. We divide problems into decision problems and optimization problems. The former contain a yes or no question, as in the partition problem example above, and the latter an optimization question like "what schedule finishes processing all jobs the earliest?".

A decision problem can also be stated as a formal language. A language is a (finite or infinite) set of words over some finite alphabet; formally, some language $L$ over alphabet $\Sigma$ is defined as $L \subseteq \Sigma^*$. The Kleene star operator denotes the set of all possible concatenations of any number of elements in $\Sigma$, e.g. $\{a,b\}^* = \{\varepsilon, a, b, aa, ab, ba, bb, aaa, \dots\}$, where $\varepsilon$ denotes an empty string. A decision problem can be represented by a formal language $L$, where a word $w$ is part of the language $L$ if the problem instance encoded by $w$ is a "yes"-instance.

When speaking about the efficiency of algorithms, we want to state the runtime of an algorithm in relation to the *input size* (or problem's *instance size*). This is where a problem arises: A problem definition is, in a way, encoding agnostic. Take, for example, the partition problem mentioned above and an instance with numbers 1, 5, and 8. We can represent that as $1;11111;11111111$ (unary) or $1;101;1000$ (binary), resulting in a representation length of $\sum a_i = \mathcal{O}(|A| \max a_i)$ and $\sum \log a_i = \mathcal{O}(|A| \log(\max a_i))$ respectively. An algorithm that is allowed runtime linear in the input size could have exponentially more time using the former representation than with the latter. However, for a decision problem that is given as a language that would be well defined, as a word from a language is some specific character sequence and therefore has some specific length.

Following usual conventions, we will, in slight abuse of formal notation, consider problems like languages with some unspecified *efficient* encoding. What exactly an efficient encoding entails remains a bit vague, but the usual requirement is that numbers are encoded in a way that they need logarithmic space (e.g., binary). However, there are instances where it is interesting to look at an inefficient encoding, which we will describe in the following section.

## 1.3 P vs. NP and NP-Hardness

The famous *P vs. NP-Problem*, which is nowadays widely recognized as the most important open question in (theoretical) computer science, was introduced by Cook in 1971 in Canada [16] and independently by Levin in Russia in 1973 [61]. However, the real implications of the problem classes only became obvious after Karp's "Reducibility Among Combinatorial Problems" in 1972 [46]. P and NP are defined as follows:

**Definition 1.1** A decision problem (language) $L$ is in P, if and only if there is a deterministic algorithm that can decide if $I \in L$ for any instance $I$ in time $poly(|I|)$.

**Definition 1.2** A decision problem (language) $L$ is in NP if and only if there is a non-deterministic algorithm that can decide if $I \in L$ for any instance $I$ in time $poly(|I|)$.

Another way to define NP is: given a solution, can we verify the correctness of the solution in polynomial time? $|I|$ describes the size of the problem instance, which is inferred by some *efficient* encoding, as described in the previous section. The *P vs. NP-Problem* is the open question: is P = NP or P $\neq$ NP? For the rest of this thesis, we will assume that P $\neq$ NP (and therefore P $\subset$ NP), as is usual in complexity theoretical research. We want to distinguish the complexity of problems into "solvable in polynomial time" (in P) and "at least as hard as every other problem in NP" (NP-hard) The notion of NP-hardness is already apparent in the paper by Karp from 1972 [46] though the term NP-hardness was coined by Knuth in 1974 [50]:

**Definition 1.3** A decision problem (language) $L$ is NP-hard, if and only if for every decision problem (language) $L' \in$ NP there is a polynomial reduction such that $L' \leq_p L$. A problem that is both in NP and NP-hard is called NP-complete.

In other words, an NP-hard problem $L$ is at least as hard as every other problem in NP. A polynomial reduction $L' \leq_p L$ means that there is a polynomial function $f$ that converts an instance $x$ to $f(x)$, such that $x \in L' \Leftrightarrow f(x) \in L$. Consequently, if we can solve $L$, we can also solve $L'$ by transforming it into an instance of $L$.

Following our earlier discussion about encoding efficiency, we can make an even finer distinction on the complexity of problems. This was formalized by Garey and Johnson in 1978 [30]. Assume an NP-hard Problem, but its (numerical) values are given in an *inefficient* encoding (unary). If the problem is now solvable in time polynomial to its new encoding size, we call it *weakly* NP-hard. If the unary version remains NP-hard, we call the problem *strongly* NP-hard.

**Definition 1.4** A decision problem (language) $L$ is *weakly* NP-hard, if and only if it is NP-hard, but the problem in unary encoding is in P.

Directly implied by this is the existence of a pseudo-polynomial algorithm, an algorithm that has a polynomial runtime if the numerical values of the input are bounded polynomially in the input representation length.

**Definition 1.5** A decision problem (language) $L$ is *strongly* NP-hard if the problem in unary encoding is still NP-hard.

A strongly NP-hard problem can have no pseudo-polynomial algorithm, assuming that P $\neq$ NP. This also helps in identifying generalizations or special cases that impact the hardness of a problem: The *single machine weighted number of tardy jobs* problem, mentioned in Section 1.1, is weakly NP-hard, though if we add release times to the jobs, the resulting problem turns strongly NP-hard. Two other examples of weakly and strongly NP-hard problems are the aforementioned partition problem, and the boolean satisfiability problem SAT, respectively.

Technically, the problem classes P and NP are defined on languages or decision problems; an instance is either part of the problem (language) or not. We are mostly interested in optimization problems, but we can still use the above definitions. Instead of asking, "what is the shortest schedule for these jobs?" we can ask the yes/no question, "is there a schedule of length at most $d$?" for some given value $d$. Naturally, if one can find an optimal solution, one can also answer the question if there exists a solution of at least a certain quality. Since finding an optimal solution is at least as hard as answering the decision problem, we can show that the former is hard by proving the hardness of the latter.

## 1.4 Approximation Algorithms and Schemes

As discussed earlier, we assume that P $\neq$ NP and, consequently, that NP-hard problems cannot be solved in polynomial time. Consequently, we are interested in polynomial algorithms that, while not necessarily giving optimal solutions, provide *good enough* solutions. More precisely, we want to bound the ratio between optimal solutions and those given by our algorithms.

Consider a minimization problem and an algorithm $ALG$. For some problem instance $I$ we denote by $ALG(I)$ the objective value of the solution produced by $ALG$; on the other hand, $OPT(I)$ denotes the objective value of an optimal solution for $I$.

**Definition 1.6** *ALG* $\alpha$-approximates, or is called an $\alpha$-approximation, for a minimization problem if for every instance $I$: $ALG(I) \leq \alpha OPT(I)$. If not specified otherwise, an approximation algorithm is required to adhere to a polynomial runtime.

Note here that we only give the definition of approximation algorithms for minimization problems since none of the problems considered in this thesis are maximization problems.

**R** We define $\alpha$-approximation algorithms by producing solutions that have *at most* value of $\alpha OPT(I)$. Consequently, an $\alpha$-approximation algorithm is also a $\beta$-approximation algorithm for every $\beta \geq \alpha$. If there exists a problem instance $I$ such that $ALG(I) = \alpha OPT(I)$ (or an instance that converges towards equality), we call the approximation factor $\alpha$ *tight*. For complex algorithms, it is often difficult to prove a tight approximation factor; therefore, some algorithms might be better than our given approximation factor suggests.

It is possible to design approximation schemes with scaleable approximation quality. There are different qualities of approximation schemes, distinguished by the impact the scaling parameter has on the runtime.

**Definition 1.7** $ALG_\varepsilon$ is a polynomial time approximation scheme (PTAS) if it produces a polynomial time $(1+\varepsilon)$-approximation for every fixed $\varepsilon > 0$. Based on how runtime depends on $\varepsilon$, we further distinguish between the following:

- E(fficient)PTAS: the runtime is in $\mathcal{O}(\text{poly}(|I|) \cdot f(1/\varepsilon))$ for any $f : \mathbb{R} \mapsto \mathbb{R}$
- F(ully)PTAS: the runtime is in $\mathcal{O}(\text{poly}(|I|, 1/\varepsilon))$

Consequently, FPTAS $\subset$ EPTAS $\subset$ PTAS.

**R** Note here that a strongly NP-hard problem can not have an FPTAS if it has a polynomially bounded objective function [84] (which is the case for all problems discussed in this thesis), assuming that P $\neq$ NP.

## 1.5 Formal Scheduling Notation

Though we leave the concrete formal description of the models discussed in this thesis to the individual chapters in which they are used, we give a small rundown on formally notating scheduling problems. Like motivating examples for scheduling problems, there are uncountably many variations on theoretical models.

In general, a scheduling problem consists of a set of machines $\mathcal{M} = \{M_1, M_2, \ldots, M_m\}$ and a set of jobs $\mathcal{J} = \{j_1, j_2, \ldots, j_n\}$. A job ($j \in \mathcal{J}$) can be processed on any machine ($i \in \mathcal{M}$) and needs (uninterrupted) time of $p_i(j)$. Each machine can only process one job in parallel. The most common objective is then to create a schedule, a mapping of jobs to specific time slots and machines, such that all jobs are finished as soon as possible. In scenarios with more than one machine (which is the case in all models discussed in this thesis), one can distinguish between different machine relations:

- Identical machines: $p_i(j)$ depends only on the job, not on the machine. In that case, we simply write $p_j$ for readability.
- Related machines: Each machine has a speed value $s_i$, and $p_i(j) = p_j/s_i$
- Unrelated machines: $p_i(j)$ is any function $\mathcal{J} \mapsto \mathbb{N}$ or $\mathcal{J} \mapsto \mathbb{R}_+$

Additional requirements might be imposed on the problem. For example, jobs can have precedence relations (as in Chapter 2: Server Cloud Scheduling), only be eligible on specific machines (as in Chapter 3: Restricted Assignment with Interval Restrictions), or need additional resources (as in Chapter 4: Scheduling with Many Shared Resources). We leave the specifics to the corresponding model sections.

## 1.6 Contents of this Thesis

With scheduling being the vast topic it is, algorithmic approaches to approximate different problems can be equally as varied. In the following three chapters, we each discuss a particular scheduling problem, its complexity theoretical hardness, and an algorithmic approach that is suitable for the problem at hand. We start with our original model *server cloud scheduling* and dynamic programming approaches, continue with *restricted assignment interval* and linear programming solutions, and end with *many shared resources scheduling* and combinatorial algorithms. In addition to the concrete results, this also functions as an exemplary overview of techniques commonly useable for scheduling problems. We give an overview of the following content.

**Chapter 2: Server Cloud Scheduling** Motivated by the practice of offloading big computation tasks to large-scale cloud providers, we present our original model *server cloud scheduling*. Consider a scenario where a customer owns a small local computation unit (the server) and has access to an unlimited number of rentable machines (the cloud). Now, that customer has to process a large task, representing a collection of smaller jobs in some precedence relation. More formally, the set of jobs forms a directed acyclic task graph with a fixed source and sink. The edges of this graph model precedence constraints, and the jobs have to be scheduled with respect to those. Processing times on the server and in the cloud are given for each job. For each edge in the task graph, a communication delay is included in the input and has to be taken into account if one of the two jobs is scheduled on the server and the other in the cloud. The server processes jobs sequentially, whereas the cloud can serve as many as needed in parallel but induces costs. We consider both makespan minimization regarding a budget, as well as cost minimization regarding a deadline.

We start by looking at simple instances of the problem where the task graph is either fully sequential (a chain) or fully parallel. Both cases are already weakly NP-hard, even if there are no communication delays. On the constructive side, we give simple FPTAS results for both types of graphs (and both objectives). We continue with a further generalizing combination of fully sequential and fully parallel task graphs, which we call *extended chains*. We prove that our model with extended chain task graphs is strongly NP-hard. Constructively, we give a $(2+\varepsilon)$-approximation algorithm for the budget-constrained makespan minimization on extended chains and discuss special cases where that algorithm yields an FPTAS. Next, we look at graphs with a constant maximum cardinality source and sink dividing cut. Imagine the task graph during a schedule and divide the jobs into already finished and still to be processed. We count the number of edges in the task graph going from jobs in the former set to jobs in the latter. We look at instances where this number of crossing edges is bounded by some constant for every possible intermediate schedule. Intuitively, this means that only a constant number of already processed jobs can affect the remaining schedule via their communication delays. For those graphs, we are able to give an FPTAS with regard to the budget-constrained makespan minimization. Both more

involved approximation algorithms just mentioned are built upon dynamic programming approaches and can be seen as one of the chapter's main constructive contributions.

After that, we concentrate on stronger hardness results, which form the other main results of the chapter. We start with the unit case (all processing times and delays equal 1) of the scheduling problem with arbitrary task graphs. We give a reduction from 3-SAT, which proves that this case is strongly NP-hard. Modifying this approach for the general case of the problem, we can change the resulting scheduling instance so that, for a satisfying assignment, no job has to spend processing time on the cloud, which in turn means that there will be no costs. However, a schedule that corresponds to an unsatisfying assignment has to place jobs on the cloud to meet the deadline. Resulting from that, we get that there is no (fixed quality) approximation algorithm for the deadline-constrained cost minimization problem since we could otherwise decide 3-SAT.

We finish the chapter with some interesting smaller results. First, we give two approximation results for the unit case and a simple approximation for instances with no communication delays. After that, we discuss generalizations of the server cloud scheduling problem and how those can be incorporated into the two main approximation results of the chapter. Examples of generalizations are multiple server machines, multiple cloud providers, and a different cost model. Finally, we discuss a way to imagine the problem as a two-parameter optimization problem instead of using one as the objective and the other as a constraint. We modify and reuse some of our earlier results to approximate a Pareto-front for this resulting problem. Roughly speaking, the Pareto-front is the set of all problem solutions where an improvement in one optimization parameter would necessarily be a deterioration in the other.

This chapter is based on our submitted journal paper *Server Cloud Scheduling* [63], which is in turn based on our conference paper of the same name [64], which was published in the proceedings of WAOA 2021.

**Chapter 3: Restricted Assignment Interval** In Section 1.5, we briefly discussed different machine models. Of the presented ones, the most general is the unrelated machines model. Consider a simple version of that model: each job $j$ has some value $p_j$, and for every machine $M_i$, the processing time of $j$ is either $p_j$ or $\infty$. In other words, each job has some fixed processing time but is only eligible on a specific subset of machines. This case is also called *restricted assignment*. In Chapter 3, we mainly discuss a seemingly simple variant of restricted assignment, called *restricted assignment interval* (RAI). Here, the machines $\mathcal{M}$ are given in some specific order, and every job is eligible on some consecutive interval of the machines. For the general unrelated machines problem, both a 2-approximation and a 1.5-inapproximability result are known [57]. Naturally, two questions arise. First, since constructive results for the general case also hold for the special case, is there an approximation algorithm with a ratio better than 2 for RAI? Second, are there also inapproximability results for some of the (other) subproblems of restricted assignment? The first of the two questions was posed as an open challenge in previous works [41, 79, 85] as early as 2010.

We were able to answer that question in the affirmative and present a $(2-\frac{1}{24})$-approximation. We give a linear programming (LP) formulation for the problem and use a three-stage LP-rounding procedure, placing very large jobs first, then large jobs, and finally, the remaining small jobs. We could also answer the second question for some subproblems. We give the definition of those subproblems and a summary of the results to better understand how the constructive result relates to present lower bounds.

This chapter is based on our paper *(In-)Approximability Results for Interval, Resource Restricted, and Low Rank Scheduling* [67] which was published in the proceedings of ESA 2022.

**Chapter 4: Many Shared Resources** In the last chapter of this thesis, we look at a scheduling extension where jobs need additional resources. This model is called *many shared resources scheduling* or MSRS. The general model includes a set of identical machines, a set of jobs, and a set of resources (with a type and how many are available). Each job then also has resource requirements, meaning how much of which resource(s) needs to be assigned to the job so that it can process. Resources are not consumed and only blocked for as long as the job is processing.

We mostly consider a variant where each resource is available exactly once, and each job needs exclusive access to exactly one resource. In other words, we can separate the jobs into distinct classes where at any time, only one job per class can be processed in parallel. We present two combinatorial approximation algorithms that cleverly schedule classes of jobs of specific sizes. More specifically, we start with a short $5/3$-approximation, which works in three steps. The algorithm gives a good intuition that we have to focus on how to handle big jobs or classes that are large in overall processing time. Notably, the algorithm already beats the previously best-known result. The second algorithm is a $3/2$-approximation that is much more involved. We handle two cases separately. First, we give an algorithm that only works if no jobs have a processing time of more than $3/4$ of the overall makespan. Second, we give an algorithm for the general case, which first tries to place all the jobs with a processing time of more than $3/4$ of the makespan, and then applies the first algorithm.

In the original paper, we did some work on approximation schemes for the problem at hand. We give a short summary of the approaches and the results they yielded.

Lastly, we show some inapproximability results for cases where each job may need more than a single resource, though all different resources are still available only once. To be more specific, we prove a $5/4$-inapproximability for the case where each job may need up to 3 resources and all jobs have processing time 1, 2, or 3. A slight variation gives a $5/4$-inapproximability for the case where each job may need up to 5 resources, but all jobs have a processing time of 1. Finally, another reduction gives a $4/3$-inapproximability for the case where each job may need any number of resources, but all jobs again have a processing time of 1.

This chapter is based on our paper *Scheduling with Many Shared Resources* [21] which was accepted for publication in the proceedings of IPDPS 2023

# 2. Server Cloud Scheduling

We introduce a new scheduling problem, called *server cloud scheduling*, motivated by current trends in cloud computing. In this model large tasks are given as a collection of smaller jobs with some precedence relation between themselves. Jobs can be processed on a strictly sequential, but free, server or offloaded to the cloud, which allows infinite parallelization but incurs costs. Among other results, we study the model's complexity and show how dynamic programming approaches can be used to approximate it.

This chapter is based on our submitted journal paper *Server Cloud Scheduling* [63], which is in turn based on our conference paper of the same name [64], which was published in the proceedings of WAOA 2021. The new sections introduced in the journal version are Section 2.3, Section 2.7 and Section 2.8. The short Section 2.6.1 contains my refinement of a result based on the bachelor thesis of Bastian Franke. I supervised the thesis and recommended promising approaches.

## 2.1 Introduction

Scheduling with precedence constraints with the goal of makespan minimization is widely considered a fundamental problem. It has already been studied in the 1960s by Graham [33] and receives a lot of research attention up to this day (see e.g. [31, 51, 60]). One problem variant that has received particular attention recently, is the variant with communication delays (e.g. [19, 20, 51]). Another, more contemporary topic concerns scheduling using external resources like, for instance, machines from the cloud. Several models in this context have been considered of late (e.g. [1, 68, 76]). In this chapter, we introduce and study a model closely connected to both settings, where jobs with precedence constraints may either be processed on a single server machine or on one of many cloud machines. Here, communication delays may occur only if the computational setting is changed. The server and cloud machines may behave heterogeneously, i.e., jobs may have different processing times on the server and in the cloud, and scheduling in the cloud incurs costs proportional to the computational load performed in the cloud. Both makespan and cost minimization is considered. We believe that the present model provides a useful link between scheduling with precedence constraints and communication delays on the one hand and cloud scheduling on the other.

### 2.1.1 Problem Definition

We consider a scheduling problem *SCS* where a task graph $G = (\mathcal{J}, E)$ has to be scheduled on a combination of a local machine (server) and a limitless number of remote machines (cloud). The task graph is a directed, acyclic graph with exactly one source $\mathcal{S} \in \mathcal{J}$ and exactly one sink $\mathcal{T} \in \mathcal{J}$. Each job $j \in \mathcal{J}$ has a processing time on the server $p_s(j)$ and on the cloud $p_c(j)$. We consider $p_s(\mathcal{S}) = p_s(\mathcal{T}) = 0$ and $p_c(\mathcal{S}) = p_c(\mathcal{T}) = \infty$, i.e., the first and the last job must be processed on the server. For every other job, the values of $p_s$ and $p_c$ can be arbitrary in $\mathbb{N}_0$, meaning that the server and the cloud are unrelated machines in our default model. An edge $e = (i, j)$ denotes precedence, i.e., job $i$ has to be fully processed before job $j$ can start. Furthermore, an edge $e = (i, j) \in E$ has a communication delay of $c(i, j) \in \mathbb{N}_0$, which means that after job $i$ finished, $j$ has to wait for an additional $c(i, j)$ timesteps before it can start if $i$ and $j$ are not both scheduled on the same type of machine (server or cloud).

A schedule $\pi$ is given as a tuple $(\mathcal{J}^s, \mathcal{J}^c, C)$. $\mathcal{J}^s$ and $\mathcal{J}^c$ are a proper partition of $\mathcal{J}$: $\mathcal{J}^s \cap \mathcal{J}^c = \emptyset$ and $\mathcal{J}^s \cup \mathcal{J}^c = \mathcal{J}$. The sets $\mathcal{J}^s$ and $\mathcal{J}^c$ denote jobs that are processed on the server or cloud in $\pi$, respectively. Lastly, $C : \mathcal{J} \mapsto \mathbb{N}_0$ maps jobs to their completion time. We introduce some notation before we formally define the validity of a schedule. Let $p^\pi(j)$ be equal to $p_s(j)$ if $j \in \mathcal{J}^s$, and $p_c(j)$ if $j \in \mathcal{J}^s$. The value $p^\pi(j)$ denotes the actual processing time of job $j$ in $\pi$. Let $E^* := \{(i, j) \in E \mid (i \in \mathcal{J}^s \wedge j \in \mathcal{J}^c) \vee (i \in \mathcal{J}^c \wedge j \in \mathcal{J}^s)\}$ be the set of edges between jobs on different computational contexts (server or cloud). Intuitively, for all the edges in $E^*$ we have to take the communication delays into consideration, for all edges in $E \setminus E^*$ we only care about the precedence. We call a schedule $\pi$ valid if and only if the following conditions are met:

a) There is always at most one job processing on the server:
$\forall_{i \in \mathcal{J}^s} \forall_{j \in \mathcal{J}^s \setminus \{i\}} : (C(i) \leq C(j) - p^\pi(j)) \vee (C(i) - p^\pi(i) \geq C(j))$

b) Tasks are not started before the preceding tasks have been finished and the required communication is done:
$\forall_{(i,j) \in E \setminus E^*} : (C(i) \leq C(j) - p^\pi(j))$
$\forall_{(i,j) \in E^*} : (C(i) + c(i, j) \leq C(j) - p^\pi(j))$

The makespan (*mspan*) of a schedule is given by the completion time of the sink $C(\mathcal{T})$. The cost (*cost*) of a schedule is given by the time it spends processing tasks on the cloud: $\sum_{i \in \mathcal{J}^c} p^\pi(i)$. Note here, that by requiring $p_s(\mathcal{S}) = p_s(\mathcal{T}) = 0$ and $p_c(\mathcal{S}) = p_c(\mathcal{T}) = \infty$, we assume every job to start and end on the server. This is done only for convenience as it defines a clear start and end state for each schedule.

Naturally, two different optimization problems arise from the definition. First, given a deadline $d$, find a schedule with the lowest cost and $mspan = C(\mathcal{T}) \leq d$. Second, given a cost budget $b$, find a schedule with the smallest makespan and $cost = \sum_{i \in \mathcal{J}^c} p^\pi(i) \leq b$. In both instances the $d$, respectively the $b$, is strict. The natural decision variant is: given both $d$ and $b$ find a schedule that adheres to both if one exists.

**R** Instances of *SCS* might contain schedules with a makespan (and therefore cost) of 0. We can check for those in polynomial time: First, by removing all edges with communication delay 0, we get a set of connected components $K$. If and only if $\forall_{k \in K} \left(\forall_{j \in k}\ p_s(j) = 0\right) \vee \left(\forall_{j \in k}\ p_c(j) = 0\right)$, then there is a schedule with makespan of 0. For the rest of the chapter, we will assume that our algorithms check that beforehand and are only interested in schedules with $mspan > 0$.

### 2.1.2 Our Results

We start by establishing (weak) NP-hardness already for the case without communication delays and very simple task graphs. More precisely, when the task graph forms one chain starting with the source and ending with the sink (*chain case*) and when the graph is fully parallel, i.e., each job $j \in \mathcal{J} \setminus \{\mathcal{S}, \mathcal{T}\}$ is only preceded by the source and succeeded by the sink (*fully parallel case*). On the other hand, we establish FPTAS results for both the chain and fully parallel case with arbitrary communication delays and with respect to both objective functions. These results are discussed in Section 2.2.

In Section 2.3 we generalize the previous two task graph models (chain and fully parallel) into one, called extended chain graphs. Extended chain graphs are a chain, where any number of chain nodes can be replaced by a fully parallel graph. We prove that the resulting scheduling problem is now strongly NP-hard, by giving a reduction from the strongly NP-hard $1 \mid r_j \mid \sum w_j U_j$ problem [56]. We present a $(2+\varepsilon)$-approximation for the budget-constrained makespan minimization for this class of task graphs. We end the section, by discussing some small assumptions on the problem instance, which allow us to achieve FPTAS results instead.

Table 2.1: An overview of the results of this chapter.

| Algorithmic Results | |
|---|---|
| fully parallel or chain task graph | FPTAS w.r.t. cost and makespan |
| extended chain task graph | $(2+\varepsilon)$-approximation w.r.t. makespan |
| extended chain + additional assumptions | FPTAS w.r.t. makespan |
| extended chain task graph + generalizations | $(4+\varepsilon)$-approximation w.r.t. makespan |
| task graph with constant $\psi$ | FPTAS w.r.t. makespan |
| task graph with constant $\psi$ | $\alpha$-approximation of Pareto front, for any $\alpha > 0$ |
| task graph with constant $\psi$ + generalizations | FPTAS w.r.t. makespan |
| $c = 0$, $p_c = p_s$ (no delays, identical machines) | 2-approximation w.r.t. makespan |
| $c = 0$, $p_c = p_s = 1$ | polynomial w.r.t. makespan and cost |
| $c = p_c = p_s = 1$ (unit delays, unit sizes) | $\frac{1+\varepsilon}{2\varepsilon}$-approximation w.r.t. cost with makespan at most $(1+\varepsilon)d$, 3-approximation w.r.t makespan |
| **Hardness Results** | |
| fully parallel or chain task graph, $c = 0$ | (weakly) NP-hard |
| extended chain task graph | (strongly) NP-hard |
| $\forall j \in \mathcal{J} : c(j) = 0, p_c(j), p_s(j) \in \{1,2\}$ | (strongly) NP-hard |
| $c = p_c = p_s = 1$ (unit delays, unit sizes) | (strongly) NP-hard |
| general problem | no constant approximation w.r.t. cost |

In Section 2.4 we aim to generalize the previous FPTAS results regarding the makespan as much as possible. We show that an FPTAS can be achieved as long as the *maximum cardinality source and sink dividing cut* $\psi$ is constant. Intuitively, this parameter upper bounds the number of edges that have to be considered together in a dynamic program and in many relevant problem variants it can be bounded or replaced by the longest anti-chain length. We provide a formal definition in Section 2.4.

We turn our attention to strong NP-hardness results in Section 2.5. We show, that a classical reduction due to Lenstra and Rinnooy Kan [55] can be adapted to prove NP-hardness already for the variant of *SCS* without communication delays and processing times equal to one or two. *SCS* with unit processing times and without communication delays, on the other hand, can be trivially solved in polynomial time. Hence we are interested in the case with unit processing times and communication delays. We design an intricate reduction to show that this very basic case is NP-hard as well. Note that in this setting the server and the cloud machines are implicitly identical, except costs. Furthermore, we are able to show that a slight variation of the reduction implies that no constant approximation with respect to the cost objective can be achieved regarding the general problem.

In Section 2.6, we consider approximation algorithms for the case of unit processing times and delays. We show that a relatively simple approach yields a $\frac{1+\varepsilon}{2\varepsilon}$-approximation for $\varepsilon \in (0,1]$ regarding the cost objective if we allow a makespan of $(1+\varepsilon)d$, aswell as another approach that gives a 3-approximation for the makespan objective. Furthermore, we present a 2-approximation for the case without delays and identical server and cloud machines ($p_c = p_s$) but arbitrary task graphs and the makespan objective. Here, we also show that the respective algorithm can also be used to solve the problem optimally with respect to both objectives in the case of unit processing times. In Section 2.7, we establish some natural generalizations on the model and sketch how those can be solved by slight adaptations of our algorithms for extended chain and constant $\psi$ graphs. Lastly, in Section 2.8 we show how to give an $\alpha$-approximation, for any chosen $\alpha > 0$, on the pareto front of a problem with a task graph with constant $\psi$, when we look at the problem as a multi-objective optimization problem. This means, that for any point in the actual pareto front, we give a nearby feasible point that is only worse by a factor of $1+\alpha$ in both dimensions. In Table 2.1 we give an overview of the important results.

### 2.1.3 Related Work

Probably the closest related model to ours was studied by Aba et al. [1]. In that paper a problem instance is very similar to a problem instance of *SCS*, however, while we have a limited server and an unlimited cloud, they have an an unbounded number of machines in both and the goal is simply makespan minimization (as there are no costs). The authors show NP-hardness on the one hand and identify cases that can be solved in polynomial time on the other. In the conclusion of that paper, a model very similar to the one studied in this work is mentioned as an interesting research direction. For a further discussion of related models, we refer to the preprint version of the above work [1].

The present model is closely related to the classical problem of makespan minimization on parallel machines with precedence constraints, where a set of jobs with processing times, a precedence relation on the jobs (or a task graph), and a set of $m$ machines are given. The goal is to assign the jobs to starting times and machines such that the precedence constraints are met and the last job finishes as soon as possible. In the 1960s, Graham [33] introduced

the list scheduling heuristic for this problem and proved it to be a $(2-\frac{1}{m})$-approximation. Interestingly, to date, this is essentially the best result for the general problem. On the other hand, Lenstra and Rinnooy Kan [55] showed that no better than $\frac{4}{3}$-approximation can be achieved for the problem with unit processing times, unless P = NP. In more recent days, there has been a series of exciting new results for this problem starting with a paper by Svensson [82] who showed that no better than 2-approximation can be hoped for assuming a variant of the unique games conjecture. Furthermore, Levey and Rothvoss [60] presented an approximation scheme with nearly quasi-polynomial running time for the variant with unit processing times and a constant number of machines, and Garg [31] improved the running time to quasi-polynomial shortly thereafter. These results utilized so-called LP-hierarchies to strengthen linear programming relaxations of the problems. This basic approach has been further explored in a series of subsequent works (e.g. [19, 20, 51]), which in particular also investigate the problem variant where a communication delay is incurred for pairs of precedence-constrained jobs running on different machines. The latter problem variant is closely related to our setting as well.

Lastly, there is at least a conceptual relationship to problems where jobs are to be executed in the cloud. For example, a problem was considered by Saha [76] in which cloud machines have to be rented in fixed time blocks in order to schedule a set of jobs with release dates and deadlines minimizing the costs which are proportional to the rented time blocks. Another example is a work by Mäcker et al. [68] in which machines of different types can be rented from the cloud and machine-dependent setup times have to be paid before they can be used. Jobs arrive in an online fashion and the goal is again cost minimization. Both papers reference further work in this context.

## 2.2 Chains and Fully Parallel Task Graphs

In this section, we collect some first results that give an intuition concerning the complexity and approximability of our problem. In particular, we show weak NP-hardness already for cases with very simple task graphs and without communication delays. Furthermore, we discuss complementing FPTAS results for those cases and both objectives.

### 2.2.1 Hardness

We show that *SCS* is NP-hard even for two very simple types of task graphs and in a case where every communication delay is 0. For both of these reductions we use the decision variant of the problem: given a deadline $d$ and a budget $b$, find a schedule that satisfies both. Naturally, this shows the hardness of both the cost minimization as well as the makespan minimization problem.

**Chain Graph Case** We start by reducing the decision version of knapsack to *SCS* with a chain graph as its task graph. The knapsack problem is given as a capacity $C$, a value threshold $V$ and a set of items $\{1,\dots,n\}$ with weights $w_i$ and values $v_i$. The question is; does there exist a subset of items $S$ such that $\sum_{i\in S} w_i \leq C$ and $\sum_{i\in S} v_i \geq V$? We create the respective *SCS* problem as follows. For every item $i \in \{1,\dots,n\}$ create a task with $p_s(i) = w_i + v_i$ and $p_c(i) = v_i$. Consider a task graph with those tasks as a chain (in an arbitrary order) and each resulting edge $(i,j)$ has $c(i,j) = 0$. We set the deadline to $d = \sum_{1\leq i\leq n} v_i + C$ and the budget to $b = \sum_{1\leq i\leq n} v_i - V$. It is left to show, that there is a solution to the knapsack problem if and only if there is a schedule for our transformed

problem. We show that there is a one-to-one relation between our schedules and knapsack solutions. Assume there is some feasible solution (a subset of items $S$) for the knapsack problem with value $V'$. For each $i \in S$, we put the respective task in $\mathcal{J}^s$ and the rest in $\mathcal{J}^c$. Since the task graph is a chain we can compute a minimal makespan from this partition: $\sum_{1\leq i\leq n} v_i + \sum_{i\in S} w_i$ which is smaller or equal to $d$ if and only if $\sum_{i\in S} w_i \leq C$. The cost for the schedule is equal to $\sum_{1\leq i\leq n} v_i - V'$. Therefore, the cost for the schedule is smaller or equal to $b$ exactly when $V' \geq V$. It is easy to see that we can construct a knapsack solution from a schedule in a similar vein, therefore we conclude:

**Theorem 2.1** The *SCS* problem is weakly NP-hard for chain graphs and even without communication delays.

**Fully Parallel Case** Secondly, we look at problems with fully parallel task graphs, which means that every job $j$ besides $\mathcal{S}$ and $\mathcal{T}$ has exactly two edges: $(\mathcal{S}, j)$ and $(j, \mathcal{T})$. Here we do a simple partition reduction. Given a set $S$ of natural numbers, the question is, if there is a partition into sets $S_1$ and $S_2$ such that $\sum_{i\in S_1} i = \sum_{i\in S_2} i$. For every element $i$ in $S$ we create a task with $p_s(j) = p_c(j) = i$, set $d = b = \frac{1}{2}\sum_{i\in S_1} i$. We arrange the tasks into a fully parallel task graph where each edge $(i, j)$ has $c(i, j) = 0$. Imagine a solution $S_1$, $S_2$ for the partition problem. We schedule every task related to an integer in $S_1$ on the server and every other task on the cloud. Since everything is fully parallel and there are no communication delays we can conclude a makespan of $\max\{\sum_{i\in S_1} i, \max_{i\in S_2} i\}$ and costs of $\sum_{i\in S_2}$. This is a correct solution for the scheduling problem if and only if $\sum_{i\in S_1} i = \sum_{i\in S_2} i$. Again it is easy to see that an equivalent argument can be made for the other direction.

**Theorem 2.2** The *SCS* problem is weakly NP-hard for fully parallel graphs and even without communication delays.

### 2.2.2 Algorithms

In the following, we present complementing FPTAS results for the variants of *SCS* with fully parallel and chain task graphs, without any limitations on processing times or communication delays.

**Fully Parallel Case** We present how we can deal with the variant of *SCS* with a fully parallel task graph using by using well-known results and techniques. In particular, we can design two simple dynamic programs for the search version of the problem. We consider for each job the two possibilities of scheduling them on the cloud or on the server and compute for each possible budget or deadline the lowest makespan or cost, respectively, that can be achieved with the jobs considered so far. These dynamic programs can then be combined with suitable rounding procedures that reduce the number of considered states and search procedures for approximate values for the optimal cost or makespan, respectively, yielding:

**Theorem 2.3** There are FPTAS for *SCS* with fully parallel task graphs for the cost objective and for the makespan objective.

*Proof.* We start by designing the dynamic programs for the search version of the problem with budget $b$ and deadline $d$. Without loss of generality, we assume $\mathcal{J} = \{0,1,\ldots,n,n+1\}$ with $\mathcal{S} = 0$, $\mathcal{T} = n+1$ and set $c(j) = c(\mathcal{S},j) + c(j,\mathcal{T})$.

For each deadline $d' \in \{0,1,\ldots,d\}$ and $j \in \mathcal{J}$, we want to compute the smallest cost $C[j,d']$ of all the schedules of the jobs $0,1,\ldots,j$ adhering to the deadline $d'$ on the server ($j = 0$ denotes the trivial case that no job after the source has been scheduled). We initialize $C[0,d'] = 0$ for each $d'$. For all other jobs $j$ we consider the two possibilities of scheduling it on the cloud or server. In particular, let $C_1[j,d'] = C[j-1,d'] + p_c(j)$ if $p_c(j) + c(j) \leq d$ and $C_1[j,d'] = \infty$ otherwise, and, furthermore, $C_2[j,d'] = C[j-1,d' - p_s(j)]$ if $p_s(j) \leq d'$ and $C_2[j,d'] = \infty$ otherwise. Then, we may set $C[j,d'] = \min\{C_1(j,d'), C_2(j,d')\}$. Now, if $C[n+1,d] > b$, we know that there is no feasible solution for the search version, and otherwise, we can use backtracking starting from $C[n+1,d]$ to find one. The time and space complexity is polynomial in $d$ and $n$.

In the second dynamic program, we compute the smallest makespan $M[j,b']$ of all the schedules of the jobs $0,1,\ldots,j$ adhering to the budget $b'$, for each budget $b' \in \{0,1,\ldots,b\}$ and $j \in \mathcal{J}$. Again, we set $M[0,b'] = 0$ for each $b'$ and consider the two possibilities of scheduling job $j$ on the cloud or server. To that end, let $M_1[j,b'] = \max\{M[j-1,b' - p_c(j)], p_c(j) + c(j)\}$ if $p_c(j) + c(j) \leq d$ and $b' - p_c(j) \geq 0$. Otherwise, set $M_1[j,b'] = \infty$, furthermore, $M_2[j,b'] = M[j-1,b'] + p_s(j)$. Then, we may set $M[j,b'] = \min\{M_1(j,b'), M_2(j,b')\}$. Again, if $M[n+1,b] > d$, we know that there is no feasible solution for the search version, and otherwise, we can use backtracking starting from $M[n+1,b]$ to find one. The time and space complexity is polynomial in $b$ and $n$.

For both programs, we can use rounding and scaling approaches to trade the complexity dependence in $d$ or $b$ with dependence in $\text{poly}(n, \frac{1}{\varepsilon})$ incurring a loss of a factor $(1+\mathcal{O}(\varepsilon))$ in the makespan or cost, respectively, if a solution is found. This can then be combined with a suitable search procedure for approximate values of the optimal makespan or cost. For details, we refer to Section 2.4, where such techniques are used and described in more detail. In addition to the techniques mentioned there, the possibility of a zero-cost solution has to be considered which can easily be done in this case. ■

**Chain Graph Case** We present FPTAS results for the variant of *SCS* with chain task graph. The basic approach is very similar to the fully parallel case.

**Theorem 2.4** There are FPTAS for *SCS* with chain task graphs for the cost objective and for the makespan objective.

*Proof.* We again start by designing dynamic programs for the search version of the problem with budget $b$ and deadline $d$. Without loss of generality, we assume $\mathcal{J} = \{0,1,\ldots,n+1\}$ with $\mathcal{S} = 0$, $\mathcal{T} = n+1$, and $j \in \{0,1,\ldots,n+1\}$ being the $j$-th job in the chain.

For each deadline $d' \in \{0,1,\ldots,d\}$, job $j \in \{0,1,\ldots,n+1\}$, and location $loc \in \{s,c\}$ (referring to the server and cloud) we want to compute the smallest cost $C[d',j,loc]$ of all the schedules of the jobs $1,\ldots,j$ adhering to the deadline $d'$ and with the job $j$ being scheduled on $loc$. To that end, we set $C[d',0,s] = 0$, $C[d',0,c] = \infty$, and with slight abuse of notation use the convention $C[z,j,loc] = \infty$ for $z < 0$. Further values can be computed via the following recurrence relations:

$$C[d',j,s] = \min\{C[d' - p_s(j) - c(j-1,j), c], C[d' - p_s(j), s]\}$$
$$C[d',j,c] = \min\{C[d' - p_c(j), c] + p_c(j), C[d' - p_c(j) - c(j-1,j), s] + p_c(j)\}$$

If $C[d, n+1, s] > b$, we know that there is no feasible solution for the search version, and otherwise, we can use backtracking starting from $C[d, n+1, s]$ to find one. The time and space complexity is polynomial in $d$ and $n$.

In the second dynamic program, we compute the smallest makespan $M[j, b', loc]$ of all the schedules of the jobs $0, \dots, j$ adhering to the budget $b'$ and with job $j$ placed on location $loc$, for each $b' \in \{0, 1, \dots, b\}$, $j \in \{0, 1, \dots, n+1\}$ and $loc \in \{s, c\}$. We set $M[b', 0, s] = 0$, $M[b', 0, c] = \infty$, use the convention $M[z, j, loc] = \infty$ for $z < 0$, and the recurrence relations:

$$M[b', j, s] = \min\{M[b', c] + p_s(j) + c(j-1, j), M[b', s] + p_s(j)\}$$
$$M[b', j, c] = \min\{M[b' - p_c(j), c] + p_c(j), M[b' - p_c(j), s] + p_c(j) + c(j-1, j)\}$$

If $M[b, n+1, s] > d$, we know that there is no feasible solution for the search version, and otherwise, we can use backtracking starting from $M[b, n+1, s]$ to find one. The time and space complexity is polynomial in $b$ and $n$.

Like in the fully parallel case, we can use rounding and scaling approaches to trade the complexity dependence in $d$ or $b$ with a dependence in $\text{poly}(n, \frac{1}{\varepsilon})$ incurring a loss of a factor $(1 + \mathcal{O}(\varepsilon))$ in the makespan or cost, respectively, if a solution is found. This can then be combined with a suitable search procedure for approximate values of the optimal makespan or cost. For details, we again refer to Section 2.4, where such techniques are described in more detail. In addition to the techniques mentioned there, the possibility of a zero-cost solution has to be considered which can easily be done in this case as well. ■

## 2.3 The Extended Chain Model ($SCS^e$)

As a first step towards more general models, we introduce the extended chain model. The main idea here is to find a unifying generalization for the chain and fully parallel case. Informally one can imagine an extended chain as a chain graph where any number of edges were replaced with fully parallel graphs. After giving a formal definition of these graphs we give a reduction to show that this problem is strongly NP-hard. Then, we introduce a $(2+\varepsilon)$-approximation for the budget-constrained makespan minimization. Both the algorithm and the hardness proof use reductions to single machine weighted number of tardy jobs scheduling. Therefore, we briefly discuss this problem here before actually giving the reduction. We finish the constructive side by exploring some assumptions on problem instances that allow us to achieve FPTAS results with our approach.

**Model** We give a constructive description of extended chain graphs. Let $G = (\mathcal{J}, E)$ with $\mathcal{S} \in \mathcal{J}$ and $\mathcal{T} \in \mathcal{J}$ be a chain graph. For any number of edges $e = (j-1, j) \in E$ we may remove the edge $e$ and introduce a set of jobs $\mathcal{J}_j$ and for every $j' \in \mathcal{J}_j$ two edges, namely $(j-1, j')$ and $(j', j)$. The resulting graph $G' = (\mathcal{J}', E')$ is an extended chain graph. We denote by $N$ the total number of jobs (nodes) in the graph. Denote the *SCS* problem on extended chains by $SCS^e$. For an example, we refer to Figure 2.1. Note here, that the introduced subgraphs are fully parallel graphs as described earlier and consequently fully parallel graphs, as well as chain graphs, are a subset of extended chain graphs. This also directly infers that $SCS^e$ is at least weakly NP-hard as shown in Theorem 2.1 and Theorem 2.2.

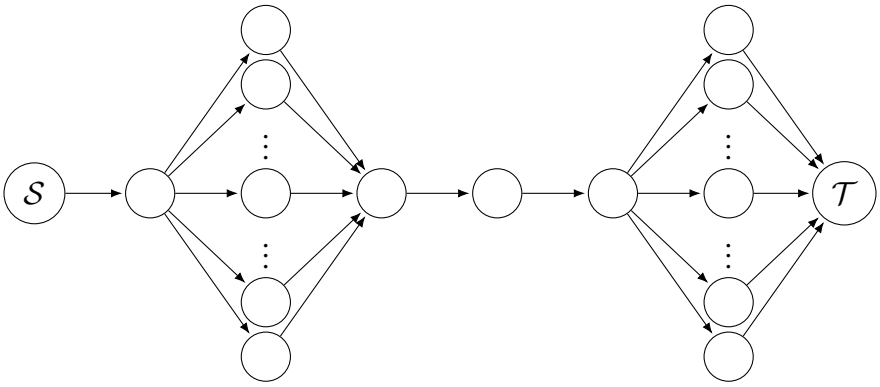



Figure 2.1: An example extended chain with two parallel parts.

### 2.3.1 A Preliminary Problem: Single Machine Weighted Number of Tardy Jobs

As mentioned before this section reduces some intermediate steps in the algorithm to the single machine weighted tardiness problems, for which we will reuse an already established algorithm.

The single machine weighted number of tardy jobs (WNTJ) problem, or $1 \mid \mid \sum w_j U_j$ in three field notation [34], can be defined as follows: On a single machine, where only one job at a time can be processed, $n$ jobs are to be scheduled. Each job has an integer processing time $p_j$, weight $w_j$ and due date $d_j$. A job is called *late* (or tardy) if its scheduled completion time $C_j > d_j$ and *early* if $C_j \leq d_j$. The goal is to find a schedule that minimizes the sum over the weights of the tardy (late) jobs. Pseudo-polynomial dynamic programs with runtime in $\mathcal{O}(n \min\{\sum_j p_j, \max_j d_j\})$ and $\mathcal{O}(n \min\{\sum_j p_j, \sum_j w_j, \max_j d_j\})$, respectively, were given by Lawler and Moore [52] and later Sahni [77]. Denote the former by `wTardyJobs`.

A natural extension of the model is denoted by $1 \mid r_j \mid \sum w_j U_j$. It also includes release dates $r_j$ per job $j$, marking the earliest time at which a job can start processing. Besides that addition, the model stays the same as WNTJ ($1 \mid \mid \sum w_j U_j$). This extended problem is strongly NP-hard [56]. To the best of our knowledge, no approximation algorithms with a provable approximation factor are known for this problem. There are however practical algorithms, which have been tested empirically. Used approaches contain mixed integer programming [22], genetic algorithms [80] and branch-and-bound algorithms [69].

For a more comprehensive survey on both (and related) problems, we refer to [2].

### 2.3.2 Strong NP-Hardness of Scheduling Extended Chains

As already noted, this problem is at least weakly NP-hard, following from Theorem 2.1 as well as Theorem 2.2. We show that this problem is actually strongly NP-hard, by giving a reduction from the strongly NP-hard $1 \mid r_j \mid \sum w_j U_j$ problem [56]. As in Section 2.2.1 we use decision variants of the considered problems, giving us results for both deadline-constrained cost reduction and budget-constrained makespan minimization.

**Theorem 2.5** The $SCS^e$ problem is strongly NP-hard.

*Proof.* $1 \mid r_j \mid \sum w_j U_j$ is defined as follows: Given a set of jobs $\mathcal{J} = \{1, \ldots, n\}$, each with processing time $p_j$, release date $r_j$, deadline $d_j$ and weight $w_j$, schedule the jobs (without

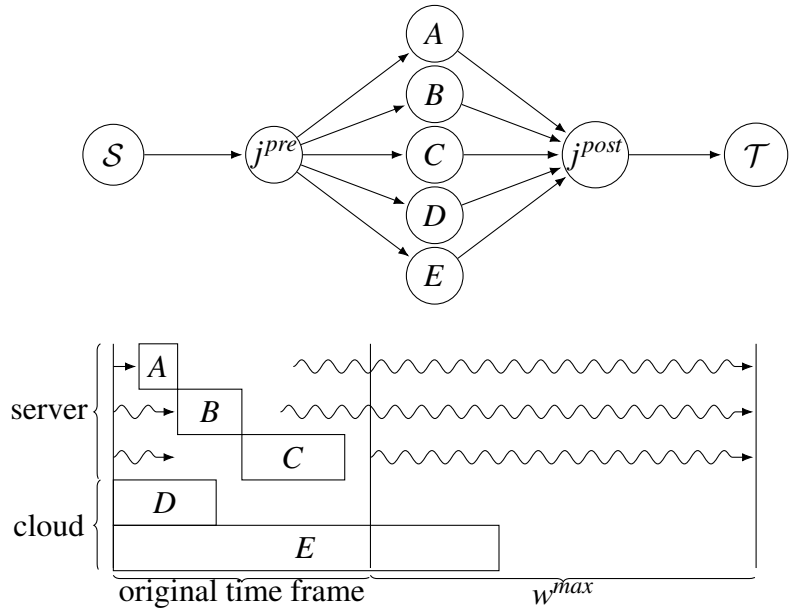



Figure 2.2: Schematic example of resulting $SCS^e$ problem for 5 jobs, squiggly arrows represent communication delays and model release dates and deadlines.

preemption) on a single machine, such that the sum of weights of late jobs is smaller or equal to a given $b$ ($\sum w_j U_j \leq b$). A job $j$ is late ($U_j = 1$) if it finishes processing after $d_j$, $U_j = 0$ otherwise.

Given an instance of $1 \mid r_j \mid \sum w_j U_j$, create the following decision version of $SCS^e$. Note that we will substitute "an edge $(j, j')$ with communication delay $c(j, j') = k$" simply by "an edge $c(j, j') = k$" to keep this readable. As per definition create $\mathcal{S}$ and $\mathcal{T}$ with $p_s(\mathcal{S}) = p_s(\mathcal{T}) = 0$ and $p_c(\mathcal{S}) = p_c(\mathcal{T}) = \infty$. Create jobs $j^{pre}$ and $j^{post}$ with $p_s(j^{pre}) = p_s(j^{post}) = \infty$ and $p_c(j^{pre}) = p_c(j^{post}) = 0$ and edges $c(\mathcal{S}, j^{pre}) = 0$ and $c(j^{post}, \mathcal{T}) = 0$. Set $w^{max} = \max_{j \in \mathcal{J}} w_j$ and $d^{max} = \max_{j \in \mathcal{J}} d_j$. For every $j \in \mathcal{J}$ create a job $j'$ with $p_s(j') = p_j$, $p_c(j') = w_j$ and edges $c(j^{pre}, j') = r_j$, $c(j', j^{post}) = w^{max} + d^{max} - d_j$. Set the deadline to $d' = w^{max} + d^{max}$ and the budget $b' = b$. Trivially, in all schedules $\mathcal{S}$ and $\mathcal{T}$ are scheduled on the server, $j^{pre}$ and $j^{post}$ on the cloud. Note that neither of these jobs contributes processing time to the resulting schedule. For better comprehension, we give an example of the structure in Figure 2.2.

It remains to show, that there is a schedule with $\sum w_j U_j \leq b$ for the original $1 \mid r_j \mid \sum w_j U_j$ problem, if and only if there is a schedule with $cost \leq b'$ and $mspan \leq d'$ for the $SCS^e$ problem. Assume that there is a schedule with $\sum w_j U_j \leq b$. We can partition the jobs into two sets $\mathcal{J}^{early}$ and $\mathcal{J}^{late}$, which contain all jobs that are on time or late, respectively. Place all jobs that correspond to a job from $\mathcal{J}^{late}$ on the cloud and start them immediately. All of them finish before $d' = w^{max} + d^{max}$, since $w^{max} \geq p_c(j')$. Place all remaining jobs ($\mathcal{J}^{early}$) on the server and let them start at the same time as in the original schedule. Since no job starts before its release date no communication delay is violated in the new schedule. Since all jobs from $\mathcal{J}^{early}$ end before their deadline, no communication delay hinders us from scheduling $j^{post}$ and $\mathcal{T}$ at $d' = w^{max} + d^{max}$. The cost of that schedule is equal to the value of $\sum w_j U_j$ in the original schedule and therefore $cost \leq b$. One can confirm that the other direction works analogously by keeping the schedule of jobs on the cloud intact and simply processing all jobs from the cloud after that schedule in any order.

■

### 2.3.3 A $(2+\varepsilon)$-approximation (Makespan) on the Extended Chain

**Theorem 2.6** There is a $(2+\varepsilon)$-approximation algorithm for the budget-constrained makespan minimization $SCS^e$ problem.

We design a pseudo-polynomial algorithm, that given a feasible makespan estimate $T$ ($T \geq mspan_{OPT}$) calculates a schedule with makespan at most $\min\{2T, 2mspan_{OPT}\}$. Otherwise ($T < mspan_{OPT}$) the algorithm calculates a schedule with makespan at most $\min\{2T, 2mspan_{OPT}\}$ or no schedule at all. We can use a binary search to find $T \approx OPT$, beginning with the trivial upper bound $T = \sum_{j \in \mathcal{J}'} p_s(j) \geq mspan_{OPT}$.

We first introduce notation that follows the constructive description of extended chains above. We assume $\mathcal{J} = \{0, 1, \ldots, n+1\}$ with $\mathcal{S} = 0$, $\mathcal{T} = n+1$, and $j \in \{1, \ldots, n\}$ being the $j$-th job in the original chain. If there is a parallel subgraph between some jobs $j-1$ and $j$ we denote the jobs in it by $\mathcal{J}_j = \{0^j, 1^j, \ldots, m^j\}$.

We reuse the state description from Theorem 2.4, but this time we iteratively create all reachable states by going over the jobs $\{0, 1, \ldots, n+1\}$. A state is a combination of timestamp $t \in \{0, 1, \ldots, T\}$, job $j \in \{0, 1, \ldots, n+1\}$, and location $loc \in \{s, c\}$ (referring to server and cloud respectively). The value of a state is the smallest cost of all the schedules of the jobs $0, 1, \ldots, j$ finishing processing during or before timestamp $t$, with $j$ being scheduled on $loc$, denoted by $[t, j, loc] = cost$. Note, that we have not mentioned the parallel subgraphs in the description above. We start with the trivial start state [0,0(=$\mathcal{S}$),s] = 0.

Let StateList$^{j-1}$ be the list of states for some job $j-1$ of the chain. We create StateList$^j$ in the following way: First we create a set of state extensions Extensions$^j$, each of form $[\Delta t, loc_{j-1} \to loc_j] = cost$. Then we form every (fitting) combination of a state from StateList$^{j-1}$ with an extension from Extensions$^j$, which forms StateList$^j$. Lastly, we cull all dominated states from StateList$^j$ and continue with $j+1$.

**Calculate Extensions$^j$:**

1. If there is no parallel subgraph between $j-1$ and $j$ we can simply enumerate all state extensions:
   a. $j-1$ on server, $j$ on server: $[p_s(j),\ s \to s] = 0$
   b. $j-1$ on server, $j$ on cloud: $[p_c(j) + c(j-1, j),\ s \to c] = p_c(j)$
   c. $j-1$ on cloud, $j$ on server: $[p_s(j) + c(j-1, j),\ c \to s] = 0$
   d. $j-1$ on cloud, $j$ on cloud: $[p_c(j),\ c \to c] = p_c(j)$
2. Otherwise, there is a parallel subgraph between $j-1$ and $j$ with jobs $\mathcal{J}_j = \{0^j, 1^j, \ldots, m^j\}$. Enumerate (or approximate) possible extensions:
   a. $j-1$ on server, $j$ on server:
   Set $\Delta^{max} = \min\{\sum_{j' \in \mathcal{J}_j} p_s(j'), T\}$, for every $\Delta^i$ in $\{0, \ldots, \Delta^{max}\}$, do the following: Set $\mathcal{J}^s = \emptyset$ and $\mathcal{J}^c = \emptyset$. For every $j' \in \mathcal{J}_j$ check:
   - $p_s(j') > \Delta^i$ and $c(j-1, j') + p_c(j') + c(j', j) > \Delta^i$:
     break and go to next $\Delta^i$ ($[\Delta^i,\ s \to s]$ not feasible)
   - $p_s(j') > \Delta^i$ and $c(j-1, j') + p_c(j') + c(j', j) \leq \Delta^i$:
     add $j'$ to $\mathcal{J}^c$ ($j'$ has to be put on the cloud)
   - $p_s(j') \leq \Delta^i$ and $c(j-1, j') + p_c(j') + c(j', j) > \Delta^i$:
     add $j'$ to $\mathcal{J}^s$ ($j'$ has to be put on the server)

If $\sum_{j' \in \mathcal{J}^s} p_s(j') > \Delta^i$ break and go to next $\Delta^i$. More load has to be placed on the server than there is time. Create a WNTJ instance as follows: For every job $j' \in \mathcal{J}_j \setminus (\mathcal{J}^s \cup \mathcal{J}^c)$ create a job $j''$ with processing time $p_{j''} = p_s(j')$, deadline $d_{j''} = \Delta^i - \sum_{j' \in \mathcal{J}^s} p_s(j')$ and weight $w_{j''} = p_c(j')$. Solve this problem with `wTardyJobs`, let $V$ be the cost of the solution. Add $[\Delta^i,\ s \to s] = \sum_{j' \in \mathcal{J}^c} p_c(j') + V$ to EXTENSIONS$^j$. (This could also be solved as a knapsack problem, but we need WNTJ later either way.)

b. $j-1$ on server, $j$ on cloud:
Set $\Delta^{max} = \min\{\sum_{j' \in \mathcal{J}_j} p_s(j') + \max_{j' \in \mathcal{J}_j} c(j', j), T\}$.
For every $\Delta^i$ in $\{0, \ldots, \Delta^{max}\}$, do the following: Set $\mathcal{J}^s = \emptyset$ and $\mathcal{J}^c = \emptyset$.
For every $j' \in \mathcal{J}_j$ check:
- $p_s(j') + c(j', j) > \Delta^i$ and $c(j-1, j') + p_c(j') > \Delta^i$:
  break and go to next $\Delta^i$ (state extension $[\Delta^i,\ s \to c]$ not feasible)
- $p_s(j') + c(j', j) > \Delta^i$ and $c(j-1, j') + p_c(j') \leq \Delta^i$:
  add $j'$ to $\mathcal{J}^c$ ($j'$ has to be put on the cloud)
- $p_s(j') + c(j', j) \leq \Delta^i$ and $c(j-1, j') + p_c(j') > \Delta^i$:
  add $j'$ to $\mathcal{J}^s$ ($j'$ has to be put on the server)

If $\sum_{j' \in \mathcal{J}^s} p_s(j') > \Delta^i$ break and go to next $\Delta^i$. More load has to be placed on the server than there is time. Create a WNTJ instance as follows: For every job $j' \in \mathcal{J}_j \setminus \mathcal{J}^c$ create a job $j''$ with processing time $p(j'') = p_s(j')$, deadline $d_{j''} = \Delta^i - c(j', j)$ and weight $w_{j''} = p_c(j')$ if $j' \notin \mathcal{J}^s$, $w_{j''} = \infty$ otherwise. Solve this problem with `wTardyJobs`, let $V$ be the cost of the solution if $V = \infty$ break. Otherwise, add $[\Delta^i,\ s \to c] = \sum_{j' \in \mathcal{J}^c} p_c(j') + V$ to EXTENSIONS$^j$.

c. $j-1$ on cloud, $j$ on server:
This works analogously to the previous case. Simply replace each instance of $c(j', j)$ by $c(j-1, j')$ and vice versa. Add the resulting extensions to EXTENSIONS$^j$. Note, that for the reduction there is no computational difference between a common release date and different deadlines and different release dates but a common deadline.

d. $j-1$ on cloud, $j$ on cloud:
We 2-approximates the resulting extensions, by precisely handling the communication to the server, but upper bounding the communication from the server. Repeat case 2b with the two following changes:
For the checks before the problem conversion use $c(j-1, j') + p_s(j') + c(j', j)$ and $p_c(j')$ instead of $p_s(j') + c(j', j)$ and $c(j-1, j') + p_c(j')$, respectively. Let $\mathcal{J}^{s'} \subseteq \mathcal{J}_j$ be the set of jobs actually put on the server in this step. Add $[\Delta^i + \max_{j' \in \mathcal{J}^{s'}} c(j-1, j'),\ c \to c] = \sum_{j' \in \mathcal{J}^c} p_c(j') + V$ instead of $[\Delta^i,\ c \to c] = \sum_{j' \in \mathcal{J}^c} p_c(j') + V$ to EXTENSIONS$^j$. We wait for the biggest communication delay to pass until we schedule the first job on the server. Note, that $\Delta^i + \max_{j' \in \mathcal{J}^{s'}} c(j', j) \leq 2\Delta^i$ by construction.

For every pair of a state $([t, j-1, loc] = cost) \in$ STATELIST$^{j-1}$ and $([\Delta t, loc_{j-1} \to loc_j] = cost') \in$ EXTENSIONS$^j$ with $loc = loc_{j-1}$ add $[t + \Delta t, j, loc_j] = cost + cost'$ to STATELIST$^j$. After that process, for every triple $t, j, loc$ that has multiple states in STATELIST$^j$ keep only the state with the lowest cost. We can also discard states with $cost > b$ and timestamp $t > 2T$. Repeat this process with $j \to j+1$ until we computed

StateList$^{n+1}$, simply move through that list and select the state with the lowest timestamp $t$. If there is no such state, there exists no schedule with makespan smaller or equal to $T$.

**Lemma 2.1** Given a feasible $T$, the described procedure calculates a 2-approximation on the optimal makespan in time poly$(N,T)$

*Proof.* We start by showing the approximation factor. Everywhere except step 2d we simply enumerate all (non-dominated) ways to extend a running schedule, which is trivially optimal. Assume that we added $[\Delta^i,\ c \to c] = \sum_{j' \in \mathcal{J}^c} p_c(j') + V$ instead of $[\Delta^i + \max_{j' \in \mathcal{J}^{s'}} c(j',j),\ c \to c] = \sum_{j' \in \mathcal{J}^c} p_c(j') + V$ in step 2d above. That hypothetical algorithm would calculate a (possibly infeasible) solution with makespan $mspan_{ALG}^{hypo} \leq mspan_{OPT}$, since step 2d underestimates the needed time (by ignoring one of the communication delays), and everything else is calculated precisely. The actual algorithm has makespan $mspan_{ALG} \leq 2mspan_{ALG}^{hypo}$ and therefore also $mspan_{ALG} \leq 2mspan_{OPT}$.

We show the runtime of the algorithm by bounding the time needed for each iteration of 1. constructing state extensions Extensions$^j$, 2. combining the extensions with the previous StateList$^{j-1}$ and 3. culling duplicates from the resulting StateList$^j$.

1. For directly connected jobs $j-1$ and $j$ we can trivially calculate the 4 options in constant time. Therefore, we are interested in the runtime of steps 2a, 2b, 2c and 2d for some parallel subgraph with jobs $\mathcal{J}_j$. The steps get repeated for $\Delta^i$ in $\{0,\dots,\Delta^{max}\}$, where $\Delta^{max} < T$. The preprocessing in each iteration of all four steps needs time linear in the size of $\mathcal{J}_j$. Using `wTardyJobs` in the steps needs time in $\mathcal{O}(|\mathcal{J}_j| \min\{\sum_{j' \in \mathcal{J}_j \setminus \mathcal{J}^c} p_s(j'), \max_{j' \in \mathcal{J}_j \setminus \mathcal{J}^c} d_{j''}\}) \leq \mathcal{O}(T \cdot N^2)$. Overall we need time in poly$(T,N)$ to calculate Extensions$^j$, with $|\text{Extensions}^j| \leq \mathcal{O}(T)$
2. StateList$^{j-1}$ contains at most $2T \cdot (n+2) \cdot 2$ (timestamp, job, location) different states (after the previous culling). We may simply bruteforce all possible combinations from StateList$^{j-1} \times$ Extensions$^j$. Since both of these sets have at most poly$(T,N)$ elements, the resulting set StateList$^j$ also has polynomial size.
3. By culling states from StateList$^j$ we reduce it back to size at most $2T \cdot (n+2) \cdot 2$. It should be obvious, that we can identify duplicate states in polynomial time.

Note that we iterate the above steps for each job $j \in \{1,\dots,n+1\}$. Therefore we have a polynomial repetition of steps needing polynomial time. We prevent exponential build-up in the state lists, by culling duplicates after each iteration. ■

Now we have to scale our instance, such that our pseudo-polynomial algorithm runs in proper polynomial time. For that, we scale $T$ and all $p_c$, $p_s$ and $c$ by $\frac{N\varepsilon'}{T}$ and round down to the next integer. Then, we run our algorithm with the scaled values, but still use the unscaled $p_c$ to calculate the *value (cost)* of states, as those calculations only factor logarithmically in the runtime, a $p_c$ exponential in the input size is fine. The algorithm now needs time in poly$(N, \lfloor \frac{T \cdot N\varepsilon'}{T} \rfloor) \leq$ poly$(N,\varepsilon')$ and finds a 2-approximation for the scaled instance (given a feasible $T$). After scaling back up each job and communication delay might need up to $\frac{T}{N\varepsilon'}$ additional time, delaying our whole schedule by at most $3N \cdot \frac{T}{N\varepsilon'} \leq 3\varepsilon' T$. For $\varepsilon = 3\varepsilon'$ and $T = mspan_{OPT}$ our resulting schedule has a makespan of $mspan_{ALG} \leq 2mspan_{OPT} + \varepsilon T = (2+\varepsilon)mspan_{OPT}$. Via a binary search, we can find such a $T$ by repeating our procedure at most $\log \sum_{j \in \mathcal{J}'} p_s(j)$ times. This concludes the proof of Theorem 2.6.

**Corollary 2.1** There is a polynomial algorithm for the deadline-constrained cost minimization problem on extended chains, that finds a schedule with at most optimal cost, but a makespan of $(2+\varepsilon)d$.

**R** With argumentation similar to the reduction in Theorem 2.5, one can show that the $1 \mid r_j \mid \sum w_j U_j$ problem is embedded in step 2d of this section's algorithm. This leads to the observation, that we might be able to use approximation results for $1 \mid r_j \mid \sum w_j U_j$ to improve our handling of that case. Sadly, as noted earlier, there seem to be no known approximation algorithms with a provable approximation factor for this problem. If one becomes known in the future, it might be worthwhile to reevaluate the approach in step 2d.

### 2.3.4 Cases with FPTAS

We reconsider the approximation result for three special cases of instances which allow us to improve the result. Looking back at Theorem 2.6, we build an algorithm that would be an FPTAS if it were not for case 2d where we needed to double our time frame $\Delta^i$ to fit the unaccounted communication delay. In the following part, we will only describe how to approach that case, since everything else can stay as it was.

First, we assume locally small delays in the parallel subgraphs, meaning that the smallest processing time in the subgraph is at least as big as the largest communication delay. More precisely, for every $\mathcal{J}_e$ with $e = (j-1, j)$ it holds that

$$\min_{j' \in \mathcal{J}_e} \min\{p_s(j'), p_c(j')\} \geq \max_{j' \in \mathcal{J}_e} \max\{c((j-1, j')), c((j', j))\}.$$

In this case, only the first $j^\alpha$, and the last job $j^\omega$ to be processed on the server are actually affected by their communication delay, since all other delays fit in the time frame, where $j^\alpha$ and $j^\omega$ are processed. After the preprocessing of a given $\Delta^i$, for each pair of jobs $j^\alpha, j^\omega \in \mathcal{J}_j \setminus \mathcal{J}^c$ with $j^\alpha \neq j^\omega$ do the following: Assume $j^\alpha, j^\omega$ are the first and last job to be processed on the server, respectively. Add $j^\alpha$ and $j^\omega$ to $\mathcal{J}^s$. Now create the WNTJ instance as follows: For every job $j' \in \mathcal{J}_j \setminus (\mathcal{J}^s \cup \mathcal{J}^c)$ create a job $j''$ with processing time $p_{j'} = p_s(j')$, deadline $d_{j''} = \Delta^i - (c(j-1, j^\alpha) + c(j^\omega, j)) - \sum_{j' \in \mathcal{J}^s} p_s(j')$ and weight $w_{j''} = p_c(j')$. Solve this problem with `wTardyJobs`, let $V$ be the cost of the solution and note $[\Delta^i,\ c \to c]_{j^\omega}^{j^\alpha} = \sum_{j' \in \mathcal{J}^c} p_c(j') + V$. After all ($\mathcal{O}(N^2)$) combinations have been tested, add the smallest $[\Delta^i,\ c \to c]_{j^\omega}^{j^\alpha}$ to $\text{EXTENSIONS}^j$.

Secondly, we assume a constant upper bound $c_{max}$ on the communication delays inside parallel subgraphs. More precisely, for every $\mathcal{J}_e$ with $e = (j-1, j)$ it holds that

$$c_{max} \geq c(j-1, j') \text{ and } c_{max} \geq c(j', j).$$

Instead of brute forcing only a first and last job, we brute force the first and last $c_{max}$ timesteps. Trivially, jobs with $p_s = 0$ can be put on the server, and therefore there are at most $\mathcal{O}(N^c_{max} \cdot N^c_{max})$ combinations we have to work through. The remaining part works analogously to the first case.

Lastly, we assume that each job produces some output, that has to be sent to all of its direct successors in full, meaning that all outgoing communication delays of a job are equivalent. More precisely, for every $\mathcal{J}_e$ with $e = (j-1, j)$ it holds that

$$\forall j', j'' \in \mathcal{J}_e : c(j-1, j') = c(j-1, j'').$$

Here we can simply reuse the result from step 2b, but subtract $c(j-1,j')$ from the $\Delta^i$ used in the WNTJ problem. Since all $c(j-1,j')$ are equal, no job could be processed on the server in the first $c(j-1,j')$ timesteps and all jobs are available after those $c(j-1,j')$ timesteps.

All these, in combination with the previously described scaling approach, lead to FPTAS results:

**Theorem 2.7** There is an FPTAS for the budget-constrained makespan minimization problem on extended chains if at least one of the following holds for every parallel subgraph $\mathcal{J}_e$ with $e=(j-1,j)$:

1. $\min_{j'\in\mathcal{J}_e}\min\{p_s(j'),p_c(j')\} \geq \max_{j'\in\mathcal{J}_e}\max\{c((j-1,j')),c((j',j))\}$
2. $c_{max} \geq c(j-1,j')$ and $c_{max} \geq c(j',j)$
3. $\forall j',j'' \in \mathcal{J}_e : c(j-1,j') = c(j-1,j'')$

## 2.4 Constant Cardinality Source and Sink Dividing Cut ($SCS^{\psi}$)

In this section, we introduce a concept to bound the number of jobs that can be relevant for scheduling decisions at the same time. Consider the task graph during a running schedule and partition the jobs into a set of completed and a set of pending jobs. Count the number of edges connecting jobs in the completed set to jobs in the pending set. For a given instance, the *maximum cardinality source and sink dividing cut* denotes the maximum number of crossing edges for all possible intermediate schedules. We examine cases where this number is bounded by a constant.

**Model** We introduce the notion of a *maximum cardinality source and sink dividing cut*. For $G=(\mathcal{J},E)$, let $\mathcal{J}_\mathcal{S}$ be a subset of jobs, such that $\mathcal{J}_\mathcal{S}$ includes $\mathcal{S}$ and there are no edges $(j,k)$ with $j \in \mathcal{J}\setminus\mathcal{J}_\mathcal{S}$ and $k\in\mathcal{J}_\mathcal{S}$. In other words, in a running schedule $\mathcal{J}_\mathcal{S}$ and $\mathcal{J}\setminus\mathcal{J}_\mathcal{S}$, could represent already processed jobs and still to be processed jobs respectively. Denote by $\mathcal{J}_\mathcal{S}^G$ the set of all such sets $\mathcal{J}_\mathcal{S}$. We define

$$\psi := \max_{\mathcal{J}_\mathcal{S}\in\mathcal{J}_\mathcal{S}^G} |\{(j,k)\in E \mid j\in\mathcal{J}_\mathcal{S} \wedge k\in\mathcal{J}\setminus\mathcal{J}_\mathcal{S}\}|,$$

the maximum number of edges between any set $\mathcal{J}_\mathcal{S}$ and $\mathcal{J}\setminus\mathcal{J}_\mathcal{S}$ in $G$. In a series-parallel task graph $\psi$ is equal to the maximum anti-chain size of the graph.

### 2.4.1 Dynamic Programming for *SCS*

We discuss how to solve or approximate problems with a constant size $\psi$, but otherwise arbitrary task graphs ($SCS^{\psi}$). We first consider the deadline-constrained cost minimization, in Theorem 2.9 we show how to adapt this to the budget-constrained makespan minimization. We give a dynamic program to optimally solve instances of *SCS* with arbitrary task graphs. At first, we will not confine the algorithm to polynomial time. Consider a given problem instance with $G=(\mathcal{J},E)$, its source $\mathcal{S}$ and sink $\mathcal{T}$, processing times $p_s(j)$ and $p_c(j)$ for each $j\in\mathcal{J}$, communication delays $c(i,j)$ for each $(i,j)\in E$ and a deadline $d$.

We define intermediate states of a (running) schedule, as the states of our dynamic program (see Section 2.4.1). Such a state contains two types of variables. First, we have two global variables, the timestamp $t$ and the number of timesteps the server has been

unused $f_s$. In other words, the server has not finished processing a job since $t - f_s$. The second type is defined per *open edge*. An open edge is a $e = (j,k)$ where $j$ has already been processed, but $k$ has not. For each such edge add the variables $e = (j,k)$ (the edge itself), $loc_j \in \{s,c\}$ denoting if $j$ was processed on the server ($s$) or the cloud ($c$) and $f_j$ denoting the number of timesteps that have passed since $j$ finished processing. If a job $j$ is contained in multiple open edges, $loc_j$ and $f_j$ are still only included once. Write the state as $[t, f_s, e^1 = (j^1,k^1), loc_{j^1}, f_{j^1}, \ldots, e^m = (j^m,k^m), loc_{j^m}, f_{j^m}]$, where $e^1, \ldots, e^m$ denote all open edges. Note here, that there is information that we purposefully drop from a state: the completion time and location of every processed job without open edges, as those are not important for future decisions anymore. There might be multiple ways to reach a specific state, but we only care about the minimum possible cost to achieve that state, which is the *value* of the state.

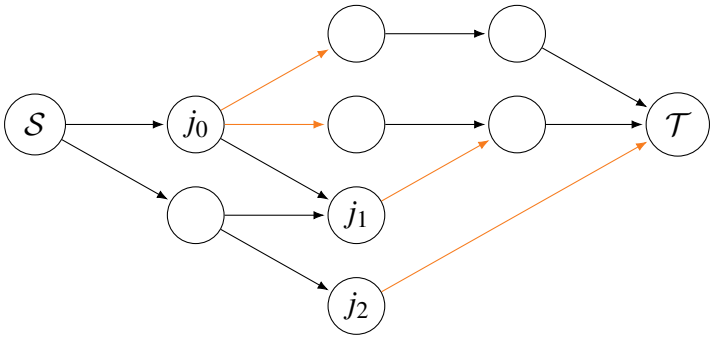



Figure 2.3: Example state of a running schedule, *open edges* are orange, $loc_{j_i}$ and $f_{j_i}$ kept for $j_0$, $j_1$ and $j_2$.

We iteratively calculate the value of every reachable state with $t = 0, 1, 2, \ldots$. We start with the trivial state $[t = 0, f_s = 0, e^1, \ldots, e^m, loc_{\mathcal{S}} = s, f_{\mathcal{S}} = 0] = 0$, where $e^1, \ldots, e^m \in E$ with $e^i = (\mathcal{S}, j)$. This state forms the beginning of our *(sorted) state list*. We keep this list sorted in ascending order of state values (costs) at all times. We exhaustively calculate every state that is reachable during a specific timestep, given the set of states reachable during the previous timestep. Intuitively, we try every possible way to "fill up" the still undefined time windows $f_s$ and $f_j$. After the dynamic program is finished, we iterate through the state list one last time and take the first state $[t = d, f_s]$. The value of that state is the minimum cost possible to schedule $G$ in time $d$. One can easily adapt this procedure to also yield such a schedule, by annotating each kept state with a list of all processed jobs containing their location and completion time. Note here, that we would consider two states that only differ in this list as identical, and only keep one of them (with either of the lists). Finally, we give the actual dynamic program:

**DPFGG – Dynamic Program for General Graphs:**

1. Initialize the state list $SL$ with the start state (as defined above)
2. For all $state \in SL$:
   a. Let $\mathcal{J}^{state}$ be the set of all jobs that are endpoints in open edges from $state$
   b. For all $j \in \mathcal{J}^{state}$:
      i. Check if $\forall (k,j) \in E : (k,j)$ is also an open edge in $state$, if not, break and go to next job. Are all predecessors of $j$ already processed?
      ii. Can $j$ be processed on the server?
      Check $f_s \geq p_s(j)$ and for every $(k,j) \in E$: $loc_k = s \wedge f_k < p_s(j)$ or $loc_k = c \wedge f_k < p_s(j) + c(k,j)$. Is there enough free time on the server, and did all predecessors finish sufficiently early, considering the processing time as well as possible communication delays? If yes:
         A. Calculate resulting state $state'$ as a copy of $state$ with the same cost value and the following changes:
         For all $(k,j) \in E$ remove $(k,j)$ from $state'$. If $j$ is the last open successor of $k$, remove $f_k$ and $loc_k$ from $state'$. Add $f_j = 0, loc_j = s$ and all new open edges to $state'$. Set $f_s = 0$.
         B. If $state'$ is already in $SL$: update the cost value of $state'$ in $SL$, if the new value is lower. Then move $state'$ to correct position in $SL$.
         C. Otherwise, add $state'$ to correct position in $SL$ (always after $state$).
      iii. Can $j$ be processed on the cloud?
      Check for every $(k,j) \in E$: $loc_k = s \wedge f_k < p_c(j) + c(k,j)$ or $loc_k = c \wedge f_k < p_c(j)$. Did all predecessors finish sufficiently early, considering the processing time as well as possible communication delays? If yes:
         A. Calculate resulting state $state'$ as a copy of $state$ with the following changes:
         For all $(k,j) \in E$ remove $(k,j)$ from $state'$. If $j$ is the last open successor of $k$, remove $f_k$ and $loc_k$ from $state'$. Add $f_j = 0, loc_j = c$ and all new open edges to $state'$. Increase the cost value of $state'$ by $p_c(j)$.
         B. If $state'$ is already in $SL$: update the cost value of $state'$ in $SL$, if the new value is lower. Then move $state'$ to correct position in $SL$.
         C. Otherwise, add $state'$ to correct position in $SL$ (always after $state$).
3. Check end condition:
   If there is a state $[t = d, f_s] \in SL$, return such state with the lowest cost value.
4. Else, if $t < d$:
   a. Move from $t$ to $t+1$:
   For all $state \in SL$ increase $t$, $f_s$ and each $f_j$ in $state$ by 1.
   b. Back to step 2.

**Lemma 2.2** DPFGG's runtime is bounded in $\mathcal{O}(d^{2\psi+3} \cdot n^{2\psi+1})$.

*Proof.* At any point, there are a maximum of $\mathcal{O}(d \cdot (d \cdot n)^{\psi})$ states in the state list, given by the maximum value of $f_s$ and the possible combination of open edges with their timestamp. For every $t$ we look at every state. Since we never insert a state in front of the state we are currently inspecting (costs can only increase), this traverses the list exactly once. For each of those states, we calculate every possible successor, of which there are $\mathcal{O}(\psi)$ and

traverse the state list an additional time to correctly insert or update the state. We iterate from $t = 0$ to $d$ and therefore get a runtime of: $\mathcal{O}(d \cdot ((d \cdot (d \cdot n)^{\psi}) \cdot \psi \cdot (d \cdot (d \cdot n)^{\psi}))) = \mathcal{O}(d^3 \cdot n \cdot (d \cdot n)^{2\psi}) = \mathcal{O}(d^{2\psi+3} \cdot n^{2\psi+1})$. ■

### 2.4.2 Scaling and Rounding the Dynamic Program

We use a rounding approach on DPFGG to get a program that is polynomial in $n = |\mathcal{J}|$, given that $\psi$ is constant. We scale $d$, $c$, $p_c$, and $p_s$ by a factor $\varsigma := \frac{\varepsilon \cdot d}{2n}$. Denote by $\hat{d} := \lceil \frac{d}{\varsigma} \rceil \leq \frac{2n}{\varepsilon} + 1$, $\hat{p}_s(j) := \lfloor \frac{p_s(j)}{\varsigma} \rfloor$, $\hat{p}_c(j) := \lfloor \frac{p_c(j)}{\varsigma} \rfloor$ and $\hat{c}(x) := \lfloor \frac{c(x)}{\varsigma} \rfloor$. Note here, that we round up $d$ but everything else down. We run the dynamic program with the rounded values but still calculate the cost of a state with the original unscaled values.

We transform the output $\pi'$ to the unscaled instance, by trying to start every job $j$ at the same (scaled back up) point in time as in the scaled schedule. Since we rounded down, there might now be points in the schedule where a job $j$ can not start at the time it is supposed to. This might be due to the server not being free, a parent node of $j$ that has not been fully processed or an unfinished communication delay. We look at the first time this happens and call the mandatory delay on $j$ $\Delta$ and increase the start time of every remaining job by $\Delta$. Repeat this process until all jobs are scheduled. We introduce no new conflicts with this procedure since we always move everything together as a block. Call this new schedule $\pi$.

**Theorem 2.8** DPFGG combined with the scaling technique finds a schedule $\pi$ with at most optimal cost and a makespan $\leq (1+\varepsilon) \cdot d$ in time $\mathcal{O}(n^{4\psi+4} \cdot \frac{1}{\varepsilon}^{2\psi+3})$, for any $\varepsilon > 0$. For problem instances with a constant maximum cardinality source and sink dividing cut ($\psi$), this is in time poly$(n, \frac{1}{\varepsilon})$.

*Proof.* We start by proving the runtime of our algorithm. We can scale the instance in polynomial time, this holds for both scaling down and scaling back up. The dynamic program now takes time in $\mathcal{O}(\hat{d}^{2\psi+3} \cdot n^{2\psi+1})$, where $\hat{d} \leq \frac{2n}{\varepsilon} + 1$, resulting in $\mathcal{O}(n^{4\psi+4} \cdot \frac{1}{\varepsilon}^{2\psi+3})$. Since $\psi$ is constant this results in a dynamic program runtime in poly$(n, \frac{1}{\varepsilon})$. In the end, we transform the schedule as described above, by going through the schedule once and delaying every job no more than $n$ times. Trivially, this can be done in polynomial time as well.

Secondly, we show that the makespan of $\pi$ is at most $(1+\varepsilon) \cdot d$. Every valid schedule for the unscaled problem is also valid in the scaled problem, meaning that there is no possible schedule we overlook due to the scaling. In the other direction, this might not hold. First, while scaling everything down we rounded the deadline up. This means, that scaled back we might actually work with a deadline of up to $d + \varsigma$. Secondly, we had to delay the start of jobs to make sure that we only start jobs when it is actually possible. In the worst case, we delay the sink $\mathcal{T}$ a total of $n-2$ times, once for every job other than $\mathcal{S}$ and $\mathcal{T}$. Each time we delay all remaining jobs we can bound the respective $\Delta < 2 \cdot \varsigma$. This is due to the fact that each of the delaying options can not delay by more than $\varsigma$ (as that is the maximum timespan not regarded in the scaled problem) and only a direct predecessor job and the communication from it needing longer can coincide to a non-parallel delay. Taking both of these into account, a valid schedule for the scaled problem might use time up to

$$d + \varsigma + (n-2) \cdot (2\varsigma) \leq d + 2n\varsigma = (1+\varepsilon) \cdot d$$

in the unscaled instance.

Lastly, we take a look at the cost of $\pi$. While rounding, we did not change the calculation of a state's value, and with every valid schedule of the unscaled instance being still valid in the scaled instance we can conclude that the cost of $\pi$ is smaller or equal to an optimal solution of the original problem. ■

**Theorem 2.9** DPFGG combined with the scaling technique and a binary search over the deadline yields an FPTAS for the budget-constrained makespan minimization problem, for graphs with a constant maximum cardinality source and sink dividing cut ($\psi$).

*Proof.* Theorem 2.8 can be adapted to solve this, assuming that we know a reasonable makespan estimate of an optimal solution to use in our scaling factor. During the algorithm discard any state with costs bigger than the budget and terminate when the first state $[t, f_s]$ is reached. The $t$ gives us the makespan.

Using a makespan estimate that is too big will lead to a rounding error that is not bounded by $\varepsilon \cdot mspan_{OPT}$, a too-small estimate might not find a solution. To solve this, we start with an estimate that is purposefully large. Let $d^{max} = \sum_{j \in \mathcal{J}} p_s(j)$ be the sum over all processing times on the server. There is always a schedule with 0 costs and makespan $d^{max}$. We run our algorithm with the scaling factor $\varsigma^0 := \frac{\varepsilon \cdot d^{max}}{4n}$. Iteratively repeat this process with scaling factor $\varsigma^i = \frac{1}{2^i}\varsigma^0$ for increasing $i$ starting with 1. At the same time half the original deadline estimate in each step, which leads to $\hat{d}$, and therefore the runtime, to stay the same in each iteration. End the process when the algorithm does not find a solution for the current $i$ and deadline estimation. This infers that there is no schedule with the wanted cost budget and a makespan smaller or equal to $\frac{1}{2^i}d^{max}$ (in the unscaled instance), therefore $\frac{1}{2^i}d^{max} < mspan_{OPT}$. We look at the result of the previous run $i-1$: The scaled result was optimal, therefore the unscaled version has a makespan of at most

$$mspan_{ALG} \leq mspan_{OPT} + 2n \cdot \varsigma^{i-1} \tag{2.1}$$

$$= mspan_{OPT} + 2n \cdot \frac{1}{2^{i-1}} \cdot \frac{\varepsilon \cdot d^{max}}{4n} \tag{2.2}$$

$$= mspan_{OPT} + \varepsilon \cdot \frac{1}{2^i} d^{max} \leq (1+\varepsilon) mspan_{OPT}. \tag{2.3}$$

It should be easy to infer from Lemma 2.2 that each iteration of this process has polynomial runtime. Combined with the fact that we iterate at most $\log d^{max}$ times we get a runtime that is in $\text{poly}(n, \frac{1}{\varepsilon})$. ■

**R** The results of this section work, as written, for a constant $\psi$. Note here, that for series-parallel digraphs, this is equivalent to a constant anti-chain size. The algorithms can also be adapted to work on any graph with constant anti-chain size, if the communication delays are (bounded by) some constant or are *locally small*. Delays are locally small, if for every $(j,k) \in E$, $c(j,k)$ is smaller or equal than every $p_c(k')$, $p_s(k')$, $p_c(j')$ and $p_s(j')$, where $k'$ is every direct successor of $j$ and $j'$ every direct predecessor of $k$ [70].

## 2.5 Strong NP-Hardness

In this section, we consider more involved reductions than in Section 2.2 in order to gain a better understanding of the complexity of the problem. First, we show that a classical result

due to Lenstra and Rinnooy Kan [55] can be adapted to prove that already the variant of *SCS* without communication delays and processing times equal to one or two is NP-hard. This already implies strong NP-hardness. Remember that we did show in Section 2.2 that *SCS* without communication delays and with unit processing times can be solved in polynomial time. Hence, it seems natural to consider the problem variant with unit processing times and communication delays. We prove this problem to be NP-hard as well via an intricate reduction from 3*SAT* that can be considered the main result of this section. Lastly, we show that the latter reduction can be easily modified to get a strong inapproximability result regarding the general variant of *SCS* and the cost objective.

### 2.5.1 No Delays and Two Sizes

We show strong hardness for the case without communication delays and $p_c(j), p_s(j) \in \{1,2\}$ for each job $j$. The reduction is based on a classical result due to Lenstra and Rinnooy Kan [55].

Let $G = (V,E)$, $k$ be a clique instance with $\mid E \mid > \binom{k}{2}$, and let $n = \mid V \mid$ and $m = \mid E \mid$. We construct an instance of the cloud server problem in which the communication delays all equal zero and both the deadline and the cost bound are $2n+3m$. There is one vertex job $J(v)$ for each node $v \in V$ and one edge job $J(e)$ for each edge $e \in E$ and $J(\{u,v\})$ is preceded by $J(u)$ and $J(v)$. The vertex jobs have a size of 1 and the edge jobs a size of 2 both on the server and on the cloud.

Furthermore, there is a dummy structure. First, there is a chain of $2n+3m$ many jobs called the anchor chain. The $i$-th job of the anchor chain is denoted $A(i)$ for each $i \in \{0,\dots 2n+3m-1\}$ and has size 1 on the cloud and size 2 on the server. Next, there are gap jobs each of which has a size of 1 both on the server and the cloud. Let $k^* = \binom{k}{2}$ and $v \prec w$ indicate that an edge from $v$ to $w$ is included in the task graph. There are four types of gap jobs, namely $G(1,i)$ for $i \in \{0,\dots k-1\}$ with edges $A(2i) \prec G(1,i) \prec A(2(i+1))$, $G(2,i)$ for $i \in \{0,\dots k^*-1\}$ with $A(2k+3i+1) \prec G(2,i) \prec A(2k+3(i+1))$, $G(3,i)$ for $i \in \{0,\dots (n-k)-1\}$ with $A(2k+3k^*+2i) \prec G(3,i) \prec A(2k+3k^*+2(i+1))$, and $G(4,i)$ for $i \in \{0,\dots (m-k^*)-1\}$ with $A(2n+3k^*+3i+1) \prec G(4,i) \prec A(2n+3k^*+3(i+1))$ for $i < (m-k^*)-1$ and $A(2n+3m-2) \prec G(4,(m-k^*)-1)$. Lastly, there are the source and the sink which precede or succeed all of the above jobs, respectively.

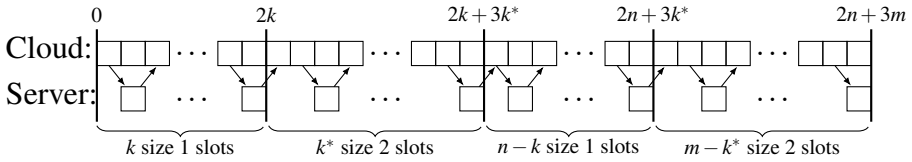



Figure 2.4: The dummy structure for the reduction from the clique problem to a special case of *SCS*. Time flows from left to right, the anchor chain jobs are positioned on the cloud, and the gap jobs on the server.

**Lemma 2.3** There is a $k$-clique, if and only if there is a schedule with length and cost at most $2n+3m$.

*Proof.* First, note that in a schedule with a deadline of $2n+3m+1$ the anchor chain has to be scheduled completely on the cloud. If the schedule additionally satisfies the cost bound,

all the other jobs have to be scheduled on the server. Furthermore, for the gap and anchor chain jobs there is only one possible time slot due to the deadline. In particular, $A(i)$ starts at time $i$, $G(1,i)$ at time $2i+1$, $G(2,i)$ at time $2k+3i+2$, $G(3,i)$ at time $2k+3k^*+2i+1$, and $G(4,i)$ at time $2n+3k^*+3i+2$. Hence, there are $k$ length 1 slots positioned directly before the $G(1,i)$ jobs left on the server, as well as, $k^*$ length 2 slots directly before the $G(2,i)$ jobs, $n-k$ length 1 slots directly before the $G(3,i)$ jobs, and $m-k^*$ length 2 slots directly before the $G(2,i)$ jobs (see also Figure 2.4). The $m$ edge jobs have to be scheduled in the length 2 slots, and hence the vertex jobs have to be scheduled in the length 1 slots.

$\Longrightarrow$ : Given a $k$-clique, we can position the $k$ clique vertices in the first $k$ length 1 slots, the corresponding $k^*$ edges in the first length 2 slots, the remaining vertex jobs in the remaining length 1 slots, and the remaining edge jobs in the remaining length 2 slots.

$\Longleftarrow$ : Given a feasible schedule, the vertices corresponding to the first length 1 slots have to form a clique. This is the case because there have to be $k^*$ edge jobs in the first length 2 slots and all of their predecessors are positioned in the first length 1 slots. This is only possible if these edges are the edges of a $k$-clique. ■

Hence, we have:

**Theorem 2.10** The *SCS* problem with job sizes 1 and 2 and without communication delays is strongly NP-hard.

In the above reduction, the server and the cloud machines are unrelated relative to each other due to the different sizes of the anchor chain jobs. However, it is easy to see that the reduction can be modified to a uniform setting where the cloud machines have a speed of 2 and the server speed of 1. If we allow communication delays, even identical machines can be achieved.

### 2.5.2 Unit Size and Unit Delay ($SCS^1$)

We consider a unit time variant of our model in which all $p_c = p_s = 1$ and all $c = 1$. Note here, that this also implies that the server and the cloud are identical machines (the cloud still produces costs, while the server does not). We again look at the decision variant of the problem: Is there a schedule with a cost smaller or equal to $b$ while adhering to the deadline $d$.

**Theorem 2.11** The $SCS^1$ problem is strongly NP-hard.

We give a reduction $3SAT \leq_p SCS^1$. Let $\phi$ be any boolean formula in 3-CNF, denote the variables in $\phi$ by $\mathcal{X} = \{x_1, x_2, \ldots, x_m\}$ an the clauses by $\mathcal{C} = \{C_1^\phi, C_2^\phi, \ldots, C_n^\phi\}$. Before we define the reduction formula we want to give an intuition and a few core ideas used in the reduction.

The main idea is that we ensure that nearly everything has to be processed on the cloud, there are only a few select jobs that can be handled by the server. For each variable, there will be two jobs, of which one can be processed on the server, and the selection will represent an assignment. For each clause, there will be a job per literal in that clause, only one of which can be processed on the server, and only if the respective variable job is 'true'. Only if for each variable and for each clause one job is handled by the server the schedule will adhere to both the cost and the time limits.

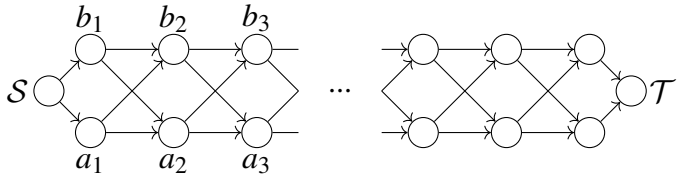



Figure 2.5: Schematic representation of an anchor chain.

A core technique of the reduction is the usage of an anchor chain. An anchor chain of length $l$ consists of two chains of the same length $l := d-2$, where we interlock the chains by inserting $(a_i, b_{i+1})$ and $(b_i, a_{i+1})$ for two parallel edges $(a_i, a_{i+1})$ and $(b_i, b_{i+1})$. The source $\mathcal{S}$ is connected to the two start nodes of the anchor chain, and the two nodes at the end of the chain are connected to $\mathcal{T}$.

**Lemma 2.4** If the task graph of a $SCS^1$ problem contains an anchor chain, every valid schedule has to schedule all but one of $a_1$,$b_1$ and one of $a_l$,$b_l$ on the cloud. For every job $a_i, b_i$ $1 < i < l$ the timestep in which it will finish processing on the cloud in every valid schedule is $i+1$.

Finally we give the reduction function $f(\phi) = G, d, b$, where $G = (\mathcal{J}, E)$. Set $d = 12+m+n$ and $b = \mid \mathcal{J} \mid -(2+m+n)$. We define $G$ by constructively giving which jobs and edges are created by $f$. Create an anchor chain of length $d-2$, this will be used to limit parts of a schedule to certain time frames. Note that by Lemma 2.4 we know that every valid schedule of $G = (\mathcal{J}, E), d, b$ has every node pair of the anchor chain (besides the first and last) on the cloud at a specific fixed timestamp. More specifically, the completion time of $a_i$ and $a_{i+j}$ differ by exactly $j$ time units. For each variable $x_i \in \mathcal{X}$ create two jobs $j_{x_i}$ and $j_{\bar{x}_i}$ and edges $(a_{1+i}, j_{x_i}), (a_{1+i}, j_{\bar{x}_i})$ and $(j_{x_i}, a_{5+i}), (j_{\bar{x}_i}, a_{5+i})$. For each clause $C_p^\phi$ create a clause job $j_{C_p^\phi}$ and edges $(a_{7+m+p}, j_{C_p^\phi})$ and $(j_{C_p^\phi}, a_{9+m+p})$. Let $L_1^p, L_2^p, L_3^p$ be the literals in $C_p^\phi$. Create jobs $j_{L_1^p}, j_{L_2^p}, j_{L_3^p}$ and edges $(j_{L_1^p}, C_p^\phi), (j_{L_2^p}, C_p^\phi), (j_{L_3^p}, C_p^\phi)$ for these literals. For every literal job $j_{L_1^p}$ connect it to the corresponding variable job $j_{x_i}$ or $j_{\bar{x}_i}$ by a chain of length $1+(m-i)+p$. Also, create an edge from $a_{3+i}$ to the start of the created chain and an edge from the end of the chain to $a_{6+m+p}$.

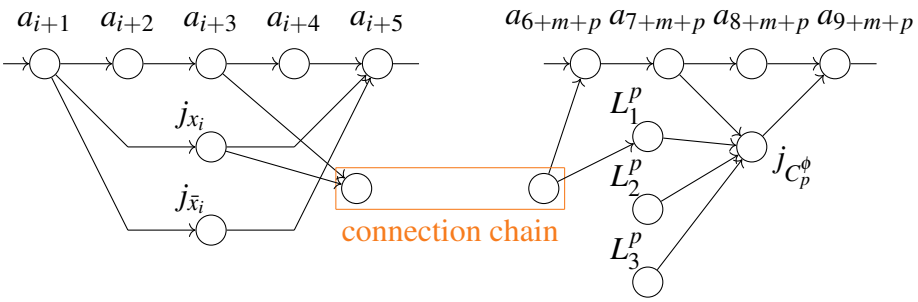



Figure 2.6: Schematic representation of the variable and clause gadgets and their connection.

It remains to show that there is a schedule of length at most $d$ with costs at most $b$ in $f(\phi) = G, d, b$ if and only if there is a satisfying assignment for $\phi$.

**Lemma 2.5** In a deadline adhering schedule for $f(\phi) = G,d,b$ every job in the anchor chain (except one at the front and one at the end), every job in the variable and clause literal connecting chains and every clause job has to be scheduled on the cloud.

*Proof.* By Lemma 2.4 we already know that every node in the anchor chain except one of $a_1$,$b_1$ and one of $a_l$,$b_l$ has to be scheduled on the cloud. We also know that the jobs in the anchor chain have fixed timesteps in which they have to be processed. We look at some chain and their connection to the anchor chain. The start of the chain of length $1+(m-i)+p$ is connected to $a_{3+i}$, the end to $a_{6+m+p}$. Between the end of $a_{3+i}$ and the start of $a_{6+m+p}$ are $6+m+p-1-(3+i) = 2+m+p-i$ timesteps. So with the processing time required to schedule all $1+(m-i)+p$ jobs of the chain, there is only one free timestep, but we would need at least 2 free timesteps to cover the communication cost to and from the server. (Recall here that both $a_{3+i}$ and $a_{6+m+p}$ have to be processed on the cloud). The same simple argument fixes each clause job on the server to a specific timestep. ■

**Lemma 2.6** In a deadline adhering schedule for $f(\phi) = G,d,b$ only one of $j_{x_i}$ and $j_{\bar{x}_i}$ can be processed on the server for every variable $x_i \in \mathcal{X}$. The same is true for $j_{L_1^p}, j_{L_2^p}, j_{L_3^p}$ of clause $C_p^\phi$.

*Proof.* $j_{x_i}$ and $j_{\bar{x}_i}$ are both fixed to the same time interval via the edges $(a_{1+i}, j_{x_i})$, $(a_{1+i}, j_{\bar{x}_i})$ and $(j_{x_i}, a_{5+i}), (j_{\bar{x}_i}, a_{5+i})$. Since $a_{1+i}$ and $a_{5+i}$ will be processed on the cloud and keeping communication delays in mind, only the middle of the three timesteps in between can be used to schedule $j_{x_i}$ or $j_{\bar{x}_i}$ on the server. Since the server is only a single machine only one of them can be processed on the server. Note here that the other job can be scheduled a timestep earlier (on the cloud) which we will later use. The argument for $j_{L_1^p}, j_{L_2^p}, j_{L_3^p}$ works analogously to the statement above. ■

**Lemma 2.7** There is a deadline adhering schedule for $f(\phi) = G,d,b$ with costs of $\mid \mathcal{J} \mid -(2+m+n)$ if and only if there is a satisfying assignment for $\phi$. The variable jobs processed on the cloud represent this satisfying assignment

*Proof.* From Lemma 2.4, Lemma 2.5 and Lemma 2.6 we can infer that a schedule with costs of $\mid \mathcal{J} \mid -(2+m+n)$ has two jobs of the anchor chain, one job for each pair of variable jobs and one job per clause on the server. Two jobs of the anchor chain can always be placed on the server, and the choice of variable jobs is also free. It remains to show, that we can only schedule a literal job per clause on the server if and only if the respective clause is fulfilled by the assignment inferred by the variable jobs.

The clause job $j_{C_p^\phi}$ of $C_p^\phi$ has to be processed in timestep $9+m+p$ (between $a_{7+m+p}$ and $a_{9+m+p}$). Therefore, $j_{L_1^p}$ has to be processed no later than $8+m+p$ or $7+m+p$ if it is processed on the cloud or server respectively. Let $j_{x_i}$ be the variable job connected to $j_{L_1^p}$ via a connection chain.

If $j_{x_i}$ is *true* (scheduled on the cloud), it can finish processing at timestep $3+i$, which does not delay the start of the connection chain (which is connected to $a_{3+i}$, finishing in timestep $4+i$). This means that the chain can finish in timestep $4+i \ + \ 1+(m-i)+$

$p = 5+m+p$, the timestep $6+m+p$ can be used for communication, allowing $j_{L_1^p}$ to be processed by the server in $7+m+p$.

If $j_{x_i}$ is *false* (scheduled on the server), it finishes processing at timestep $4+i$, which, combined with the induced communication delay, delays the start of the chain by 1. Therefore, the chain only finishes in timestep $6+m+p$, and $j_{L_1^p}$ has to be processed on the cloud since there is not enough time for the communication back and forth. Trivially, the same argument holds true for $j_{L_2^p}$ and $j_{L_3^p}$. ■

It should be easy to see that the reduction function $f$ is computable in polynomial time. Combined with Lemma 2.7, this concludes the proof of our reduction $3SAT \leq_p SCS$[1]. The correctness of Theorem 2.11 directly follows from that.

### 2.5.3 Inapproximability of the General Case

Adapting the previous reduction we can show an even stronger result for the general case of *SCS*. Basically, we are able to degenerate the reduction output in a way, that a satisfying assignment results in a schedule with a cost of 0, while every other assignment (schedule) has costs of at least 1. It should be obvious, that this also means that there is no (cost minimization) approximation algorithm for this problem with a fixed multiplicative performance guarantee if P $\neq$ NP.

This reduction uses processing times and communication delays of 0, $\infty$ and values in between. Note that $\infty$ can simply be replaced by $d+1$. To keep the following part readable we again substitute "an edge $(j,j')$ with communication delay $c(j,j') = k$" simply by "an edge $c(j,j') = k$".

We follow the same general structure (an anchor chain, variable-, clause- and connection gadgets). The *anchor chain* now looks as follows: For every timestep create two jobs $a_i$ and $a_i'$ with $p_s(a_i) = 0$, $p_c(a_i) = \infty$, $p_s(a_i') = \infty$, $p_c(a_i') = 0$ and an edge $c(a_i, a_i') = 0$. These chain links are then connected by an edge $c(a_i', a_{i+1}) = 1$. Finally we create $c(\mathcal{S}, a_1) = 1$ and $c(a_d, \mathcal{T}) = 0$. Every schedule will process $a_i$ and $a_i'$ in timestep $i$ on the server and the cloud respectively. This gives us anchors to the server and to the cloud for every timestep, without inducing congestion or costs. Since the anchor jobs themselves have a processing time of 0, the "usable" time interval between some $a_i$ and $a_{i+1}$ is one full timestep. By this construction, waiting in the anchor chain is purely realized by communication and, therefore, cost free.

For each variable $x_i \in \mathcal{X}$ create two jobs $j_{x_i}$, $j_{\bar{x}_i}$ with $p_s(j_{x_i}) = p_s(j_{\bar{x}_i}) = 1$ and $p_c(j_{x_i}) = p_c(j_{\bar{x}_i}) = 0$. Create edges $c(a_i, j_{x_i}) = 1$, $c(a_i, j_{\bar{x}_i}) = 1$ and $c(j_{x_i}, a_{i+1}) = 0$, $c(j_{\bar{x}_i}, a_{i+1}) = 0$. In short, only one of them can be processed on the server, the other on the cloud. Both will finish in timestep $i+1$, the one processed on the server is *true*, therefore processing both on the cloud is possible, but not helpful.

For each clause $C_p^\phi$ create a clause job $j_{C_p^\phi}$ with $p_s(j_{C_p^\phi}) = \infty$, $p_c(j_{C_p^\phi}) = 0$ and edges $c(a'_{5+m+3p}, j_{C_p^\phi}) = \infty$ and $c(j_{C_p^\phi}, a'_{6+m+3p}) = \infty$. This means, that $j_{C_p^\phi}$ has to finish processing by timestep $6+m+3p$. Let $L_1^p, L_2^p, L_3^p$ be the literals in $C_p^\phi$. Create jobs $j_{L_1^p}, j_{L_2^p}, j_{L_3^p}$ each with $p_c = p_s = 1$ and edges $c(j_{L_1^p}, C_p^\phi) = 0$, $c(j_{L_2^p}, C_p^\phi) = 0$, $c(j_{L_3^p}, C_p^\phi) = 0$ for these literals. Create edges $c(a_3+m+3p, j_{L_1^p}) = 0$, $c(a_3+m+3p, j_{L_2^p}) = 0$ and $c(a_3+m+3p, j_{L_3^p}) = 0$, so that, in theory, all three of the literal jobs can be processed on the server, finishing in timesteps $4+m+3p$, $5+m+3p$ and $6+m+3p$ respectively. Lastly, for every literal

job $j_{L_1^p}$ connect it to the corresponding variable job $j_{x_i}$ (or $j_{\bar{x}_i}$) by an edge with a communication delay of $m-i+3p+3$. Since $j_{x_i}$ (or $j_{\bar{x}_i}$) finish processing in timestep $i+1$, this means that $j_{L_1^p}$ can start no earlier than $m+3p+4$ (and therefore finish processing in $5+m+3p$), if $j_{x_i}$ (or $j_{\bar{x}_i}$) were processed on the cloud.

Recall here, that a variable job being scheduled on the server denotes that it is *true*. So only a literal job that evaluates to true, can be scheduled so that it finishes processing in timestep $4+m+3p$ on the cloud.

It follows directly, that a schedule for this construction will have costs of 0 if and only if the assignment derived from the placement of the variable jobs fulfills every clause.

**Theorem 2.12** There is no approximation algorithm for the deadline-constrained cost minimization *SCS* problem that has a fixed performance guarantee, assuming that P $\neq$ NP.

## 2.6 Algorithms for $SCS^1$ and Instances Without Delays

We explore some simple algorithms on instances with arbitrary task graphs but constraints on job sizes and communication delays. We start by examining the unit case $SCS^1$ (all processing times and communication delays are one) for which we showed strong NP-hardness in Theorem 2.11. We give both an algorithm for the budget-constrained makespan minimization as well as the deadline-constrained cost minimization. Secondly, we give an algorithm for budget-constrained makespan minimization for instances with arbitrary processing times, but without communication delays.

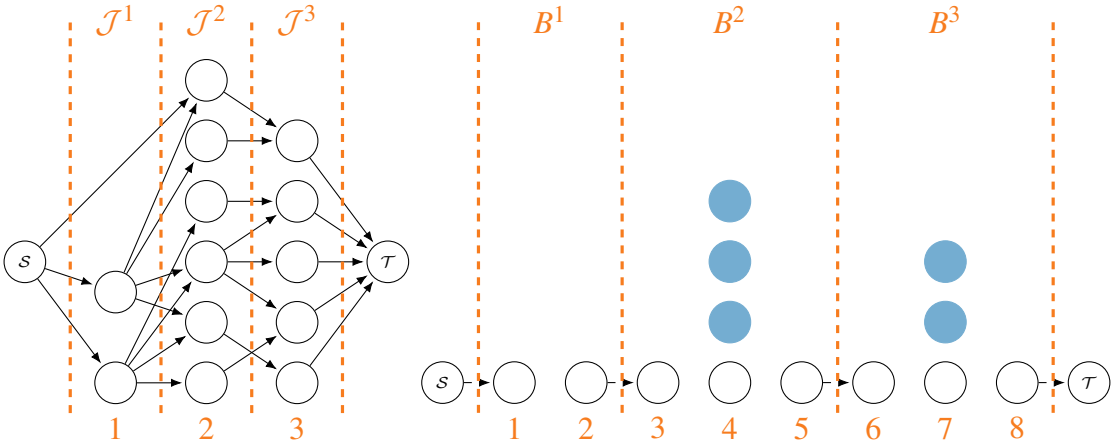
### 2.6.1 A 3-Approximation (Makespan) for $SCS^1$

Since we are looking at the unit version of the problem, we can make a general observation about the relevance of communication delays. In timestep $t$, we only need to care about communication delays of jobs that were processed in timestep $t-1$. In other words, after a single timestep, it is irrelevant for future jobs, if a job was processed on the server or on the cloud. We will make heavy use of that by creating a schedule that schedules jobs in blocks, beginning and ending with a synchronization step and a parallelization step in the middle.

We start by constructing a hypothetical schedule $\pi^h$, where every job starts as soon as all of its predecessors finished processing. This hypothetical schedule is neither bound in the number of parallel tasks, nor in its cost. For every $t$ from 1 to $mspan^h$ let $\mathcal{J}^t$ be the set of jobs finished in timestep $t$ by $\pi^h$. Following the definition, we know that jobs in $\mathcal{J}^t$ have no predecessors in $\mathcal{J}^t$ or $\mathcal{J}^{t+i}$ for any $i$.

**Algorithm:** We construct our schedule $\pi$ as follows:

- *remaining-budget* $= b$
- For every t in $\{1, mspan^h\}$:
  - If $|\mathcal{J}^t| <= 3$, place all jobs in $\mathcal{J}^t$ on the server, in any order.
  - Else, place 3 jobs from $\mathcal{J}^t$ on the server, call the middle one $j_t$.
    As long as *remaining-budget* $> 0$ and there are still unscheduled jobs in $\mathcal{J}^t$, place one of those jobs on the cloud, on the same timestep as $j_t$. Repeat this until all jobs in $\mathcal{J}^t$ are scheduled, or *remaining-budget* $= 0$. If *remaining-budget* $= 0$, place all remaining jobs from $\mathcal{J}^t$ onto the server.




(a) Hypothetical schedule $\pi^h$ (b) Resulting schedule $\pi$, blue jobs scheduled on the cloud

Figure 2.7: Sets $\mathcal{J}^t$ of $\pi^h$ and the corresponding blocks $B^t$ of $\pi$

**Theorem 2.13** There is a 3-approximation for the budget-constrained makespan minimization $SCS^1$ problem.

*Proof.* We start by showing that the given algorithm does indeed construct a valid schedule. Our algorithm schedules jobs from some $\mathcal{J}^t$ strictly after all jobs from any $\mathcal{J}^{t-i}$. Call the time frame in which jobs from $\mathcal{J}^t$ are scheduled block $B^t$. Since the blocks are non-overlapping and scheduled in increasing order, no precedence constraints will be violated. Furthermore, a block with jobs on the cloud consists of at least 3 timesteps. In both the first and third timestep a single job is placed on the server (synchronization steps), while jobs on the cloud are always scheduled on the second timestep of a block (parallelization step). That gives us, that the first and third timestep of a block can always be used to fulfill all arising communication delays from and to the second timestep since there are no edges between jobs of the same block.

We prove the approximation ratio by looking at the *remaining-budget* at the end.
Case *remaining-budget* $= 0$: That means that we process as many jobs on the cloud as possible. Furthermore, by construction there is no timestep where the server idles, giving us $mspan_{ALG} = n - b$. Since the optimal solution is also limited by the budget and the sequentiality of the server, we know that $n - b = mspan_{OPT} = mspan_{ALG}$.
Case *remaining-budget* $\neq 0$: That means that our algorithm never got out of budget while placing blocks. We can conclude that each $B^t$ consists of at most 3 timesteps, which directly infers that $mspan_{ALG} \leq 3 \cdot mspan^h$. Since $mspan^h$ is the optimal makespan under the assumption that we only have to adhere to precedence constraints, we know that $mspan^h \leq mspan_{OPT}$. Combined we get $mspan_{ALG} \leq 3 \cdot mspan_{OPT}$, concluding our proof. ■

### 2.6.2 A $\frac{1+\varepsilon}{2\varepsilon}$-Approximation (Cost) for $SCS^1$ with Resource Augmentation

We use resource augmentation and ask: given a $SCS^1$ problem instance with deadline $d$, find a schedule in polynomial time that has a makespan of at most $(1+\varepsilon) \cdot d$ that *approximates* the optimal cost in regards to the actual deadline $d$.

**Algorithm:** If there is a chain of length $d$ or $d-1$, that chain has to be scheduled on the server, since there is no time for the communication delay. For instances with a chain of size $d$ that is trivially optimal, for those with $d-1$ we can check in polynomial time if any other job also fits on the server, again, finding an optimal solution. From now we assume that there is no chain of length more than $d-2$.

First, construct a schedule that places every job on the cloud, as fast as possible. The resulting schedule from timestep 1 to $(1+\varepsilon)\cdot d$ looks as follows: one timestep of communication, at most $d-2$ timesteps of processing on the server, another timestep for communication followed by at least $\varepsilon d$ empty timesteps. Now pull (one of) the last job(s) that is processed on the cloud to the last empty timestep and process it on the server instead. Repeat this process until the last job can not be moved to the server anymore. Do the whole procedure again, but this time starting with the cloud schedule at the end of the schedule, and each time pulling the first job to the beginning. Keep the result with lower costs.

Note that one can always fill the timestep being used solely for communicating from the server to the cloud by processing one job on the server, which otherwise would be one of the first jobs being processed on the cloud (the same holds for the other direction).

**Theorem 2.14** There is a $\frac{1+\varepsilon}{2\varepsilon}$-approximation for the deadline-constrained cost minimization $SCS^1$ problem with resource augmentation that allows a deadline of $(1+\varepsilon)\cdot d$ instead of $d$.

*Proof.* Case $n \leq (1+\varepsilon)d$: The algorithm places all jobs on the server, the cost is 0 and therefore optimal.
Case $(1+\varepsilon)d < n < (1+2\varepsilon)d$: Assume that the preliminary cloud-only schedule needs $d-2$ timesteps on the cloud, if that is not the case, we *stretch* the schedule to that length. There are $n$ jobs distributed onto $d-2$ timesteps. Therefore, either from the front or from the end, there is an interval of length $\frac{d}{2}-1$ with at least $\frac{d}{2}-1$ and at most $\frac{n}{2} < \frac{(1+2\varepsilon)d}{2} = \frac{d}{2}+\varepsilon d$ many jobs. The algorithm will schedule those at most $\frac{d}{2}+\varepsilon d-1$ jobs to the $\frac{d}{2}-1$ plus the free $\varepsilon d$ many time slots. If the interval included less than $\frac{d}{2}+\varepsilon d-1$ jobs, it will simply continue until the $\frac{d}{2}-1+\varepsilon d$ timesteps are filled with jobs being processed on the server. With the one job we can process on the server during the communication timestep we process $\frac{d}{2}+\varepsilon d$ jobs on the server and have costs of $n-(\frac{d}{2}+\varepsilon d)$. An optimal solution has costs of at least $n-d$. For $\varepsilon \geq 0.5$ it holds that: $cost_{ALG} = n-(\frac{d}{2}+\varepsilon d) \leq n-d \leq cost_{OPT}$, otherwise:

$$\frac{cost_{ALG}}{cost_{OPT}} \leq \frac{n-(\frac{d}{2}+\varepsilon d)}{n-d} \leq \frac{(1+\varepsilon)d-(\frac{d}{2}+\varepsilon d)}{(1+\varepsilon)d-d} \leq \frac{0.5d}{\varepsilon d} = \frac{1}{2\varepsilon}$$

Case $(1+2\varepsilon)d \leq n$: In this case, we simply observe that our algorithm places at least $\varepsilon d$ many jobs on the server. For $\varepsilon \geq 1$ it holds that: $cost_{ALG} = n-\varepsilon d \leq n-d \leq cost_{OPT}$, otherwise:

$$\frac{cost_{ALG}}{cost_{OPT}} \leq \frac{n-\varepsilon d}{n-d} \leq \frac{(1+2\varepsilon)d-\varepsilon d}{(1+2\varepsilon)d-d} = \frac{d+\varepsilon d}{2\varepsilon d} = \frac{1+\varepsilon}{2\varepsilon}$$

■

### 2.6.3 A 2-Approximation (Makespan) on Identical Machines and no Delays

We design a simple heuristic for the case in which the server and the cloud machines behave the same (except costs), that is, $p_c(j) = p_s(j)$ for each job $j$ (except for the source and sink), and the communication delays all equal zero. In this case, we may define the length of a chain in the task graph as the sum of the processing times of the jobs in the chain.

**Algorithm:** The first step in the algorithm is to identify a longest chain in the task graph, which can be done in polynomial time. The jobs of the longest chain are scheduled on the server and the remaining jobs on the cloud each as early as possible. Now, the makespan of the resulting schedule is the length of a longest chain, which is optimal (or better) and there are no idle times on the server. However, the schedule may not be feasible since the budget may be exceeded. Hence, we repeatedly do the following: If the budget is still exceeded, we pick a job scheduled on the cloud with maximal starting time and move it onto the server right before its first successor (which may be the sink). Some jobs on the server may be delayed by this but we can do so without causing idle times.

If all the processing times are equal this procedure produces an optimal solution, otherwise, there may be an additive error of up to the maximal job size. Hence, we have:

**Theorem 2.15** There is a 2-approximation for the budget-constrained makespan minimization *SCS* problem without communication delays and identical server and cloud machines.

It is easy to see, that the analysis is tight considering an instance with three jobs: One with size $b$, one with size $b+\varepsilon$, and one with size $2\varepsilon$. The first job precedes the last one. Our algorithm will place everything on the server, while the first job is placed on the cloud in the optimal solution.

Note that we can take a similar approach to find a solution with respect to the cost objective by placing more and more jobs on the server as long as the deadline is still adhered to. However, an error of one job can result in an unbounded multiplicative error in the objective in this case. On the other hand, it is easy to see that in the case of unit processing times, there will be no error at all in both procedures yielding:

**Corollary 2.2** The variant of *SCS* without communication delays and unit processing times can be solved in polynomial time with respect to both the makespan and the cost objective.

## 2.7 Generalizations of Server Cloud Scheduling

In this section, we introduce some generalizations to the *SCS* problem. We consider different aspects like multiple clouds and server machines and direction-specific delays. We sketch how to adapt our algorithms for $SCS^e$ and $SCS^\Psi$ to cover those new generalizations.

### 2.7.1 Changes in the Definitions

We shortly define the changes to the model that we explore in this section.

**Machine Model:** So far we imagined a single server machine and one homogeneous cloud in our problem definition. Now, instead of a single server machine, there can be any (constant) number of identical server machines: $\text{SERVER} = \{s_1, \ldots, s_z\}$. Instead of one homogeneous cloud, there can be any number of different cloud contexts: $\text{CLOUDS} = \{c_1, \ldots, c_k\}$. Each cloud context still consists of an unlimited number of parallel machines.

**Jobs:** Jobs are still given as a task graph $G = (\mathcal{J}, E)$. A job $j \in \mathcal{J}$ has processing time $p_s(j)$ on any server machine and processing time $p_{c_i}(j)$ on a machine of cloud context $c_i$. An edge $e = (i, j)$ and machine contexts $m_1, m_2 \in \{s, c_1, \ldots, c_k\}$ have a communication delay of $c_{m_1 \triangleright m_2}(i, j) \in \mathbb{N}_0$, which means, that after job $i$ finished on a machine of type $m_1$, $j$ has to wait an additional $c_{m_1 \triangleright m_2}(i, j)$ timesteps before it can start on a machine of type $m_2$. For $m_1 = m_2$ we set $c_{m_1 \triangleright m_2}(i, j) = 0$. Note that this function does not need to be symmetric, e.g. $c_{m_1 \triangleright m_2}(i, j)$ and $c_{m_2 \triangleright m_1}(i, j)$ may be unequal.

**Costs and Schedules:** Previously we defined cost simply by "time spent on the cloud". While considering multiple clouds, that is not sensible anymore. A faster cloud will not be universally cheaper than a slower one. We define a cost function based on the cloud context and job, $cost : \mathcal{J} \times \text{CLOUDS} \mapsto \mathbb{N}_0$. A schedule still consists of $C : \mathcal{J} \mapsto \mathbb{N}_0$ (maps jobs to their completion time), but instead of a partition we give a mapping function $\eta : \mathcal{J} \mapsto \{s_1, \ldots, s_z\} \cup \{c_1, \ldots, c_k\}$. Note that $s_i$ refers to one specific server machine, while $c_i$ refers to a cloud context, consisting of infinitely many machines.

We call a schedule $\pi = (C, \eta)$ valid if and only if the following conditions are met:

a) There is always at most one job processing on each server:

$$\forall_{i,j \in \mathcal{J}, i \neq j : \eta(i) = \eta(j) \in \text{SERVER}} : (C(i) \leq C(j) - p_s(j)) \vee (C(i) - p_s(i) \geq C(j))$$

b) Tasks are not started before the previous tasks has been finished/ the required communication is done:

$$\forall_{(i,j) \in E} : (C(i) + c_{\eta(i) \triangleright \eta(j)}(i, j) \leq C(j) - p_{\eta(j)} j)$$

The makespan (*mspan*) of a schedule is still given by the completion time of the sink $\mathcal{T}$: $C(\mathcal{T})$. The cost (*cost*) of a schedule is given by:

$$\sum_{j \in jobs : \eta(j) \in clouds} cost(j, \eta(j)).$$

### 2.7.2 Revisiting $SCS^e$

We briefly sketch how to adapt the algorithm from Section 2.3 to incorporate the previously defined changes in the model. We will use the observations, that multiple server machines only affect the scheduling of parallel parts and that we can always calculate an optimal cloud location for a job in a given situation (part of the schedule, time frame and location of predecessor and successor).

**Theorem 2.16** There is a $(4+\varepsilon)$-approximation algorithm for the budget-constrained makespan minimization problem on extended chains, even when there are $z$ server machines, $k$ different cloud contexts, the communication delays are directionally dependent on the machine context, and costs are given as an arbitrary cost function $cost : \mathcal{J} \times \text{CLOUDS} \mapsto \mathbb{N}_0$.

*Proof.* We adapt the pseudo-polynomial algorithm from Section 2.3 that given a feasible makespan estimate $T (T \geq mspan_{OPT})$ calculates a schedule with makespan of at most $\min\{2T, 2mspan_{OPT}\}$, such that it incorporates the changes to the model and calculates a schedule with makespan of at most $\min\{4T+\varepsilon', 4mspan_{OPT}+\varepsilon'\}$. The only change in the state description is that $loc \in \{s, c_1, \ldots, c_k\}$ instead of $loc \in \{s, c\}$. As the state description is used for the chain parts of the extended chain, we do not differentiate the server machines here. The creation of the state extension list $\text{EXTENSIONS}^j$ (each of form $[\Delta t, loc_{j-1} \to loc_j] = cost$), has the following changes:

- Instead of the four combinations $s \to s$, $s \to c$, $c \to s$, $c \to c$, we consider all combinations from $\{s, c_1, \ldots, c_k\} \times \{s, c_1, \ldots, c_k\}$.
- Substitute the corresponding values, for example $[p_c(j) + c(j-1, j),\ s \to c] = p_c(j)$ becomes $[p_{c_i}(j) + c_{s \triangleright c_i}(j-1, j),\ s \to c_i] = cost(j, c_i)$.
- If there is a parallel subgraph between $j-1$ and $j$ we adapt the calculation in the following way:
  - Calculate $\Delta^{max}$ as before (the sum over all processing times on the server plus the biggest relevant in- and outgoing communication delays)
  - Iterate over $\Delta^i$ in $\{0, \ldots, \Delta^{max}\}$:
    - As before, check for each job if it fits: (1) only on the servers, (2) not on the servers but on at least one cloud context, (3) on both, (4) on none. If at least one job falls into (4) break.
    - Calculate for each job $j$ in (2) or (3) the cheapest fitting option to schedule that job on some available cloud in the time frame $\Delta^i$. Use that cost $c_j$ for $j$ for the remainder of the iteration.
    - Greedily put jobs in (1) onto server machines (1 to $k$) until the current server has load $\geq \Delta^i$, proceed with the next machine and so on. If not all jobs in (1) can be placed this way break, as there is not enough space to place jobs on the server that do not fit on the cloud in the given time frame.
    - Sort the jobs in (3) by their ratio of cost $c_j$ to processing time on the server (highest to lowest cost per time). Continue by greedily placing those on the server machines as before. When all jobs in (3) are placed, or all server machines have load $\geq \Delta^i$, put all remaining jobs from (3) on their corresponding cheapest cloud context.
    - Put all jobs from (2) on their corresponding cheapest cloud context.
    - insert time in the front and back corresponding to the biggest communication delay invoked by the (sub-)schedule for the parallel part.

The rest of the algorithm behaves as before. The changes to state extensions spanning a parallel subgraph calculate solutions that have at most optimal cost for a time frame of $\Delta^i$, while using a time frame of $4\Delta^i$. The 4 times correspond to: at most $2\Delta^i$ time for all in- and outgoing communication delays since the communication delays have to fit into $\Delta^i$ to be considered, at most $2\Delta^i$ time for our greedy packing of the server machines since we can add a job of size $\Delta^i$ to a machine currently having load $\Delta^i - \varepsilon$. It should be easy to see that

the greedy packing of "highest cost jobs", with what is essentially resource augmentation of a multiple knapsack problem, gives at most optimal cost. Note that we could also utilize a PTAS for multiple knapsack here to stay in a time frame of $3\Delta^i$, but we want to find a solution with optimal cost (or lower), to remain strictly budget-adhering. It remains to simply use the same scaling technique used in Section 2.3 to get the $4+\varepsilon$-approximation. ■

If the communication delays are (bounded by some) constant the result can be easily adapted to yield a $2+\varepsilon$-approximation, by getting rid of the added time for communication delays.

### 2.7.3 Revisiting $SCS^{\Psi}$

In a similar vein as the previous subsection, we briefly sketch how to adapt the results from Section 2.4 to include most of the previously defined model generalizations. Naturally, we still require the *maximum cardinality source and sink dividing cut* ($\psi$) to be bounded by a constant. In contrast to the previous result, we require the number of server machines to be a constant.

**Theorem 2.17** There is an FPTAS for the budget-constrained makespan minimization problem for graphs with a constant maximum cardinality source and sink dividing cut ($\psi$), even when there are a constant number of server machines, $k$ different cloud contexts, the communication delays are directionally dependent on the machine context, and costs are given as an arbitrary cost function $cost : \mathcal{J} \times \text{CLOUDS} \mapsto \mathbb{N}_0$.

*Proof.* We make the following two changes to the state definition: We consider $loc_j \in \{s, c_1, \ldots, c_k\}$ instead of $loc_j \in \{s, c\}$, we track the unused time of every server machine individually so instead of a single $f_s$ the state contains $f_{s^1}, \ldots, f_{s^z}$. The dynamic program needs only minor tweaks. When iterating through the jobs that are open (and of which all predecessors have been processed) use the server $s^i$ with the smallest fitting $f_{s^i}$ and set $f_{s^i} = 0$. Instead of checking if the job fits on "the cloud" we simply go through all clouds, and add corresponding states for each fitting location. While calculating the value of a state use the new cost function *cost* instead of $p_c$, while checking if a job fits we use the directional communication delays. After a full iteration, increase each $f_{s^i}$ by one (instead of only increasing the singular $f_s$). It should be easy to see, that these adaptations do not change the correctness of the algorithm. The runtime (after the rounding technique) naturally increases to $\text{poly}(n^z, k, \frac{1}{\varepsilon})$, which is polynomial, if and only if $z$ (the number of server machines) is a constant. ■

## 2.8 Approximating the Pareto Front

The problem variants we describe and analyze in this chapter are multi-criteria optimization problems. To simultaneously handle the two criteria cost and makespan, we either looked at decision variants "is there a schedule with makespan $\leq d$ and cost $\leq b$" or we used one of them as a constraint and asked "given a budget of $b$, minimize the makespan" (or vice versa). Naturally, one might be interested in finding an assortment of different efficient solutions, without giving a specific budget or deadline. A solution is called efficient, or Pareto optimal, if we can not improve one of the criteria, without worsening the other.

The set of all Pareto optimal solutions is called the Pareto front. In the following, we will use the term *point* to refer to the makespan and cost of a feasible solution of a given *SCS* problem.

For our NP-hard problems, we will not be able to efficiently calculate the exact Pareto front, but we can find a set of points that is close to the optimum. In the literature, one can find slightly different definitions for such approximations. In [54], the authors scale each criteria to an interval from 0 to 1. A set of points is an $\alpha$-approximation if, for each point in the actual Pareto front, there is a point where each dimension is offset by at most an additional $\pm\alpha$. We follow the definition of Pareto front approximations given in [75] (adapted to our case with exactly 2 objectives):

**Definition 2.1** A set of points $S$ is an $\alpha$-approximation of a Pareto front $P$, if for each point $p = (mspan^p, cost^p) \in P$ there is a point $p' = (mspan^{p'}, cost^{p'})$ in $S$ with $mspan^{p'} \leq (1+\alpha)mspan^p$ and $cost^{p'} \leq (1+\alpha)cost^p$.

The dynamic programming algorithms established in this chapter can be used to find such an approximation. We use the results from Section 2.4 to show how this is done, but note that a similar approach can be used for other results of this chapter.

Intuitively our dynamic programs calculate a collection of possible results but only report a single one, where the "best" is selected based on the current objective. Imagine that one of our deadline-constrained algorithms with approximation factor $(1+\varepsilon)$ reports *every* non-dominated solution it finds instead. The result for $d = 10$ and $\varepsilon = 0.1$ could look like Figure 2.8. For every reported point $(mspan, cost)$ we can infer a lower bound on the makespan of $mspan - \varepsilon \cdot d$ any schedule with a given *cost* has, due to the approximation factor of the algorithm. Note that the gap is in relation to a given $d$ and therefore results with a smaller makespan are less precise. We will circumvent that by repeating the algorithm with smaller values for $d$.

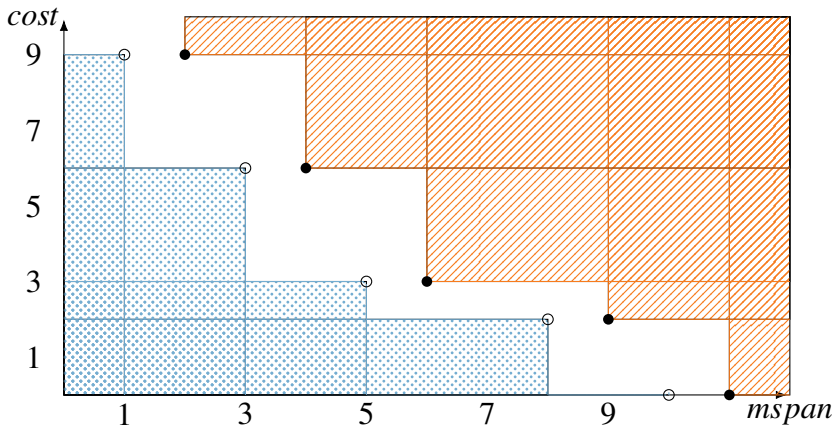



Figure 2.8: Reported solutions by our algorithm, filled circles and empty circles represent reported points and best possible solutions due to the approximation factor, respectively. Blue-dotted region is infeasible, the orange-striped region is feasible but dominated.

**Theorem 2.18** Using DPFGG (see Section 2.4) one can $\alpha$-approximate the Pareto front of a $SCS^{\Psi}$ problem with constant maximum cardinality source and sink dividing cut ($\psi$) in polynomial time, for any $\alpha > 0$.

*Proof.* Given some $SCS^{\psi}$ problem with constant $\psi$ run DPFGG (with the rounding approach) with $d = \sum_{j \in \mathcal{J}} p_s(j)$. Normally the algorithm reported the lowest cost state $[\hat{d}, f_s] = cost$ it found. Now, instead let the algorithm find the first state $[t, f_s] = cost$ for every $t \in (0.5\hat{d}, \hat{d}]$. For each of those states calculate an upper bound on the makespan for the respective schedule in the unscaled instance. Following the argumentation in the proof for Theorem 2.8, we know that the makespan is $\leq t + \varsigma + (n-2)2\varsigma = (t + 2n - 4)\varsigma$. Report the point $(mspan = (t + 2n - 4)\varsigma, cost)$ and add it to $S$. After that full algorithm iteration, set $d := 0.5d$ and repeat the process. Do this until $d = 1$. Finally, return the reported point set $S$.

We want to show that for every point $p = (mspan^p, cost^p)$ of a Pareto front, there is a reported point $p' = (mspan^{p'}, cost^{p'})$ with $mspan^{p'} \leq (1+\alpha)mspan^p$ and $cost^{p'} \leq (1+\alpha)cost^p$. Given some point $p = (mspan^p, cost^p)$, look at the iteration where $0.5d < mspan^p \leq d$. Since there is a feasible schedule with $mspan^p$ and $cost^p$ at some point during that iteration we found a feasible scaled schedule with $t = \lfloor \frac{mspan^p}{\varsigma} \rfloor$ and $cost \leq cost^p$. The calculated upper bound for that schedule in unscaled is then $(\lfloor \frac{mspan^p}{\varsigma} \rfloor + 2n - 4)\varsigma \leq mspan^p + (2n-4)\varsigma = mspan^p + (2n-4)\frac{\varepsilon \cdot d}{2n} \leq mspan^p + \varepsilon d \leq (1+2\varepsilon)mspan^p$ (recall: $\varsigma := \frac{\varepsilon \cdot d}{2n}$). Therefore, a point $p' = (mspan^{p'}, cost^{p'})$ with $mspan^{p'} \leq (1+2\varepsilon)mspan^p$ and $cost^{p'} \leq cost^p$ got reported. Setting $\varepsilon = 0.5\alpha$ and noting that we repeat the process no more than $log(\sum_{j \in \mathcal{J}} p_s(j))$ times concludes the proof. ■

## 2.9 Future Work

We give a small overview of some future research directions that emerge from our work. **SCS$^{\mathbf{e}}$**: If good approximations for $1 \mid r_j \mid \sum w_j U_j$ become established, the algorithm given in Section 2.3 for the extended chain could probably be improved. One could model the incoming communication delay with release dates and get an equivalent subproblem to solve, instead of the approximate subproblem currently used. **SCS**: Section 2.5 gives a strong inapproximability result for the general case with regard to the cost function. For two easy cases (chain and fully parallel graphs) we could establish FPTAS results, for graphs with a constant source and sink dividing cut ($\psi$) we have an algorithm that finds optimal solutions with a $(1+\varepsilon)$ deadline augmentation. Here one could explore if there are FPTAS results for different assumptions, whether there are approximation algorithms without resource augmentation for constant $\psi$ instances, and lastly, if there are approximation algorithms with resource augmentation for the general case. For the makespan objective, we already have an FPTAS for graphs with a constant $\psi$. It remains to explore approximation algorithms or inapproximability results for the general case of this problem. **SCS$^{\mathbf{1}}$**: We show strong NP-hardness even for this simplified problem. Since this is a special case of the general problem all constructive results still hold, additionally we were able to give first simple algorithms for both cost and makespan optimization in general graphs. Here it would be interesting to look into more involved approximation algorithms that give better performance guarantees.

# 3. Restricted Assignment Interval

We consider the *restricted assignment interval* problem, which is a special case of the well studied unrelated scheduling problem. In this problem, machines are given in some order, and each job is eligible on a specific interval of those machines. We show how to extend known linear programs and develop a rounding strategy that gives the first better than 2-approximation for this problem and establish some new inapproximability results for the problem family.

This chapter is based on our paper *(In-)Approximability Results for Interval, Resource Restricted, and Low Rank Scheduling* [67] which was published in the proceedings of ESA 2022. The original paper is divided into two main parts, a constructive part containing an approximation algorithm and a complexity-theoretical part containing several inapproximability results. The first part, which is given in full here, was written and developed by myself, with help from Marten Maack. I added a small section on the integrality gap of the considered LP to this thesis, which was not part of our paper. The second part was mostly the work of Marten Maack and Anna Rodriguez Rasmussen, with only small contributions by myself. Therefore, we only summarize the results, give an intuition for the used approaches, and direct the interested reader to the full version of the paper.

## 3.1 Introduction

Makespan minimization on unrelated parallel machines, or unrelated scheduling for short, is considered a fundamental problem in approximation and scheduling theory. We consider the family of subproblems included in the general problem, with a focus on the restricted assignment interval problem.

### 3.1.1 Problem Definition

First consider the general problem of *makespan minimization on unrelated parallel machines*, or *unrelated scheduling* for short. In this problem, a set $\mathcal{J}$ of jobs has to be assigned to a set $\mathcal{M}$ of machines via a schedule $\pi : \mathcal{J} \to \mathcal{M}$. Each job $j$ has a processing time $p_{ij}$ depending on the machine $i$ it is assigned to, and the goal is to minimize the makespan $C_{\max}(\pi) = \max_{i \in \mathcal{M}} \sum_{j \in \pi^{-1}(i)} p_{ij}$.

A natural special case is the *restricted assignment problem*. In it, each job $j$ has a size $p_j$ and $p_{ij} \in \{p_j, \infty\}$ for each machine $i$. For each job $j$ we denote its set of eligible machines by $\mathcal{M}(j) = \{i \in \mathcal{M} \mid p_{ij} = p_j\}$.

We look at the following special cases of both unrelated scheduling and restricted assignment.

**Interval Restrictions.** In the variant of restricted assignment with interval restrictions, denoted as RAI in the following, there is a total order of the machines and each job $j$ is eligible on a discrete interval of machines, i.e., $\mathcal{M} = \{M_1, M_2, \ldots, M_m\}$ and $\mathcal{M}(j) = \{M_\ell, M_{\ell+1}, \ldots M_r\}$ for some $\ell, r \in \{1, \ldots, m\}$.

**Resource Restrictions.** In the restricted assignment problem with $R$ resource restrictions, or RAR($R$), a set $\mathcal{R}$ of $R$ (renewable) resources is given, each machine $i$ has a resource capacity $c_r(i)$ and each job $j$ has a resource demand $d_r(j)$ for each $r \in \mathcal{R}$. The eligible machines are determined by the corresponding resource constraints, i.e., $\mathcal{M}(j) = \left\{i \in \mathcal{M} \,\middle|\, \forall r \in \mathcal{R} : d_r(j) \leq c_r(i)\right\}$ for each job $j$.

**Low Rank Scheduling.** In the rank $D$ version of unrelated scheduling, or LRS($D$), the processing time matrix $(p_{ij})$ has a rank of at most $D$. Alternatively (see [14]), we can assume that each job $j$ has a $D$ dimensional size vector $s(j)$ and each machine $i$ a $D$ dimensional speed vector $v(i)$ such that $p_{ij} = \sum_{k=1}^{D} s_k(j) v_k(i)$.

### 3.1.2 Relation Between the Models and the State of the Art

As stated earlier, restricted assignment is a subproblem of unrelated scheduling. We can formulate a few additional special cases and relationships between the models. Consider the hierarchical case of RAI in which each job is eligible on an interval of the form $\{M_1, M_2, \ldots M_r\}$, i.e., the first machine is eligible for each job. It should be easy to see that this problem is equivalent to RAR(1). Sort the machines in RAR(1) from most to least capable; the set of eligible machines of each job now is an interval from the first up to the last machine, which is capable enough for the respective job. For the other direction, we assign a strictly descending resource capability to the machines, and each job's resource demand is exactly the capability of the last eligible job. In a similar vein, RAI is a subset of RAR(2), which we can show by restating any RAI problem as a RAR(2) problem [66]. For each machine $M_i \in \{M_1, M_2, \ldots, M_m\}$ set $c_1(i) = i$ and $c_2(i) = m - i$. Now we can use the first resource to model the left border of a job's interval and the second resource for the right border, i.e., for every job $j$ with $\mathcal{M}(j) = \{M_\ell, M_{\ell+1}, \ldots M_r\}$, set $d_1(j) = \ell$ and $d_2(j) = r$. As RAR($R$) is a special case of restricted assignment, which in turn is a special case of unrelated scheduling, we get the following picture.

**Theorem 3.1** There is the following relationship between special cases of unrelated scheduling and restricted assignment:

$$\text{hierarchical RAI} = \text{RAR}(1) \subsetneq \text{RAI} \subsetneq \text{RAR}(2) \subsetneq \text{restricted assign.} \subsetneq \text{unrelated sched.}$$

Naturally, that means that every constructive result also works for the included problems, and every inapproximability result also holds for the larger problems.

Unrelated Scheduling is considered a fundamental problem in approximation and scheduling theory. In 1990, Lenstra, Shmoys, and Tardos [57] presented a 2-approximation

for it and further showed that no better than 1.5-approximation can be achieved (unless P = NP) already for the restricted assignment problem. Closing or narrowing the gap between 2-approximation and 1.5-inapproximability is a famous open problem in approximation [86], and scheduling theory [78]. We give an overview of the state of the art and the most relevant results concerning the discussed subproblems of unrelated scheduling.

**Interval Restrictions.** There are several variants and special cases of this problem that are known to admit a polynomial time approximation scheme (PTAS), see [27, 41, 48, 62, 72, 79], the most prominent of which is probably the previously mentioned hierarchical case [62]. For RAI, on the other hand, there is an $(1+\delta)$-inapproximability result for some small but constant $\delta > 0$ [66]. Furthermore, Schwarz [79] designed a $(2-2/(\max_{j\in\mathcal{J}} p_j))$-approximation (assuming integral processing times); and Wang and Sitters [85] studied an LP formulation that provides an optimal solution for the special case with two distinct processing times and some additional assumption.

**Resource Restrictions.** As mentioned, RAR$(1)$ corresponds to the hierarchical RAI case, which admits a PTAS [62]. On the other hand, there can be no approximation algorithm with ratios smaller than $48/47 \approx 1.02$ or 1.5 for RAR$(2)$ and RAR$(4)$, respectively, unless P = NP, see [66]. While the hierarchical case, i.e., RAR$(1)$, has been studied extensively before, RAR$(R)$ was first introduced in a paper by Bhaskara et al. [6], who mentioned it as a special case of the next problem that we consider.

**Low Rank Scheduling.** Now, LRS$(1)$ is exactly makespan minimization on uniformly related parallel machines, which is well known to admit a PTAS [37]. Bhaskara et al. [6], who introduced LRS$(D)$, presented a QPTAS for LRS$(2)$ along with some initial inapproximability results for $D > 2$. Subsequently, Chen et al. [15] showed that there can be no better than 1.5-approximation for LRS$(4)$ unless P = NP, and for LRS$(3)$ the same authors together with Marx [14] ruled out a PTAS. On an intuitive level, resource restrictions can be seen as a restricted assignment version of low rank scheduling. However, there is a more direct relationship between the two problems: For each RAR$(R)$ instance, there exist LRS$(R+1)$ instances that are arbitrarily good approximations of the former (see [66]). Hence, any approximation algorithm for LRS$(R+1)$ can also be used for RAR$(R)$, and any inapproximability result for RAR$(R)$ carries over to LRS$(R+1)$. In fact, many (but not all) inapproximability results for low rank scheduling essentially have this form.

### 3.1.3 Our Results

The main result of this chapter is the improved approximation result for RAI with a ratio $2-\frac{1}{24} \approx 1.96$ presented in Section 3.2; This marks the first constant better than 2 result for RAI, which was posed as an open challenge in previous works [41, 79, 85], the earliest of which is from 2010. When considering the respective results in [79] and [85], in particular, it seems highly probable that the actual goal of the research was to address exactly that challenge. The presented approximation algorithm follows the approach of solving and rounding a relaxed linear programming formulation of the problem, which has been used in the classical work by Lenstra et al. [57] and many of the results thereafter. In particular, we extend the so-called assignment LP due to Lenstra et al. [57] and design a customized rounding approach. Both the linear programming extension and the rounding approach utilize extensions and refinements of ideas from [79] and [85]. Our result joins the relatively short list of special cases of the restricted assignment problem that do not allow a PTAS

and for which an approximation algorithm with a rate smaller than 2 is known. Other notable entries are the restricted assignment problem with only two processing times [13] and the so-called graph balancing case [24], where each job is eligible on at most two machines.

The other results from this chapter's underlying paper are inapproximability results for the introduced subproblems of unrelated scheduling. The four reductions result in the following (unless P = NP):

- a 1.5-inapproximability result, for RAR(3) presented in Section 3.3.1;
- a $8/7$-inapproximability result for RAR(2) presented in Section 3.3.2;
- a $9/8$-inapproximability result for RAI presented in Section 3.3.3;
- and a 1.5-inapproximability result for LRS(3) (not furter presented in this thesis).

In this chapter, we give a summary of three of them and sketch the main ideas. The inapproximability results we summarize here directly build upon the results presented in the paper [66], which in turn utilizes many of the previously published ideas, e.g., from [6, 14, 15, 24, 57]. We use a specifically tailored satisfiability problem presented in [66] as the starting point for all of the reductions. For the RAI result in particular, we refine and restructure the respective results from [66] aiming for a significantly better ratio. The respective reduction involves a sorting process, and curiously the main improvement in the reduction involves changing a sorting process resembling insertion sort into one resembling bubble sort. Due to this change, the construction becomes locally less complex, enabling the use of smaller processing times and hence a stronger inapproximability result. Furthermore, the simplified construction in the result enables us to use the basic structure of the reduction as a starting point for the second main result of the underlying paper, namely, the 1.5-inapproximability result for RAR(3). For this reduction, several additional considerations and gadgets are needed, arguably making it the most elaborate of the reductions presented in this chapter.

The search for an inapproximability result with a reasonably big ratio for RAR(3) was stated as an open challenge in the long version of [14]. Adding the new result yields a very clear picture regarding the approximability of low rank makespan minimization: There is a PTAS for LRS(1), a QPTAS for LRS(2), and a 1.5-inapproximability result for LRS($D$) with $D \geq 3$. The last two reductions regarding RAR(2) and RAR(3) yield much-improved inapproximability results for the respective problems using comparatively simple and elegant reductions. The result regarding RAR(3), in particular, closes a gap in the results of [66] and also yields an (arguably) easier, alternative proof for the result of [15]. Finally, we note that all of the inapproximability results regarding restricted assignment with resource restrictions can be directly applied to the so-called fair allocation or Santa Claus versions of the problems. In these problem variants, we maximize the minimum load received by the machines rather than minimization of the maximum load, i.e., the objective function is given by $C_{\min}(\pi) = \min_{i \in \mathcal{M}} \sum_{j \in \pi^{-1}(i)} p_{ij}$ in this case.

### 3.1.4 Further Related Work

We already discussed the most relevant related results in Section 3.1.2. We refer to [66], the corresponding long version [65], and the references therein for a more detailed discussion of related work and only briefly discuss some further references. Regarding the problem of closing the gap between the 2-approximation and 1.5-approximability, there has been a promising line of research (e.g. [3, 43]) in the last decade, starting with a breakthrough result due to Svensson [83], which in turn was preceded by corresponding results for the

fair allocation (Santa Claus) version of the problem, see, e.g., [5, 29]. These results are based on local search algorithms that usually do not run in polynomial time but can be used to prove a small integrality gap for a certain linear program, which, therefore, can be used to approximate the optimum objective value in polynomial time without actually producing a schedule. In the online setting, versions of restricted assignment with different types of restrictions, and variants of RAR(1) in particular, have been intensively studied. We refer to the surveys [53, 58, 59] for an overview. Lastly, we note that the low rank scheduling has also been considered from the perspective of fixed-parameter tractable algorithms [14].

## 3.2 A $(2-\frac{1}{24})$-Approximation

In this section, we establish the first approximation algorithm for RAI with an approximation factor better than 2:

**Theorem 3.2** There is a $(2-\gamma)$-approximation for RAI with $\gamma = \frac{1}{24}$.

The particular value of the parameter $\gamma$ is justified in the end. To achieve this result, we first formulate a customized linear program based on the assignment LP due to Lenstra et al. [57] and develop a rounding approach that places different types of jobs in phases. Note that the placement of big jobs with a size close to OPT (where OPT is the makespan of an optimal schedule) is often critical when aiming for an approximation ratio of smaller than 2 for a makespan minimization problem. For instance, the classical 2-approximation [57] for restricted assignment produces a schedule of length at most $\text{OPT} + \max_{j\in\mathcal{J}} p_j$ where OPT is the makespan of an optimal schedule and hence the approximation ratio is better if $\max_{j\in\mathcal{J}} p_j$ is strictly smaller than OPT. This is also the case with our approach – the main effort goes into the careful placement of such big jobs. In particular, we place the largest jobs in the first rounding step and the remaining big jobs in the second. All of these jobs have the property that each machine should receive at most one of them, and they are placed accordingly. Moreover, the placement is designed to deviate not too much from the fractional placement due to the LP solution. In the last step, the remaining jobs are placed. Each rounding step is based on a simple heuristic approach that considers the machines from left to right and places the least flexible eligible jobs first, i.e., the jobs that have not been placed yet, are eligible on the current machine, and have a minimal last eligible machine in the ordering of the machines. Both the LP and the rounding approach reuse ideas from [79, 85]. Hence, the main novelty lies in the much more elaborate approach for placing the mentioned big jobs in two phases.

In the following, we first establish some preliminary considerations; then briefly discuss the least flexible first heuristic utilized in the rounding approach; next, we formulate the LP and argue that it is indeed a relaxation of the problem at hand, shortly discuss its integrality gap and then introduce and analyze the different phases of the rounding procedure step by step.

### 3.2.1 Preliminaries.

We apply the standard technique (see [57]) of using a binary search framework to guess a candidate makespan $T$ ($\geq \max_{j\in\mathcal{J}} p_j$). The goal is then to either correctly decide that no schedule with makespan $T$ exists or to produce a schedule with makespan at most $(2-\gamma)T$. Given this guess $T$, we divide the jobs $j$ into small ($p_j \leq 0.5T$), large ($0.5T <$

$p_j \leq (0.5+\xi)T$) and huge ($(0.5+\xi)T < p_j$) jobs depending on some parameter $\xi = \frac{1}{24}$ which is justified later on. We denote the sets of small, large, and huge jobs as $\mathcal{S}$, $\mathcal{L}$, and $\mathcal{H}$, respectively. Furthermore, we fix the (total) order of the machines such that each job is eligible on consecutive machines. This is possible since we are considering RAI. For the sake of simplicity, we assume $\mathcal{M} = \{0,\dots,m-1\}$ with the ordering corresponding to the natural one and set $\mathcal{M}(\ell,r) = \{\ell,\dots,r\}$ for each $\ell,r \in \mathcal{M}$. When considering the machines, we use a left-to-right intuition with predecessor machines on the left and successor machines on the right. Note that for each job $j$, there exists a left-most and right-most eligible machine, and we denote these by $\ell(j)$ and $r(j)$, respectively, i.e., $\mathcal{M}(j) = \mathcal{M}(\ell(j),r(j))$. For a set of jobs $J \subseteq \mathcal{J}$, we call a job $j \in J$ *least flexible* in $J$ if $r(j)$ is minimal in $\{r(j') \mid j' \in J\}$, and a job $j$ is called *less flexible* than a job $j'$ if $r(j) \leq r(j')$. Lastly, we set $J(\ell,r) = \left\{ j \in J \,\middle|\, \mathcal{M}(j) \subseteq \mathcal{M}(\ell,r) \right\}$ for each set of jobs $J \subseteq \mathcal{J}$ and pair of machines $\ell,r \in \mathcal{M}$, and $p(J) = \sum_{j\in\mathcal{J}} p_j$.

**Least Flexible First.** Consider the least flexible first heuristic for RAI: The optimum makespan OPT is lower bounded by the maximum job size $\max_{j\in\mathcal{J}} p_j$ as well as the average load $p(\mathcal{J}(\ell,r))/|\mathcal{M}(\ell,r)|$ of jobs that have to be placed in any given interval of machines $\mathcal{M}(\ell,r)$ in a feasible schedule. Let $L \leq \text{OPT}$ be the maximum of all of the above lower bounds. Starting with the left-most machine in the ordering, the heuristic works as follows:

- Let $i^*$ be the current machine and $J$ the set of jobs that have not been placed yet and are eligible on $i^*$.
- If $i^*$ has received a load of at most $L$ up to now and $J \neq \emptyset$, place a *least flexible* job $j \in J$ on $i^*$, i.e., a job $j \in J$ with minimal $r(j)$, and consider $i^*$ again.
- Otherwise, consider the next machine in the ordering or stop if there is none.

It is easy to see that this simple approach yields a 2-approximation:

**Lemma 3.1** The least flexible first heuristic places each job, and each machine receives a load of at most $L + \max_{j\in\mathcal{J}} p_j \leq 2\,\text{OPT}$.

*Proof.* The heuristic obviously never places a load greater than $L + \max_{j\in\mathcal{J}} p_j$ on any machine. Now assume for the sake of contradiction that there exists a job that is not placed by this heuristic. Let $j^*$ be a job that is not placed, i.e., after considering the right-most eligible machine $r^* = r(j^*)$, the job $j^*$ has not been placed by the algorithm. Then $r^*$ did receive a load greater than $L$, and we have $r(j) \leq r^*$ for each job $j$ placed on $r^*$. Let $\ell^* \leq r^*$ be the left-most machine with the properties that each machine in $\mathcal{M}(\ell^*,r^*)$ did receive a load greater than $L$ and $r(j) \leq r^*$ for each job $j$ placed on $\mathcal{M}(\ell^*,r^*)$. Furthermore, let $J^*$ be the set of jobs placed on $\mathcal{M}(\ell^*,r^*)$ by the algorithm together with $j^*$. Then $\ell(j) \geq \ell^*$ for each $j \in J^*$ since otherwise there exists a machine $i$ directly preceding $\ell^*$ that could have received $j$ as well but did not. This would imply that $i$ did receive a load greater than $L$ of jobs less flexible than $j$ yielding a contradiction to the choice of $\ell^*$. Hence, $J^* \subseteq \mathcal{J}(\ell^*,r^*)$ yielding the contradictory statement $p(\mathcal{J}(\ell^*,r^*)) \geq p(J^*) > |\mathcal{M}(\ell^*,r^*)|L \geq p(\mathcal{J}(\ell^*,r^*))$ (considering the definition of $L$). ■

We are not aware of this observation being published before, but consider it very likely that it was already known, in particular, since variants thereof are used in [79, 85].

### 3.2.2 Linear Program.

The classical assignment LP (see [57]) is given by assignment variables $x_{ij} \in [0,1]$ for each $i \in \mathcal{M}$ and $j \in \mathcal{J}$ and the following constraints:

$$\sum_{i \in \mathcal{M}} x_{ij} = 1 \qquad \forall j \in \mathcal{J} \tag{3.1}$$

$$\sum_{j \in \mathcal{J}} p_j x_{ij} \leq T \qquad \forall i \in \mathcal{M} \tag{3.2}$$

$$x_{ij} = 0 \qquad \forall j \in \mathcal{J}, i \in \mathcal{M} \setminus \mathcal{M}(j) \tag{3.3}$$

Equation (3.1) guarantees that each job is (fractionally) placed exactly once; Equation (3.2) ensures that each machine receives at most a load of $T$; and due to Equation (3.3), jobs are only placed on eligible machines. We add additional constraints that have to be satisfied by any integral solution. In particular, we add the following constraints using parameters $UB(\ell, r)$ for each $\ell, r \in \mathcal{M}$ with $\ell \leq r$, which will be properly introduced shortly:

$$\sum_{j \in \mathcal{L} \cup \mathcal{H}} x_{ij} \leq 1 \qquad \forall i \in \mathcal{M} \tag{3.4}$$

$$\sum_{i \in \mathcal{M}(\ell,r)} \sum_{j \in \mathcal{H}} x_{ij} \leq UB(\ell, r) \qquad \forall \ell, r \in \mathcal{M}, \ell \leq r \tag{3.5}$$

Equation (3.4) captures the simple fact that no machine may receive more than one job of size larger than $0.5T$ and was used in [24] as well. The bound $UB(\ell, r)$, on the other hand, is defined in relation to the total load of small jobs that has to be scheduled in the respective interval $\mathcal{M}(\ell, r)$. In particular, we consider the overall load of small jobs that have to be placed in the interval together with the load due to huge jobs with their sizes rounded down to their minimum size. The respective load has to be bounded by $T$ times the number of machines in the interval, i.e., $\sum_{i \in \mathcal{M}(l,r)} \sum_{j \in \mathcal{H}} (0.5 + \xi) T x_{ij} + p(\mathcal{S}(\ell, r)) \leq T|\mathcal{M}(l, r)|$. Since the number of huge jobs placed in an interval is integral for an integral solution, we can therefore set $UB(\ell, r) = \left\lfloor (T|\mathcal{M}(l, r)| - p(\mathcal{S}(\ell, r)))/((0.5 + \xi)T) \right\rfloor$. We note that a constraint similar to Equation (3.5) is also used in [79, 85]. Summing up, we try to solve the linear program given by Equations (3.1) to (3.5), which is indeed a relaxation for RAI. If this is not successful, we reject $T$ and otherwise round the solution $x$ using the procedure described in the following and yield a rounded solution $\bar{x}$.

### 3.2.3 Integrality Gap of the Linear Program

The classical assignment LP as given by Equations (3.1) to (3.3) has an integrality gap of (at least) 2, which would naturally not be sufficient to get a better than 2-approximation with our approach. Consider an instance with $m$ jobs of size $1 - 1/m$ and a single job of size 1, where each job is eligible on every machine ($m$ denotes the number of machines), and $T = 1$. The LP can place one size $1 - 1/m$ job on each machine and distribute the size 1 job onto the machines evenly, resulting in an LP solution of 1. The integral solution, however, has to place two jobs on one machine, resulting in a makespan of at least $2 - 2/m$, which approaches 2 for big $m$.

This specific example would be captured by Equation (3.4) since the LP solution places a total of $1 + 1/m$ parts of large or huge jobs on one machine. However, we can simply transform the $m$ jobs of size $1 - 1/m$ into very small jobs with the same total load, where for every machine $i$, a set of jobs with total load $1 - 1/m$ is eligible exclusively on $i$. This construction results in the same integrality gap of (at least) 2.

With Equation (3.5), no (fractional) part of a huge job may be placed on a machine with local small load of more than $(0.5-\xi)T$. Therefore, the LP has no fractional solution for the instance mentioned above unless we increase $T$ until the huge job is not huge anymore, or the small load only makes up $(0.5-\xi)T$ total load per machine. Specifically, that would mean to increase $T$ from 1 to $1/(0.5+\xi)$, and lower the integrality gap of that instance from 2 to $1+2\xi$. We have two options for similar constructions. The first maximizes the small load such that the LP can fractionally distribute one huge job over two machines; the second does a similar thing with one big job over many machines. Consider an instance with 2 machines, each with a local small load of $0.5-\xi$, as well as a huge job of size 1 and two jobs of size $\xi$, each eligible on both machines. The fractional LP solution is 1, while the best integral solution is $1.5-\xi$. The second construction looks as follows. We have $m$ machines, each with small local jobs of total size $1-1/m$ and one large job of size $0.5+\xi$ eligible on all machines. As in the first example, we can distribute the large job over all machines, resulting in an LP solution of $\approx 1$, while the best integral solution has size $\approx 1.5+\xi$ (for big $m$). The integrality gap of our LP is, therefore, at least $1.5+\xi$. The distinction between big and huge jobs is not motivated by the constructions given above, as they would imply the best result for $\xi = 0$. However, Equation (3.5) becomes more restrictive with increasing $\xi$, which allows our following three-phase algorithm to achieve an approximation ratio better than 2.

### 3.2.4 The Rounding Algorithm.

Assume that we have successfully solved the LP as given above, giving us a solution $x$. We start by placing the huge jobs as follows.

**Step 1 Placement of Huge Jobs:** Starting with the first machine in the ordering, do the following:

- Let $i^*$ be the current machine and $H$ the set of huge jobs that have not been placed yet and are eligible on $i^*$.
- If $\lfloor \sum_{i\in\mathcal{M}(0,i^*)} \sum_{j\in\mathcal{H}} x_{ij} \rfloor > \lfloor \sum_{i\in\mathcal{M}(0,i^*-1)} \sum_{j\in\mathcal{H}} x_{ij} \rfloor$ and $H \neq \emptyset$, place a least flexible job $j \in H$ on $i^*$, i.e., we set $\bar{x}_{i^*j} = 1$.
- Consider the next machine in the ordering or stop if there is none.

We denote the set of machines that are considered by the above procedure as $\mathcal{X}$, i.e., $\mathcal{X} = \{i^* \in \mathcal{M} \mid \lfloor \sum_{i\in\mathcal{M}(0,i^*)} \sum_{j\in\mathcal{H}} x_{ij} \rfloor > \lfloor \sum_{i\in\mathcal{M}(0,i^*-1)} \sum_{j\in\mathcal{H}} x_{ij} \rfloor\}$.

This procedure indeed works, and we preserve a connection to the original LP solution:

**Lemma 3.2** All of the huge jobs are placed (on eligible machines) by the above procedure and, for each $\ell, r \in \mathcal{M}$ with $\ell \leq r$, we have $\sum_{i\in\mathcal{M}(\ell,r)} \sum_{j\in\mathcal{H}} \bar{x}_{ij} \leq \lceil \sum_{i\in\mathcal{M}(\ell,r)} \sum_{j\in\mathcal{H}} x_{ij} \rceil$.

*Proof.* The second statement directly follows from the fact that we only place a new job if the sum of fractional huge jobs placed up to the current machine in the LP solution reaches a new integer. Regarding the first, assume for the sake of contradiction that there exists a huge job $j^*$ that is not placed. We set $r^* = r(j^*)$. Note that $\mathcal{X} \cap \mathcal{M}(j^*) \neq \emptyset$ since $j^*$ was placed fractionally by the LP and for the same reason the last such machine $r' \leq r^*$ or some predecessor did receive some of the fractional load of $j^*$ in the LP as well. Then $r'$ did receive a huge job $j$ with $r(j) \leq r(j^*)$. Let $\ell^*$ be the left-most machine such that each machine in $\mathcal{M}(\ell^*, r^*) \cap \mathcal{X}$ did receive a huge job $j$ with $r(j) \leq r(j^*)$ and let $H^*$ be the set

of huge jobs placed by the procedure on $\mathcal{M}(\ell^*, r^*) \cap \mathcal{X}$ together with $j^*$. Then we have $\ell^* \leq \ell(j)$ for each $j \in H^*$ since otherwise there exists a machine $i \in \mathcal{X}$ directly preceding $\ell^*$ that may have received a job from $H^*$. Since this did not happen, it must have received a less flexible job, which is a contradiction to the choice of $\ell^*$. Hence, $H^* \subseteq \mathcal{H}(\ell^*, r^*)$ but $|\mathcal{M}(\ell^*, r^*) \cap \mathcal{X}| < |H^*|$. This is a contradiction since each job in $H^*$ was completely placed in $\mathcal{M}(\ell^*, r^*)$ by the LP which implies $|\mathcal{M}(\ell^*, r^*) \cap \mathcal{X}| \geq |H^*|$. ■

We note that this first rounding step is very similar to the first rounding step in [85].

In the next step, we divide the machines into regions, where each region did receive fractional large load of (roughly) one.

**Step 2 Mapping out the Regions:** We define a set of border machines $\mathcal{B}$ as the machines considered from left to right where the sum of fractionally placed large jobs hits a new integer, i.e., $\mathcal{B} = \left\{ i' \in \mathcal{M} \,\middle|\, \left\lfloor \sum_{i \in \mathcal{M}(0,i')} \sum_{j \in \mathcal{L}} x_{ij} \right\rfloor > \left\lfloor \sum_{i \in \mathcal{M}(0,i'-1)} \sum_{j \in \mathcal{L}} x_{ij} \right\rfloor \right\}$. Moreover, let $\mathcal{B} = \{ i_1, \ldots, i_q \}$ with $i_1 < \cdots < i_q$ and $i_0$ be the left-most machine with $\sum_{j \in \mathcal{L}} x_{i_0 j} > 0$. For each $s \in \{0, \ldots, q-1\}$, we may initially define the $s$-th region as $R^s = \mathcal{M}(i_s, i_{s+1})$. At this point, consecutive regions overlap by one machine. We change this while guaranteeing that each region retains at least one *candidate machine* that may receive a large job in the following. In particular, a machine $i \in \mathcal{M}$ is a candidate if it did receive some fractional large or huge job in the LP solution, i.e., $\sum_{j \in \mathcal{H} \cup \mathcal{L}} x_{ij} > 0$, but no huge job afterward, i.e., $\sum_{j \in \mathcal{H}} \bar{x}_{ij} = 0$. We denote the set of candidate machines as $\mathcal{C}$. For each $s \in [q-1]$, apply the following procedure in incremental order:

- Check whether region $R^s$ needs the last machine to have at least one candidate, i.e., $\mathcal{M}(i_s, i_{s+1} - 1) \cap \mathcal{C} = \emptyset$.
  - If this is the case, set $R^{s+1} = \mathcal{M}(i_{s+1} + 1, i_{s+2})$.
  - Otherwise set $R^s = \mathcal{M}(i_s, i_{s+1} - 1)$.

After applying this procedure, we have:

**Lemma 3.3** The regions are non-overlapping and each contain at least one candidate.

*Proof.* The first statement is obvious, and we show the second via contradiction. Assume that there is a region $R^r$ without a candidate machine. After running the procedure the original left and right borders $i_s$ and $i_{s+1}$ of a region $R^s$ may or may not be included in $R^s$ and we set $\texttt{inner}(R^s) = \mathcal{M}(i_s + 1, i_{s+1} - 1)$ for each $s \in [q]$. Since $R^r$ does not contain candidates, we know that $\texttt{inner}(R^r)$ cannot contain candidates, $i_{r+1}$ was assigned to $R^r$ by the algorithm and is not a candidate either. Let $\ell \leq r$ be maximal with the property that $R^\ell$ did receive $i_\ell$ in the algorithm and $R^s$ did receive $i_{s+1}$ for each $s \in \{\ell, \ldots, r\}$. Then the rules of the algorithm imply that $i_\ell$ is not a candidate and $\texttt{inner}(R^s)$ does not contain a candidate either for each $s \in \{\ell, \ldots, r\}$. Hence, the only possible remaining candidate machines in the respective regions are the borders $\mathcal{B} \cap \mathcal{M}(i_{\ell+1}, i_r)$. Let $C$ be the set of candidates in the respective regions, i.e., $C = \mathcal{C} \cap \bigcup_{s \in \{\ell, \ldots, r\}} R_s$, and $k = |\{\ell, \ldots, r\}|$. Then the above implies $|C| \leq k - 1$. For the remainder of the proof, we introduce some additional notation: the set of assigned huge jobs in $\bigcup_{s \in \{\ell, \ldots, r\}} R_s$ is given by $H$ and the fractional number of large or huge jobs placed in these regions according to $x$ is denoted as $\texttt{fracLarge}$ or $\texttt{fracHuge}$, respectively, i.e., $\texttt{fracLarge} = \sum_{i \in \mathcal{M}(i_\ell, i_{r+1})} \sum_{j \in \mathcal{L}} x_{ij}$ and $\texttt{fracHuge} = \sum_{i \in \mathcal{M}(i_\ell, i_{r+1})} \sum_{j \in \mathcal{H}} x_{ij}$. Since $i_\ell \in R^\ell$ and $i_{r+1} \in R^r$, the definition of the borders yields $\texttt{fracLarge} \geq k$. Note that each machine in the regions that did receive a

fractional large or huge job in the LP solution but is not a candidate subsequently received a huge job. Hence, we have

$$|H| + |C| \overset{(3.4)}{\geq} \texttt{fracHuge} + \texttt{fracLarge} \geq \texttt{fracHuge} + k$$

and therefore $|H| \geq \texttt{fracHuge} + 1$. However, Lemma 3.2 gives us $|H| \leq \lceil \texttt{hugeLoad} \rceil < \texttt{hugeLoad} + 1$. ↯ ■

Before proceeding with the placement of the large jobs, we note the following technical observation:

**Lemma 3.4** Let $\ell, r \in \mathcal{M}$ with $\ell \leq r$, $\ell \in R^s$, $r \in R^t$, $k = |\{s, \ldots, t\}|$, and $\texttt{fracLarge} = \sum_{i \in \mathcal{M}(\ell,r)} \sum_{j \in \mathcal{L}} x_{ij}$. Then we have

- $k - 2 < \texttt{fracLarge} < k + 2$.

Furthermore,

- $\texttt{fracLarge} < k + 1$ if either $\ell > i_s$ or $r < i_{t+1}$,
- $\texttt{fracLarge} < k$ if both of these conditions hold.

*Proof.* There are at least $k-2$ regions that are completely included in $\mathcal{M}(\ell, r)$, including their original outer borders. Hence, the definition of the regions yields $k - 2 < \texttt{fracLarge}$. For the remaining statements, we consider the definition of the regions more closely. Note that there exist numbers $\lambda_u \in [0,1)$ and $\rho_u \in (0,1]$ for each $u \in [q]$ such that $\lambda_u + \left(\sum_{i \in \texttt{inner}(R^u)} \sum_{j \in \mathcal{L}} x_{ij}\right) + \rho_u = 1$ (using the notation of the last proof); and furthermore $\sum_{j \in \mathcal{L}} x_{i_0 j} = \lambda_0$ if $i_0 \neq i_1$, $\sum_{j \in \mathcal{L}} x_{i_u j} = \rho_{u-1} + \lambda_u$ for $u \in \{1, \ldots, q-1\}$, and $\sum_{j \in \mathcal{L}} x_{i_q j} = \rho_{q-1}$. We assume for now $s > 0$ and $t < q - 1$. Then we have $\texttt{fracLarge} \leq \rho_{s-1} + k + \lambda_{t+1} < k + 2$, $\texttt{fracLarge} \leq k + \lambda_{t+1} < k + 1$ if $\ell > i_s$, $\texttt{fracLarge} < \rho_{s-1} + k \leq k + 1$ if $r < i_{t+1}$, and $\texttt{fracLarge} < k$ if both of the conditions hold. If $s = 0$ or $t = q$, we can prove the statement analogously. ■

Using the regions, we place the large jobs via the following procedure.

**Step 3 Placement of Large Jobs:** Starting with the first region, do the following:

- Let $R^*$ be the current region and $L$ the set of large jobs that have not been placed yet and are eligible on at least one candidate machine from $R^*$.
- Do the following *twice*: Pick a least flexible large job $j \in L$, place it on the leftmost eligible candidate machine $i \in R^*$, i.e. $\bar{x}_{ij} = 1$, and update $L$.
- Consider the next region in the ordering or stop if there is none.

Observe that the placement of both the large and huge jobs guarantees that only machines that did receive fractional large or huge load in the LP solution may receive any large or huge job, and each such machine receives at most one such job. We argue that this procedure works and also retains some connection to the original LP solution $x$.

**Lemma 3.5** All large jobs are placed (on eligible machines) by the described procedure and, for each $\ell, r \in \mathcal{M}$ with $\ell \leq r$, we have $\sum_{i \in \mathcal{M}(\ell,r)} \sum_{j \in \mathcal{L}} \bar{x}_{ij} < 2(\sum_{i \in \mathcal{M}(\ell,r)} \sum_{j \in \mathcal{L}} x_{ij} + 2)$.

*Proof.* Regarding the second statement, note that we place at most two jobs in each region, and hence Lemma 3.4 directly yields the proof. As usual, we prove the first statement by

contradiction. To that end, assume that there exists a large job $j^*$ that is not placed by the procedure. First, note that there is at least one eligible candidate machine for $j^*$. To see this, consider the set $M$ of eligible machines $i \in \mathcal{M}(j)$ that either received fractional load of $j^*$ or some huge load, i.e., $x_{ij} > 0$ for $j \in \{j^*\} \cup \mathcal{H}$. Then Equation (3.4) implies $\sum_{i \in M} \sum_{j \in \mathcal{H}} x_{ij} \leq |M| - 1$. Hence, at most $|M| - 1$, many huge jobs are placed on machines from $M$ due to Lemma 3.2, and therefore, at least one of these machines is a candidate. There are two possibilities why $j^*$ was not placed on such a machine: either a less flexible job got placed on the machine, or two other less flexible jobs were already placed in the same region. Let $r^* = r(j^*)$ and $\ell^* \leq \ell$ be minimal with the property that each large job that was placed in $\mathcal{M}(\ell^*, r^*)$ is less flexible than $j^*$ and each free candidate machine in the interval is free because two other machines in the same respective region already received a large job less flexible than $j^*$. Furthermore, let $J^*$ be the set of large jobs placed in $\mathcal{M}(\ell^*, r^*)$ together with $j^*$. We argue that $\ell(j) \geq \ell^*$ for each $j \in J^*$. Otherwise, there exists a job $j \in J^*$ eligible on machine $\ell^* - 1$. Then there are three possibilities regarding this machine. It was not a candidate before the procedure; it was a candidate and received a job less flexible than $j$ (and therefore also less flexible than $j^*$); or it was a candidate and did not receive a large job because two other machines in the same region received a job less flexible then $j$. Each yields a contradiction to the definition of $\ell^*$. Let $\texttt{fracLarge} = \sum_{i \in \mathcal{M}(\ell^*, r^*)} \sum_{j \in \mathcal{L}} x_{ij}$ be the sum of fractional large jobs in $\mathcal{M}(\ell^*, r^*)$ according to $x$. Note that we did show $J^* \subseteq \mathcal{J}(\ell^*, r^*)$ and hence $\texttt{fracLarge} \geq |J^*|$.

Let $M^* \subseteq \mathcal{M}(\ell^*, r^*)$ be the set of machines that did receive a fraction of a job from $J^* \cup \mathcal{H}$. Then Equation (3.4) implies $\sum_{i \in M^*} \sum_{j \in \mathcal{H}} x_{ij} \leq |M^*| - |J^*|$, and furthermore Lemma 3.2 yields that at most $|M^*| - |J^*|$ huge jobs are placed on machines from $|M^*|$. Hence, there are at least $|J^*|$ candidate machines in $\mathcal{M}(\ell^*, r^*)$. Since not all of the jobs from $J^*$ have been placed by the procedure, there is, therefore, at least one free machine in $i^* \in \mathcal{M}(\ell^*, r^*)$. The definition of $\ell^*$ yields that two jobs less flexible than $j^*$ have been placed in the same region as $i^*$ and these jobs have to be included in $J^*$ (and the machines they are placed on in $\mathcal{M}(\ell^*, r^*)$).

We now take a closer look at the regions (partially) included in $\mathcal{M}(\ell^*, r^*)$. Let $\ell^* \in R^s$, $r^* \in R^t$, and $k = |\{s, \dots, t\}|$. We consider three cases: If we have $\ell^* = i_s$ and $r^* = i_{t+1}$, i.e., the borders of the interval correspond to the (original) outer borders of their regions, then each of the regions $R^s, \dots, R^t$ did receive at least one job from $J^*$ and one received at least two yielding $k \leq |J^*| - 2 \leq \texttt{fracLarge} - 2$. Moreover, if $\ell^* > i_s$ or $r^* < i_{t+1}$, then one of the regions $R^s, \dots, R^t$ may not have received a job from $J^*$ changing the inequality to $k \leq \texttt{fracLarge} - 1$. Lastly, if both $\ell^* > i_s$ and $r^* < i_{t+1}$, then the two outer regions may have received no job from $J^*$ yielding $k \leq \texttt{fracLarge}$. However, Lemma 3.4 considers the same three cases, yielding $\texttt{fracLarge} < k + 2$, $\texttt{fracLarge} < k + 1$, and $\texttt{fracLarge} < k$, respectively. ↯ ■

Lastly, we place the small jobs.

**Step 4 Placement of Small Jobs:** Starting with the first machine, do the following:

- Let $i^*$ be the current machine and $J$ the set of jobs that have not been placed yet and are eligible on $i^*$.
- Successively place least flexible jobs $j$ on $i^*$, i.e., set $\bar{x}_{i^* j} = 1$, until either $J = \emptyset$ or placing the next job would raise the load of $i^*$ above $(2 - \gamma)T$.
- Consider the next machine in the ordering or stop if there is none.

We argue that this procedure works under certain conditions:

**Lemma 3.6** All small jobs are placed (on eligible machines) by the described procedure if $\gamma \leq \xi$, $\gamma + \xi \leq \frac{1}{12}$, and $8\xi + 7\gamma \leq 0.75$ hold. In the resulting schedule, each machine has a load of at most $(2-\gamma)T$.

*Proof.* For the sake of easier presentation, we assume $T = 1$ in the following (this can be established via scaling). As usual, the second statement is easy to see, and we prove the first via contradiction. To that end, let $j^*$ be a small job we cannot place. Let $\texttt{load}(i) = \sum_{j \in \mathcal{J}} p_j \bar{x}_{ij}$ be the load machine $i \in \mathcal{M}$ did receive. We call a machine *full* if we stop placing small jobs on it because placing another job would have caused a load of more than $2-\gamma$. Note that $\texttt{load}(i) > 1.5 - \gamma$ for full machines $i \in \mathcal{M}$. Let $r^* = r(j^*)$. Then $r^*$ is full since we were not able to place $j^*$ and we have $\texttt{load}(i) + p_{j^*} > 2 - \gamma$. Moreover, all the small jobs placed on $r^*$ are less flexible than $j^*$. Let $\ell^*$ be the left-most machine with the property that each machine in $\mathcal{M}(\ell^*, r^*)$ is full and each small job placed on such a machine is less flexible than $j^*$, and let $S^*$ be the set of small jobs placed on $\mathcal{M}(\ell^*, r^*)$ together with $j^*$. We have $\ell(j) \geq \ell^*$ for each $j \in S^*$ since otherwise machine $\ell^* - 1$ has to be full and each small job placed on this machine must be less flexible than $j$ yielding a contradiction to the choice of $\ell^*$. Hence, we have $S^* \subseteq \mathcal{S}(\ell^*, r^*)$.

We establish some further notation. Let `fracHuge`, `fracLarge`, and `fracSmall`, be the summed up number of fractional huge, large, or small jobs, respectively, in $\mathcal{M}(\ell^*, r^*)$, e.g., $\texttt{fracHuge} = \sum_{i \in \mathcal{M}(\ell^*, r^*)} \sum_{j \in \mathcal{H}} x_{ij}$. Furthermore, let $H^*$ and $L^*$ be the sets of huge and large jobs placed in $\mathcal{M}(\ell^*, r^*)$ by the rounding procedure, and $k = |\mathcal{M}(\ell^*, r^*)|$ the length of the interval of machines. Now, we have already established:

$$p(S^*) + p(L^*) + p(H^*) = p_{j^*} + \sum_{i \in \mathcal{M}(\ell^*, r^*)} \texttt{load}(i) > (k-1)(1.5 - \gamma) + (2 - \gamma) \tag{3.6}$$

Furthermore, we have $|H^*| \leq \lceil \texttt{fracHuge} \rceil \leq \lceil \lfloor (k - p(\mathcal{S}(\ell^*, r^*)))/(0.5 + \xi) \rfloor \rceil \leq (k - p(\mathcal{S}(\ell^*, r^*)))/(0.5 + \xi)$ due to Lemma 3.2 and Equation (3.5) yielding:

$$p(S^*) \leq p(\mathcal{S}(\ell^*, r^*)) \leq k - (0.5 + \xi)|H^*| \tag{3.7}$$

On the other hand, already the classical assignment LP constraints upper bound the load in the interval by $k$, which implies:

$$\sum_{i \in \mathcal{M}(\ell^*, r^*)} \sum_{j \in \mathcal{S}} p_j x_{ij} \leq k - \sum_{i \in \mathcal{M}(\ell^*, r^*)} \sum_{j \in \mathcal{H} \cup \mathcal{L}} p_j x_{ij} < k - 0.5 \cdot \texttt{fracLarge} - (0.5 + \xi) \cdot \texttt{fracHuge}$$

Hence, Lemma 3.2 and Lemma 3.5 give us:

$$p(S^*) \leq \sum_{i \in \mathcal{M}(\ell^*, r^*)} \sum_{j \in \mathcal{S}} p_j x_{ij} \leq k - 0.5 \cdot \frac{|L^*| - 4}{2} - (0.5 + \xi) \cdot (|H^*| - 1) \tag{3.8}$$

We conclude the proof by considering two cases. In particular, if $|L^*| \leq 6$, we have:

$$\begin{aligned}
(2 - \gamma) &\overset{(3.6)}{<} p(S^*) + p(L^*) + p(H^*) - (k-1)(1.5 - \gamma) \\
&\overset{(3.7)}{\leq} k - (0.5 + \xi)|H^*| + (0.5 + \xi)|L^*| + |H^*| - (k-1)(1.5 - \gamma) \\
&= (\gamma - 0.5)(k - |H^*| - |L^*|) - \xi|H^*| + \xi|L^*| + \gamma|H^*| + \gamma|L^*| + 1.5 - \gamma \\
&\leq \xi|L^*| + \gamma|L^*| + 1.5 - \gamma \leq 6\xi + 6\gamma + 1.5 - \gamma \leq 2 - \gamma
\end{aligned}$$

Note that we did use $\gamma \leq 0.5$, $\gamma \leq \xi$, and $\gamma + \xi \leq \frac{1}{12}$. If $|L^*| \geq 7$, on the other hand, we have:

$$\begin{aligned}
(2-\gamma) &\overset{(3.6)}{<} p(S^*) + p(L^*) + p(H^*) - (k-1)(1.5-\gamma) \\
&\overset{(3.8)}{<} k - \frac{|L^*|-4}{4} - (0.5+\xi)(|H^*|-1) + (0.5+\xi)|L^*| + |H^*| - (k-1)(1.5-\gamma) \\
&= (\gamma - 0.5)(k - |H^*| - |L^*|) - \frac{|L^*|}{4} - \xi|H^*| + \xi + \xi|L^*| + \gamma|H^*| + \gamma|L^*| + 3 - \gamma \\
&\leq (\xi + \gamma - \frac{1}{4})|L^*| + \xi + 3 - \gamma \\
&\leq (\xi + \gamma - \frac{1}{4})7 + \xi + 3 - \gamma = 1.25 + 8\xi + 7\gamma - \gamma \leq 2 - \gamma
\end{aligned}$$

This time, we used $\gamma \leq 0.5$, $\gamma \leq \xi$, $\xi + \gamma \leq 0.25$ and $8\xi + 7\gamma \leq 0.75$. Since we did reach the contradiction $(2-\gamma) < (2-\gamma)$ in both cases, the proof is complete. ■

Lastly, we choose values for $\xi$ and $\gamma$, which satisfy all the requirements of the above lemma and maximize $\gamma$. The biggest $\gamma$ is achieved by setting $\gamma = \xi = \frac{1}{24}$. This concludes the proof of Theorem 3.2.

## 3.3 A Summary of Our Complexity Results

We want to introduce our complexity results to frame the context of our constructive algorithm. We give an intuitive summary here without going too much into the technical details. We give the underlying basic reduction in more detail and sketch how to adapt the techniques already present there for the other problems. The complexity results build upon the ones in [66]. In that work, a satisfiability problem denoted as 3-SAT* was introduced and shown to be NP-hard, and all reductions in the present work start from this problem. An instance of the problem 3-SAT* is a conjunction of clauses with exactly 3 literals each. Each of the clauses is either a 1-in-3-clause or a 2-in-3-clause; that is, they are satisfied if exactly one or two of their literals, respectively, evaluate to *true* in a given truth assignment. We denote a 1-in-3-clause (2-in-3-clause) with literals $x$, $y$, and $z$ as $(x,y,z)_1$ ($(x,y,z)_2$). Furthermore, we use the notation $[n] = \{0,\ldots,n-1\}$ for each integer $n$. There are as many 1-in-3-clauses in a 3-SAT* instance as there are 2-in-3-clauses, and each literal occurs exactly twice. Hence, a minimal example for a 3-SAT* instance is given by $(x_0,x_1,\neg x_2)_1 \wedge (\neg x_0,x_1,x_2)_1 \wedge (x_0,\neg x_1,\neg x_2)_2 \wedge (\neg x_0,\neg x_1,x_2)_2$. We have two 1-in-3-clauses and two 2-in-3-clauses, and two occurrences of $x_i$ and $\neg x_i$ for each $i \in [3]$. The formula is satisfied if we map every variable to *false*.

In each reduction, we start with an instance $I$ of 3-SAT* with $m$ many 1-in-3-clauses $C_0,\ldots,C_{m-1}$, $m$ many 2-in-3-clauses $C_m,\ldots,C_{2m-1}$ and $n$ variables $x_0,\ldots,x_{n-1}$. Since there are $2m$ clauses with 3 literals each and 4 occurrences for each variable, we have $6m = 4n$. In the following, the precise positions of the occurrences of the variables are important, and we have to make them explicit. To this end, let for each $j \in [n]$ and $t \in [4]$ the pair $(j,t)$ correspond to the first or second positive occurrence of variable $x_j$ if $t = 0$ or $t = 1$, respectively, and to the first or second negative occurrence of variable $x_j$ if $t = 2$ or $t = 3$. Furthermore, let $\kappa : [n] \times [4] \to [2m] \times [3]$ be the bijection that maps $(j,t)$ to the corresponding clause index and position in that clause. For instance, in the above example we have $\kappa(0 \mathrel{\hat{=}} x_0, 2 \mathrel{\hat{=}} \text{third occurance}) = (1 \mathrel{\hat{=}} \text{second clause}, 0 \mathrel{\hat{=}} \text{first variable of clause})$ and $\kappa(2,1) = (3,2)$.

Next, we construct an instance $I'$ of the problem considered in the respective case. For the restricted assignment type problems, all job sizes are integral and upper bounded by some constant $T$ such that the overall size of the jobs equals $|\mathcal{M}|T$. Hence, if a machine receives jobs with an overall size of more or less than $T$, the objective function value is worse than $T$ for both the makespan and fair allocation case. The goal is to show that there is a schedule with makespan $T$ for $I'$, if and only if $I$ is a yes-instance. This rules out approximation algorithms with a rate lower than $(T+1)/T$ for the makespan problem since the overall load is $|\mathcal{M}|T$. For the low rank problem, we first design a restricted assignment reduction using the above approach and then show that there exist low rank scheduling instances that approximate the restricted assignment instance with arbitrary precision.

**Simple Reduction.** We start with a simple reduction for the general restricted assignment problem (with arbitrary restrictions), introducing several ideas and gadgets relevant to the following reductions. Note that the reduction is very similar to the one by Ebenlendr et al. [24] and to a reduction in [66].

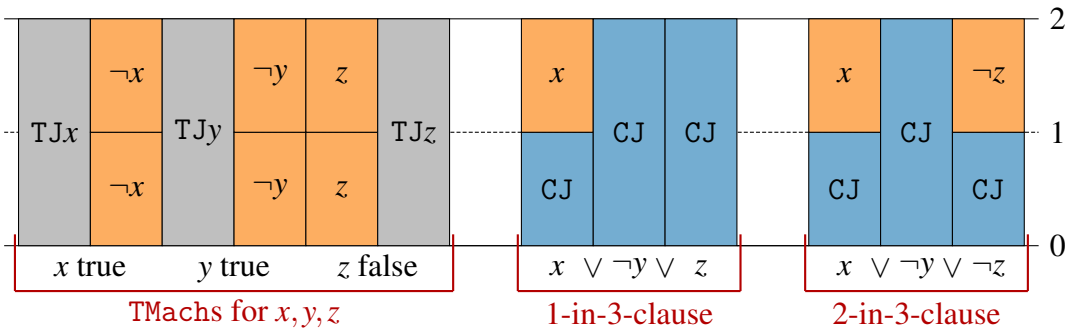



Figure 3.1: (Incomplete) example schedule from the reduction. TJobs (grey) form the assignment, in this case, $x$ and $y$ are true, and $z$ is false. The VJobs (orange) corresponding to the chosen assignment can fill empty slots on the clause machines (CMachs), while the VJobs not corresponding to the assignment can be put on the respective TMach. The CMachs have one or two empty slots if they represent a 1-in-3-clause or 2-in-3-clause, respectively. The location of these slots is flexible, as the CJobs (blue) of a clause can be scheduled in any permutation on the respective machines.

We have three types of basic jobs and machines, namely, truth assignment machines and jobs that are used to assign truth values to variables, clause machines and jobs that model clauses being satisfied, and variable jobs that connect the first two types. More formally:

- **Truth assignment machines:** $\texttt{TMach}(j,0)$ and $\texttt{TMach}(j,1)$ for each $j \in [n]$.
- **Truth assignment jobs:** $\texttt{TJob}(j)$ with size 2, eligible on $\{\texttt{TMach}(j,0), \texttt{TMach}(j,1)\}$ for each $j \in [n]$.
- **Clause machines:** $\texttt{CMach}(i,s)$ for each $i \in [2m]$ and $s \in [3]$ .
- **Clause jobs:** $\texttt{CJob}(i,s)$ eligible on $\{\texttt{CMach}(i,s') \mid s' \in [3]\}$, for each $i \in [2m]$ and $s \in [3]$. The job $\texttt{CJob}(i,0)$ has size 1, $\texttt{CJob}(i,2)$ has size 2, and $\texttt{CJob}(i,1)$ has size 2 if clause $C_i$ is a 1-in-3-clause and size 1 otherwise.
- **Variable jobs:** $\texttt{VJob}(j,t)$ with size 1, eligible on $\{\texttt{TMach}(j, \lfloor \frac{t}{2} \rfloor), \texttt{CMach}(\kappa(j,t))\}$ for each $j \in [n]$ and $t \in [4]$.

Note that the overall job size $\sum_{j\in\mathcal{J}} p(j)$ is equal to $2|\mathcal{M}|$. Consider the case that we have a satisfying truth assignment for instance $I$ (as in Figure 3.1). If variable $x_j$ is assigned to *true*, we place $\texttt{TJob}(j)$ on $\texttt{TMach}(j,0)$, $\texttt{VJob}(j,0)$ and $\texttt{VJob}(j,1)$ on $\texttt{CMach}(\kappa(j,0))$ and $\texttt{CMach}(\kappa(j,1))$, respectively, together with local size 1 clause jobs. Furthermore, $\texttt{VJob}(j,2)$ and $\texttt{VJob}(j,3)$ are placed on $\texttt{TMach}(j,1)$ and $\texttt{CMach}(\kappa(j,2))$ and $\texttt{CMach}(\kappa(j,3))$ each receive a local size 2 clause job. If variable $x_j$ is assigned to *false*, we place $\texttt{TJob}(j)$ on $\texttt{TMach}(j,1)$, and the placement strategy of the positive and negative variable jobs is reversed. The placement of the clause jobs has to work out since the truth assignment is satisfying. This approach yields a schedule with a makespan of 2. The other way around, we can create a satisfying truth assignment for $I$, by basing the assignment of $x_j$ on the placement of $\texttt{TJob}(j)$, and hence we have:

**Lemma 3.7** There is a satisfying truth assignment for $I$, if and only if there is a schedule with makespan 2 for $I'$.

This reduction forms the basis of all the other ones considered in this section.

### 3.3.1 Three Resources

The RAR(3) case is the cleanest usage of the previous construction. In the end, schedules look the same as before, though the eligible machines are defined through resources and are slightly different. Imagine all truth assignment machines TMach on a line. With one resource that is increasing throughout the machines and one that is decreasing, we can easily choose demands, such that each $\texttt{TJob}(j)$ is only eligible on $\texttt{TMach}(j,0)$ and $\texttt{TMach}(j,1)$. The clause machines CMach are interleaved with the truth assignment machines corresponding to the same variable on that conceptual machine line. Though the specific choices of resource capacities and demands are a bit technical, the construction ensures that the truth assignment and variable jobs have the same sets of eligible machines as before. The clause jobs have slightly different eligibilities. We use the third resource to model that the $i$th set of three clause jobs is eligible on all clause machines $\texttt{CMach}(i',s)$ with $i' \geq i$ (instead of $i' = i$ as before). Since each clause machine needs exactly one clause job to ensure a makespan of 2, placing the last set of clause jobs fixes the machines for the penultimate set of clause jobs, and so on.

Hence, Lemma 3.7 works the same as before, and we have:

**Theorem 3.3** There is no better than 1.5-approximation for RAR(3) and no better than 2-approximation for the fair allocation version of this problem, unless P = NP.

### 3.3.2 Two Resources

The reduction for RAR(2) follows the same ideas but is more technical. We have to reintroduce the restrictions we are losing from the third resource via arguments over job sizes and the target makespan. For that, we have to increase the target makespan of a schedule corresponding to a yes-instance of the 3-SAT$^*$ to 7, which also changes the inapproximability result reached in the end. Through this technical adaptation that we omit here, we get:

**Theorem 3.4** There is no better than $\frac{8}{7}$-approximation for RAR$(2)$ and no better than $\frac{7}{6}$-approximation for the fair allocation version of this problem, unless P $=$ NP.

### 3.3.3 Interval Restrictions

In order to motivate the new ideas for the RAI reduction and to make them easier to understand, it is helpful to revisit the reduction from [66] first. One of the main ingredients in that result is a simple trick we use extensively.

**Pyramid Trick.** Consider the case depicted in Figure 3.2. We have $2\ell$ consecutive machines and $\ell$ pairs of jobs. The $i$-th pair of jobs is eligible on the $i$-th machine and up to and including the $(2\ell+1-i)$-th machine. Furthermore, we assume that each machine has to receive at least one of the jobs. Then the first and last machine each have to receive one job from the first pair because there are no other eligible jobs that can be processed on these machines. Now, the same argument can be repeated for the second and second to last machine and so on. Hence, machine $i$ and $(2\ell+1-i)$ each have to receive exactly one job from pair $i$.

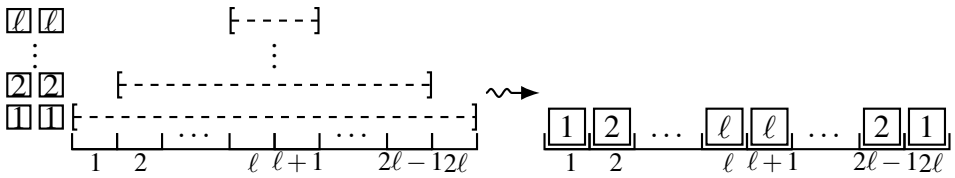



Figure 3.2: A visualization of the pyramid trick. Squares represent jobs, and the intervals in brackets next to them represent their sets of eligible machines.

**Sorting.** Since we are considering the interval case, we have to fix some ordering of the machines. Imagine that the truth assignment machines are placed on the left and the clause machines on the right. We could use similar truth assignment and clause jobs as in the reductions before. However, variable jobs were eligible on one truth assignment machine and one clause machine. Since they have to be eligible on an interval, all machines in between would also be eligible in a naive adaptation of the reduction. We want to use the pyramid trick to deal with this problem. The main work in [66] was to sort the information regarding the variables made in the truth assignment gadget to enable the use of the pyramid trick. Several gadgets have been introduced that were intertwined with the truth assignment gadget and carefully built up the ordered information using the pyramid trick and interlocking job sizes. The problem with this approach is that the job sizes can get big rather fast if too many different job types are eligible on the same machines resulting in a high value for $T$ (and, therefore, a weak inapproximability result). The main idea in our work is to decouple the decision and the sorting process and make the sorting process as simple as possible to enable smaller job sizes and, therefore, a stronger result. Curiously, the sorting process in [66] could be interpreted as some variant of insertion sort, while the one used in the present reduction resembles bubble sort.

Keeping in mind the general ideas from before, we roughly sketch the new construction. We have a truth assignment gadget that determines the truth values of the variables and is followed by a gateway gadget whose sole purpose is to decouple the used job sizes

in the truth assignment gadget and the sorting gadget. Next, there is the sorting gadget that slowly reorders the information about the decisions in the truth assignment gadget. Lastly, there is the clause gadget in which the truth assignment is evaluated. We again refer to our paper [67] for the concrete reduction. Using that reduction, we can show the following:

**Theorem 3.5** There is no better than $\frac{9}{8}$-approximation for RAI and no better than $\frac{8}{7}$-approximation for the fair allocation version of this problem, unless $\mathrm{P} = \mathrm{NP}$.

## 3.4 Future Work

We conclude this chapter with a brief discussion of possible future research directions in this topic area. There are some obvious questions that can be pursued, building directly upon the presented results. Most notably, it might be possible that the analysis of our approximation algorithm is not tight, and a better parameter could therefore be chosen. Further narrowing, or even closing, the gap between the best-known approximation and the inapproximability result for RAI could also be approached via different algorithmic techniques or by even stronger inapproximability results for RAI. Of course, an improved approximation ratio (or inapproximability) for any problem of the family would be interesting to develop. We would like to highlight $\mathrm{LRS}(2)$, in particular, as the, in some sense, easiest problem in the family without a known polynomial time approximation with a ratio better than 2. Lastly, only very little is known regarding fixed-parameter tractable algorithms for this family of problems, even though fixed-parameter tractability, in general, has gotten a lot of attention lately. For instance, it is open whether $\mathrm{RAR}(1)$ is fixed-parameter tractable with respect to the objective value.

# 4. Many Shared Resources

We consider the *many shared resources scheduling* problem. In this model, we are given a set of resources in addition to the machines and jobs. A job may require exclusive access to one of the resources during its processing time, or in other words, two jobs needing the same resource may not be scheduled in parallel. We show how combinatorial algorithms can be used to tackle this problem and greatly improve the state of the art. We also prove how jobs that need more than one resource increase the complexity of the problem.

This chapter is based on our paper *Scheduling with Many Shared Resources* [21] which was accepted for publication in the proceedings of IPDPS 2023. Overall the paper was developed and written in equal parts by my coauthors and myself. I did more work on the approximation algorithms (see Section 4.2 and Section 4.3) and less work on the approximation schemes. The approximation schemes are, therefore, only summarized in this thesis (see Section 4.4). The part on inapproximability in Section 4.5 is completely my work and includes three results, of which the last two are original to this thesis.

## 4.1 Introduction

The study of scheduling problems with additional resources has a long and rich tradition. Already in 1983, Blazewicz et al. [9] provided a classification for such problems along with basic hardness results, and several additional surveys have been published since then [7, 8, 25]. The problem of makespan minimization on identical parallel machines with many shared resources or many shared resources scheduling (MSRS) for short, in particular, was introduced by Hebrard et al. [36]. In this model, jobs need exclusive access to a specific resource from a resource set during their processing.

### 4.1.1 Problem Definition

We consider many shared resources scheduling (MSRS). In this problem, we are given $m$ identical machines, a set $\mathcal{J}$ of $n$ jobs, and a processing time or size $p_j \in \mathbb{N}_{\geq 0}$ for each job $j \in \mathcal{J}$. Furthermore, each job needs exactly one additional shared resource in order to be executed and no other job needing the same resource can be processed at the same time. Hence, the jobs are partitioned into (non-empty) classes $\mathcal{C}$, i.e., $\bigcup \mathcal{C} = \mathcal{J}$, such that each class corresponds to one of the resources. A schedule $(\pi, t)$ maps each job to a machine

$\pi : \mathcal{J} \to \{1,\dots,m\}$ and a starting time $t : \mathcal{J} \to \mathbb{N}_{\geq 0}$. It is called valid if no two jobs overlap on the same machine and no two jobs of the same class are processed in parallel, i.e.:

- $\forall j, j' \in \mathcal{J}, j \neq j'$ with $\pi(j) = \pi(j')$: $t(j) + p_j \leq t(j')$ or $t(j') + p'_j \leq t(j)$
- $\forall c \in \mathcal{C} : j, j' \in c, j \neq j'$: $t(j) + p_j \leq t(j')$ or $t(j') + p'_j \leq t(j)$

The makespan $C_{\max}$ of a schedule is defined as $\max_{j \in \mathcal{J}} t(j) + p_j$ and the goal is to find a schedule with minimum makespan. Note that MSRS also models the case in which some jobs do not need a resource since in this case private resources can be introduced.

### 4.1.2 State of the Art and Motivation

As already mentioned, the MSRS problem was introduced by Hebrard et al. [36]. In their application scenario, a satellite has a communication link with a ground station only for very limited time, during which files stored on several memory banks should be downloaded via several download channels. The download time of a file is proportional to its size, but only one file from each memory bank can be downloaded at the same time. Hence, the download channels correspond to the parallel machines, the files to the jobs, and the memory banks to the resources. In that paper, they provided a $(2m/(m+1))$-approximation for the problem. Strusevich [81] revisited MSRS and presented an additional application in human resource management. Moreover, he provided a faster, alternative $(2m/(m+1))$-approximation that is claimed to be simpler as well and a $6/5$-approximation for the case with only two machines. The work also extends the three field notation for scheduling problems based on the convention for additional resources introduced in [9] to encompass the problem at hand. In particular, MSRS is denoted as $P|res \cdot 111|C_{\max}$ (identical parallel machines | resources each available once, each job needs one | makespan objective) in this notation.

The most recent result regarding MSRS is due to Dósa et al. [23] who provided an efficient polynomial time approximation scheme (EPTAS) for MSRS with a constant number of machines. In fact, the EPTAS even works for a more general setting where each job $j$ additionally may only be assigned to a machine belonging to a given set $\mathcal{M}(j)$ of eligible machines.

Since MSRS includes makespan minimization on identical machines (without resource constraints) as a subproblem, it is NP-hard already on two machines and strongly NP-hard if the number of machines is part of the input due to straightforward reductions from the partition and 3-partition problem, respectively. Hence, approximation schemes are essentially the best we can hope for.

The MSRS problem has also been considered with regard to the total completion time objective [44, 45]. The study of this variant is motivated by a scheduling problem in the semiconductor industry. On the one hand, the authors show NP-hardness for generalizations of the problem. On the other, they argue that the approach yielding a polynomial time algorithm for total completion time minimization in the absence of resource constraints leads to a $(2-1/m)$-approximation for the considered problem.

Another way of looking at MSRS is to consider it as a variant of scheduling with conflicts, where a conflict graph is given in which the jobs are the vertices and no two jobs connected by an edge may be processed at the same time. This problem was introduced for unit processing times by Baker and Coffman in 1996 [4]. It is known to be APX-hard [28] already on two machines with job sizes at most 4 and a bipartite agreement graph, i.e., the complement of the conflict graph. There are many positive and negative results for different versions of this problem (see, e.g., [4, 28] and the references therein). For

instance, the problem is NP-hard on cographs with unit-size jobs but polynomial-time solvable if the number of machines is constant [11]. Note that in the case of MSRS, we have a particularly simple cograph, i.e., a collection of disjoint cliques.

### 4.1.3 Our Results

We present a $5/3$-approximation in Section 4.2, a $3/2$-approximation in Section 4.3, approximation schemes in Section 4.4, and inapproximability results in Section 4.5. Note that the $5/3$- and $3/2$-approximation have better approximation ratios than the previously known $(2m/(m+1))$-approximation [36] already for 6 and 4 machines, respectively.

The $5/3$-approximation is a simple and fast algorithm that is based on placing full classes of jobs taking special care of classes containing jobs with particularly big sizes and of classes with large processing time overall. While the $3/2$-approximation reuses some of the ideas and observations of the first result, it is much more involved. To achieve the second result, we first design a $3/2$-approximation for the instances in which jobs cannot be too large relative to the optimal makespan and then design an algorithm that carefully places classes containing such large jobs and uses the first algorithm as a subroutine for the placement of the remaining classes. Note that our approaches are very different from the one in [36], which successively chooses jobs based on their size and the size of the remaining jobs in their class and then inserts them with some procedure designed to avoid resource conflicts, and the one in [81], which merges the classes into single jobs to avoid resource conflicts.

We summarize the two EPTAS results from our original paper for MSRS. The first works if the number of machines is constant. The second uses resource augmentation for the general case. In particular, we need $\lfloor(1+\varepsilon)m\rfloor$ many machines in the latter result. Both results make use of the basic framework introduced in [38], which in turn utilizes relatively recent algorithmic results for integer programs (IPs) of a particular form – so-called N-fold IPs. Compared to the mentioned work by Dósa et al. [23] – which provides an EPTAS for the case with a constant number of machines as well – our result is arguably simpler and faster (going from at least triply exponential in $m/\varepsilon$ to doubly exponential). We also provide the result with resource augmentation for the general case, which may be refined in the future to work without resource augmentation as well.

Finally, we provide inapproximability results for variants of MSRS where each job may need more than one resource. In particular, we show that there is no better than $5/4$-approximation for the variant of MSRS with multiple resources per job, unless P $=$ NP, even if no job needs more than three resources and all jobs have processing time 1, 2 or 3. An alternative version of that result that we also sketch gives the same inapproximability result if all jobs have a processing time of 1 and no job needs more than five resources. A third approach shows that there exists no better than $4/3$-approximation for the variant of MSRS with multiple resources per job, unless P $=$ NP when all jobs have processing time 1, but without a limit on the number of resources per job. Previously, the APX-hardness result due to Even et al. [28] for scheduling with conflicts was known, which did focus on a different context and in particular, does not provide bounds regarding the number of resources a job may require.

### 4.1.4 Further Related Work

As mentioned above, there exists extensive research regarding scheduling with additional resources and we refer to the surveys [7–9, 25] for an overview. For instance, the variant with only one additional shared renewable resource where each job needs some fraction of the resource capacity has received a lot of attention (see [40, 42, 49, 73] for some relatively recent examples). Interestingly, Hebrard [36] pointed out that this basic setting is more closely related to MSRS than it first appears: Consider the case that we have dedicated machines, i.e., each job is already assigned to a machine, and we only have to choose the starting times; each job needs one unit of the single additional shared resource, and the shared resource has some integer capacity. This problem is equivalent to MSRS if the multiple resources take on the roles of the machines and the machines take the role of the single resource. Hence, results for variants of this setting translate to MSRS as well. For instance, MSRS can be solved in polynomial time if at most two classes include more than one job [47] and [35] yields a $(3+\varepsilon)$-approximation.

Scheduling with conflicts has also been studied from the orthogonal perspective, where jobs that are in conflict may not be processed on the same *machines*. This problem was already studied in the 1990s (see e.g. [11, 12]), and there has been a series of recent results [18, 32, 74] regarding the setting corresponding to MSRS where the conflict graph is a collection of disjoint cliques.

### 4.1.5 Preliminaries

We introduce some additional notation and a first observation that will be used throughout the following sections.

For any set of jobs $X$, let $p(X) = \sum_{j \in X} p_j$ denote its total processing time. Also, let $p(j) = p_j$ for all jobs $j \in \mathcal{J}$. While creating or discussing a schedule, for any machine $m$, denote by $p(m)$ the (current) total load of jobs on that machine $m$. Subsequently, for a set of machines $M$, $p(M) = \sum_{m \in M} p(m)$.

For any combination of a set $X \in \{\mathcal{J}, \mathcal{C}\}$, a relation $* \in \{<, \leq, \geq, >\}$, and a number $\lambda$, we define $X_{*\lambda} = \{x \in X \mid p(x) * \lambda\}$. Furthermore, given an interval $v$ let $X_v = \{x \in X \mid p(x) \in v\}$. For example it holds that $\mathcal{J}_{>1/2} = \{j \in \mathcal{J} \mid p(j) > 1/2\}$ and $\mathcal{C}_{(1/2,3/4]} = \{c \in \mathcal{C} \mid p(c) \in (1/2, 3/4]\}$.

**Observation 1** It holds that $\mathrm{OPT} \geq \max\{\frac{p(\mathcal{J})}{m}, \max_{c \in \mathcal{C}} p(c)\}$.

Hence, we assume that $m < |\mathcal{C}|$ as otherwise, there is a trivial schedule with one machine per class. Furthermore, let us assume that we sort the jobs in decreasing order of processing time. Consider the jobs $j_m$ and $j_{m+1}$ at position $m$ and $m+1$. Note that it has to hold that $\mathrm{OPT} \geq p(j_m) + p(j_{m+1})$, since either $j_{m+1}$ has to be scheduled on the same machine as one of the first $m$ jobs, or two of the first $m$ jobs have to be scheduled at the same machine.

## 4.2 A 5/3-approximation

In this section, we introduce a first simple algorithm that gives some intuition on the problem that will be used more cleverly in the next section. We start by lower bounding the makespan $T$ of an optimal schedule and construct a schedule with makespan at most $\frac{5}{3}T$. The algorithm works by placing full classes of jobs in a specific order. More precisely,

first classes that contain a job of size at least $\frac{1}{2}T$, then classes with total processing time larger than $\frac{2}{3}T$, and lastly, all residual classes get placed.

**Theorem 4.1** There exists an algorithm that, for any instance $I$ of MSRS, finds a schedule with makespan bounded by $\frac{5}{3}T$ in $\mathcal{O}(|I|)$ steps, where for the jobs $j_m$ and $j_{m+1}$ with $m$-th and $(m+1)$-st largest processing time we define

$$T := \max\{\frac{1}{m}p(\mathcal{J}), \max_{c\in\mathcal{C}} p(c), p(j_m)+p(j_{m+1})\}.$$

As noted earlier, $T$ denotes a lower bound on the makespan. We scale each job by $1/T$. As a consequence, all jobs have a processing time in $(0,1]$ and the total load is bounded by $m$. Denote by $\mathcal{C}_{B^+} := \{c \in \mathcal{C} \mid |c \cap \mathcal{J}_{>1/2}| = 1\}$ all classes containing a job of size greater than $1/2$. We aim to find a schedule with makespan in $[1,5/3]$. The following two observations are directly implied by the definition of $T$.

**Observation 2** For each class $c \in \mathcal{C}$ it holds that $|c \cap \mathcal{J}_{>1/2}| \leq 1$.

**Observation 3** It holds that $|\mathcal{C}_{B^+}| = |\mathcal{J}_{>1/2}| \leq m$.

Lastly, we address classes with a large total processing time.

**Lemma 4.1** Each class $c \in \mathcal{C}_{>2/3} \setminus \mathcal{C}_{B^+}$ can be partitioned into parts $c_1$ and $c_2 = c \setminus c_1$ such that $1/3 \leq p(c_1) \leq 2/3$ and $p(c_2) \leq 2/3$. This partition can be found in time $\mathcal{O}(|c|)$.

*Proof.* If there exists a job $j_\top$ in $c$ with $p(j_\top) > 1/3$, we define $c_1 = \{j_\top\}$ and $c_2 = c \setminus c_1$. Note that $c$ does not contain a job with processing time larger than $1/2$ and hence, $p(c_1) \in (1/3, 1/2]$ and $p(c_2) = p(c) - p(c_1) < 1 - 1/3 = 2/3$.

Otherwise, greedily add jobs from $c$ to an empty set $c_1$ until $p(c_1) \geq 1/3$ and set $c_2 = c \setminus c_1$. Since all the jobs of $c$ have processing time at most $1/3$, it holds that $p(c_1) \in [1/3, 2/3]$. Consequently, it holds that $p(c_2) \leq 2/3$ as well. ■

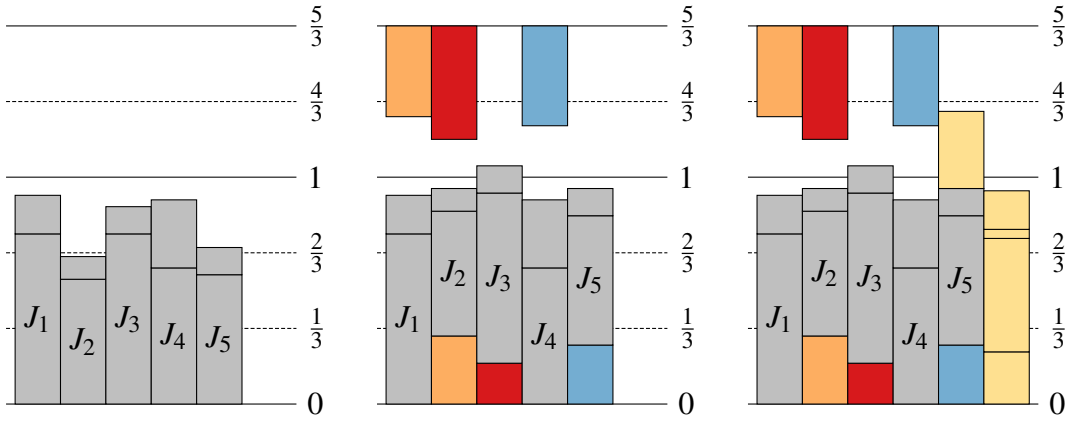
**Algorithm:** `Algorithm_5/3`

**Step 1** Consider all classes containing a job with processing time larger than $1/2$, $\mathcal{C}_{B^+}$. Each of these classes is assigned to an individual machine, and all jobs from such a class are scheduled consecutively, see Figure 4.1a.

**Step 2** Consider all remaining classes with total processing time larger than $2/3$, $\mathcal{C}_{>2/3} \setminus \mathcal{C}_{B^+}$. Try to add these classes on the machines filled with the classes $\mathcal{C}_{B^+}$ and afterward proceeds to empty machines, see Figure 4.1b. If the considered machine has load in $(1,5/3]$, *close* the machine and no longer attempt to place any other job on it. Note that after placing the classes $\mathcal{C}_{B^+}$ all machines remained open. Let $m_i$ be the machine we try to place class $c \in \mathcal{C}_{>2/3} \setminus \mathcal{C}_{B^+}$ on. If $m_i$ has load $p(m_i) \leq 5/3 - p(c)$, place the entire class on this machine and close it. Otherwise, partition the class $c$ in two parts $c_1$ and $c_2$ such that $p(c_2) \leq p(c_1) \leq 2/3$ (cf. Lemma 4.1). Place the larger

part $c_1$ on the current machine starting at $5/3 - p(c_1)$ and close it, moving to the next machine. All jobs on this machine are delayed such that the first job starts at $p(c_2)$. All jobs from $c_2$ are scheduled between 0 and $p(c_2)$ on this machine. If it has load of at least 1, this machine is closed as well.

**Step 3 — Greedy.** Finally, place the classes $\mathcal{C}_{\leq 2/3} \setminus \mathcal{C}_{B^+}$, see Figure 4.1c. Consider the residual machines one after another and add each class $c \in \mathcal{C}_{\leq 2/3} \setminus \mathcal{C}_{B^+}$ entirely to the considered machine. As soon as the load of a machine exceeds 1, close it and move to the next.




(a) Classes with large jobs (b) Placing large classes (c) Adding all other classes

Figure 4.1: The three steps of the algorithm (where $\mathcal{J}_{>1/2} = \{J_1, \ldots, J_5\}$)

## Algorithm Correctness.

**Lemma 4.2** Given any instance $I = (m, \mathcal{C})$ of MSRS, `Algorithm_5/3` produces a feasible schedule with makespan at most $\frac{5}{3}\mathrm{OPT}(I)$.

*Proof.* To prove the correctness and approximation ratio of the algorithm, we have to prove the following points:

- All jobs can be scheduled
- The processing times of two jobs from the same class never overlap.
- The latest completion time of a job is given by $5/3$

We start by proving that all jobs are scheduled by showing that the algorithm closes only machines that have a total load of at least 1. Since the total load of the jobs is bounded by $m$, when attempting to schedule the last class, there has to exist a non-closed machine. The only time the algorithm potentially closes a machine with load less than 1 is in step 2 when a class $\mathcal{C}_{>2/3} \setminus \mathcal{C}_{B^+}$ is split into two parts. Let $c_{B^+}$ be the class already on the machine, and $c_1$ and $c_2$ be the parts of the class the algorithm tries to schedule in this step, such that $p(c_1) \geq p(c_2)$. Since the class was split in two by the algorithm, it holds that $p(c_{B^+}) + p(c_1) + p(c_2) > 5/3$. Furthermore, since $p(c_1) + p(c_2) \leq 1$ and $p(c_1) \geq p(c_2)$ it holds that $p(c_2) \leq 1/2$ and hence $p(c_{B^+}) + p(c_1) > 7/6$. Hence that closed machine has a load of at least 1.

Next, we prove that the processing of two jobs from the same class never overlaps in time. Again, the only time one class is scheduled on more than one machine is in step 2. When placing the two parts, these parts do not overlap since they have a processing time of at most 1 and one of the parts starts at 0 while the other ends at $5/3$. The algorithm does not generate any overlapping by shifting jobs already on the machine since those have to originate from classes in $\mathcal{C}_{B^+}$, which each got placed on an individual machine.

Finally, we prove that the latest completion time of a job is given by $5/3$. After step 1 all the machines have a load of at most 1 since each class has a total processing time of at most 1. In step 2, we only add an entire class if the total load is bounded by $5/3$. If a class is split, the part that is added has a total processing time of at most $2/3$. Since before adding this part the machine had a load of at most 1, the load of the closed machine is bounded by $5/3$. ■

The existence and correctness of `Algorithm_5/3` directly proves Theorem 4.1.

## 4.3 A 3/2-approximation

In this section, we introduce the more involved algorithm hinted at earlier. While the general idea is similar, finding a lower bound $T$ for the makespan and then placing classes depending on included big jobs and total processing time, the steps are a lot more granular. We first give a $3/2$-approximation algorithm for instances without jobs of size bigger than $3/4T$. After that, we introduce a second $3/2$-approximation algorithm that places classes with jobs of size bigger than $3/4T$ on distinct machines and fills them with other jobs in a clever way such that we can reuse the first algorithm for the remaining classes.

**Theorem 4.2** There exists an algorithm that, for any given instance $I$ of MSRS, finds a schedule with makespan bounded by $\frac{3}{2}\text{OPT}$ in $\mathcal{O}(n+m\log(m))$ steps.

In the following, let us assume that we have scaled the instance such that $\text{OPT}=1$. In order to provide a $3/2$-approximation algorithm, we consider four different types of jobs. We split the jobs of a given instance into *huge* jobs $\mathcal{J}_H=\mathcal{J}_{>3/4}=\{j\in\mathcal{J}\mid p_j>3/4\}$, *big* jobs $\mathcal{J}_B=\mathcal{J}_{(1/2,3/4]}=\{j\in\mathcal{J}\mid p_j\in(1/2,3/4]\}$, *medium* jobs $\mathcal{J}_M=\mathcal{J}_{(1/4,1/2]}=\{j\in\mathcal{J}\mid p_j\in(1/4,1/2]\}$, and all residual jobs (with a processing time of at most $1/4$) which we refer to as *small* jobs.

Furthermore, turning to the classes $\mathcal{C}$ we define the subset $\mathcal{C}_H=\{c\in\mathcal{C}:|\mathcal{J}_H\cap c|=1\}$ of all classes containing a huge job, the subset $\mathcal{C}_B=\{c\in\mathcal{C}:|\mathcal{J}_B\cap c|=1\}$ of all classes containing a big job, the subset $\mathcal{C}_{\geq 3/4}=\{c\in\mathcal{C}\mid p(c)\geq 3/4\}$ of all classes with a total processing time of at least $3/4$, and the subset $\mathcal{C}_{(1/2,3/4)}=\{c\in\mathcal{C}\mid p(c)\in(1/2,3/4)\}$ of all classes with a total processing time in $(1/2,3/4)$.

**Lemma 4.3** For any normalized optimal schedule and the corresponding partition of $\mathcal{C}$ into $\mathcal{C}_H,\mathcal{C}_B,\mathcal{C}_{\geq 3/4}\setminus(\mathcal{C}_H\cup\mathcal{C}_B)$ and $\mathcal{C}\setminus\mathcal{C}_{\geq 3/4}$ it holds that

$$|\mathcal{C}_H|+\max\left\{|\mathcal{C}_B|,\left\lceil\frac{1}{2}\left(|\mathcal{C}_B|+|\mathcal{C}_{\geq 3/4}\setminus(\mathcal{C}_H\cup\mathcal{C}_B)|\right)\right\rceil\right\}\leq m.$$

*Proof.* Clearly, it holds that $|\mathcal{C}_H|+|\mathcal{C}_B|\leq m$. Let us consider the total load processed in the time corridor between $1/4$ and $3/4$ (over the entire schedule). For each class $c\in\mathcal{C}_H$

we have to schedule at least load $1/2$ in this corridor, since the tallest job in $c$, which has a processing time of at least $3/4$, has to start before $1/4$ and has to end after $3/4$. For each class $c \in \mathcal{C}_B$, at least load $1/4$ is scheduled in this corridor since its big job, which has a processing time in $(1/2,3/4)$, has to end after $1/2$ and has to start before $1/2$. Finally, each class in $\mathcal{C}_{\geq 3/4} \setminus (\mathcal{C}_H \cup \mathcal{C}_B)$ has load of at least $3/4$. Since at most $1/2$ of this load can be scheduled outside of the corridor, there has to be load of at least $1/4$ scheduled inside of this corridor. Hence the total load scheduled in this corridor is at least $\frac{1}{2}|\mathcal{C}_H| + \frac{1}{4}(|\mathcal{C}_B| + |\mathcal{C}_{\geq 3/4} \setminus (\mathcal{C}_H \cup \mathcal{C}_B)|)$.

Since each machine covers at most $1/2$ of this load, it holds that

$$m \geq \left\lceil \frac{\frac{1}{2}|\mathcal{C}_H| + \frac{1}{4}(|\mathcal{C}_B| + |\mathcal{C}_{\geq 3/4} \setminus (\mathcal{C}_H \cup \mathcal{C}_B)|}{\frac{1}{2}} \right\rceil = |\mathcal{C}_H| + \left\lceil \frac{1}{2}\left(|\mathcal{C}_B| + |\mathcal{C}_{\geq 3/4} \setminus (\mathcal{C}_H \cup \mathcal{C}_B)|\right) \right\rceil$$

and that proves the claim. ■

Next, we prove that in $\mathcal{O}(n+m\log(m))$ steps it is possible to find the smallest value $T$ with $\max\{\frac{1}{m}p(\mathcal{J}), \max_{c\in\mathcal{C}} p(c)\} \leq T \leq \text{OPT}$ such that the instance scaled by $1/T$ fulfills the properties from Observations 2 and 3 and Lemma 4.3. The algorithms presented in this section will find a schedule with a makespan of at most $3/2$ for this scaled instance, i.e. the schedule for the original instance will have a makespan of at most $(3/2)T$.

**Lemma 4.4** In $\mathcal{O}(n+m\log(m))$ for any given instance $I$, it is possible to find a lower bound $T \leq \text{OPT}$ such that for the instance normalized by $1/T$ and the corresponding partition of $\mathcal{C}$ into $\mathcal{C}_H, \mathcal{C}_B, \mathcal{C}_{\geq 3/4} \setminus (\mathcal{C}_H \cup \mathcal{C}_B)$ and $\mathcal{C} \setminus \mathcal{C}_{\geq 3/4}$ it holds that

$$|\mathcal{C}_H| + \max\left\{ |\mathcal{C}_B|, \left\lceil \frac{1}{2}\left(|\mathcal{C}_B| + |\mathcal{C}_{\geq 3/4} \setminus (\mathcal{C}_H \cup \mathcal{C}_B)|\right) \right\rceil \right\} \leq m.$$

*Proof.* By Observation 1, we know that we can set $T \geq \max\{\frac{1}{m}p(\mathcal{J}), \max_{c\in\mathcal{C}} p(c)\}$. Let $\tilde{p}_i$ denote the $(m+1)$-st largest processing time (in a list of processing times containing one entry per job). Since each machine can contain at most one job with processing times larger than OPT$/2$, we set $T \geq \max\{\frac{1}{m}p(\mathcal{J}), \max_{c\in\mathcal{C}} p(c), \tilde{p}_m + \tilde{p}_{m+1}\}$. It is possible to find $p_{m+1}$ in $\mathcal{O}(n)$ steps by using the famous median algorithm of Blum et al. [10].

Since each class in $\mathcal{C}_H \cup \mathcal{C}_B$ contains an item with processing time $\geq 1/2$, only the $m$ classes containing the largest items are candidates for these sets. These classes can be found in $\mathcal{O}(n)$ by identifying the largest item of each class and comparing it to $p_{m+1}$. Similarly the number of classes in $\mathcal{C}_{\geq 3/4}$ is bounded by $(4/3)m$, which can be identified in $\mathcal{O}(n)$ by comparing their processing time to $\max\{\frac{1}{m}p(\mathcal{J}), \max_{c\in\mathcal{C}} p(c), \tilde{p}_m + \tilde{p}_{m+1}\}$.

After identifying the potential classes, we have to deal with at most $\mathcal{O}(m)$ classes. For each of these classes there exist three threshold values for $T \in \mathbb{N}$ (i.e., $\lceil \frac{4}{3}(\max_{j\in c} p_j) + 1/3 \rceil, 2(\max_{j\in c} p_j) + 1$, and $\lceil \frac{4}{3}p(c) + 1/3 \rceil$), that would categorize these classes to be no longer in $\mathcal{C}_H$, $\mathcal{C}_B$, and $\mathcal{C}_{\geq 3/4}$, respectively, which after the first two steps can be found in $\mathcal{O}(m)$ for all the classes, since they depend on the largest processing time in the class and the total processing time of that class.

The algorithm can take all these values and sort them by size in $\mathcal{O}(m\log(m))$. Via binary search in $\mathcal{O}(m\log(m))$, it is possible to find the smallest value $T$ such that $T \geq$

$\max\{\frac{1}{m}p(\mathcal{J}), \max_{c\in\mathcal{C}} p(c), 2p_{m+1}\}$ and for the instance normalized by $1/T$ and the corresponding partition into of $\mathcal{C}$ into $\mathcal{C}_H, \mathcal{C}_B, \mathcal{C}_{\geq 3/4} \setminus (\mathcal{C}_H \cup \mathcal{C}_B)$ and $\mathcal{C} \setminus \mathcal{C}_{\geq 3/4}$ it holds that

$$|\mathcal{C}_H| + \max\left\{ |\mathcal{C}_B|, \left\lceil \frac{1}{2}\left(|\mathcal{C}_B| + |\mathcal{C}_{\geq 3/4} \setminus (\mathcal{C}_H \cup \mathcal{C}_B)|\right)\right\rceil \right\} \leq m. \qquad \blacksquare$$

In the following, we only consider the instance that was scaled by $1/T$. We present two Lemmas stating the possibility of partitioning some classes into two parts that will be scheduled on two different machines.

**Lemma 4.5** Let $c \in \mathcal{C}_{\geq 3/4}$ and $\max_{j\in c} p_j \leq 3/4$. Then $c$ can be partitioned into two parts $\check{c}$ and $\hat{c}$ with $p(\check{c}) \leq 1/2$ and $p(\hat{c}) \leq 3/4$ and $p(\check{c}) \leq p(\hat{c})$. Furthermore, if $\max_{j\in c} p_j \leq 1/2$, it holds that $p(\check{c}) \in (1/4, 1/2]$ or $p(\hat{c}) \in (1/4, 1/2]$.

*Proof.* Let $c \in \mathcal{C}_{\geq 3/4}$ and $\max_{j\in c} p_j \leq 3/4$. If $\max_{j\in c} p_j > 1/2$, we set $\hat{c}$ to include the job from $c$ with size bigger than $1/2$ and $\check{c} = c \setminus \hat{c}$. If $\max_{j\in c} p_j \in (1/4, 1/2]$, then let $c'$ include a maximal job from $c$ and let $\hat{c}, \check{c} \in \{c', c \setminus c'\}$ be distinct such that $p(\check{c}) \leq p(\hat{c})$. Lastly, if $\max_{j\in c} p_j \leq 1/4$, then we construct $c'$ by greedily adding jobs from $c$ to $c'$ until $p(c') > 1/4$ and again define $\hat{c}, \check{c} \in \{c', c \setminus c'\}$ to be distinct such that $p(\check{c}) \leq p(\hat{c})$. $\blacksquare$

**Lemma 4.6** Let $c \in \mathcal{C}$ with $p(c) \in (1/2, 3/4)$ and $\max_{j\in c} p_j \leq 1/2$. Then $c$ can be partitioned into two parts $\check{c}$ and $\hat{c}$ with $p(\check{c}) \leq p(\hat{c}) \leq 1/2$ and $1/4 < p(\hat{c})$.

*Proof.* Let $c \in \mathcal{C}$ with $p(c) \in (1/2, 3/4)$ and $\max_{j\in c} p_j \leq 1/2$. If $\max_{j\in c} p_j \in (1/4, 1/2]$, then let $c'$ include a maximal job from $c$ and let $\hat{c}, \check{c} \in \{c', c \setminus c'\}$ be distinct such that $p(\check{c}) \leq p(\hat{c})$. If $\max_{j\in c} p_j \leq 1/4$, then we construct $c'$ by greedily adding jobs from $c$ to $c'$ until $p(c') > 1/4$ and again define $\hat{c}, \check{c} \in \{c', c \setminus c'\}$ to be distinct such that $p(\check{c}) \leq p(\hat{c})$. $1/4 < p(\hat{c})$ follows directly from the fact that $p(\check{c}) \leq p(\hat{c})$ and $1/2 < p(\check{c}) + p(\hat{c})$. $\blacksquare$

In the following, we will present two algorithms. The first can only handle instances with classes that do not possess an item with a processing time larger than $3/4$. This algorithm will be used as a subroutine for the second algorithm, which can handle all instances.

### 4.3.1 Algorithm for Instances without Huge Jobs

Here we give an algorithm for instances with $|\mathcal{C}_H| = 0$. We assume that the instance was scaled by a value $1/T$ and the classes are categorized as described earlier. The main idea is to repeatedly take combinations of classes with specific parameters which conveniently fill one, two or three machines without opening additional ones. *Fill* in this case means that the average load of *full* machines is in $[1, 3/2]$. We start with taking two classes with total size in $(1/2, 3/4)$ each, as those fill one machine. Then we continue with four classes with total size $\geq 3/4$ each and show how those can be arranged to fill three machines. The procedure continues with different combinations of classes until all jobs are scheduled. We show the correctness of the algorithm by arguing that closed machines have on average load of at least 1, and every scheduled job is finished at $3/2$. At some point in the algorithm, we reach a state where only jobs of classes with a total load of at most $1/2$ are left. Those can be scheduled greedily by placing full classes on residual machines until a machine has load at least 1.

Since we repeatedly have to refer to the jobs which have not been scheduled, we introduce the notation of $\bar{\mathcal{C}}_X \subseteq \mathcal{C}_X$ to denote the subset of classes that have not been scheduled at the described step for any class specifier $X$. Note that at the beginning of the algorithm, we have $\bar{\mathcal{C}}_X = \mathcal{C}_X$ for all the sets. Furthermore, the algorithm will close some of the machines during the construction of the schedule and will not add jobs to closed machines. We denote the set of closed machines as $M_c$. The algorithm is as follows:

**Algorithm:** `Algorithm_no_huge`

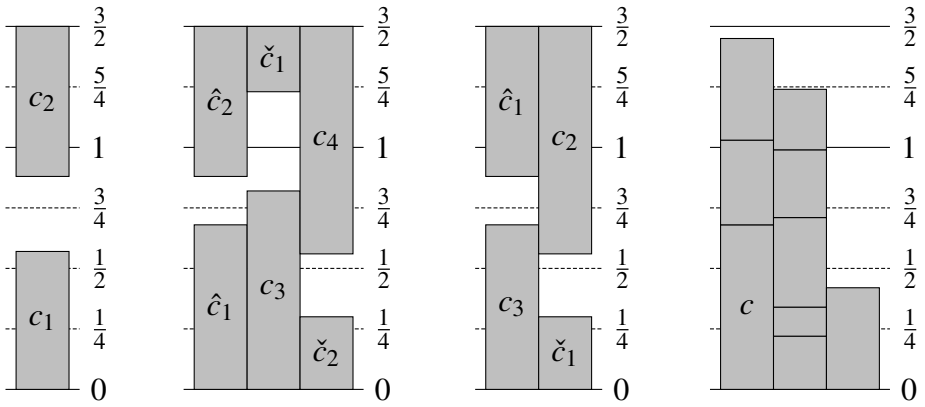
**Step 1** By applying Lemma 4.5, partition every class $c \in \mathcal{C}_{>3/4}$ into two parts $\check{c}, \hat{c} \subseteq c$ with $p(\check{c}) \leq p(\hat{c}) \leq 3/4$ and $p(\check{c}) \leq 1/2$.

**Step 2** While $|\bar{\mathcal{C}}_{(1/2,3/4)}| \geq 2$: Take $c_1, c_2 \in \bar{\mathcal{C}}_{(1/2,3/4)}$. Schedule $c_1$ and $c_2$ on one machine such that $c_1$ starts at 0 and $c_2$ ends at $3/2$.

**Claim** The load of each machine closed in this step is in $(1, 3/2)$. After this step, it holds that $|\bar{\mathcal{C}}_{(1/2,3/4)}| \leq 1$, the partial schedule is feasible, and the total load of closed machines $M_c$ is at least $|M_c|$.

**Step 3** While $|\bar{\mathcal{C}}_{\geq 3/4}| \geq 4$: Take $c_1, c_2, c_3, c_4 \in \bar{\mathcal{C}}_{\geq 3/4}$. On the first machine schedule $\hat{c}_1$ and $\hat{c}_2$, such that $\hat{c}_1$ starts at 0 and $\hat{c}_2$ ends at $3/2$. On the second machine schedule $\check{c}_1$ and $c_3$, such that $\check{c}_1$ ends at $3/2$ and starts after 1. On the third machine schedule $\check{c}_2$ and $c_4$, such that $\check{c}_2$ starts at 0 and ends before $1/2$ followed by $c_4$, see Figure 4.2b for an example. Close all three machines.

**Claim** After this step $|\bar{\mathcal{C}}_{(1/2,3/4)}| \leq 1$ and $|\bar{\mathcal{C}}_{\geq 3/4}| \leq 3$, the partial schedule is feasible, and the total load of closed machines $M_c$ is at least $|M_c|$. Furthermore, all scheduled jobs are finished by $3/2$.




(a) Step 2 (b) Step 3 (c) Step 4 (d) Step 5

Figure 4.2: Examples for Steps 2 to 5

**Step 4** If $|\bar{\mathcal{C}}_{\geq 3/4}| \geq 2$ and $|\bar{\mathcal{C}}_{(1/2,3/4)}| = 1$: Take $c_1, c_2 \in \bar{\mathcal{C}}_{\geq 3/4}$ and $c_3 \in \bar{\mathcal{C}}_{(1/2,3/4)}$. Schedule $c_3$ on the first machine, followed by $\hat{c}_1$ such that it ends at $3/2$. Schedule

$\check{c}_1$ on the second machine followed by the jobs from $c_2$ and close both machines, see Figure 4.2c for an example.

**Claim** After this step it holds that $|\bar{\mathcal{C}}_{(1/2,3/4)}| = 0$ and $|\bar{\mathcal{C}}_{\geq 3/4}| \leq 3$ or it holds that $|\bar{\mathcal{C}}_{(1/2,3/4)}| = 1$ and $|\bar{\mathcal{C}}_{\geq 3/4}| \leq 1$. This implies that $|\bar{\mathcal{C}}_{>1/2}| \leq 3$ after this step and that $\bar{\mathcal{C}}_{>1/2}$ contains at most one class with total processing time less than $3/4$. Furthermore, the partial schedule is feasible, the total load of closed machines $M_c$ is at least $|M_c|$, and no scheduled job finishes after $3/2$.

Depending on the size of $|\bar{\mathcal{C}}_{>1/2}|$ the algorithm chooses one of three procedures:

**Step 5** If $|\bar{\mathcal{C}}_{>1/2}| \leq 1$: Place this class $c$ on one machine. Fill this machine and the residual machines greedily with the residual classes in $\bar{\mathcal{C}}_{\leq 1/2}$.

**Claim** After this step, it either holds that $2 \leq |\bar{\mathcal{C}}_{>1/2}| \leq 3$ or all jobs have been scheduled feasibly with no job finishing after $3/2$.

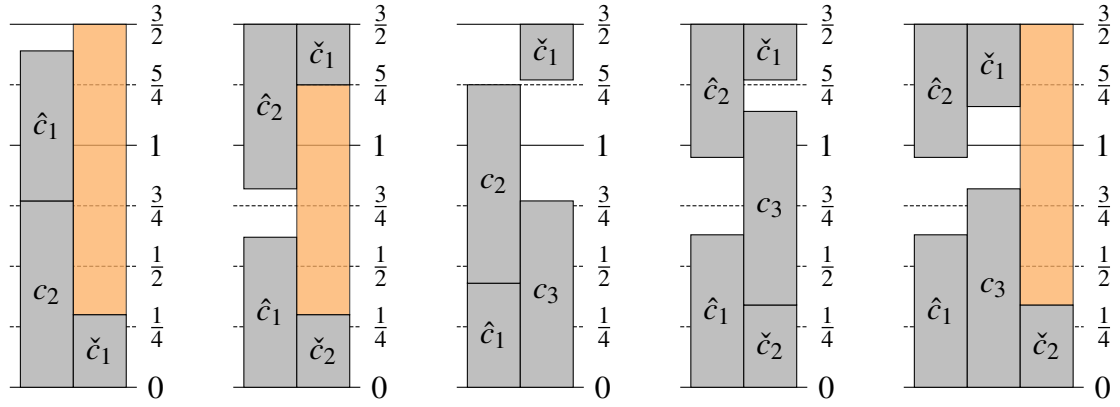
*Proof.* In the latter case, we can place all remaining jobs since there are at least as many open machines as there is open load because, before this step, we had $p(M_c) \geq |M_c|$. Each opened machine will be filled with load in $[1,3/2]$ since each residual job has a size of at most $1/2$. ■

**Step 6** If $|\bar{\mathcal{C}}_{>1/2}| = 2$: Let $\bar{\mathcal{C}}_{>1/2} = \{c_1, c_2\}$ with $p(c_1) \geq p(c_2)$. We know that $p(c_1) \geq 3/4$.

1. If $p(c_2) \leq 3/4$:
   a. If $p(c_1) + p(c_2) \leq 3/2$: Schedule both on one machine (with $c_1$ starting at 0 and $c_2$ ending at $3/2$), close it, and continue greedily with the residual jobs.
   b. If $p(c_1) + p(c_2) > 3/2$: Place $c_2$ on one machine followed by $\hat{c}_1$ such that $\hat{c}_1$ ends at $3/2$ and close the machine. Place $\check{c}_1$ on the next machine and continue greedily with the residual jobs in $\bar{\mathcal{C}}_{\leq 1/2}$.
2. If $p(c_2) \geq 3/4$:
   a. If $p(\hat{c}_1) + p(\hat{c}_2) \leq 1$: Schedule $c_2$ followed by $\hat{c}_1$ on one machine and close it. Start $\check{c}_1$ at 0 on the next machine and continue greedily with the residual jobs in $\bar{\mathcal{C}}_{\leq 1/2}$.
   b. If $p(\hat{c}_1) + p(\hat{c}_2) > 1$: Then place $\hat{c}_1$ and $\hat{c}_2$ on one machine such that $\hat{c}_1$ starts at 0 and $\hat{c}_2$ ends at $3/2$. Place $\check{c}_2$ at the bottom and $\check{c}_1$ at the top of the next machine. Continue greedily with the residual classes in $\bar{\mathcal{C}}_{\leq 1/2}$. Start placing them between $\check{c}_2$ and $\check{c}_1$ until the load of that machine is at least 1 and then continue with the empty machines.

**Claim** After this step, it either holds that $|\bar{\mathcal{C}}_{>1/2}| = 3$ or all jobs have been scheduled feasibly with no job finishing after $3/2$.

*Proof.* We will prove the latter case. If $p(c_2) \leq 3/4$, the load of the machines that contains either $c_1$ and $c_2$ or only $c_2$ and $\hat{c}_1$ has a load in $(1,3/2]$. Each residual class (or part of a class) has a total processing time of at most $1/2$. Furthermore, up to this step, it holds that $p(M_c) \geq |M_c|$. As a consequence, greedily scheduling the residual classes starting with $\check{c}_1$ is possible.




(a) Step 6.2a (b) Step 6.2b (c) Step 7.1 (d) Step 7.2a (e) Step 7.2b

Figure 4.3: Examples for Step 6 and Step 7. Orange blocks represent space for residual classes.

If, on the other hand, $p(c_2) > 3/4$ holds, the machine containing $c_2$ and $\hat{c}_1$ (or $\hat{c}_2$ and $\hat{c}_1$ respectively) has a total load of at least 1 in either case and placing $\check{c}_1$ (and $\check{c}_2$) as described does not provoke an overlapping of two jobs requiring the same resource (see Figure 4.3). Furthermore, the machine containing $c_2$ and $\hat{c}_1$ (or $\hat{c}_2$ and $\hat{c}_1$ respectively) has a total load of at most $3/2$ since $p(\check{c}_2) + p(\hat{c}_1) + p(\hat{c}_2) \leq 1/2 + 1$ if $p(\hat{c}_1) + p(\hat{c}_2) \leq 1$ and $p(\hat{c}_i) \leq 3/4$ for $i \in \{1,2\}$. The residual classes can again be scheduled greedily. This is easy to see in the case $p(\hat{c}_1) + p(\hat{c}_2) \leq 1$ and otherwise, we have $p(\check{c}_2) + p(\check{c}_1) \in [0,1)$, and hence the remaining gap has a size of at least $1/2$. Since all remaining classes have total load of at most $1/2$ it is possible to greedily add such classes until the total load of that machine is at least 1 or all remaining classes have been placed. ■

**Step 7** If $|\bar{\mathcal{C}}_{>1/2}| = 3$: Then $\bar{\mathcal{C}}_{>1/2} = \bar{\mathcal{C}}_{\geq 3/4}$. Let $\bar{\mathcal{C}}_{\geq 3/4} = \{c_1, c_2, c_3\}$.

1. If there exists an $i \in \{1,2,3\}$ such that $\hat{c}_i \leq 1/2$: Let w.l.o.g. $\hat{c}_1 \leq 1/2$. On the first machine, schedule $\hat{c}_1$ followed by all the jobs from $c_2$. On the next machine, schedule all the jobs from $c_3$ and the job $\check{c}_1$ such that it ends at $3/2$ and close both machines. Greedily schedule the jobs in $\bar{\mathcal{C}}_{\leq 1/2}$ on the non-closed machines.
2. If $\hat{c}_i > 1/2$ for all $i \in \{1,2,3\}$: Place $\hat{c}_1$ and $\hat{c}_2$ on one machine such that $\hat{c}_1$ starts at 0 and $\hat{c}_2$ ends at $3/2$.
   a. If $p(\check{c}_1) + p(\check{c}_2) + p(c_3) \leq 3/2$: On the next machine place $\check{c}_2$ followed by $c_3$ and $\check{c}_1$ and let $\check{c}_1$ end at $3/2$. Close both machines.
   b. If $p(\check{c}_1) + p(\check{c}_2) + p(c_3) > 3/2$: Then w.l.o.g. $p(\check{c}_1) > 1/4$ and we place $c_3$ and $\check{c}_1$ on the next machine, such that $\check{c}_1$ ends at $3/2$. Close both machines. On the next machine, place $\check{c}_2$ such that it starts at 0.

   Greedily schedule the jobs in $\bar{\mathcal{C}}_{\leq 1/2}$ on the non-closed machines.

**Claim** After this step, all scheduled jobs are finished by $3/2$, and the schedule is feasible.

*Proof.* Note that the two machines containing the classes $c_1$, $c_2$, and $c_3$ (or $c_1$, $\hat{c}_2$, and $c_3$ respectively) have a total load of at least 2. As a consequence, all machines $M_c$ closed to this point have a load of at least $|M_c|$. Therefore, their residual load fits on the residual

machines. When greedily scheduling the classes, each machine is overloaded by at most $1/2$ since each residual class has a processing time of at most $1/2$. ■

**Lemma 4.7** Given an instance $I = (m, \mathcal{J}, \mathcal{C})$ that does not contain a huge job, the algorithm `Algorithm_no_huge` finds a schedule with makespan at most $\frac{3}{2}T$, where $T = \max\{\frac{1}{m}p(\mathcal{J}), \max_{c\in\mathcal{C}} p(c), \tilde{p}_m + \tilde{p}_{m+1}\}$.

### 4.3.2 Algorithm for the General Case

Now we present the above-mentioned algorithm that can handle any instance of the problem and uses the previous algorithm in a subroutine. More specifically, this algorithm places all classes which contain a huge job on a separate machine and fills those machines with jobs from other classes. This is done by working through different combinations of classes until we reach a point where we can handle the remaining classes and machines as a separate problem instance, at which point the previous algorithm is used. As before, we assume that the instance is scaled by a value $1/T$, and the classes are categorized as described earlier.

We keep the following invariant of the remaining instance over the whole algorithm.

**Invariant** The total load of unscheduled jobs and jobs placed on open machines is upper bounded by the number of open machines (open machines are all machines not explicitly closed), i.e., $p(\bar{M}_H) + p(\bar{\mathcal{C}}) \leq |\bar{M}_u| + |\bar{M}_H|$, and in each step the cardinality of the set of unused machines $\bar{M}_u$ is bounded analogous to Lemma 4.3:

$$|\bar{M}_u| \geq \max\{|\bar{\mathcal{C}}_B|, \lceil(|\bar{\mathcal{C}}_B| + |\bar{\mathcal{C}}_{\geq 3/4} \setminus (\mathcal{C}_H \cup \mathcal{C}_B)|)/2\rceil\}.$$

**Algorithm:** `Algorithm_3/2`

**Step 1** Combine specific jobs of the same class into one job. The simplification is done as follows: Iterate all classes $c \in \mathcal{C}$

- If $c \in \mathcal{C}_H$ combine all jobs in $c$ to one huge job.
- Else if $p(c) > 3/4$ partition it into parts $\hat{c}$ and $\check{c}$ with $p(\check{c}) \leq p(\hat{c}) \leq 3/4$ and $p(\check{c}) \leq 1/2$. Introduce for each part a new job with processing time $p(\hat{c})$ and $p(\check{c})$, see Lemma 4.5.
- Else if $c \in \mathcal{C}_{(1/2,3/4)} \cap \mathcal{C}_B$: partition it into $\hat{c}$ and $\check{c}$, such that $\hat{c}$ contains the largest job and $\check{c}$ contains the rest.
- Else if $c \in \mathcal{C}_{(1/2,3/4)} \setminus \mathcal{C}_B$ partition it into parts $\hat{c}$ and $\check{c}$ with $p(\check{c}) \leq p(\hat{c}) \leq 1/2$, see Lemma 4.6.
- Else if $p(c) \leq 1/2$ introduce one job of size $p(c)$.

**Claim** This partition is feasible, and every solution for this simplified instance will still be a solution for the original instance.

**Step 2** For each $c \in \mathcal{C}_H$: Open one new machine and assign class $c$ to it. Let $M_H$ be the set of these opened machines. Close all the machines that have load exactly 1. Denote by $\bar{M}_H$ the set of currently open machines containing a class from $\mathcal{C}_H$.

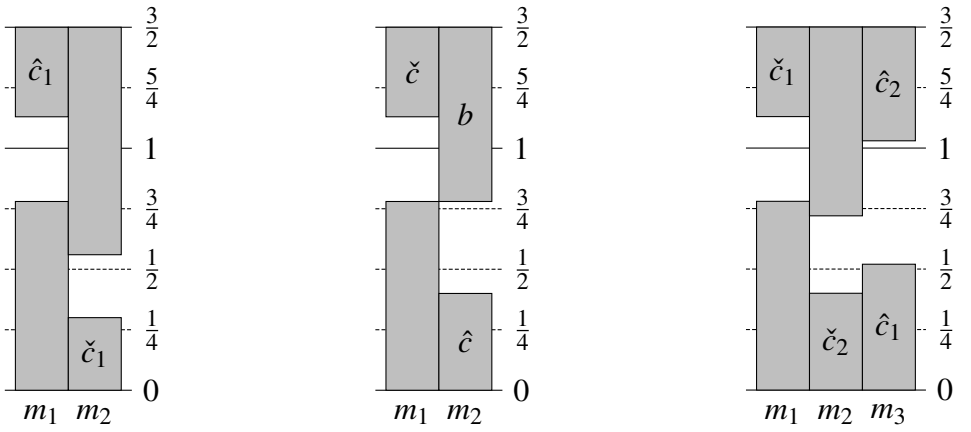
**Claim** After this step, there are $|\bar{M}_H|$ open machines with load in $(3/4, 1)$, $|\bar{\mathcal{C}}_H| = 0$, and the Invariant holds.

**Step 3** Assign classes $c_s$ with $p(c_s) \leq 1/2$ greedily to machines $\bar{M}_H$ and close each machine with load at least 1. Continue until either no machines in $\bar{M}_H$ with load less than 1 is left, or no class with load at most $1/2$ is left. If $|\bar{M}_H| = 0$, continue with `Algorithm_no_huge` on the residual instance.

**Claim** After this step, either all jobs are scheduled feasibly, or it holds that $|\bar{M}_H| \geq 1$ and $|\bar{\mathcal{C}}_{\leq 1/2}| = 0$. Furthermore, the partial schedule is feasible, all scheduled jobs are finished by $3/2$, and the Invariant holds.

*Proof.* Since we only closed machines with load at least one in this step and did not open any new machines, the invariant on the number of unused machines is trivially true. Hence, if we have used `Algorithm_no_huge` on the residual instance, by Lemma 4.7 it generates a schedule with makespan at most $3/2$ because $p(\bar{\mathcal{C}}) \leq |\bar{M}_u|$ at that point and no class was scheduled partially. ■

**Step 4** While $|\bar{M}_H| \geq 2$ and $|\bar{\mathcal{C}}_{(1/2,3/4)} \setminus \bar{\mathcal{C}}_B| \geq 1$: Take $m_1, m_2 \in \bar{M}_H$, $c \in \bar{\mathcal{C}}_{(1/2,3/4)} \setminus \mathcal{C}_B$. Shift the huge job on $m_2$ up such that it ends at $3/2$ and starts at or after $1/2$. Schedule $\hat{c}$ on $m_1$ such that it ends at $3/2$, schedule $\check{c}$ on $m_2$ starting at 0 and close both machines, see Figure 4.4a. If $|\bar{M}_H| = 0$, continue with `Algorithm_no_huge` on the residual instance.




(a) Step 4 (b) Step 6 (c) Step 8

Figure 4.4: Examples for Step 4, Step 6, and Step 8

**Claim** After this step, either all jobs are scheduled feasibly, or one of the following two conditions holds: $|\bar{M}_H| = 1$ and $|\bar{\mathcal{C}}_{\leq 1/2}| = 0$, or $|\bar{M}_H| \geq 2$ and $|\bar{\mathcal{C}} \setminus (\mathcal{C}_B \cup \bar{\mathcal{C}}_{\geq 3/4})| = 0$. Furthermore, the partial schedule is feasible, all scheduled jobs are finished by $3/2$ and all machines not in $\bar{M}_H$ are either closed or empty, and the Invariant holds.

*Proof.* Note that we have not opened any other machine in this step. Hence the lower bound on $|\bar{M}_u|$ is trivially true. The total load of $m_1, m_2$ and $c$ is at least $2 \cdot 3/4 + 1/2 = 2$.

Hence in each of these steps, we close two machines but also reduce the residual load by at least 2, proving the upper bound on the residual load. Hence, if we have used `Algorithm_no_huge` on the residual instance, by Lemma 4.7 it generates a schedule with makespan at most $3/2$ because $p(\bar{\mathcal{C}}) \leq |\bar{M}_u|$ at that point and no class was scheduled partially. ■

**Step 5** If $|M_H| = 1$:
- If there exists $c \in \bar{\mathcal{C}} \setminus \mathcal{C}_B$: Choose $c' \in \{\hat{c}, \check{c}\}$ with $c' \in (1/4, 1/2]$. Schedule $c'$ on the last open machine $m_0$. Use `Algorithm_no_huge` to schedule the residual instance, including the job $c'' \in c \setminus c'$. "Rotate" the load on $m_0$, such that $c'$ does not overlap with $c''$.
- If $\bar{\mathcal{C}} \setminus \mathcal{C}_B$ is empty: Assign all the residual classes to an individual machine.

**Claim** After this step, all jobs have been scheduled feasibly or $|\bar{M}_H| \geq 2$ and $|\bar{\mathcal{C}} \setminus (\mathcal{C}_B \cup \bar{\mathcal{C}}_{\geq 3/4})| = 0$. Additionally, the partial schedule is feasible, all scheduled jobs are finished by $3/2$, and the Invariant holds.

*Proof.* First consider the case that $\bar{\mathcal{C}} \setminus \mathcal{C}_B \neq \emptyset$. We know that such a required $c'$ exists. This is given by Lemma 4.6 and Lemma 4.5 for classes in $\bar{\mathcal{C}}_{(1/2,3/4)} \setminus \mathcal{C}_B$ and $\bar{\mathcal{C}}_{\geq 3/4} \setminus \mathcal{C}_B$, respectively. The residual instance will be scheduled with the algorithm for instances without huge jobs. This generates a feasible schedule since all machines that are non-empty before the start of this subroutine have load of at least 1. Furthermore, only class $c$ is partially scheduled, and the load on $m_0$ can be rotated, such that $\hat{c}$ and $\check{c}$ do not overlap. This rotation is always possible: The residual job $c''$ of the class is smaller than $3/4$ and will therefore be scheduled consecutively by `Algorithm_no_huge`. No matter when $c''$ gets scheduled, before or after, it will be a large enough gap that fits $c'$ since $c'$ got scheduled starting at 0 (or ending at $3/2$ after the rotation). Therefore, a correct rotation is possible.

In the case that $\bar{\mathcal{C}} \setminus \mathcal{C}_B = \emptyset$, we put the residual classes to individual machines. This is possible since only classes in $\mathcal{C}_B$ are left and the number of residual machines is at least $|\bar{\mathcal{C}}_B|$. ■

**Step 6** While $|\bar{M}_H| \geq 1$, $|\bar{\mathcal{C}}_{(1/2,3/4)} \cap \mathcal{C}_B| \geq 1$, and $|\bar{\mathcal{C}}_{\geq 3/4}| \geq 1$: Take $m_1 \in \bar{M}_H$, $b \in \bar{\mathcal{C}}_{(1/2,3/4)} \cap \mathcal{C}_B$ and $c \in \bar{\mathcal{C}}_{\geq 3/4}$. Open one new machine $m_2$. Schedule $\check{c}$ on $m_1$ such that it ends at $3/2$. Schedule $\hat{c}$ on $m_2$ such that it starts at 0 and ends before $3/4$. Schedule $b$ at $m_2$ such that it ends at $3/2$, see Figure 4.4b. Close both machines. If $|\bar{M}_H| = 0$, continue with `Algorithm_no_huge` on the residual instance.

**Claim** After this step, all jobs are scheduled feasibly or $|\bar{M}_H| \geq 1$ and $|\bar{\mathcal{C}} \setminus (\bar{\mathcal{C}}_{(1/2,3/4)} \cap \mathcal{C}_B)| = 0$ or $|\bar{\mathcal{C}} \setminus \bar{\mathcal{C}}_{\geq 3/4}| = 0$. Furthermore, all jobs are scheduled feasibly in this step, all scheduled jobs are finished by $3/2$, and the Invariant holds.

*Proof.* Note that we open one more machine in each iteration of the step. This machine has to exist since in each of these steps, we have $|\bar{\mathcal{C}}_B| + |\bar{\mathcal{C}}_{\geq 3/4} \setminus \mathcal{C}_B| \geq 2$. In this step, we have reduced $|\bar{\mathcal{C}}_B| + |\bar{\mathcal{C}}_{\geq 3/4} \setminus \mathcal{C}_B|$ by 2 and $|\bar{\mathcal{C}}_B|$ at least by 1. Hence there still have to exist $\max\{\, |\bar{\mathcal{C}}_B|, \lceil (|\bar{\mathcal{C}}_B| + |\bar{\mathcal{C}}_{\geq 3/4} \setminus \mathcal{C}_B|)/2 \rceil \,\}$ unused machines. In each iteration of this step, we

close two machines but also reduce the residual load by at least $3/4+1/2+3/4=2$, proving the upper bound on the residual load.

Hence, if we have used `Algorithm_no_huge` on the residual instance, by Lemma 4.7 it generates a schedule with makespan at most $3/2$ because $p(\bar{\mathcal{C}}) \leq |\bar{M}_u|$ at that point and no class was scheduled partially. ■

**Step 7** If $|\bar{\mathcal{C}}_{(1/2,3/4)} \cap \mathcal{C}_B| \neq 0$, open one machine for each of these classes.

**Claim** After this step, all jobs are feasibly scheduled, or it holds that $|\bar{M}_H| \geq 1$ and all residual classes have a total processing time of at least $3/4$, all scheduled jobs are finished by $3/2$, and the Invariant holds.

*Proof.* Note that if $|\bar{\mathcal{C}}_{(1/2,3/4)} \cap \mathcal{C}_B| \neq 0$ this set is the only set containing unscheduled classes. Since we still have $|\bar{M}_u| \geq |\bar{\mathcal{C}}_B|$ unused machines, we can feasibly open one machine for each of these classes and are done. ■

**Step 8** While $|\bar{M}_H| \geq 2$ and $|\bar{\mathcal{C}}_{\geq 3/4}| \geq 2$: Take $m_1, m_2 \in \bar{M}_H$, $c_1, c_2 \in \bar{\mathcal{C}}_{\geq 3/4}$ starting with the classes in $\bar{\mathcal{C}}_B$. Shift all jobs on $m_2$ to the top, such that the last job ends at $3/2$. Schedule $\check{c}_1$ on $m_1$ as one block that ends at $3/2$ and all the jobs from $\check{c}_2$ as one block on $m_2$ that starts at 0. Open one more machine $m_3$ where we start the jobs from $\hat{c}_1$ at 0 and let the last job from $\hat{c}_2$ end at $3/2$, see Figure 4.4c. Close all three machines $m_1, m_2, m_3$. If $|\bar{M}_H| = 0$, continue with `Algorithm_no_huge` on the residual instance.

**Claim** After this step, all jobs are scheduled, or it holds that either $|\bar{M}_H| = 1$ or $|\bar{\mathcal{C}}_{\geq 3/4}| \leq 1$. Furthermore, $|\bar{\mathcal{C}} \setminus \bar{\mathcal{C}}_{\geq 3/4}| = 0$, and in each iteration, the partial schedule is feasible, all scheduled jobs are finished by $3/2$, and the Invariant holds.

*Proof.* In each of these steps, no two jobs from the same class overlap. Note that we open one more machine in each iteration of the step. This machine has to exist, since $|\bar{\mathcal{C}}_B| + |\bar{\mathcal{C}}_{\geq 3/4} \setminus \mathcal{C}_B| \geq 2$ before this step. Since all remaining classes have load at least $3/4$ it holds that $|\bar{\mathcal{C}}_B| = |\bar{\mathcal{C}}_B \cap \bar{\mathcal{C}}_{\geq 3/4}$. Therefore, if $|\bar{\mathcal{C}}_B| \neq 0$ we used at least one such class and reduced $|\bar{\mathcal{C}}_B|$ by at least 1. We also reduced $|\bar{\mathcal{C}}_B| + |\bar{\mathcal{C}}_{\geq 3/4} \setminus \mathcal{C}_B|$ by 2 and hence there are still $\max\{\, |\bar{\mathcal{C}}_B|, \lceil (|\bar{\mathcal{C}}_B| + |\bar{\mathcal{C}}_{\geq 3/4} \setminus \mathcal{C}_B|)/2 \rceil \,\}$ unused machines. Lastly, in each of these steps, we close three machines but also reduce the residual load by at least 3 (4 classes with processing time at least $3/4$ each), proving the upper bound on the residual load.

Hence, if we have used `Algorithm_no_huge` on the residual instance, by Lemma 4.7 it generates a schedule with makespan at most $3/2$ because $p(\bar{\mathcal{C}}) \leq |\bar{M}_u|$ at that point and no class was scheduled partially. ■

**Step 9** If $|\bar{M}_H| \geq 2$ or $|\bar{\mathcal{C}} \setminus \mathcal{C}_B| = 0$, open one machine for each of the remaining classes.

**Claim** After this step, either all jobs are scheduled, or it holds that $|\bar{M}_H| = 1$, $|\bar{\mathcal{C}} \setminus \bar{\mathcal{C}}_{\geq 3/4}| = 0$, $\bar{\mathcal{C}} \setminus \mathcal{C}_B \neq \emptyset$, the partial schedule is feasible, all scheduled jobs are finished by $3/2$, and the Invariant holds.

*Proof.* Due to the previous steps $|\bar{M}_H| \geq 2$ implies $|\bar{\mathcal{C}}_{\geq 3/4}| \leq 1$, and if $|\bar{\mathcal{C}}_{\geq 3/4}| = 0$ we have already scheduled all the jobs. Otherwise, if $|\bar{\mathcal{C}}_{\geq 3/4}| = 1$, there has to be one unused machine because there are at least $\max\{\,|\bar{\mathcal{C}}_B|, \lceil(|\bar{\mathcal{C}}_B| + |\bar{\mathcal{C}}_{\geq 3/4} \setminus \mathcal{C}_B|)/2\rceil\,\}$ unused machines.

If, on the other hand, $|\bar{\mathcal{C}} \setminus \mathcal{C}_B| = 0$, we still have $|\bar{M}_u| \geq |\bar{\mathcal{C}}_B|$ unused machines, we can feasibly open one machine for each of these classes and are done. ■

**Step 10** If $|\bar{M}_H| = 1$, take $c \in \bar{\mathcal{C}} \setminus \mathcal{C}_B$. It holds that $p(c) \geq 3/4$ and there exists $c' \in \{\hat{c}, \check{c}\}$ with $p(c') \in (1/4, 1/2]$. Place $c'$ on $m_0 \in \bar{M}_H$. Continue with `Algorithm_no_huge` to schedule the residual jobs, including the job $c'' \in c \setminus \{c'\}$. Rotate the load on $m_0$ such that $c'$ does not overlap with $c''$.

**Claim** After this step, all jobs are scheduled feasibly, and all scheduled jobs are finished by $3/2$.

*Proof.* The algorithm for instances without huge jobs can feasibly finish the schedule with makespan at most $3/2$ by Lemma 4.7 since $p(\bar{\mathcal{C}}) \leq |\bar{M}_u|$ at that point, all non-empty machines have load at least 1 on average, and every class except $c$ is either fully scheduled, or not scheduled at all. Like in Step 5, the rotation makes sure that there is no conflict within $c$. ■

**Lemma 4.8** Given any instance $I = (m, \mathcal{C})$ of MSRS, `Algorithm_3/2` produces a feasible schedule with makespan at most $\frac{3}{2}\mathrm{OPT}(I)$.

*Proof.* This is a direct consequence when considering the state after each step of the algorithm. ■

The existence and correctness of algorithm `Algorithm_3/2` proofs Theorem 4.2.

## 4.4 A Summary of Our Work on Approximation Schemes

For a more comprehensive picture on the costructiv results fo MSRS, we give a summary of the work on approximation schemes that we did in the original paper. To keep this short and easily comprehensible, we will explain the general ideas used without too much focus on formal completeness. For the technical details, we refer to the original paper. The two results are given by the following theorem:

**Theorem 4.3** There is an EPTAS for MSRS if either the number $m$ of machines is constant or $\lfloor \varepsilon m \rfloor$ additional machines may be used, i.e., resource augmentation is allowed.

To achieve these results, we follow a framework that was introduced in [38] and also used in [39]. We use the standard technique (see [37]) of applying a binary search framework to acquire a makespan guess $T$. The goal is then to either find a schedule of length $(1+\mathcal{O}(\varepsilon))T$ or correctly report that no schedule of length $T$ exists. Assume from now on that we already know $T$.

We divide the jobs into categories of length *big*, *medium*, or *small*. The definition of medium and small is chosen in a way such that small jobs from classes in which the small jobs have overall size of a medium job and medium jobs themselves only contribute an $\varepsilon$

fraction of the total load. We then consider a simplified version of the problem: Let $I$ be the input instance and $I_1$ the instance where (i) we remove all medium jobs or (ii) remove all medium jobs from classes including at most $\varepsilon T$ medium load and the entire classes containing at least $\varepsilon T$ medium load.

**Lemma 4.9** Let (i) $m$ be a constant or (ii) $m$ be part of the input. If there is a schedule with makespan $T'$ for $I$, then there is also a schedule with makespan $T'$ for $I_1$; and if there is a schedule with makespan $T'$ for $I_1$, then there is also a schedule with makespan $T' + \varepsilon T$ for $I$ ((ii) using at most $\lfloor \varepsilon m \rfloor$ additional machines).

The first direction is obvious. For the other direction, in the case of (i), note that the overall size of the medium jobs is upper bounded by $\varepsilon T$, and hence we can place all of them at the end of the schedule on some arbitrary machine. For the case of (ii), the proof idea is to use two greedy procedures placing the removed medium jobs from classes, including at most $\varepsilon T$ medium load at the end of the schedule and the remaining removed medium jobs on the additional machines.

We define a certain structure for schedules, called layered schedules, where jobs can only start on distinct multiples of a given layer height. We round the processing time of big jobs to multiples of the layer height, discard small jobs from classes with little total load, and replace the remaining small jobs with dummy batches (that instance is called $I_3$). We prove the existence of a solution structured in that way, with only bounded loss in the objective compared to an optimal solution of the $I_1$ instance. The problem of finding such a solution can then be formulated as an integer program (IP) of a particular form. This IP can be solved efficiently using $N$-fold integer programming algorithms. Furthermore, we show how that solution for the simplified $I_3$ problem can be used to derive a solution for the original one with only little loss in the objective value. The main challenge lies in the design of the well-structured solution and the proof of its existence. This also causes the limitations of our result: A certain group of jobs may cause problems in the respective construction, and to deal with them, we either use a more fine-grained approach, yielding a polynomial running time if $m$ is constant, or place the respective jobs on (few) additional machines using resource augmentation.

## 4.5 Inapproximability Results

We consider the case in which each job may need more than a single resource. Let us assume that we have a set $\mathcal{R}$ of resources, and each job $j$ needs some subset $\mathcal{R}(j)$ in order to be processed. The classes then correspond to subsets of resources $R \subseteq \mathcal{R}$ with $\mathcal{J}(R) = \{j \in \mathcal{J} \mid \mathcal{R}(j) = R\}$. We can adapt an APX-hardness result from [28] by recreating their conflict graph with resources. This is done by creating a resource $r_e$ per edge $e = \{u, v\}$ and letting jobs $u$ and $v$ require that resource. This reduction needs two machines, job sizes in $1, 2, 3, 4$ but roughly as many distinct resources per job as there are jobs. Subsequently, we give a new unrelated reduction for an instance of the problem with a constant bound on the number of distinct resources per job.

**Theorem 4.4** There is no $5/4-\varepsilon$-approximation algorithm with $\varepsilon > 0$ for the MSRS with multiple resources per job if P $\neq$ NP. This holds true, even if no job needs more than 3 resources ($\forall j \in \mathcal{J} : |\mathcal{R}(j)| \leq 3$) and all jobs have processing time 1, 2 or 3 ($\forall j \in \mathcal{J} : p(j) \in \{1,2,3\}$). Furthermore, this also holds when the number of machines is unlimited.

*Proof.* We show this by giving a reduction from the NP-hard MONOTONE 3-SAT-(2,2) problem [17], which is a satisfiability problem with the following restrictions: The boolean formula is in 3CNF, each clause contains either only unnegated or negated variables, and each literal appears in exactly 2 clauses (and every variable in exactly 4 clauses). Note here that we only use the bounded occurrence of literals, not the monotony.

The main idea of this reduction is to create a dummy structure, variable jobs, and clause jobs in such a way that a specific time interval was only useable by satisfied clause jobs. The overall dummy structure and processing times of jobs make sure that this specific time interval has to be filled completely to get a schedule with a makespan of 4. Confer Figure 4.5 for an overview of the construction.

In the following, we write that two (or more) jobs $j$ and $j'$ "share a resource $r$", which means that $r \in \mathcal{R}(j)$ and $r \in \mathcal{R}(j')$ and for all other jobs $j^*$, $j^* \neq j$ and $j^* \neq j'$, $r \notin \mathcal{R}(j^*)$. Let $\phi$ be the given formula and $\mathfrak{C}$, $\mathfrak{X}$ the sets of clauses and variables in $\phi$, respectively. We start by creating a dummy structure that we can anchor jobs to by using shared resources. Create $|\mathfrak{C}|$ many pairs of dummy jobs $j_i^A$, $j_i^a$ with $p(j_i^A) = 3, p(j_i^a) = 1$, which share a unique resource $A_i$. Furthermore, $j_i^a$ and $j_{i+1}^A$ share a unique resource $A_{i\to i+1}$. Create $|\mathfrak{X}|$ many pairs of dummy jobs $j_i^b$, $j_i^B$ with $p(j_i^b) = p(j_i^B) = 2$, which share a unique resource $B_i$. Furthermore, $j_i^B$ and $j_{i+1}^b$ share a unique resource $B_{i\to i+1}$. Lastly, $j_{|\mathfrak{C}|}^a$ and $j_1^b$ share a unique resource $A_{\to B}$

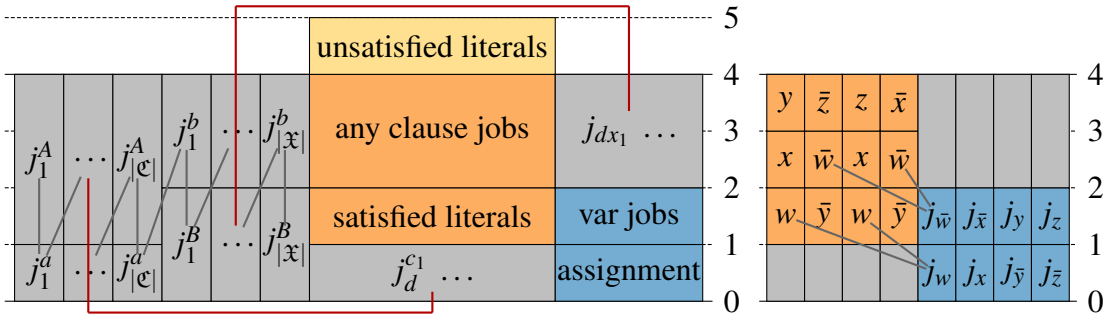
For every $x_i \in \mathfrak{X}$ create three variable jobs $j_{x_i}$, $j_{\bar{x}_i}$ and $j_{dx_i}$ which all share a resource $X_{x_i}$, moreover $j_{dx_i}$ and $j_i^B$ share a resource $B_{x_i}$. Set $p(j_{x_i}) = p(j_{\bar{x}_i}) = 1$ and $p(j_{dx_i}) = 2$.

For every $c_i \in \mathfrak{C}$ with $c_i = \{x_1^{c_i}, x_2^{c_i}, x_3^{c_i}\}$ create four clause jobs $j_{x_1}^{c_i}$, $j_{x_2}^{c_i}$, $j_{x_3}^{c_i}$ and $j_d^{c_i}$ which all share a resource $C_{c_i}$ and all with processing time 1. For every unnegated literal $x_i$ find the two clauses $c'$ and $c''$ in which it is contained. Let $j_{x_i}^{c'}$ and $j_{x_i}^{c''}$ be the two corresponding clause jobs created for $x_i$ in $c'$ and $c''$ respectively, then $j_{x_i}$ shares a unique resource with $j_{x_i}^{c'}$ and another unique resource with $j_{x_i}^{c''}$ (see connections from $j_w$ to its two occurrences in clauses in Figure 4.5b). Repeat this process for $\bar{x}_i$ and its negated occurrences. Furthermore $j_d^{c_i}$ and $j_i^A$ shares a resource $A_{c_i}$.

Finally, we set the number of machines to $2|\mathfrak{C}| + 2|\mathfrak{X}|$ (remark: we could also give an unlimited number of machines, as the resources limit the number of concurrently usable machines either way).

**Lemma 4.10** There is an optimal schedule with makespan 4 if and only if there is a satisfying assignment for the MONOTONE 3-SAT-(2,2) problem. Otherwise, the optimal schedule has a makespan of 5.

We first show that there is a trivial schedule with makespan 5 for each instance of the resulting scheduling problem. Place the dummy jobs as in Figure 4.5a, after that, for each $j_{dx_i}$ place the two corresponding jobs $j_{x_i}$ and $j_{\bar{x}_i}$ directly below it (in any order). Lastly, for every $j_d^{c_i}$ leave the timestep directly above it empty, and place $j_{x_1}^{c_i}$, $j_{x_2}^{c_i}$, $j_{x_3}^{c_i}$ above the empty timestep, finishing in timestep 5. It should be easy to see that this is always possible.




(a) Structure of emergent schedules (b) Clause-variable connections

Figure 4.5: Dummy structure and connection between clause and variable jobs. The time interval $[4,5]$ has to be used if there is no satisfying assignment for the MONOTONE 3-SAT-(2,2) problem. Grey lines represent a resource each; red lines represent pairwise resources of $j_i^A$ and $j_d^{c_i}$ ($j_i^B$ and $j_{dx_i}$, respectively).

Secondly, we show how to construct a schedule with a makespan of 4 if there is a satisfying assignment. We again start by placing the dummy jobs as in Figure 4.5a. For each $j_{dx_i}$ we place the two corresponding jobs $j_{x_i}$ and $j_{\bar{x}_i}$ below it. Now, look at the satisfying assignment for $\phi$, if $x_i$ is true (false) in the assignment, $j_{x_i}$ is placed below (above) $j_{\bar{x}_i}$. The job corresponding to the true assignment finishes at timestep 1, the other at 2. For every $j_d^{c_i}$ place $j_{x_1}^{c_i}$, $j_{x_2}^{c_i}$, $j_{x_3}^{c_i}$ above it. From the three jobs, choose one of which the corresponding literal is satisfied in the given assignment to be placed directly above $j_d^{c_i}$ (note that there has to be at least one such job since the assignment satisfies $\phi$). This is the only of the three (non-dummy) clause jobs that overlap the variable jobs placed earlier, but the variable job it shares a resource with was scheduled in the first timestep (see Figure 4.5b).

Lastly, we show how to construct a satisfying assignment from a schedule with makespan 4. One can verify that each dummy job in such a schedule has a fixed time window where it has to be scheduled due to the conflicts with other dummy jobs (we ignore that the whole schedule can be "flipped on its head" since it is equivalent). Furthermore, in such a schedule every time interval on every machine must be filled. Each pair of $j_i^A$, $j_i^a$ or $j_i^b$, $j_i^B$ occupies an interval of $[0,4]$, $j_{dx_i}$ is scheduled in $[0,1]$ and every $j_d^{c_i}$ is scheduled in $[2,4]$ (see Figure 4.5a). We count the remaining open slots: $[0,1]$: $|\mathfrak{X}|$, $[1,2]$: $|\mathfrak{X}|+|\mathfrak{C}|$ and $[2,4]$: $|\mathfrak{C}|$. The variable jobs $j_{x_i}$ and $j_{\bar{x}_i}$ can only be scheduled in $[0,2]$ (due to their dummy job) and can not be scheduled concurrently (due to their shared resource). Therefore, for every pair $j_{x_i}$ and $j_{\bar{x}_i}$ one job is scheduled in $[0,1]$ and one in $[1,2]$. After that, the remaining open slots are $[1,4]$: $|\mathfrak{C}|$. Following an analogously argumentation for each triple of clause jobs $j_{x_1}^{c_i}$, $j_{x_2}^{c_i}$, $j_{x_3}^{c_i}$ one job gets scheduled in $[1,2]$, $[2,3]$ and $[3,4]$. For every of those in $[1,2]$, the corresponding variable job has to be scheduled in $[0,1]$ (because they share a resource), which gives us, that the variable jobs in $[0,1]$ directly correspond to a satisfying assignment for the original MONOTONE 3-SAT-(2,2) Problem. ■

Following a similar construction, we can show the same inapproximability result for unit jobs and 5 or fewer resources per job. Furthermore, it is possible to give a $4/3-\varepsilon$ inapproximability result for the problem by giving a reduction from the NAE-3SAT problem. That reduction uses unit jobs but a non-constant number of resources per job. We briefly sketch both constructions here.

**Theorem 4.5** There is no $5/4-\varepsilon$-approximation algorithm with $\varepsilon > 0$ for the MSRS with multiple resources per job if P $\neq$ NP. This holds true, even if no job needs more than 5 resources ($\forall j \in \mathcal{J} : |\mathcal{R}(j)| \leq 5$) and all jobs have processing time 1 ($\forall j \in \mathcal{J} : p(j) = 1$). Furthermore, this also holds when the number of machines is unlimited.

*Proof.* The general idea here is the same as before. We only have to adapt the dummy structure in a way such that it only uses jobs with a processing time of 1. Again let $\phi$ be the given formula and $\mathfrak{C}$, $\mathfrak{X}$ the sets of clauses and variables in $\phi$, respectively. We replace the dummy structure from before as follows: Create $|\mathfrak{C}|+2|\mathfrak{X}|$ many dummy jobs $j_i^A$, $j_i^B$, $j_i^C$, $j_i^D$, each with processing time 1. The $i$th set of dummy jobs is connected to the $i+1$th set of dummy jobs in the following way:

- $j_i^A$, $j_{i+1}^B$, $j_{i+1}^C$, $j_{i+1}^D$ all share a resource $A_i$.
- $j_i^B$, $j_{i+1}^A$, $j_{i+1}^C$, $j_{i+1}^D$ all share a resource $B_i$.
- $j_i^C$, $j_{i+1}^A$, $j_{i+1}^B$, $j_{i+1}^D$ all share a resource $C_i$.
- $j_i^D$, $j_{i+1}^A$, $j_{i+1}^C$, $j_{i+1}^C$ all share a resource $D_i$.

Additionally $j_1^A$, $j_1^B$, $j_1^C$, $j_1^D$ all share a resource $A_0$. The web of resources makes sure that only dummy jobs with the same letter can be processed at the same time. Now we can use these dummy jobs as an anchor in the same way as before.

For the variable jobs replace each $j_{dx_i}$ with two jobs $j_{dx_i}^1$ and $j_{dx_i}^2$, both with processing time 1. The jobs $j_{x_i}$, $j_{\bar{x}_i}$, $j_{dx_i}^1$, and $j_{dx_i}^2$ all share a resource $X_{x_i}$. Moreover, $j_{dx_i}^1$, $j_{|\mathfrak{X}|+2i}^B$, $j_{|\mathfrak{X}|+2i}^C$, and $j_{|\mathfrak{X}|+2i}^D$ share a resource $X_{x_i}^1$. Analogously, $j_{dx_i}^2$, $j_{|\mathfrak{X}|+2i+1}^A$, $j_{|\mathfrak{X}|+2i+1}^C$, and $j_{|\mathfrak{X}|+2i+1}^D$ share a resource $X_{x_i}^2$. This fixes $j_{dx_i}^1$ and $j_{dx_i}^2$ to the timesteps of the dummy jobs $j_i^A$ and $j_i^B$, respectively. The clause jobs remain nearly identical, only $j_d^{c_i}$ now shares a resource $A_{c_i}$ with $j_i^A$, $j_i^B$, and $j_i^C$ (instead of with the respective single 3 long dummy job as before), this fixes $j_d^{c_i}$ to the timestep of the $j_i^D$. Finally, we set the number of machines to $2|\mathfrak{C}|+3|\mathfrak{X}|$ (remark: we again could give an unlimited number of machines, as the resources limit the number of concurrently usable machines either way). We omit the rest of the proof as it is nearly identical to the one above. ■

**Theorem 4.6** There is no $4/3-\varepsilon$-approximation algorithm with $\varepsilon > 0$ for the MSRS with multiple resources per job if P $\neq$ NP. This holds true, even if all jobs have processing time 1 ($\forall j \in \mathcal{J} : p(j) = 1$). Furthermore, this also holds when the number of machines is unlimited.

*Proof.* We show this via reduction from the NP-complete Not-all-equal 3-satisfiability (NAE-3SAT) problem [71]. An instance of the problem is a boolean formula in 3CNF, and the question is whether there exists an assignment such that every clause contains at least one satisfied and one unsatisfied literal. The idea of this reduction is similar to the ones above. Before, we used a dummy structure and variable jobs in such a way that a specific time interval was only useable by satisfied clause jobs. Now we remove a part of the dummy structure, such that one specific time interval is only useable by satisfied clause jobs, while another is only useable by unsatisfied clause jobs. See Figure 4.6 for a visual representation.

Again we are given a formula $\phi$ with the sets of clauses $\mathfrak{C}$ and variables $\mathfrak{X}$. All jobs created in this reduction have a processing time of 1. We create a dummy structure as in

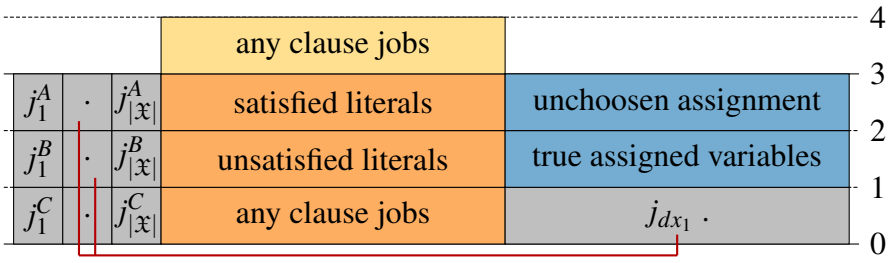



Figure 4.6: Emergent structure of schedules for the reduction from NAE-3SAT. Each job in *satisfied literals* shares a resource with a job in *true assigned variables* (analogously for *unsatisfied literals* and *unchoosen assignment*). The time interval $[3,4]$ has to be used if there is no correct assignment for the NAE-3SAT problem. The red line represents a resource $D_{x_i}$ per set $j_{dx_i}$, $j_i^A$ and $j_i^B$, fixing $j_{dx_i}$ to the same time interval as $j_i^C$.

the last reduction, but with a total length of 3 instead of 4: Create $|\mathfrak{X}|$ many dummy jobs $j_i^A$, $j_i^B$, $j_i^C$. The $i$th set of dummy jobs is connected to the $i+1$th set of dummy jobs in the following way:

- $j_i^A$, $j_{i+1}^B$, $j_{i+1}^C$ all share a resource $A_i$.
- $j_i^B$, $j_{i+1}^A$, $j_{i+1}^C$ all share a resource $B_i$.
- $j_i^C$, $j_{i+1}^A$, $j_{i+1}^B$ all share a resource $C_i$.

Additionally $j_1^A$, $j_1^B$, $j_1^C$ all share a resource $A_0$.

For every $x_i \in \mathfrak{X}$ create three variable jobs $j_{x_i}$, $j_{\bar{x_i}}$ and $j_{dx_i}$ which all share a resource $X_{x_i}$, moreover $j_{dx_i}$, $j_i^A$ and $j_i^B$ share a resource $D_{x_i}$.

For every $c_i \in \mathfrak{C}$ with $c_i = \{x_1^{c_i}, x_2^{c_i}, x_3^{c_i}\}$ create three clause jobs $j_{x_1}^{c_i}$, $j_{x_2}^{c_i}$, and $j_{x_3}^{c_i}$ which all share a resource $C_{c_i}$. The clause job $j_{x_1}^{c_i}$ (representing the literal ${x_1}^{c_i}$) and the corresponding negated or unnegated variable job $j_{x_k}$ or $j_{\bar{x_k}}$ share a resource $V_{x_k}^{c_i}$ (analogously for $j_{x_2}^{c_i}$ and $j_{x_3}^{c_i}$). Finally, we set the number of machines to $2|\mathfrak{C}| + |\mathfrak{X}|$ (remark: we again could give an unlimited number of machines, as the resources limit the number of concurrently usable machines either way).

We give an intuition why there is a schedule with makespan 3 if and only if there is an assignment such that in every clause, at least one literal is satisfied and at least one is unsatisfied. For each variable the three variable jobs $j_{x_i}$, $j_{\bar{x_i}}$ and $j_{dx_i}$ are assigned to one machine. Assume w.l.o.g. that all $j_{dx_i}$ get scheduled at $[0,1]$. For every pair $j_{x_i}$, $j_{\bar{x_i}}$ we can schedule them at $[1,2]$ and $[2,3]$ or at $[2,3]$ and $[1,2]$, respectively. The former represents $x_i$ *is true*, and the latter $x_i$ *is false*. For each clause we now have to distribute $j_{x_1}^{c_i}$, $j_{x_2}^{c_i}$, and $j_{x_3}^{c_i}$ onto $[0,1]$, $[1,2]$ and $[2,3]$ such that:

- A clause job that represents a satisfied literal has to be scheduled to $[2,3]$
- A clause job that represents an unsatisfied literal has to be scheduled to $[1,2]$
- Any clause job can be scheduled to $[0,1]$

If the jobs can not be arranged in this way, that means that a clause contains only satisfied or only unsatisfied literals. In that case, we would have to schedule one of the jobs to $[3,4]$, thus getting a makespan of 4. ■

**R** For both of the two sketched reductions, we can permutate the time intervals in any order. Note here that the dummy structure makes sure that a specific set of jobs has to be processed in the same time interval. It is of no relevance in which order these intervals are processed. Our explanations and figures simply assume some fixed order for comprehensibility.

## 4.6 Future Work

The results discussed in this chapter greatly improved the state of the art regarding the approximability of MSRS. There are several interesting directions emerging for further investigation. Firstly, there is the question of whether a PTAS for MSRS without resource augmentation can be achieved. It seems plausible that the approximation schemes results of the present work could be further refined to reach this goal. For the case with only a constant number of machines, on the other hand, an FPTAS is not ruled out at this point.

Moreover, it would be interesting to explore natural extensions of MSRS and, in particular, to investigate for which variants approximation schemes may or may not be feasible. From the negative perspective, we have already provided initial results in this chapter. We would like to point out one further question in this direction: Note that MSRS can be seen as a special case of scheduling with conflicts where the conflict graph is a cograph. This problem is known to be NP-hard already for unit size jobs [11], and it would be interesting to explore inapproximability for arbitrary sizes. Regarding the design of approximation schemes, on the other hand, variants, where the corresponding conflict graph is a particularly simple cograph, may be interesting.

Finally, from a broader perspective, it seems interesting to explore the possibilities of N-fold IPs and related concepts [26] for scheduling with additional resources.

# Bibliography


[1] Massinissa Ait Aba, Alix Munier Kordon, and Guillaume Pallez. Scheduling on Two Unbounded Resources with Communication Costs. In *Proceedings of the 25th Annual International Conference on Parallel and Distributed Computing (Euro-Par)*, volume 11725 of *Lecture Notes in Computer Science*, pages 117–128, 2019.

[2] Muminu O Adamu and Aderemi O Adewumi. A Survey of Single Machine Scheduling to Minimize Weighted Number of Tardy Jobs. *Journal of Industrial & Management Optimization*, 10(1):219, 2014.

[3] Chidambaram Annamalai. Lazy Local Search Meets Machine Scheduling. *SIAM Journal on Computing*, 48(5):1503–1543, 2019.

[4] Brenda S. Baker and Edward G. Coffman Jr. Mutual Exclusion Scheduling. *Theoretical Computer Science*, 162(2):225–243, 1996.

[5] Nikhil Bansal and Maxim Sviridenko. The Santa Claus Problem. In *Proceedings of the 38th Annual ACM Symposium on Theory of Computing (STOC)*, pages 31–40, 2006.

[6] Aditya Bhaskara, Ravishankar Krishnaswamy, Kunal Talwar, and Udi Wieder. Minimum Makespan Scheduling with Low Rank Processing Times. In *Proceedings of the 24th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 937–947, 2013.

[7] Jacek Blazewicz, Nadia Brauner, and Gerd Finke. Scheduling with Discrete Resource Constraints. In *Handbook of Scheduling - Algorithms, Models, and Performance Analysis*. 2004.

[8] Jacek Blazewicz, Klaus H. Ecker, Erwin Pesch, Günter Schmidt, Malgorzata Sterna, and Jan Weglarz. Scheduling under Resource Constraints. In *Handbook on Scheduling: From Theory to Practice*, pages 475–525. 2019.

[9] Jacek Blazewicz, Jan Karel Lenstra, and A. H. G. Rinnooy Kan. Scheduling Subject to Resource Constraints: Classification and Complexity. *Discrete Applied Mathematics*, 5(1):11–24, 1983.

[10] Manuel Blum, Robert W. Floyd, Vaughan R. Pratt, Ronald L. Rivest, and Robert Endre Tarjan. Time Bounds for Selection. *Journal of Computer and System Sciences*, 7(4):448–461, 1973.

[11] Hans L. Bodlaender and Klaus Jansen. On the Complexity of Scheduling Incompatible Jobs with Unit-Times. In *Proceedings of the 18th Annual International Symposium on Mathematical Foundations of Computer Science (MFCS)*, volume 711 of *Lecture Notes in Computer Science*, pages 291–300, 1993.

[12] Hans L. Bodlaender, Klaus Jansen, and Gerhard J. Woeginger. Scheduling with Incompatible Jobs. *Discrete Applied Mathematics*, 55(3):219–232, 1994.

[13] Deeparnab Chakrabarty, Sanjeev Khanna, and Shi Li. On (1, $\varepsilon$)-Restricted Assignment Makespan Minimization. In *Proceedings of the 26th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1087–1101, 2015.

[14] Lin Chen, Dániel Marx, Deshi Ye, and Guochuan Zhang. Parameterized and Approximation Results for Scheduling with a Low Rank Processing Time Matrix. In *Proceedings of the 34th Annual Symposium on Theoretical Aspects of Computer Science (STACS)*, volume 66 of *LIPIcs*, pages 22:1–22:14, 2017.

[15] Lin Chen, Deshi Ye, and Guochuan Zhang. An Improved Lower Bound for Rank Four Scheduling. *Operations Research Letters*, 42(5):348–350, 2014.

[16] Stephen A. Cook. The Complexity of Theorem-Proving Procedures. In *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing (STOC)*, pages 151–158, 1971.

[17] Andreas Darmann and Janosch Döcker. On simplified NP-complete variants of Monotone 3-Sat. *Discrete Applied Mathematics*, 292:45–58, 2021.

[18] Syamantak Das and Andreas Wiese. On Minimizing the Makespan when some Jobs cannot be Assigned on the same Machine. In *Proceedings of the 25th Annual European Symposium on Algorithms (ESA)*, volume 87 of *LIPIcs*, pages 31:1–31:14, 2017.

[19] Sami Davies, Janardhan Kulkarni, Thomas Rothvoss, Jakub Tarnawski, and Yihao Zhang. Scheduling with Communication Delays via LP Hierarchies and Clustering. In *Proceedings of the 61st Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 822–833, 2020.

[20] Sami Davies, Janardhan Kulkarni, Thomas Rothvoss, Jakub Tarnawski, and Yihao Zhang. Scheduling with Communication Delays via LP Hierarchies and Clustering II: Weighted Completion Times on Related Machines. In *Proceedings of the 32nd Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2958–2977, 2021.

[21] Max A. Deppert, Klaus Jansen, Marten Maack, Simon Pukrop, and Malin Rau. Scheduling with Many Shared Resources. In *Proceedings of the 37th Annual IEEE International Parallel & Distributed Processing Symposium (IPDPS), (accepted for publication, arXiv:2210.01523)*, 2023.

[22] Boris Detienne. A Mixed Integer Linear Programming Approach to Minimize the Number of Late Jobs with and without Machine Availability Constraints. *European Journal of Operational Research*, 235(3):540–552, 2014.

[23] György Dósa, Hans Kellerer, and Zsolt Tuza. Restricted Assignment Scheduling with Resource Constraints. *Theoretical Computer Science*, 760:72–87, 2019.

[24] Tomás Ebenlendr, Marek Krcál, and Jirí Sgall. Graph Balancing: A Special Case of Scheduling Unrelated Parallel Machines. *Algorithmica*, 68(1):62–80, 2014.

[25] Emrah B. Edis, Ceyda Oguz, and Irem Ozkarahan. Parallel Machine Scheduling with Additional Resources: Notation, Classification, Models and Solution Methods. *European Journal of Operational Research*, 230(3):449–463, 2013.

[26] Friedrich Eisenbrand, Christoph Hunkenschröder, Kim-Manuel Klein, Martin Koutecký, Asaf Levin, and Shmuel Onn. An Algorithmic Theory of Integer Programming. *CoRR*, arXiv:1904.01361, 2019.

[27] Leah Epstein and Asaf Levin. Scheduling with Processing Set Restrictions: PTAS Results for Several Variants. *International Journal of Production Economics*, 133(2):586–595, 2011.

[28] Guy Even, Magnús M. Halldórsson, Lotem Kaplan, and Dana Ron. Scheduling with Conflicts: Online and Offline Algorithms. *Journal of Scheduling*, 12(2):199–224, 2009.

[29] Uriel Feige. On Allocations that Maximize Fairness. In *Proceedings of the 19th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 287–293, 2008.

[30] M. R. Garey and David S. Johnson. "Strong" NP-Completeness Results: Motivation, Examples, and Implications. *Journal of the ACM*, 25(3):499–508, 1978.

[31] Shashwat Garg. Quasi-PTAS for Scheduling with Precedences using LP Hierarchies. In *Proceedings of the 45th Annual International Colloquium on Automata, Languages, and Programming (ICALP)*, volume 107 of *LIPIcs*, pages 59:1–59:13, 2018.

[32] Kilian Grage, Klaus Jansen, and Kim-Manuel Klein. An EPTAS for Machine Scheduling with Bag-Constraints. In *Proceedings of the 31st Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 135–144, 2019.

[33] Ronald L. Graham. Bounds for Certain Multiprocessing Anomalies. *Bell System Technical Journal*, 45(9):1563–1581, 1966.

[34] Ronald Lewis Graham, Eugene Leighton Lawler, Jan Karel Lenstra, and AHG Rinnooy Kan. Optimization and Approximation in Deterministic Sequencing and Scheduling: a Survey. In *Annals of discrete mathematics*, volume 5, pages 287–326. 1979.

[35] Alexander Grigoriev and Marc Uetz. Scheduling Jobs with Time-Resource Tradeoff via Nonlinear Programming. *Discrete Optimization*, 6(4):414–419, 2009.

[36] Emmanuel Hebrard, Marie-José Huguet, Nicolas Jozefowiez, Adrien Maillard, Cédric Pralet, and Gérard Verfaillie. Approximation of the Parallel Machine Scheduling Problem with Additional Unit Resources. *Discrete Applied Mathematics*, 215:126–135, 2016.

[37] Dorit S. Hochbaum and David B. Shmoys. Using Dual Approximation Algorithms for Scheduling Problems Theoretical and Practical Results. *Journal of the ACM*, 34(1):144–162, 1987.

[38] Klaus Jansen, Kim-Manuel Klein, Marten Maack, and Malin Rau. Empowering the Configuration-IP: new PTAS Results for Scheduling with Setup Times. *Mathematical Programming*, 2021.

[39] Klaus Jansen, Alexandra Lassota, and Marten Maack. Approximation Algorithms for Scheduling with Class Constraints. In *Proceedings of the 32nd Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 349–357, 2020.

[40] Klaus Jansen, Marten Maack, and Malin Rau. Approximation Schemes for Machine Scheduling with Resource (In-)dependent Processing Times. *ACM Transactions on Algorithms*, 15(3):31:1–31:28, 2019.

[41] Klaus Jansen, Marten Maack, and Roberto Solis-Oba. Structural Parameters for Scheduling with Assignment Restrictions. *Theoretical Computer Science*, 844:154–170, 2020.

[42] Klaus Jansen and Malin Rau. Closing the Gap for Single Resource Constraint Scheduling. In *Proceedings of the 29th Annual European Symposium on Algorithms (ESA)*, volume 204 of *LIPIcs*, pages 53:1–53:15, 2021.

[43] Klaus Jansen and Lars Rohwedder. A Quasi-Polynomial Approximation for the Restricted Assignment Problem. *SIAM Journal on Computing*, 49(6):1083–1108, 2020.

[44] Teun Janssen. *Optimization in the Photolithography Bay: Scheduling and the Traveling Salesman Problem*. PhD thesis, Delft University of Technology, Netherlands, 2019.

[45] Teun Janssen, Céline M. F. Swennenhuis, Abdoul Bitar, Thomas Bosman, Dion Gijswijt, Leo van Iersel, Stéphane Dauzère-Pérès, and Claude Yugma. Parallel Machine Scheduling with a Single Resource per Job. *CoRR*, arXiv:1809.05009, 2018.

[46] Richard M. Karp. Reducibility Among Combinatorial Problems. In *Proceedings of the IBM symposium on the Complexity of Computer Computations*, The IBM Research Symposia Series, pages 85–103, 1972.

[47] Hans Kellerer and Vitaly A. Strusevich. Scheduling Problems for Parallel Dedicated Machines under Multiple Resource Constraints. *Discrete Applied Mathematics*, 133(1-3):45–68, 2003.

[48] Kamyar Khodamoradi, Ramesh Krishnamurti, Arash Rafiey, and Georgios Stamoulis. PTAS for Ordered Instances of Resource Allocation Problems with Restrictions on Inclusions. *CoRR*, arXiv:1610.00082, 2016.

[49] Peter Kling, Alexander Mäcker, Sören Riechers, and Alexander Skopalik. Sharing is Caring: Multiprocessor Scheduling with a Sharable Resource. In *Proceedings of the 29th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 123–132, 2017.

[50] Donald E. Knuth. Postscript about NP-hard Problems. *SIGACT News*, 6(2):15–16, 1974.

[51] Janardhan Kulkarni, Shi Li, Jakub Tarnawski, and Minwei Ye. Hierarchy-Based Algorithms for Minimizing Makespan under Precedence and Communication Constraints. In *Proceedings of the 31st Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2770–2789, 2020.

[52] Eugene L Lawler and J Michael Moore. A Functional Equation and its Application to Resource Allocation and Sequencing Problems. *Management science*, 16(1):77–84, 1969.

[53] Kangbok Lee, Joseph Y.-T. Leung, and Michael L. Pinedo. Makespan Minimization in Online Scheduling with Machine Eligibility. *Annals of Operations Research*, 204(1):189–222, 2013.

[54] Julien Legriel, Colas Le Guernic, Scott Cotton, and Oded Maler. Approximating the Pareto Front of Multi-criteria Optimization Problems. In *Proceedings of the 16th Annual International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 6015 of *Lecture Notes in Computer Science*, pages 69–83, 2010.

[55] Jan Karel Lenstra and A. H. G. Rinnooy Kan. Complexity of Scheduling under Precedence Constraints. *Operations Research*, 26(1):22–35, 1978.

[56] Jan Karel Lenstra, AHG Rinnooy Kan, and Peter Brucker. Complexity of Machine Scheduling Problems. In *Annals of discrete mathematics*, volume 1, pages 343–362. 1977.

[57] Jan Karel Lenstra, David B. Shmoys, and Éva Tardos. Approximation Algorithms for Scheduling Unrelated Parallel Machines. *Mathematical Programming*, 46:259–271, 1990.

[58] Joseph Y-T Leung and Chung-Lun Li. Scheduling with Processing Set Restrictions: A Survey. *International Journal of Production Economics*, 116(2):251–262, 2008.

[59] Joseph Y-T Leung and Chung-Lun Li. Scheduling with Processing Set Restrictions: A Literature Update. *International Journal of Production Economics*, 175:1–11, 2016.

[60] Elaine Levey and Thomas Rothvoss. A $(1+\varepsilon)$-approximation for Makespan Scheduling with Precedence Constraints using LP Hierarchies. In *Proceedings of the 48th Annual ACM SIGACT Symposium on Theory of Computing (STOC)*, pages 168–177, 2016.

[61] Leonid A. Levin. Universal Sequential Search Problems. *Problems of Information Transmission*, 9(3):265–266, 1973.

[62] Chung-Lun Li and Xiuli Wang. Scheduling Parallel Machines with Inclusive Processing Set Restrictions and Job Release Times. *European Journal of Operational Research*, 200(3):702–710, 2010.

[63] Marten Maack, Friedhelm Meyer auf der Heide, and Simon Pukrop. Extended Server Cloud Scheduling. *CoRR*, arXiv:2108.02109, 2021.

[64] Marten Maack, Friedhelm Meyer auf der Heide, and Simon Pukrop. Server Cloud Scheduling. In *Proceedings of the 19th Annual International Workshop on Approximation and Online Algorithms (WAOA)*, volume 12982 of *Lecture Notes in Computer Science*, pages 144–164, 2021.

[65] Marten Maack and Klaus Jansen. Inapproximability Results for Scheduling with Interval and Resource Restrictions. *CoRR*, arXiv:1907.03526, 2019.

[66] Marten Maack and Klaus Jansen. Inapproximability Results for Scheduling with Interval and Resource Restrictions. In *Proceedings of the 37th Annual Symposium on Theoretical Aspects of Computer Science (STACS)*, volume 154 of *LIPIcs*, pages 5:1–5:18, 2020.

[67] Marten Maack, Simon Pukrop, and Anna Rodriguez Rasmussen. (In-)Approximability Results for Interval, Resource Restricted, and Low Rank Scheduling. In *Proceedings of the 30th Annual European Symposium on Algorithms (ESA)*, volume 244 of *LIPIcs*, pages 77:1–77:13, 2022.

[68] Alexander Mäcker, Manuel Malatyali, Friedhelm Meyer auf der Heide, and Sören Riechers. Cost-efficient Scheduling on Machines from the Cloud. *J. Comb. Optim.*, 36(4):1168–1194, 2018.

[69] Rym M'Hallah and R. L. Bulfin. Minimizing the Weighted Number of Tardy Jobs on a Single Machine with Release Dates. *European Journal of Operational Research*, 176(2):727–744, 2007.

[70] Rolf H. Möhring, Markus W. Schäffter, and Andreas S. Schulz. Scheduling Jobs with Communication Delays: Using Infeasible Solutions for Approximation (Extended Abstract). In *Proceedings of the 4th Annual European Symposium on Algorithms (ESA)*, volume 1136 of *Lecture Notes in Computer Science*, pages 76–90, 1996.

[71] Cristopher Moore and Stephan Mertens. *The Nature of Computation*. 2011.

[72] Gabriella Muratore, Ulrich M. Schwarz, and Gerhard J. Woeginger. Parallel Machine Scheduling with Nested Job Assignment Restrictions. *Operations Research Letters*, 38(1):47–50, 2010.

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

[74] Daniel R. Page and Roberto Solis-Oba. Makespan Minimization on Unrelated Parallel Machines with a few Bags. *Theoretical Computer Science*, 821:34–44, 2020.

[75] Christos H. Papadimitriou and Mihalis Yannakakis. On the Approximability of Trade-offs and Optimal Access of Web Sources. In *Proceedings of the 41st Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 86–92, 2000.

[76] Barna Saha. Renting a Cloud. In *Proceedings of the 33rd Annual IARCS Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, volume 24 of *LIPIcs*, pages 437–448, 2013.

[77] Sartaj Sahni. Algorithms for Scheduling Independent Tasks. *Journal of the ACM*, 23(1):116–127, 1976.

[78] Petra Schuurman and Gerhard J Woeginger. Polynomial Time Approximation Algorithms for Machine Scheduling: Ten Open Problems. *Journal of Scheduling*, 2(5):203–213, 1999.

[79] Ulrich M. Schwarz. *Approximation Algorithms for Scheduling and Two-dimensional Packing Problems*. PhD thesis, University of Kiel, 2010.

[80] Marc Sevaux and Stéphane Dauzère-Pérès. Genetic Algorithms to Minimize the Weighted Number of Late Jobs on a Single Machine. *European Journal of Operational Research*, 151(2):296–306, 2003.

[81] Vitaly A. Strusevich. Approximation Algorithms for Makespan Minimization on Identical Parallel Machines under Resource Constraints. *Journal of the Operational Research Society*, 0(0):1–12, 2020.

[82] Ola Svensson. Hardness of Precedence Constrained Scheduling on Identical Machines. *SIAM Journal on Computing*, 40(5):1258–1274, 2011.

[83] Ola Svensson. Santa Claus Schedules Jobs on Unrelated Machines. *SIAM Journal on Computing*, 41(5):1318–1341, 2012.

[84] Vijay V. Vazirani. *Approximation Algorithms*. 2001.

[85] Chao Wang and René Sitters. On some Special Cases of the Restricted Assignment Problem. *Information Processing Letters*, 116(11):723–728, 2016.

[86] David P. Williamson and David B. Shmoys. *The Design of Approximation Algorithms*. 2011.