UNIVERSITÄT PADERBORN
Die Universität der Informationsgesellschaft

Scheduling with Scarce Resources

Dissertation

zur Erlangung des akademischen Grades
Doktor der Naturwissenschaften (Dr. rer. nat.)
an der Fakultät für Elektrotechnik, Informatik und Mathematik
der Universität Paderborn

vorgelegt von

Sören Riechers

Paderborn, August 2017

Betreuer:

Prof. Dr. Friedhelm Meyer auf der Heide	(Universität Paderborn)

Gutachter:

Prof. Dr. Friedhelm Meyer auf der Heide	(Universität Paderborn)
Prof. Dr. Petra Berenbrink	(Universität Hamburg)

Weitere Mitglieder der Promotionskommission:

Prof. Dr. Eyke Hüllermeier	(Universität Paderborn)
Jun.-Prof. Dr. Alexander Skopalik	(Universität Paderborn)
Dr. Ulf-Peter Schröder	(Universität Paderborn)

Datum der mündlichen Prüfung:

14. November 2017

to my father, who showed me how to love mathematics,
to both of my parents, who have been there for me since the very first day,
and to farina, who is the best thing that has ever happened to me.

Zusammenfassung

Heutzutage ist in großen Daten- und Rechenzentren oft nicht mehr die Rechenkapazität der Flaschenhals des Systems, sondern der Speicher oder die verfügbare Datenrate. Scheduling-Algorithmen treffen in der Regel Entscheidungen, wie Jobs an einzelnen Knoten abgearbeitet werden, aber berücksichtigen meistens keine zusätzlichen Ressourceneinschränkungen in Bezug auf das gesamte Rechenzentrum. Diese Arbeit zielt darauf ab, solche globalen Ressourcen zu berücksichtigen.

Es werden vier Modelle eingeführt, die solche Ressourcen einbeziehen: Die ersten drei Modelle ähneln sich insofern, dass jeweils eine Ressource mit begrenzter Kapazität von mehreren Prozessoren geteilt wird, und das Ziel größtenteils darin besteht, die Gesamtabarbeitungszeit zu minimieren. Im ersten Modell wird der Fokus auf die Zuordnung der Ressource zu den Prozessoren gesetzt, während die Jobs bereits in einer festgelegten Reihenfolge auf die Prozessoren aufgeteilt sind. Im zweiten Modell werden Kommunikationsanforderungen zwischen Jobs betrachtet, die auf einem gemeinsamen Kommunikationskanal erfüllt werden müssen. Das dritte Modell ist zugleich auch das allgemeinste Modell, in dem Jobs mit bestimmten Ressourcenanforderungen an Prozessoren verteilt werden müssen, aber auch die Ressource noch zugeteilt werden muss.

Das vierte Modell erfasst dagegen mögliche Strategien für hochdynamische Systeme, in denen sich stetig verändernde Beschränkungen eingehalten werden müssen. Genauer wird hier der Energieverbrauch eines einzelnen Prozessors unter variablen Geschwindigkeitsschranken und veränderlichen Energiekosten minimiert.

Abstract

In today's data and computing centers, the available computing power of a system often is sufficient, but memory and the data rate become the bottleneck instead. Scheduling algorithms usually deal with the assignment of jobs to processors, but without any global constraint on the computing center as a whole. In this thesis, new scheduling problems incorporating such global properties are introduced. Four (slightly) different models capturing aspects of these properties are studied.

The first three models are similar in that a resource with a limited capacity is shared among multiple processors, and mostly the objective is to minimize the makespan, i.e., the time until all jobs are completed. The focus of the first model is on the assignment of the resource to the processors, where for each processor a queue of jobs is already fixed. The second model focuses on interjob communication, where given communication requirements between jobs need to be scheduled on a common communication channel. Finally, the third model is the most general case, where jobs with a certain resource requirement need to be scheduled on the different processors, but the resource has to be assigned as well.

On the other hand, the fourth model captures possible strategies for highly dynamic systems, where constraints may even change continuously over time. Here, the energy consumption of a single processor is minimized while adhering to variable speed limits and incorporating fluctuating energy costs.

Contents

Preface

Writing a PhD thesis takes more than a single person at a desk. I am grateful for everyone who crossed my path during the last years. I have met lots of wonderful people, it has been a great pleasure to work together with them, and many of them have become very good friends.

Most notably, I want to thank my advisor Friedhelm Meyer auf der Heide, who helped me understand the great vision behind the theory of computing, but also behind computer science in general. His inspirations enabled me to find interesting models and variations, but at the same time, he gave me the freedom to decide for myself which area suited me most. The brilliant vision behind the Collaborative Research Centre 901 "On-The-Fly Computing" has also broadened my horizon substantially, and thanks to the CRC, I had the opportunity to present my work at international conferences and to spend short research periods at other universities.

For improving my mood every single day, I am thankful to all of my colleagues, especially to those who shared or still share an office with me, Peter Kling, Andreas Cord-Landwehr, Alexander Mäcker and Johannes Schaefer. Our working days have consisted of many collaborative research sessions, sometimes pondering about a tiny complex problem to be solved. These sessions have been so valuable, since different ideas and view points have solved many problems in the end. Even though others would deserve to be mentioned here as well, I restrict myself to naming three of my co-authors individually. Peter Kling introduced me to the area of scheduling and his view on models has often complemented my own, making it inspirational to work with him. The game theory sessions with Maximilian Drees were always a welcome diversion from scheduling, and during the last years, Max has become a very close friend. Finally, Alexander Mäcker has always been the best help with his knowledge of scheduling literature, which I perceive as unlimited, and the research sessions with him have always been very helpful.

Particular thanks are directed to my family, especially my parents, who have always shared plenty of valuable time with us. Thanks to them, I also started to love mathematics and computer science early and I am very grateful for their everlasting support for everything I have done in my life. Last but not least, I cannot put into words my gratitude to Farina, who is there for me every second of my life.

Sören Riechers
Paderborn, 15. August 2017

CHAPTER 1

Introduction

Computing centers do not cease to grow. Even for a single chip, where one might expect that physical limits should lead to a cessation of the increase in processing power, the limits are not yet reached. Instead, Moore's law [Moo65], stating that the number of transistors on a chip will double every two years, now holds true for more than 50 years. It has been observed that a similar law also holds for the increase in data traffic [CO02]. While parallelism strongly increases, it is comprehensible that communication between machines, processors and cores must also increase. Oftentimes, it even happens that the available data rate becomes more important than the device's speed. In extreme cases, this effect may lead to the device's speed having almost no influence, that is, if the available data rate is reduced by a certain factor $x > 1$, the runtime is increased by this factor [Zhu+12]. On a smaller scale, a shared communication channel such as a data bus yields similar results. Other examples for scarce resources include memory or processing power being shared among multiple virtual machines. Scheduling decisions thus have to include the distribution of the resource in addition to how and where services are executed, and the question of how to distribute the resource often becomes more important than on which processors services are scheduled.

In general, scheduling describes the problem of allocating resources as well as defining an order in which certain tasks are completed. Scheduling decisions start with everyday tasks such as planning a day at work, where e-mails need to be answered, meetings (at fixed times) need to be attended and phone calls need to be made, possibly with additional restrictions, for example due to differences between time zones.

In computing centers, scheduling decisions typically focus on services (or *jobs*) that have to be executed. Each job has specific properties such as the required processing power or data rate. An example for a simple scheduling model is as follows [Bła+07, Ch. 4]: On a single processor, there are jobs with a processing requirement and a release time and the objective is to finish all jobs as early as possible (the total time to complete all jobs is also called the *makespan*). Indeed, an optimal solution for this problem can be found quite easily by greedily scheduling jobs in the order of non-decreasing release times. This simple procedure is called *Earliest Release Time First (ERF)*. Similarly, if deadlines instead of release times are given, in case there is a *feasible solution*, i.e., a solution such that each job is finished before or at its deadline, an optimal solution can be found by scheduling jobs in order of non-decreasing deadlines (*Earliest Deadline First (EDF)*).

Scheduling is mostly regarded as an independent research area since 1954 [PS09], originating in the seminal paper by Johnson [Joh54]. Since then, the variety of scheduling problems has strongly increased and is still researched extensively today. Naturally, new aspects have been considered. A good overview of scheduling models is given by Leung [Leu04].

Among those are scheduling under resource constraints and energy-efficient scheduling (or speed scaling). In the former area, jobs additionally require resources that are shared among processors [GG75; BLK83]. Here, a job requires its full resource requirement in order to execute jobs. As the resources need to be assigned to the jobs, additional complexity is added to the original problem. In the latter area, processors can be sped up in order to improve their performance [YDS95]. However, this comes at the cost of increased power consumption. Typically, a linear increase in speed is assumed to lead to a cubic increase in power consumption, as also observed in practice [Bro+00].

In this thesis, related problems of resource constrained scheduling are considered in Chapters 2 to 4. Here, it is assumed that the resource requirement of a job can be fulfilled in arbitrary parts (that may also differ in size), whereas most related literature assumes that the supplied resource of a job remains constant over time. For the speed scaling variant considered in Chapter 5, there is an additional upper speed limit (translating to an upper power limit) that may change almost arbitrarily over time. This captures the trend to more power consumption and heat in computing centers, making the speed limited in the sense that speed needs energy, which is already limited by itself as only a certain power can be supplied, and energy also produces heat, which is particularly critical during high temperature periods. However, the model in Chapter 5 differs from the models in Chapters 2 to 4 in that only a single processor is given, hence no resource is shared among multiple processors. For an arbitrary number of processors sharing a common energy source, the problem is in line with the problems from Chapters 2 to 4, but seemingly becomes much more difficult to cope with and is left as an open problem (see also more details in Chapter 6).

1.1 Approximation and Online Algorithms

In the following, I give a short overview of how the quality of algorithms for offline and online problems is usually measured.

Approximation Algorithms. It is widely believed that the complexity class P of problems deterministically solvable in polynomial time is a strict subset of the class NP of problems nondeterministically solvable in polynomial time. Assuming this to be true, it can be shown that many computational problems cannot be (deterministically) solved optimally in polynomial time. In particular, this is true for the class of so-called NP-hard problems, denoting those problems that are at least as hard to solve as the hardest problems in NP. As super-polynomial runtimes quickly become intractable in practice, different methods of how to cope with such problems have been developed.

For offline optimization problems, i.e., problems where the full instance is known in advance and where the objective is to mimimize or maximize a certain objective value, the most popular among those methods is the design of polynomial-time approximation algorithms. Here, algorithms are developed that guarantee to be at most by a certain factor worse than the optimal solution. Formally, for a minimization problem, denoting $A(I)$ to be the value of a solution achieved by a given algorithm A for an instance I, and $\mathrm{OPT}(I)$ defined similarly, A is an *α-approximation* for some $\alpha \geq 1$ if $\frac{A(I)}{\mathrm{OPT}(I)} \leq \alpha$ holds for all instances I. Analogously, for a maximization problem, an algorithm A is an α-approximation for some $\alpha \geq 1$ if $\frac{A(I)}{\mathrm{OPT}(I)} \geq \frac{1}{\alpha}$ holds for all instances I. An algorithm has an *asymptotic* approximation ratio of α if, on any instance I, $A(I) = \alpha \cdot \mathrm{OPT}(I) + \mathrm{o}(\mathrm{OPT}(I))$.

The strongest variant of approximation algorithms are so-called polynomial-time approximation schemes. A polynomial-time approximation scheme (PTAS) is an approximation algorithm which takes an input parameter $\varepsilon > 0$ additionally to the problem instance and returns a $(1+\varepsilon)$-approximation for a given optimization problem. However, for the runtime, ε is assumed to be a constant, hence a PTAS is only required to be polynomial in the input size and not in the parameter ε. An efficient polynomial-time approximation scheme (EPTAS) is a PTAS where the runtime is bounded by $\mathrm{O}\left(n^c\right)$ with c being a constant independent of ε. For example, runtimes such as $c'^{f(1/\varepsilon)} n^c$, where c' may arbitrarily depend on $1/\varepsilon$, but c is independent of ε, are allowed. A fully polynomial-time approximation scheme (FPTAS) is a PTAS where the runtime is polynomial in $1/\varepsilon$ and n. An asymptotic PTAS (APTAS) is defined analogously to a PTAS, but with an *asymptotic* approximation ratio of $(1+\varepsilon)$. Asymptotic EPTAS (AEPTAS) and asymptotic FPTAS (AFPTAS) are defined analogously.

However, such approximation schemes often have a runtime that is too high for practical applications. For this reason, the focus of this thesis is on approximation algorithms with guaranteed bounds, but reasonable runtimes.

Online Algorithms. For online optimization problems, i.e., problems where jobs (or other items) arrive over time and their properties only become available at their release time, algorithms are typically analyzed in terms of their competitiveness. For a minimization problem, denoting $A(I)$ and $\mathrm{OPT}(I)$ similar to above, where $\mathrm{OPT}(I)$ is the optimal *offline* solution, an online algorithm A is α-competitive for some $\alpha \geq 1$ if for any instance I, it holds $\frac{A(I)}{\mathrm{OPT}(I)} \leq \alpha$. For a maximization problem, A is α-competitive for some $\alpha \geq 1$ if for any instance I, it holds $\frac{A(I)}{\mathrm{OPT}(I)} \geq \frac{1}{\alpha}$.

In terms of online problems, one often thinks of an adversary that builds up the instance over time. This can be done as the algorithm has no knowledge about the future and deciding whether a job arrives or not (and which properties it has) can be done by the adversary on the fly.

1.2 Outline of the Thesis

The focus of this thesis is to cope with scarce resources in scheduling problems. Throughout this thesis, I usually consider the preemptive setting. That is, jobs can be interrupted and resumed at any point in time without inducing additional cost. Note that in Chapter 2, the preemptive and non-preemptive settings are equivalent. In Chapter 4, however, the non-preemptive setting is considered. In this case, the results for the non-preemptive case directly carry over to the preemptive setting (because the bounds on the optimal algorithm remain valid). Note that preemptiveness should not be confused with migration, which allows to stop jobs at arbitrary times and resume them on a different processor. Migration is not allowed in the models of this thesis or the models in related literature.

In the following, I introduce the models considered in the different chapters. In Section 1.3, I compare the models and elaborate on important differences between the models. An overview of related work regarding all parts of the thesis is given in Section 1.4. In order to put this thesis in context with my other research, I conclude the introduction with a list of my own publications in Section 1.5.

Chapters 2 to 5 contain the main content of this thesis. Note that each of these chapters is self-contained except for related work which is summarized in Section 1.4 to avoid redundancy. In Chapters 2 to 4, a shared resource needs to be assigned to a number of processors. In contrast to original resource constrained scheduling [GG75; BLK83], the resource requirement of a job is assumed to be divisible among contiguous time steps by slowing down jobs. Chapter 5 considers a variant of energy-efficient scheduling, where power consumption limits and energy cost dynamically change over time. In the following, an overview of the models and results of the individual chapters is given.

Assigning a Sharable Resource in a Multiprocessor System. In this chapter, the model contains m identical processors sharing a continuously divisible resource. An assignment of a number of jobs to the m processors and the order of the jobs on each processor are already given. The time line is assumed to be composed of discrete time steps. It is the scheduler's task to distribute the resource among the processors. Here, each job j comes with a resource requirement $r_j \in [0, 1]$ and unit size, that is, a job can always be finished in one time step if granted its full resource requirement. If receiving only an x-portion of r_j, it is processed at an x-fraction of the full speed. For example, a job with resource requirement 70% can be finished in 3 time steps by granting it 30% of the resource in the first time step and 20% of the resource in the remaining two time steps. The objective is to find a resource assignment that minimizes the makespan.

In contrast to Chapter 4, where the assignment of the jobs to the processors also has to be done by the scheduler, this model rather focuses on the assignment of the resource to the processors. It is shown that finding an optimal solution is NP-hard if the number of processors is part of the input. Positive results include a polynomial-time algorithm for any constant number of processors. Since the runtime is infeasible for practical purposes, more efficient algorithm variants are also provided: a faster optimal algorithm for two processors and a $(2 - 1/m)$-approximation algorithm for m processors.

The model, analyses and results presented in this chapter are based on the following publications (conference and journal version):

> **2014** (with A. Brinkmann, P. Kling, F. Meyer auf der Heide, L. Nagel and T. Süß). "Scheduling Shared Continuous Resources on Many-Cores". In: *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, cf. [Bri+14].

> **2017** (with E. Althaus, A. Brinkmann, P. Kling, F. Meyer auf der Heide, L. Nagel, J. Sgall and T. Süß). "Scheduling Shared Continuous Resources on Many-Cores". In: *Journal of Scheduling*, cf. [Alt+17].

Multiprocessor Scheduling with a Sharable Communication Channel. Similarly to Chapter 2, this chapter considers m identical processors sharing a common resource, but in a different manner. In particular, the common resource can (and should) be seen as a communication channel shared among the processors. A set of tasks needs to be scheduled on the processors, where each task T_i consists of a set of jobs with interjob communication demands, represented by a weighted, undirected graph G_i. The shared communication channel can be used by jobs to communicate among each other while being processed in parallel. In each time step, the scheduler assigns jobs to the processors. The scheduler allows (parts of) the communication demands between scheduled jobs to be satisfied under the restriction that the overall communication does not exceed the capacity of the channel. Again, the objective is to find a schedule with minimum makespan in which the communication demands of all jobs (i.e., the sum of the shares of the communication channel assigned to it) are satisfied.

This problem is shown to be NP-hard in the strong sense even if the number of processors is constant and the underlying graph is a single path or a forest with arbitrary constant maximum degree. Consequently, approximation algorithms with a provable (asymptotic) approximation guarantee are designed and analyzed. If the underlying graph G, the union of the G_i, is a forest, an asymptotic approximation ratio of $\min\{1.8, 1.5\frac{m}{m-1}\} + 1$ is shown; for general graphs it is $\min\left\{1.8, \frac{1.5m}{m-1}\right\} \cdot$

$\left(\mathrm{arb}(G) + \frac{5}{3}\right)$, where $\mathrm{arb}(G)$ denotes the arboricity of G, i.e., the minimum number of forests into which the edges of G can be partitioned.

Parts of the model, analyses and results presented in this chapter are based on the following publication. A journal version with additional results is currently under submission.

> **2016** (with J. König, A. Mäcker and F. Meyer auf der Heide). "Scheduling with Interjob Communication on Parallel Processors". In: *Proceedings of the 10th International Conference on Combinatorial Optimization and Applications (COCOA)*, cf. [Kön+16].

Multiprocessor Scheduling with a Sharable Resource. This chapter also models m identical processors sharing an arbitrarily divisible resource. This resource is shared similarly to Chapter 2. A number of jobs is given, but in contrast to Chapter 2, the assignment of the jobs to the processors is not yet done. That is, the scheduler must assign the jobs to the processors as well as distribute the resource among them (e.g., for three processors in shares of 20%, 15%, and 65%) and adjust this distribution over time. Each job j comes with a size $p_j > 0$ and a resource requirement $r_j > 0$. Jobs do not benefit when receiving a share larger than r_j of the resource. However, similar to Chapter 2, providing them with a fraction of the resource requirement causes a linear decrease in the processing efficiency. The objective is to find a (non-preemptive) job and resource assignment minimizing the makespan.

The main result of this chapter is an efficient approximation algorithm which achieves an approximation ratio of $2 + 1/(m-2)$. It can be improved to an (asymptotic) ratio of $1 + 1/(m-1)$ if all jobs have unit size (that is, they still have different resource requirements). The described algorithms also imply new results for a well-known bin packing problem with splittable items and a restricted number of allowed item parts per bin as well as for certain cases of the model from Chapter 3.

Based upon the above solution, an additional setting with so-called tasks is introduced, each containing several jobs. The objective is to minimize the average completion time of tasks, where a task is completed when all its jobs are completed. As an extension of the model with single jobs, this problem remains NP-hard and approximation algorithms with similar guarantees are derived.

The results presented in this chapter are based on the following publication.

> **2017** (with P. Kling, A. Mäcker and A. Skopalik). "Sharing is Caring: Multiprocessor Scheduling with a Sharable Resource". In: *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, cf. [Kli+17].

Scheduling with a Bounded Speed Limit and Variable Energy Costs. In this chapter, an extension of the dynamic speed scaling model introduced by Yao et al. [YDS95] is considered: A set of jobs, each with a release time, deadline, and workload, has to be scheduled on a single, speed-scalable processor. Both the maximum allowed speed of the processor and the energy costs may vary continuously over time. The objective is to find a feasible schedule that minimizes the total energy costs.

Theoretical algorithm design for speed scaling problems often tends to discretize problems, as the tools in the discrete realm are often better developed or understood. Using the above speed scaling variant with variable, continuous maximal processor speeds and energy prices as an example, it is demonstrated that a more direct approach via tools from variational calculus can not only lead to a very concise and elegant formulation and analysis, but also avoids the "explosion of variables/constraints" that often comes with discretizing [Ant+14]. Using well-known tools from calculus of variations, combinatorial optimality characteristics for the continuous problem are derived and a quite concise and simple correctness proof is provided. A combinatorial algorithm for this problem is suggested and the optimality characteristics are used to prove that this algorithm indeed returns an optimal solution.

The results in this chapter are based on the following publication.

> **2017** (with A. Antoniadis, P. Kling and S. Ott). "Continuous Speed Scaling with Variability: A Simple and Direct Approach". In: *Theoretical Computer Science* vol. 678, cf. [Ant+17].

1.3 Overview of the Different Models

In the following, I will evaluate the differences between the strongly related models from Chapters 2 to 4 (Section 1.3.1). I will then discuss the relation to the model from Chapter 5 in Section 1.3.2.

1.3.1 Sharing a Resource among Multiple Processors

First note that for all the models in Chapters 2 to 4, a job has a size $p_j \in \mathbb{R}_+$ and a resource requirement $r_j \in \mathbb{R}_+$. The model in Chapter 4 is the most general of these models, where p_j and r_j are both chosen arbitrarily. Also, the assignment of the jobs to the processors is not yet fixed.

In contrast, the assignment of jobs to processors is already fixed in Chapter 2. Also, all jobs have unit size $p_j = 1$ and resource requirement $r_j \leq 1$.

In Chapter 3, resource demands are communication demands among different nodes. That is, rather than being given a set of single jobs that need to be assigned to one processor each, there is a number of tasks, each consisting of a connected, undirected graph. The resource (communication) requirement is given as weights on the edges and can be any $r_j \in \mathbb{R}_+$. In order to satisfy the communication requirement of an edge, both adjacent nodes need to be scheduled on two separate processors at the same time. The size of an edge (not to be confused with the communication requirement or weight) is assumed to have unit size ($p_j = 1$), implying that the communication demand can be processed at once if the full communication requirement is supplied. Nevertheless, note that if the communication requirement of an edge is larger than the overall size of the communication channel, the full requirement of the edge cannot be fulfilled in one time step. However, similarly to both other models, an overall communication (or resource) limitation persists.

The Convenience of Freedom. As noted above, the model in Chapter 4 is the most general model. However, when looking for algorithms with a short runtime, achieving good approximation factors seems to be simpler if the assignment of jobs to processors is still necessary. By assigning jobs depending on the required resource, it can be easily avoided that ill fitting jobs are scheduled at the same time. That is, the scheduler can ensure that the sum of resource requirements of currently scheduled jobs is close to the available resource requirement. Intuitively, the algorithms used in Chapter 4 do exactly that. At each point in time, the scheduler tries to maintain high parallelism as well as high resource usage. That is, given m processors and an available resource of R, the scheduler tries, at any point in time, to schedule m jobs such that

1. $m - 1$ of them receive their full resource requirement, and
2. the full resource R is used.

Only at a point in time where it is no longer possible to maintain both, i.e., there are only jobs with a very small or with a large resource requirement left, is one of the two properties violated. However, it can be proven that choosing jobs in a particular way ensures that once this happens, the property not violated in that very time step remains valid until no more jobs need to be scheduled, which implies a good approximation guarantee.

Normalizing Job Sizes. One problem of the above algorithm is its frailty regarding job sizes. That is, as long as there are still jobs to be scheduled, it is ensured that one of the two conditions held for the whole time: either $m-1$ jobs were executed in parallel in each time step or the full resource R was used in each time step. However, as the length of the jobs is not incorporated in the algorithm and prioritization only depends on the resource requirements, it can happen for roughly the second half of the time line that only one very long job with a low resource requirement is scheduled. This increases the approximation ratio by a factor of almost two. In contrast, if all jobs have unit size, there is at most one time step where one of the two conditions is not fulfilled. This implies an asymptotic approximation ratio of $1 + \frac{1}{m-1}$, which approaches one for a high number of processors.

Fixing the Job Assignment. In contrast, the assignment of jobs to processors is already fixed in the model of Chapter 2. For a fixed number of processors, the NP-hardness of the problem dissolves, and it remains hard only if the number of processors is part of the input. However, approximating the solution with a fast approximation algorithm seems to become more difficult. This is because if trying to fit jobs well locally at some point in time, that is, such that resource utilization and parallelism are both maximized at this point, jobs at a much later or much earlier time may fit even worse. This may imply a sequence of errors and, thus, situations where jobs with large (low) resource requirement have to be scheduled at the same time, thereby reducing parallelism (resource utilization) and worsening the approximation substantially.

Intuitively, for two processors, this effect can be seen as two sawtooth patterns facing each other (where a spike represents a high resource requirement), where each pattern has additional irregularities. Now they can be shifted such that a spike always meets a low point of the other processor and a low point always meets a spike of the other processor, whereas the irregularities hurt the solution only slightly. This would result in a schedule with a small makespan. However, if the wrong jobs are prioritized, spikes and low points each meet their counterparts, which results in giving away resource and parallelism and implying a large makespan. Refer also to Section 2.4.3 with Figure 2.5 for a more detailed description and a visualization.

Nevertheless, a branch and bound algorithm with runtime $\mathrm{O}\left(n^2\right)$ that solves this problem optimally for two processors is given in Chapter 2, as one can still cope with the above problems for the two processor case. For more processors, however, this “fitting of saw teeth” becomes more demanding, and there seems to be no fast algorithm finding an optimal solution. By constructing a dynamic program through cleverly arranging possible configurations, the optimal solution can still be found, but only at the cost of high (but still polynomial) runtime.

The simple approximation algorithm introduced in Chapter 2 for a practically tractable runtime has an approximation guarantee that approaches two for a high number of processors, which is much worse than the approximation ratio in Chapter 4, which approaches one as m tends to infinity for the equivalent setting of unit size jobs. This is in accordance with the expectation that the possibility to avoid ill fitting jobs in Chapter 4 makes it easier to approximate a solution within a satisfying factor.

Shared Resources in Connected Components. In the model of Chapter 3, a number of tasks is given, each consisting of a connected, undirected graph. A weight or communication requirement is assigned to each edge. A common communication channel now represents the shared resource. In contrast to the models discussed above, the communication demand of an edge can only be satisfied by assigning both adjacent nodes to two processors at the same time. A simple way to meet all communication demands would be to schedule each edge separately. That is, each node v is split into $deg(v)$ copies, each connected only to one edge. If each task can be represented by a tree, it is later shown that this simple procedure together with the algorithm from Chapter 4 leads to a better approximation guarantee than the algorithms in Chapter 3. This is because splitting a tree into single edges results in at most a doubling the number of nodes, hence only losing a factor of at most two.

However, to capture the behavior of tasks not having the simple structure of a tree, the notion of *arboricity* is used. The arboricity of a graph denotes the minimum number of forests into which the edges of a graph can be decomposed. By using an existing result, it is also possible to decompose an arbitrary graph into $\operatorname{arb}(G)$ forests and one additional graph of degree at most 2. Hence, decomposing the graph into forests, then splitting the forest into separate edges and using the algorithm from Chapter 4 with approximation ratio $1 + 1/(m-1)$ leads to an approximation ratio of $(2m/(m-1))\operatorname{arb}(G)$, whereas the approximation algorithm described in Chapter 3 guarantees a ratio of $\min\{1.8, 1.5m/(m-1)\} \cdot (\operatorname{arb}(G) + 5/3)$. For large m, the latter approximation thus has a similar or better guarantee for any $\operatorname{arb}(G) \geq 5$. For small m (i.e., if the minimum equals 1.8) this is already the case for a smaller arboricity.

1.3.2 Shared Resources and Energy-Efficient Scheduling

In the model of Chapter 5, only one speed-scalable processor is given, but with maximum speed and energy cost both varying continuously over time. Hence, this model does not cover limits of resources shared among multiple processors. The variant where a variable maximum power is the shared resource used by multiple processors is left as an open question.

However, the variant from Chapter 5 dealing with only one processor gives insights about the necessary techniques to cope with such flexible limits, for example energy limits in computing centers. Also, as most of the energy is absorbed as heat, the temperature of a processor can be associated with a maximum power consumption. In order to avoid overheating, a maximum speed that changes over time is determined, which also results in the kind of problem dealt with in this chapter.

1.4 Related Work

In the following, I give an overview of related literature in the area of scheduling. In particular, I review scheduling problems where the distribution of scarce resources among processors is the main challenge.

The Origins of Scheduling. The area of scheduling is believed [PS09] to be seen as a distinct research area since Johnson [Joh54] composed his paper about production schedules, which is called flow shop nowadays. In his paper, he considers a problem where different items undergo a production process. Items have to be processed by one machine first and by a second machine afterwards. Each item has an overall processing time for each of the two machines, which is regarded as the sum of setup time and work time. Johnson gives an optimal algorithm for this problem by arranging the processing times of the items in two columns for the first and the second machine. The smallest processing time among all listed times (i.e., a processing time of any job on any machine) is picked, where ties are broken arbitrarily. If the processing time concerns the first machine, the respective job is scheduled first;

if it applies to the second machine, the respective job is scheduled last. Repeating this procedure until all jobs are processed (hence building the schedule from the ends to the middle) results in an optimal schedule. For the restricted case of a similar problem with three machines, where for any pair of items, the processing time of the first item on the first machine is larger than the processing time of the second item on the second machine, an optimal solution can be found using a similar algorithm. The same applies if a similar property holds for the third instead of the first machine.

Multiprocessor Scheduling. In the classical multiprocessor makespan scheduling problem, a set of jobs, each having a specific processing time, needs to be scheduled on m identical machines so as to minimize the makespan. For this problem, an EPTAS is known [Alo+98] if m is part of the input. For fixed m even an FPTAS is possible [HS76]. Compared to the models in Chapters 2 to 4, this model does not incorporate additional resource requirements. However, reducing the resource requirements of each job in the model of Chapter 4 to an infinitesimal amount, it becomes equivalent to the model without resource requirements.

In terms of fast algorithms, Graham [Gra69] introduces list scheduling algorithms which are used today as a typical example achieving a reasonable approximation quality. Here, jobs are added one after another from a sorted list. Each job is added to the processor with the lowest workload, that is, the processor that would finish all jobs first with respect to the current schedule. Graham [Gra69] proves that for the setting with m machines, an arbitrarily ordered list results in a $(2 - \frac{1}{m})$-approximation. If the list is sorted by non-increasing job size (also called *longest processing time first (LPT)*), it achieves an approximation ratio of $\frac{4}{3} - \frac{1}{3n}$. For the general case, the authors also prove the following result. Assume n jobs are given and the list starts with the k longest of these jobs in an order resulting in an optimal solution (limited to these k jobs). No matter in which order the remaining jobs are added to the list, the resulting list scheduling algorithm achieves an approximation ratio of $1 + \frac{1-1/n}{1+\lceil k/n \rceil}$. For example, this also implies that if the largest m jobs are distributed to the processors (one job for each processor), and the remaining list is ordered arbitrarily, the resulting solution is at most by a factor of $\frac{3}{2} - \frac{1}{2n}$ worse than the optimum.

Resource Constrained Scheduling. Research on scheduling with resource constraints originates from the 1970s. In [GG75], Garey and Graham introduce a model with m processors and a set of k resources. A number of jobs has to be scheduled, where each job has a processing time and a specific demand for each resource. In their model, the execution of a job may not be interrupted and resumed later, which corresponds to the non-preemptive setting described earlier. For any time, it is the scheduler's task to assign a set of jobs to the processors such that for any resource, the sum of supplied resource shares of the jobs does not exceed the available resource. This is in contrast to the models from Chapters 2 to 4, where the scheduler can assign a lower resource to a job than its requirement, leading to a slower execution of this job. The authors consider list-scheduling algorithms and prove an approximation factor of at most $\min\{\frac{m+1}{2}, k+2-\frac{2k+1}{m}\}$ for this problem. They also consider this problem with precedence constraints: that is, each job may depend on the completion of one or multiple other jobs. For this case, they show that their list-scheduling algorithm achieves a tight approximation factor of m, which is the same as the trivial algorithm assigning all jobs to the same processor would achieve.

For the restriction to a single resource and no precendence constraints, the results discussed above directly imply that their list scheduling algorithm achieves an approximation factor of $3-\frac{3}{m}$. In [NW15], the authors improve these results by presenting a $(2+\varepsilon)$-approximation algorithm for this problem using techniques such as grouping and linear programming. They also prove that even for unit size jobs, this problem cannot be approximated within an (absolute) approximation ratio less than $\frac{3}{2}$ unless $P = NP$ by a straightforward reduction from the *Partition* problem. For this unit size case, Epstein and Levin [EL10] introduce an asymptotic fully polynomial-time approximation scheme (AFPTAS).

Finally, Jansen et al. [JMR16] very recently published new results where they present an AFPTAS for the general problem (however, still with a single resource and without precedence constraints). They also introduce an AFPTAS for the machine scheduling problem with resource dependent processing times. This model is quite similar to the models studied in Chapters 2 to 4 in assuming that a job supplied with a smaller part of the resource than its requirement cannot be finished with full speed. However, in their model the resource supplied to a job must remain constant during the full execution time window, whereas a job's share of the resource in Chapters 2 to 4 may be changed in any time step. They also assume the overall resource as well as each job's share of the resource to be an integer number. On the other hand, Jansen et al. [JMR16] also allow other dependencies than the linear decline in processing speed, that is, they introduce a processing time function relating the set of possible resource shares for a job to arbitrary processing speeds. The models from Chapters 2 to 4 are more realistic for applications where the resource can be split arbitrarily, for example if a common data rate is involved, as a job only needs to receive a certain amount of data as soon as possible. In contrast, the model considered in [JMR16] seems more realistic for applications where the job is configured for a specific share of the resource, resulting in the requirement that the resource remains constant during the full execution time. For example, a job may have a low memory configuration that comes with a longer processing time, but allowing it to have a higher memory consumption for a part of the processing time may not speed its execution.

For a deeper insight into resource constrained scheduling, for example with multiple resources, the interested reader is referred to [Leu04, Chs. 23-24] and [Bła+07, Ch. 12].

Bin Packing. Bin packing has a long history in computer science research and a huge body of literature on several variants of bin packing problems emerged in the past. In its most basic variant, n items of sizes $0 < s_i \leq 1$ need to be placed into as few bins of capacity 1 as possible. Similar to resource constrained scheduling, this problem is easily seen to be NP-hard by a reduction from the *Partition* problem, which also directly gives an inapproximability for an (absolute) approximation ratio below $\frac{3}{2}$ unless $P = NP$. This bound is actually achieved by the well-known *First Fit Decreasing* strategy, which first sorts the items in decreasing order by their sizes and then places the current item to be packed into the first bin it fits into. In [Dós+13], Dósa et al. also prove that *First Fit Decreasing* uses at most $\frac{11}{9}\mathrm{OPT} + \frac{6}{9}$ bins and that this bound is tight, implying an exact asymptotic approximation ratio of $\frac{11}{9}$. When considering asymptotic approximation algorithms, even (fully) polynomial-time approximation schemes (A(F)PTAS) are known [VL81; KK82].

While there are dozens of variants of this basic problem, the problem supposedly closest related to the problems from Chapters 2 to 4 is *bin packing with cardinality constraints and splittable items* as introduced in [Chu+06]. In this problem, a set of n items needs to be packed into as few bins of capacity one as possible. In contrast to standard bin packing, items can have an arbitrary size in $(0, \infty)$ and may be split and distributed among different bins. However, there is a constraint on the maximum number of (parts of) different items that may be packed into a single bin given by some predefined value k. Chung et al. [Chu+06] prove this problem to be strongly NP-hard for $k = 2$ and provide a simple approximation algorithm with an asymptotic approximation ratio of $3/2$ (also for $k = 2$). In [ES11], the authors extend the NP-hardness to any fixed $k \geq 2$. They also give efficient algorithms with asymptotic approximation ratio $7/5$ for $k = 2$ and an absolute approximation ratio of $2 - 1/k$ for $k \geq 2$, respectively. Finally, Epstein et al. [ELS12] present an EPTAS for the case $k = \mathrm{o}(n)$. They also prove that for $k = \Theta(n)$ a polynomial-time approximation algorithm with a ratio smaller than $3/2$ cannot exist unless $\mathrm{P} = \mathrm{NP}$.

Note that bin packing with cardinality constraints and splittable items is, except for the lack of the notion "preemption", equivalent to the main problem from Chapter 4 with unit size jobs: If items correspond to jobs of size 1 and each bin is identified with one time step, the packing of a bin describes the jobs executed in this time step and the part size of an item corresponds to the share of the resource the respective job gets. The cardinality constraint k corresponds to having k processors.

Discrete-Continuous Scheduling. The notion of *discrete-continuous scheduling* traces back to several papers by Józefowska and Weglarz, first and foremost [JW98]. While most results in this area study scenarios where the amount of allocated resources influences the processing time or release dates of jobs (see [JJL07] for a survey), Józefowska and Weglarz [JW98] consider the case where the amount of allocated resources influences the processing *speed* of jobs. More precisely, if the function $R_j \colon \mathbb{R}_{\geq 0} \to [0, 1]$ models the share of the resource that job j gets assigned at some time $t \in \mathbb{R}_{\geq 0}$, its workload is processed at a speed of $f_j(R_j(t))$. Here, f_j

models how a job's processing speed is affected by the received resource amount and is assumed to be continuous and non-decreasing with $f_j(0) = 0$. Using this resource model, the authors consider the problem of scheduling n non-preemptable and independent jobs on m processors. They propose an analysis framework based on a mathematical programming formulation and demonstrate its use for the objective of minimizing the schedule's makespan. For certain classes of f_j, this yields a simple analytical solution [JW98; Józ+99]. This holds especially for convex functions f_j, which encourage the scheduler to assign the full resource to a single processor. Finding an optimal solution for more realistic cases (especially concave f_j) remains infeasible. The results in [JW98] initiated several research efforts in this area, including a transfer of the methodology to other scheduling variants (e.g., average flow time instead of makespan [JW96]) as well as several heuristic approaches to obtain practical solutions in the general case [Józ+00; Józ+02; Kis05; Wal11]. A detailed survey about these results can be found in [Weg+11] (especially Section 7).

The scheduling models with shared resources in this thesis have several characteristics in common with discrete-continuous scheduling problems. In particular, the jobs' resource requirements can be modeled via concave functions f_j of the form $f_j(R) = \min({}^{R}/_{r_j}, 1)$, where the value r_j denotes the resource requirement of job j (cf. Section 2.1.1). That is, the speed used to process a job depends linearly on the share of the resource it receives, but is capped at one. In contrast to the results presented in Chapters 2 to 4, most of the aforementioned results for the discrete-continuous setting are of heuristic nature and do not provide any provable quality guarantees with respect to the resulting schedules, and cases that can be analyzed analytically turn out to feature quite simple solution structures [JW98; Józ+99].

Order Scheduling Models. With respect to the second part of Chapter 4, where a model generalization for tasks that are composed of multiple jobs is considered, [LLP05] should be mentioned. Here, a production model is considered where tasks represent orders and each job of an order must be processed on a subset of specific machines. However, note that these *order scheduling models* do not consider resource sharing in the sense of the models in this thesis, but instead only the allocation to the (non-identical) machines.

Energy-Efficient Scheduling. The area of energy efficient scheduling, often also described as speed scaling, has been initiated by Yao et al. [YDS95]. They assume to have a single speed-scalable processor: that is, one processor that can be sped up arbitrarily, but at the cost of increased power consumption. In their model, the power the processor requires is described by a convex power function $P\colon \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$. More exactly, when running with speed s, they assume that the processor has a power consumption of $P(s) = s^{\alpha}$, where $\alpha > 1$ is a constant called the *energy exponent.* This assumption is natural, as the typical power consumption of CMOS devices can roughly be estimated by s^3 and CMOS devices will presumably remain the dominant technology in the near future [BKP07]. The authors present an optimal algorithm for this problem, which is called the YDS algorithm. The main idea of the algorithm is to develop a sense of density of jobs, that is, determining intervals containing a large amount of workload per time unit. Recursively identifying the densest interval, then scheduling all contained jobs within this interval with minimal uniform speed, and proceeding by deleting the interval from the timeline yields an optimal solution. Ideas from this algorithm are also used for the algorithm in Chapter 5. The authors proceed by introducing two online algorithms for this problem, which they call *Average Rate* and *Optimal Available*. Intuitively, the *Average Rate* algorithm processes each job such that its processing speed is the same over the full interval between its release time and deadline. Hence, at any point in time, the overall processing speed is the sum of the job's processing speeds. Yao et al. [YDS95] prove that *Average Rate* has a competitive ratio of at most $2^{\alpha-1}\alpha^{\alpha}$. In [Ban+08; Ban+11], the authors show that the analysis is almost tight by providing a lower bound of $\frac{(2-\delta)^{\alpha}}{2} \cdot \alpha^{\alpha}$, where δ approaches 0 when α approaches infinity. The *Optimal Available* algorithm is computationally more intensive: At any point in time where a new job arrives, the optimal solution of all currently available jobs is computed, for example by executing the YDS algorithm. Bansal et al. [BKP07] prove that the competitive ratio of *Optimal Available* is exactly α^{α}. This implies that *Optimal Available* is superior to *Average Rate* in terms of competitiveness, but this comes with a computational overhead. Bansal et al. [BKP07] also present a new algorithm, which they call *BKP* and which estimates the density of available jobs in a different way in order to behave similar to the YDS offline algorithm. They prove that *BKP* has a competitive ratio of at most $2\left(\frac{\alpha e}{\alpha-1}\right)^{\alpha}$, thus having a stronger guarantee than *Optimal Available* for $\alpha \geq 5$. In Bansal et al. [Ban+09], the authors show that the exponential dependency is inherent to the problem: that is, they prove that any online algorithm has a competitive ratio of at least $\frac{e^{\alpha-1}}{\alpha^{\alpha}}$. See also [Alb10] and [IP05] for broader surveys on energy-efficient algorithms.

Energy-Efficient Scheduling with Maximum Speed or Varying Energy Prices. Special cases of both the maximum speed and the electricity tariff setting have been studied before. Chan et al. [Cha+09] and Li [Li11] assume that there is a constant upper bound on the available speed, and one wants to maximize the throughput of the schedule while minimizing the power consumption. Chan et al. [Cha+09] present an $\mathrm{O}(1)$-competitive algorithm in terms of throughput and energy. Allowing the maximum speed of the online scheduler to be $(1+\varepsilon)$ times the original maximum speed for some $\varepsilon > 0$ enables the authors to improve the competitive ratio on throughput to any value $1+\delta$ with $\delta > 0$. However, the competitive ratio on power consumption remains a larger constant. Li [Li11] consider the offline variant of this problem. They present an algorithm which is a 3-approximation of the throughput and a $\frac{(\alpha-1)^{\alpha-1}(3^{\alpha}-1)^{\alpha}}{2\alpha^{\alpha}(3^{\alpha-1}-1)^{\alpha-1}}$-approximation of the power consumption.

On the other hand, Fang et al. [Fan+15] consider electricity tariffs, but without an upper bound on the speed and in a much more restricted setting: their model is equivalent to considering only one-job instances and discrete dynamics in the problem from Chapter 5. They develop an optimal polynomial-time algorithm by a technique which to some extent resembles ours. However, since Chapter 5 deals with a significantly more general setting, several important aspects not appearing in [Fan+15] have to be considered, in particular the KKT optimality conditions need to be extended using variational calculus. Electricity tariffs have also been considered beyond the speed-scaling setting, see for example [KT11].

Further, Thang [Tha13] uses the Lagrangian dual of a mathematical program in order to analyze several online scheduling algorithms with flow-time objectives. Although [Tha13] also has the same view of optimizing over a set of arbitrary speed functions, it differs from the approach in Chapter 5 in that Lagrangian duality is used more as a tool for analyzing the approximation ratio, rather than for characterizing an optimal solution and deriving an optimal algorithm. Finally, Bansal et al. [BCP09] consider a speed scaling problem where energy is supplied at a limited rate. However, their supply rate does not vary over time. In fact, there is another significant difference between their model and the model from Chapter 5, as they consider also a storage device and seek to minimize the constant supply rate.

1.5 Own Publications

In the following, I present a list of my own publications that I co-authored while studying the topics of this thesis. The publications are given in reverse chronological order. This list merely serves to put the topics from this thesis in context with my other research.

2017 (with M. Drees, M. Feldotto and A. Skopalik). “Pure Nash Equilibria in Restricted Budget Games”. In: *Proceedings of the 23rd International Computing and Combinatorics Conference (COCOON)*, cf. [Dre+17].

2017 (with A. Antoniadis, P. Kling and S. Ott). “Continuous Speed Scaling with Variability: A Simple and Direct Approach”. In: *Theoretical Computer Science* vol. 678, cf. [Ant+17].

2017 (with E. Althaus, A. Brinkmann, P. Kling, F. Meyer auf der Heide, L. Nagel, J. Sgall and T. Süß). “Scheduling Shared Continuous Resources on Many-Cores”. In: *Journal of Scheduling*, cf. [Alt+17].

2017 (with P. Bemmann, F. Biermeier, J. Bürmann, A. Kemper, T. Knollmann, S. Knorr, N. Kothe, A. Mäcker, M. Malatyali, F. Meyer auf der Heide, J. Schaefer and J. Sundermeier). “Monitoring of Domain-Related Problems in Distributed Data Streams (to appear)”. In: *Proceedings of the 24th International Colloquium on Structural Information and Communication Complexity (SIROCCO)*, cf. [Bem+17].

2017 (with P. Kling, A. Mäcker and A. Skopalik). “Sharing is Caring: Multiprocessor Scheduling with a Sharable Resource”. In: *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, cf. [Kli+17].

2017 (with A. Mäcker, M. Malatyali and F. Meyer auf der Heide). “Non-Clairvoyant Scheduling to Minimize Max Flow Time on a Machine with Setup Times (to appear)”. In: *Proceedings of the 15th Workshop on Approximation and Online Algorithms (WAOA)*, cf. [Mäc+17b].

2017 (with S. Li, A. Mäcker, C. Markarian and F. Meyer auf der Heide). “Towards Flexible Demands in Online Leasing Problems (to appear)”. In: *Algorithmica*, cf. [Li+17].

2017 (with A. Mäcker, M. Malatyali and F. Meyer auf der Heide). “Cost-Efficient Scheduling on Machines from the Cloud”. In: *Journal of Combinatorial Optimization*, cf. [Mäc+17a].

2016 (with A. Mäcker, M. Malatyali and F. Meyer auf der Heide). “Cost-Efficient Scheduling on Machines from the Cloud”. In: *Proceedings of the 10th Annual International Conference on Combinatorial Optimization and Applications (COCOA)*, cf. [Mäc+16].

2016 (with J. König, A. Mäcker and F. Meyer auf der Heide). “Scheduling with Interjob Communication on Parallel Processors”. In: *Proceedings of the 10th International Conference on Combinatorial Optimization and Applications (COCOA)*, cf. [Kön+16].

2015 (with M. Drees, M. Feldotto and A. Skopalik). “On Existence and Properties of Approximate Pure Nash Equilibria in Bandwidth Allocation Games”. In: *Proceedings of the 8th International Symposium on Algorithmic Game Theory (SAGT)*, cf. [Dre+15].

2015 (with A. Mäcker, M. Malatyali and F. Meyer auf der Heide). “Non-Preemptive Scheduling on Machines with Setup Times”. In: *Proceedings of the 14th International Symposium on Algorithms and Data Structures (WADS)*, cf. [Mäc+15].

2015 (with S. Li, A. Mäcker, C. Markarian and F. Meyer auf der Heide). “Towards Flexible Demands in Online Leasing Problems”. In: *Proceedings of the 20th International Computing and Combinatorics Conference (COCOON)*, cf. [Li+15].

2014 (with A. Brinkmann, P. Kling, F. Meyer auf der Heide, L. Nagel and T. Süß). “Scheduling Shared Continuous Resources on Many-Cores”. In: *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, cf. [Bri+14].

2014 (with M. Drees and A. Skopalik). “Budget-Restricted Utility Games with Ordered Strategic Decisions”. In: *Proceedings of the 7th International Symposium on Algorithmic Game Theory (SAGT)*, cf. [DRS14].

CHAPTER 2

Assigning a Sharable Resource in a Multiprocessor System

The processor scheduling problem considered in this chapter is motivated by the observation that, in many cases, it is not a device's speed or energy consumption that limits the progress of a given computation but the fact that data cannot be provided at the necessary rate. At first glance, this seems more a network issue than a problem of interest for processor scheduling. After all, bandwidth bottlenecks are typically imposed by the interconnection of devices (e.g., networks or data buses), and there is a huge body of literature concerned with such issues on the network layer. However, the analysis in this area typically concentrates on the *network's* performance. In contrast, our model focuses on how the distribution of the bandwidth shared by a fixed set of processing units can affect their *computational* performance. That is, given some information about the bandwidth requirement of a program (e.g., when does it need how much bandwidth to progress at full speed), the scheduler can speed up critical jobs by a suitable assignment of the available bandwidth to the different processors. Typical examples for such settings are many-core systems: They provide an immense computing power through the sheer number of processor cores. Yet, many (if not all) of the chip's cores share a single data bus to the outside world. If such a system has to process I/O-intensive tasks (as typical for scientific computing), the available bandwidth becomes the computational bottleneck, and the bandwidth distribution becomes the decisive scheduling factor.

A First Glimpse at the Model. From a more abstract point of view, the aforementioned bandwidth scheduling can be seen as a variant of resource constrained scheduling, the bandwidth being an example for the resource. Imagine a system consisting of several identical processors that run at a fixed speed and share a given resource. Assume that the resource is the system's performance bottleneck, in the sense that the runtime of programs (tasks) depends directly (that is to say, linearly) on the share of the resource they are allowed to use. Each task provides information about its resource requirements by stating what share of the resource it needs at different phases of its processing to run at full speed. Thus, we can imagine a task i to consist of a number n_i of jobs that must be processed sequentially, one after another. Each job represents a phase of the task's processing where the resource requirement is constant. The length of the phase (i.e., the job's processing time) is minimal at full speed and increases by a factor of $1/x$ if only a portion $x \in [0, 1]$ of the requested resource share is provided. We use the term CRSHARING to refer to this problem of sharing continuous resources; see Section 2.1.1 for a more formal description.

We approach the problem by concentrating on the assignment of resources, removing the (classical) scheduling aspect almost completely. That is to say, we consider a scenario in which each processor has exactly one task, and each task consists of jobs of unit workload (but different resource requirements). Moreover, we assume discrete time steps, such that the scheduler can change the resource assignment only at the beginning of such a time step. As we will see, even this simple setting proves to be challenging.

Outline. Section 2.1 starts with a formal model description of the CRSHARING problem in Section 2.1.1, an overview of our contribution in Section 2.1.2, and some basic definitions and results in Sections 2.1.3 to 2.1.5. Our main results are given in Sections 2.2 to 2.4, where we study the complexity of the CRSHARING problem and present algorithmic options for the CRSHARING problem.

2.1 Preliminaries

In Section 2.1.1, we start by defining the model for the general version of the CRSharing problem, which considers jobs of arbitrary sizes. Afterward, we discuss an alternative interpretation of our model that will ease our argumentation in the analysis part. Note that while the model description considers jobs of arbitrary sizes, from Section 2.1.2 on, which summarizes our results, we only consider problem instances in which all jobs are of unit size. In Section 2.1.3, a graphical representation of schedules is introduced, supplying the reader with an idea of the underlying structure. Section 2.1.4 is intended to equip her with the tools needed for the analysis in later sections by discussing and proving some basic structural properties. Finally, we analyze a simple round robin algorithm in Section 2.1.5.

2.1.1 Model & Notation

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

Alternative Model Interpretation. An alternative interpretation of our scheduling problem can be obtained by the following observation: Consider a job (i,j) whose processing is started at time step t_1. It receives a share $R_i(t_1) \in [0,1]$ of the resource.

By the previous model definition, exactly $\min({}^{R_i(t_1)}/_{r_{ij}}, 1)$ units of its processing volume are processed. Similarly, in the next time step $\min({}^{R_i(t_1+1)}/_{r_{ij}}, 1)$ units of its processing volume are processed. Consequently, the job is finished at the minimal time step $t_2 \geq t_1$ such that $\sum_{t=t_1}^{t_2} \min({}^{R_i(t)}/_{r_{ij}}, 1) \geq p_{ij}$ or, equivalently if $r_{ij} > 0$, at the minimal time step $t_2 \geq t_1$ with

$$\sum_{t=t_1}^{t_2} \min(R_i(t), r_{ij}) \geq r_{ij}p_{ij} =: \tilde{p}_{ij}. \tag{2.1}$$

This observation allows us to get rid of the resource aspect by considering *variable speed* processors instead of fixed speed processors. The speed of such variable speed processors can be changed at runtime[1]. For our reinterpretation, think of a job (i, j) to have size $\tilde{p}_{ij}$ and of a processor i to be of variable speed. The value $R_i(t)$ denotes the speed processor i is set to during time step t. The scheduler is in control of these processor speeds, but it must ensure that the aggregated speed of all processors does never exceed one. Moreover, in addition to the system's speed limit, each job (i, j) is annotated with the maximum speed r_{ij} it can utilize. In this light, our CRSHARING problem becomes a speed scaling problem to minimize the makespan in which the scheduler is limited by both the system's maximum aggregated speed and a per-job speed limit. The unit size restriction for the CRSHARING problem translates into the restriction that job sizes $\tilde{p}_{ij}$ equal the corresponding resource requirements r_{ij}. In other words, all jobs must be processable in one time step if run at maximum speed.

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

Observation 2.1. Any feasible schedule needs at least $\sum_{i=1}^{m}\sum_{j=1}^{n_i} r_{ij}p_{ij}$ time steps to finish a given set of jobs with resource requirements r_{ij} and sizes p_{ij}.

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

Additional Notation & Notions. The following additional notions and notation will turn out to be helpful in the analysis and discussion. For a processor i with n_i jobs, we define $n_i(t)$ as the number of unfinished jobs at the start of time step t. In particular, we have $n_i(1) = n_i$. The value $j_i(t) := n_i - n_i(t)$ denotes the number of jobs completed on machine i at the start of step t. A processor i is said to be *active* at time step t if $n_i(t) > 0$. Similarly, we say that job (i, j) is *active* at time step t if $j_i(t) = n_i - n_i(t) = j - 1$ (i.e., if processor i has finished exactly $j - 1$ jobs at the

[1]This is also known as *speed scaling* (cf. [YDS95]).

start of time step t). We use $M_j := \{ i \mid n_i \geq j \}$ to denote the set of all processors having at least j jobs to process. Finally, we define $n := \max_i n_i$ as the maximum number of jobs any processor has to process.

2.1.2 Contribution

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

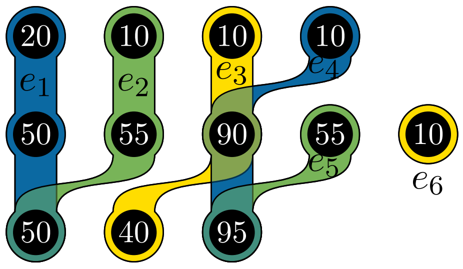
2.1.3 Graphical Representation

For the remainder of this chapter, we assume that all jobs have unit size. This section introduces a hypergraph notation for CRSHARING schedules.

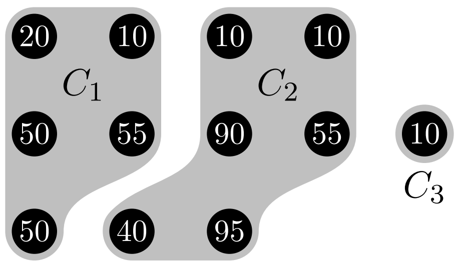
Given a problem instance of CRSHARING with unit size jobs and a corresponding schedule S, we can define a weighted hypergraph $H_S = (V, E)$ as follows: The nodes of H_S and their weights correspond to the jobs and their resource requirements, respectively. That is, the node set is given by $V = \{(i,j) | i = 1, 2, \ldots, m \wedge j = 1, 2, \ldots, n_i\}$, and the weight of a node $(i,j) \in V$ is r_{ij}. The edges of H_S correspond to the schedule's time steps and contain the currently active jobs. More formally, the edge $e_t \subseteq V$ for time step t is defined as $e_t := \{ (i,j) \mid n_i(t) > 0 \wedge j = n_i - n_i(t) + 1 \}$. Thus, if we abuse S to also denote the makespan of schedule S, the edge set of H_S can be written as $E = \{ e_1, e_2, \ldots, e_S \}$. We call H_S the *scheduling (hyper)graph* of S. See Figure 2.1a for an illustration.

Connected Components. In Section 2.1.4 and during the analysis in Section 2.4, we will see that the connected components formed by the edges of a scheduling graph H_S carry a lot of structural information about the schedule. To make use of this information, let us introduce some notation that allows us to directly argue via such components. We start with an observation that follows from the construction of H_S.

Observation 2.2. Consider a connected component $C \subseteq V$ of H_S and two time steps $t_1 \leq t_2$ with $e_{t_1} \cup e_{t_2} \subseteq C$. Then, for all $t \in \{ t_1, t_1 + 1, \ldots, t_2 \}$ we have $e_t \subseteq C$.



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



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

Figure 2.1: Hypergraph representation of a schedule for three processors. Resource requirements are given as node labels (in percent). Nodes are laid out such that each row corresponds to the job sequence of one processor (from left to right). Edges correspond to the schedule that prioritizes jobs in order of increasing remaining resource requirement.

Let N denote the total number of connected components and let C_k denote the k-th connected component (for $k \in \{1, 2, \dots, N\}$). Moreover, we use $\#_k$ to denote the number of edges of the k-th component. That is, we have $\#_k = |\{ e_t \in E \mid e_t \subseteq C_k \}|$. Observation 2.2 implies that a component C_k consists of $\#_k$ *consecutive* time steps. This allows us to order the components such that, for any two components k, k' and edges $e_t \subseteq C_k, e_{t'} \subseteq C_{k'}$ with $t \leq t'$, we have $k \leq k'$. That is, we can think of the components being processed by the processors from left to right. See Figure 2.1b for an illustration.

The maximal size of an edge in the k-th component, which equals the size of its first edge, gives us a rough estimate for the amount of potential parallelism available during the corresponding time steps. Note that while the size of edges e_t is monotonously decreasing in t, a schedule that tries to balance the number of remaining jobs on each processor will decrease the edge size only at the end of a component (for all components but the last one). We will make use of this fact in the proof of Lemma 2.21. For now, let us honor its foreshadowed importance by the following definition:

Definition 2.3 (Component Class)**.** Given a component C_k, we define its *class* q_k as the size of its first edge. That is, $q_k := |e_t|$ with $t = \min\{ t' \mid e_{t'} \subseteq C_k \}$.

Besides being an upper bound on the size of a component's edges, the class q_k is also decreasing in k. Moreover, Lemma 2.11 will show that a component's class allows us to formulate an important relation between its size and the total number of its edges.

2.1.4 Structural Properties

Let us use the introduced notions to point out some structural properties of schedules for the CRSHARING problem with unit size jobs. We start by defining three properties of schedules and show in Lemma 2.7 that we can restrict our analysis to schedules which have them.

Definition 2.4 (Non-wasting)**.** We call a schedule *non-wasting* if it finishes all active jobs during every time step t with $\sum_{i=1}^{m} R_i(t) < 1$.

Definition 2.5 (Progressive)**.** A schedule is *progressive* if, among all jobs that are assigned resources, at most one job is only partially processed during any time step t. More formally, we require that

$$|\{\, i \mid n_i(t) = n_i(t+1) \land R_i(t) > 0 \,\}| \leq 1 \tag{2.2}$$

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

Definition 2.6 (Nested)**.** Let $S(i,j)$ and $C(i,j)$ denote the starting step and the completion step of job (i,j), respectively. A schedule is *nested* if, at no time t, there are two jobs (i,j) and (i',j') such that $S(i,j) < S(i',j') \leq t < C(i',j')$, $S(i',j') < C(i,j)$ and (i,j) is running during step t.

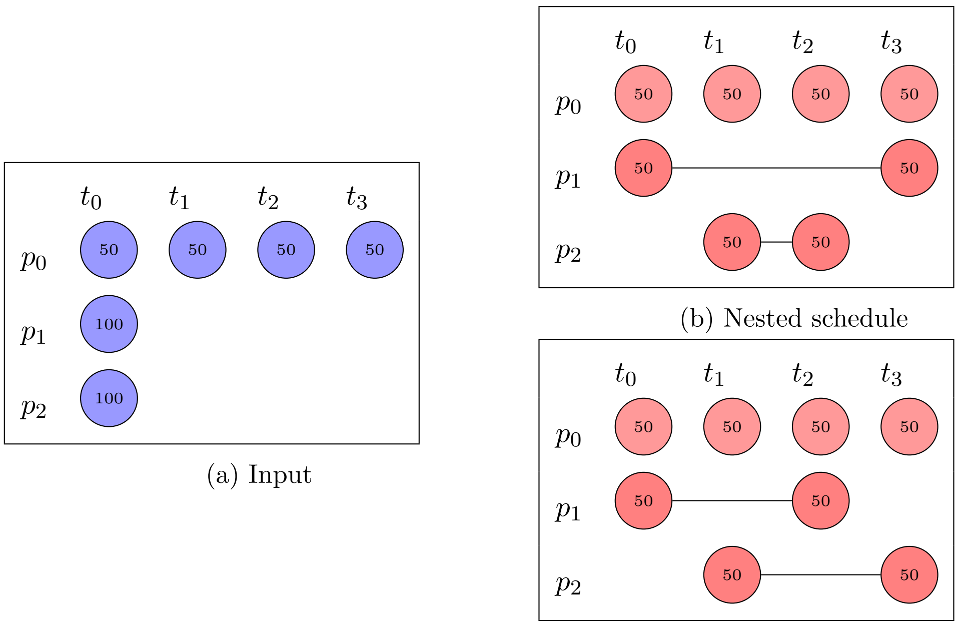
This last property intuitively means that among the partially processed jobs, we always prefer to run and complete the job that started at the latest step. Note that the condition of a nested schedule in particular implies that, for no jobs (i,j) and (i',j'), $S(i,j) < S(i',j') < C(i,j) < C(i',j')$. Otherwise we could choose $t = C(i,j)$ and job (i,j) would run in step $t = C(i,j)$. An example for a nested and an unnested schedule is given in Figure 2.2.

Lemma 2.7. *Every schedule S can be transformed into a schedule S' which is non-wasting, progressive and nested without increasing its makespan.*

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

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

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



(a) Input

(b) Nested schedule

(c) Unnested schedule

Figure 2.2: The schedules in Figure 2.2b and 2.2c are based on the input in Figure 2.2a and observe a resource limit of 100. Both schedules are non-wasting and progressive, but only the schedule in Figure 2.2b is nested. In the other schedule, p_1's job is already running when p_2's job is started, and completed before p_2's job is completed.

so that no new violating pair is created. In this way we can eliminate the violating pairs one by one.

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

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

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

Lemma 2.7 allows us to narrow our study to the subclass of non-wasting, progressive and nested schedules, and from now on we will assume any schedule to have these properties (if not stated otherwise).

Balanced Schedules. Intuitively, good schedules should try to balance the number of remaining jobs on each processor. This may provide the scheduler with more choices to prevent the underutilization of the resource later on (e.g., when only one processor with many jobs of low resource requirements remains). The better part of Section 2.4 serves the purpose of confirming this intuition. In the following, we formalize this property of balanced schedules and, subsequently, work out further formal and concise properties of balanced schedules.

Definition 2.8 (Balanced)**.** We say a schedule is *balanced* if, whenever a processor i finishes a job at a time step t, any processor i' with $n_{i'}(t) > n_i(t)$ also completes a job.

Proposition 2.9. *Every balanced schedule features the following properties:*

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

Proof. Both statements follow easily from the definition of balanced schedules. To see this, first note that both properties hold for $t = 1$, since $n_i(1) = n_i$ for all processors i. Moreover, at any time step t, the number $n_i(t)$ of remaining jobs cannot increase, and decreases by at most one during the current time step. Thus, it is sufficient to show that if one of the statements holds at some time step t with equality, it still holds at time step $t + 1$. For Property 1, $n_{i_1}(t) = n_{i_2}(t) - 1$ and the balance property imply that if i_1 finishes its job, then so must i_2. Thus, we have $n_{i_1}(t + 1) \geq n_{i_2}(t + 1) - 1$. The very same argument works for Property 2. □

Proposition 2.10. *Consider a balanced schedule and the set M_j of processors having at least j jobs. Let (i, j) be a job that is active at time step t and assume $n_i(t) > 1$ (i.e., it is not the last job on processor i). Then all processors $i' \in M_j$ are active at time step t.*

Proof. Let $i' \in M_j$ be a processor with at least j jobs and consider the case $n_{i'} \geq n_i$. By Proposition 2.9, Property 1, we have $n_{i'}(t) \geq n_i(t) - 1 > 0$, so processor i is active at time t. If $n_{i'} < n_i$, we can apply Proposition 2.9, Property 2 and get

$$n_{i'}(t) \geq n_{i'} - (n_i - n_i(t)) = n_{i'} - (j - 1) \geq 1. \tag{2.3}$$

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

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

Lemma 2.11. *Consider a non-wasting, progressive, and balanced schedule. The number of nodes and edges and the size of the first edge in a component are related via the following properties:*

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

Proof. The second statement follows immediately from Lemma 2.7, which (by the schedule being progressive) states that in each time step (i.e., for each edge) at least one job is finished.

For the first statement, fix a $k \in \{1, 2, \ldots, N-1\}$ and consider the first edge e_t of the component C_k. By definition, this edge consists of q_k different nodes. We now show that each of the remaining $\#_k - 1$ edges adds at least one new node to the component. So fix an edge $e_{t'} \subseteq C_k$ with $t' > t$ and consider the time step $t' - 1$. Since we know that at least one job is finished in every time step (Lemma 2.7) and that S is balanced, at least one of the processors having the maximal number of remaining jobs finishes its current job. More formally, there is some processor $i' = \arg\max_i n_i(t'-1)$ that finishes its currently active job at time step $t'-1$. Because of $k \neq N$, we also know that $n_{i'}(t'-1) > 1$, such that there is a new active job for processor i' at time step t'. This yields the lemma's first statement. □

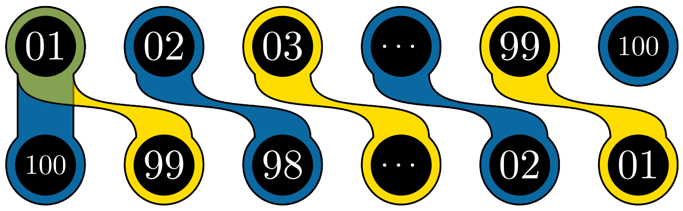
2.1.5 Warm-up: Round Robin Approximation

Consider the following simple round robin algorithm for the CRSharing problem (with unit size jobs): Given a problem instance where the maximal number of jobs on a processor is n, the algorithm operates in n phases. During phase j, it processes the j-th job on each processor, assigning the resource in an arbitrary way to any processors that have not yet finished their j-th job. Note that this algorithm may waste resources (although only between two phases) and is possibly non-progressive. Still, the following theorem shows that it results in schedules that are not too bad.

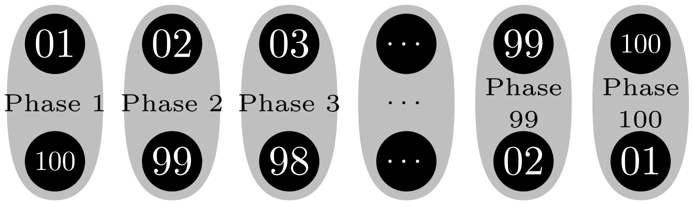
Theorem 2.12. *The* RoundRobin *algorithm for the* CRSharing *problem with unit job sizes has a worst-case approximation ratio of exactly* 2.

Proof. We start with the upper bound on the approximation ratio. RoundRobin algorithm needs exactly $\lceil \sum_{i \in M_j} r_{ij} \rceil$ time steps to finish the j-th phase (cf. "Alternative Model Interpretation" in Section 2.1.1). Thus, the makespan of a RoundRobin schedule can be bounded by

$$\sum_{j=1}^{n} \left\lceil \sum_{i \in M_j} r_{ij} \right\rceil \leq n + \sum_{j=1}^{n} \sum_{i \in M_j} r_{ij}. \tag{2.4}$$



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



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

Figure 2.3: Worst-case example for RoundRobin schedule. Node labels give the jobs' resource requirements in percent.

Since any processor can finish at most one job per time step, even an optimal schedule has a makespan of at least n. Observation 2.1 yields another lower bound on the optimal makespan, namely $\sum_{j=1}^{n} \sum_{i \in M_j} r_{ij}$. Together, we get that RoundRobin computes a 2-approximation.

For the lower bound on the approximation ratio, consider the following CRSharing problem instance with unit size jobs on two processors: Let $n \in \mathbb{N}, \varepsilon := 1/n > 0$ and define the resource requirements for the first processor as $r_{1j} := j \cdot \varepsilon$ for $j \in \{1, 2, \ldots, n\}$. For the second processor, we define $r_{2j} := (1+\varepsilon) - r_{1j}$. Note that each processor has to process n jobs. Figure 2.3 illustrates the instance as well as the resulting optimal and RoundRobin schedules for $n = 100$. An optimal schedule, shown in Figure 2.3a, will waste no resource at all. In contrast, the RoundRobin schedule, as indicated in Figure 2.3b, wastes a share of $1 - \varepsilon$ of the resource in every second time step. As a result, the RoundRobin schedule needs $2n$ time steps, while an optimal schedule can finish the same workload in $n+1$ time steps. Thus, for $n \to \infty$ we get an approximation ratio of 2. □

2.2 Problem Complexity

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

Theorem 2.13. CRSharing *with unit size jobs is NP-hard if the number of processors is part of the input.*

Proof. In the following, we prove the NP-hardness of the CRSharing problem with unit size jobs via a reduction from the Partition problem. Our reduction transforms a Partition instance of n elements into a CRSharing instance on n processors, each having three jobs to process.

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

Assume we are given a YES-instance of Partition and let, w.l.o.g., the first k elements form one partition. The schedule shown in Figure 2.4a is feasible and has makespan 4. Now assume we are given a NO-instance and an optimal schedule for the corresponding CRSharing instance. W.l.o.g., exactly the first k processors finish their jobs in the first time step. This implies $\sum_{i=1}^k \tilde{a}_i \leq 1$, yielding the inequality $\sum_{i=1}^k a_i \leq A + \delta < A + 1$. Since the given Partition instance is a NO-instance, we also have $\sum_{i=1}^k a_i \neq A$. Together this implies $\sum_{i=1}^k a_i \leq A - 1$, which, in turn, yields $\sum_{i=k+1}^n a_i \geq A + 1$. Since we have not yet finished the jobs $(k+1, 1), (k+2, 1), \dots, (n, 1)$, we need at least two more time steps until we can start working on $(k+1, 3), (k+2, 3), \dots, (n, 3)$. Their total resource requirement is at least

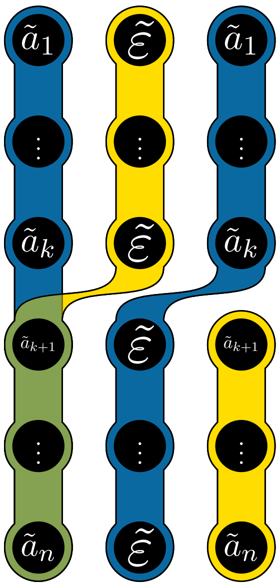
$$\sum_{i=k+1}^{n} \tilde{a}_i \frac{\sum_{i=k+1}^n a_i}{A+\delta} \geq \frac{A+1}{A+\delta} > 1. \tag{2.5}$$

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

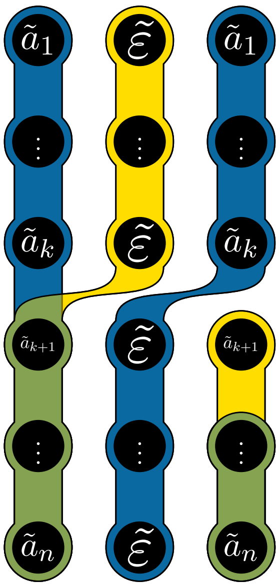
Note that we also get the following lower bound from the proof of Theorem 2.13:

Corollary 2.14. *It is NP-hard to approximate* CRSharing *with a factor better than* $5/4$.

While Theorem 2.13 proves NP-hardness of our problem, it leaves the question concerning the problem's complexity for *constant m*. In the next two sections we will show that in this case the problem is polynomial-time solvable.



(a) Optimum for YES-instances.



(b) Optimum for NO-instances.

Figure 2.4: Problem instance and schedules used for the reduction from PARTITION to CRSHARING with unit size jobs.

2.3 Optimal Algorithms

We start with a simple optimal algorithm for the variant with two processors in Section 2.3.1. In Section 2.3.2, we present a dynamic programming approach to solve the problem in polynomial time for an arbitrary number of processors.

2.3.1 Algorithm for Two Processors

While the previous section proves NP-hardness in the number of processors, there are exact polynomial-time algorithms for a fixed number of processors. Before we state and analyze the algorithm for arbitrary $m \geq 2$ in Section 2.3.2, we introduce a faster algorithm for two processors. Algorithm OptResAssignment traces out all reasonable scheduling decisions. To keep this approach feasible, we use Lemma 2.7 (implying the existence of an optimal schedule that finishes at least one job in each time step) and another structural property (see Lemma 2.15). These allow us to discard bad scheduling decisions early on.

Algorithm Description. The OptResAssignment algorithm uses a dynamic programming approach. To this end, it maintains a two-dimensional array B of size $n_1 \times n_2$. Each entry holds a tuple $B[i_1, i_2] = (r, t)$, which states that there is a schedule that, at time step t, has finished all jobs $(1, j_1)$ with $j_1 < i_1$ and $(2, j_2)$ with $j_2 < i_2$, and for which the remaining resource requirements of $(1, i_1)$ and $(2, i_2)$ sum up to r. OptResAssignment fills B in $n_1 + n_2 - 1$ phases, one phase for each diagonal of B. It maintains the invariant that, from the start of phase ℓ on, all entries on the $(\ell - 1)$-th diagonal (i.e., all $B[i_1, i_2]$ with $i_1 + i_2 = \ell$) are optimal. More precisely, such entries correspond to subschedules with minimal t (and, for this t, minimal r) reaching the jobs $(1, i_1)$ and $(2, i_2)$. See Listing 2.1 for the pseudocode. Note that in our algorithm description, we compute only the makespan (and not a corresponding schedule) of an optimal solution. However, given the array B, one can easily trace back the final entry and derive an explicit schedule in linear time.

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

Lemma 2.15. *Consider two non-wasting and progressive schedules S and S' as well as a time step t such that $n_i(t) \leq n'_i(t)$ for $i \in \{1, 2\}$. Let $v_i(t)$ and $v'_i(t)$ be the remaining resource requirement of the job that is active at time t on processor $i \in \{1, 2\}$ in schedule S and S', respectively. If*

1. *$n_1(t) < n'_1(t)$ or $n_2(t) < n'_2(t)$, or*

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

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

```
[resource requirements are stored in $A_1$ and $A_2$]
[subschedules are stored in two-dimensional array $B$]
[extend $A_1$ as well as $A_2$ by an extra 0-entry]
$n_1 = \text{length}(A_1)$; $n_2 = \text{length}(A_2)$;
initialize array $B[1 \dots n_1, 1 \dots n_2]$ with null entries
$B[1,1] = \big(A_1[1] + A_2[1], 0\big)$
for $\ell$ from 2 to $n_1 + n_2 - 1$
  for $i_1$ from $\max\{1, \ell - n_2\}$ to $\min\{\ell - 1, n_1\}$
    $i_2 = \ell - i_1$
    $(r, t) = B[i_1, i_2]$
    if $i_1 = n_1$
      $add\big(i_1, i_2 + 1, 0, A_2[i_2 + 1], t + 1\big)$
    else if $i_2 = n_2$
      $add\big(i_1 + 1, i_2, A_1[i_1 + 1], 0, t + 1\big)$
    else if $r \le 1$
      $add\big(i_1 + 1, i_2 + 1, A_1[i_1 + 1], A_2[i_2 + 1], t + 1\big)$
      $add\big(i_1, i_2 + 1, 0, A_2[i_2 + 1], t + 1\big)$
      $add\big(i_1 + 1, i_2, A_1[i_1 + 1], 0, t + 1\big)$
    else
      $add\big(i_1, i_2 + 1, A_1[i_1] + A_2[i_2] - 1, A_2[i_2 + 1], t + 1\big)$
      $add\big(i_1 + 1, i_2, A_1[i_1 + 1], A_1[i_1] + A_2[i_2] - 1, t + 1\big)$
min = $B[n_1, n_2]$

function $add(i_1, i_2, v_1, v_2, t)\}$
$r = v_1 + v_2$
$(r_{old}, t_{old}) = B[i_1, i_2]$
if $(r_{old}, t_{old}) = null \lor t < t_{old} \lor (t = t_{old} \land r < r_{old})$
  $B[i_1, i_2] = (r, t)$
```

Listing 2.1: Algorithm OptResAssignment computes an optimal solution for the two processor case in a runtime of $\mathrm{O}\left(n^2\right)$.

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

1. Without loss of generality, assume $n_1(t) < n'_1(t)$. If S' finishes only one job, S can complete a job on the same processor and hence maintains the inequalities. If S' finishes both jobs, this yields $n'_i(t+1) = n'_i(t) - 1$ for $i \in \{1, 2\}$. Thus, if S finishes a job on processor 2 and assigns the remaining resource to the job on processor 1, this results in $n_1(t+1) = n_1(t) \le n'_1(t+1)$ and $n_2(t+1) = n_2(t) - 1 \le n'_2(t+1)$. If equality applies (otherwise Property 1 holds), then the same jobs are active at time $t+1$ in S' and S, say j_1 and j_2. This yields $v_1(t+1) + v_2(t+1) \le r_{1j_1} + r_{2j_2} = v'_1(t+1) + v'_2(t+1)$, therefore Property 2 applies.

2. Now suppose $v_1(t) + v_2(t) \le v'_1(t) + v'_2(t)$. If S' finishes both jobs, S can do the same and Property 2 holds with equality. If S' only finishes one job (w.l.o.g., job $j - 1$ on processor 1), S can also finish that job. If $v_1(t) + v_2(t) \le 1$, it also completes a second job and therefore Property 1 applies. On the other hand, if $v_1(t) + v_2(t) > 1$, this results in $v_1(t + 1) + v_2(t + 1) = r_{1j} + (v_1(t) + v_2(t) - 1) \le r_{1j} + (v'_1(t) + v'_2(t) - 1) = v'_1(t + 1) + v'_2(t + 1)$, thus Property 2 applies.

□

Theorem 2.16. *Consider a* CRSharing *instance with unit size jobs and two processors. The following statements hold:*

1. OptResAssignment *computes an optimal solution.*
2. OptResAssignment *has runtime* $\mathrm{O}(n^2)$.

Proof. The correctness of Statement 2 is immediate, as OptResAssignment runs in $\mathrm{O}(n)$ phases and each phase considers the $\mathrm{O}(n)$ entries on the corresponding diagonal. It remains to prove the correctness of Statement 1.

Remember the invariant from the algorithm description: At the beginning of phase ℓ, for each entry $B[i_1, i_2] = (r, t)$ on the $(\ell - 1)$-th diagonal the following holds: t is the earliest time at which all jobs preceding $(1, i_1)$ and $(2, i_2)$ can be finished and r is, for this t, the smallest possible sum of the remaining resource requirements of $(1, i_1)$ and $(2, i_2)$. If this invariant holds for phase $n_1 + n_2$, the correctness follows immediately (we use dummy jobs, so the last diagonal entry corresponds to all non-dummy jobs being fully processed). For the first phase, the invariant's correctness is obvious from the initialization, as there are no jobs preceding $(1, 1)$ and $(2, 1)$. Now assume the invariant holds for the first ℓ phases and consider an entry $B[i_1, i_2]$ processed in the $(\ell + 1)$-th phase. This entry corresponds to a subschedule that has processed all jobs preceding $(1, i_1)$ and $(2, i_2)$. Since each processor can finish at most one job in one time step, this subschedule must originate from one of the subschedules S_1, S_2, or S_3 that have finished all jobs preceding (i) $(1, i_1 - 1)$ and $(2, i_2)$, (ii) $(1, i_1)$ and $(2, i_2 - 1)$, and (iii) $(1, i_1 - 1)$ and $(2, i_2 - 1)$, respectively. By our induction hypothesis, the entries in $B[i_1 - 1, i_2]$, $B[i_1, i_2 - 1]$, and $B[i_1 - 1, i_2 - 1]$ correspond to the best possible such schedules. Since the algorithm uses these to compute $B[i_1, i_2]$ (Lines 9 to 21) and the best of them is chosen as predecessor (Line 27, correct by Lemma 2.15), the invariant is established for entry $B[i_1, i_2]$ (and, similarly, for all remaining entries on the same diagonal). □

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

2.3.2 Algorithm for m Processors

While in the previous section we discussed OptResAssignment, an exact algorithm for $m = 2$ having a worst case runtime of $\mathrm{O}\left(n^2\right)$, this section shows that there is even a polynomial-time algorithm for any fixed m; we call it OptResAssignment2. In the proof we will restrict the schedules to nested ones (see Definition 2.6) and use the new notion of an (extended) configuration representing the current state of a schedule. We argue that only a polynomial number of extended configurations has to be considered and show that this implies a polynomial runtime.

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

Definition 2.17 ((Extended) configuration; core; support)**.** A *configuration* γ is a vector $(t, j_1(t), \dots, j_m(t), v_1(t), \dots, v_m(t))$ where $j_i(t) \in \{0, \dots, n_i\}$ and $v_i(t) \in [0,1]$. The *core* of γ is defined as $\operatorname{core}(\gamma) = (j_1(t), \dots, j_m(t))$ and its *support* as $\operatorname{supp}(\gamma) = \{ i \mid v_i(t) > 0 \}$. Further, we define the *extended configuration* of γ as the tuple $E(\gamma) := (\gamma, (i, \gamma_i)_{i \in \operatorname{supp}(\gamma)})$, where γ_i is the configuration after the time step in which processor i received resource for the last time.

We say two configurations are *step-equal* if they are in the same time step and if their corresponding cores are equal. Two extended configurations $E(\gamma) = (\gamma, (i, \gamma_i)_{i \in \operatorname{supp}(\gamma)})$ and $E(\gamma') = (\gamma', (i, \gamma_i')_{i \in \operatorname{supp}(\gamma')})$ are *step-equal* if 1) γ and γ' are step-equal, 2) they have the same support and 3) γ_i and γ_i' are step-equal for all $i \in \operatorname{supp}(\gamma)$.

In order to obtain a polynomial-time algorithm, we reduce the number of relevant configurations to a polynomial number. Obviously, if both configurations $(t, j_1(t), \dots, j_m(t), v_1(t), \dots, v_m(t))$ and $(t', j_1'(t'), \dots, j_m'(t'), v_1'(t'), \dots, v_m'(t'))$ are feasible with $t \le t'$, $j_\ell(t) \ge j_\ell'(t')$ and $v_\ell(t) \ge v_\ell'(t')$ for all $1 \le \ell \le m$, we do not need the second configuration, as the first one is always to be preferred. We say that the first configuration *dominates* the second one. The following lemma proves a natural connection between this property of domination and step-equal configurations.

Lemma 2.18. *If two extended configurations are step-equal, then one dominates the other.*

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

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

Now consider any two non-dominated and step-equal extended configurations $(\gamma, (i, \gamma_i)_{i \in \operatorname{supp}(\gamma)})$ and $(\gamma', (i, \gamma_i')_{i \in \operatorname{supp}(\gamma')})$ with $|\operatorname{supp}(\gamma)| = |\operatorname{supp}(\gamma')| \ge 2$. For all $i \in \operatorname{supp}(\gamma)$, denote by t_i the time step of γ_i (and γ_i'), and let k such that

```
$\mathcal{C}_1 = \{ (1,0,\dots,0,0,\dots,0) \}$
for $t$ from 2 to $\infty$
  $\mathcal{C}_t := \emptyset$
  for all $\gamma \in \mathcal{C}_{t-1}$
    $succ(\gamma)$ = successors of $\gamma$
    store link between $\gamma$ and each $\gamma' \in succ(\gamma)$
    $\mathcal{C}_t = \mathcal{C}_t \cup succ(\gamma)$
  if $(t, n_1+1, \dots, n_m+1, 0, \dots, 0) \in \mathcal{C}_t$
    output path to this configuration
    break
  for all $\gamma \in \mathcal{C}_t$
    for all $\gamma' \in \mathcal{C}_t \setminus \{ \gamma \}$
      if $\gamma$ dominates $\gamma'$
        remove $\gamma'$ from $\mathcal{C}_t$
```

Listing 2.2: Algorithm OptResAssignment2 computes an optimal solution for the case with a constant number of processors in polynomial time.

$t_k = \max \{ t_i \mid i \in \operatorname{supp}(\gamma) \}$. (Note that the t_i are pairwise distinct because there is at most one partly processed job in each time step.)

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

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

Algorithm. In order to find an optimal schedule, our algorithm OptResAssignment2 (Listing 2.2) enumerates all configurations that are not dominated by another configuration. Starting from the initial configuration $(1, 0, \dots, 0, 0, \dots, 0)$, it computes the configurations of the next time step on the basis of the configurations of the current time step. While doing this, it makes sure that the respective schedules remain non-wasting, progressive, and nested. In each time step, it additionally removes all dominated configurations by a pairwise comparison of the new configurations. When the algorithm hits an end configuration, it outputs the path to it and stops.

Theorem 2.19. OptResAssignment2 *computes an optimal schedule in time polynomial in n.*

Proof. In each pass of the outer for-loop, OptResAssignment2 creates all subschedules of t steps which are non-wasting, progressive and nested and whose current configuration is not dominated by another one. As soon as a final configuration is reached, the algorithm outputs the results and stops. Therefore, the correctness of the algorithm follows from Lemma 2.7 which states that there is at least one optimal schedule among all non-wasting, progressive and nested schedules.

In order to show the runtime, we will roughly bound the number of configurations that are computed by the algorithm: the non-dominated ones as well as the dominated ones (that are discarded right away). From Lemma 2.18 we know that there is exactly one configuration that dominates all the other step-equal configurations.

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

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

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

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

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

□

2.4 Balanced Schedules

This section builds up to our last result, an approximation algorithm with a tight approximation ratio of $2 - 1/m$, in Theorem 2.22. While the quality of the result is obviously worse compared to OptResAssignment2, it can be achieved by running a simple linear-time algorithm called GreedyBalance. We start by providing two lower bounds for optimal schedules in terms of a given non-wasting and balanced schedule, respectively.

2.4.1 Lower Bounds for Optimal Schedules

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

Lemma 2.20. *Let* OPT *denote the minimal makespan of a given problem instance and consider the scheduling graph H_S of a non-wasting schedule S. Then* OPT *can be bounded by*

$$\mathrm{OPT} \geq \sum_{k=1}^{N} (\#_k - 1). \tag{2.6}$$

Proof. From Observation 2.1, we immediately get that $\mathrm{OPT} \geq \sum_{i=1}^{m} \sum_{j=1}^{n_i} r_{ij}$. Consider a connected component C_k of our schedule containing the edges $t_1, t_1 + 1, \ldots, t_2$. Since S is non-wasting, $\sum_{i=1}^{m} R_i(t) = 1$ holds for all time steps $t \in \{\, t_1, t_1 + 1, \ldots, t_2 - 1 \,\}$. If there were such a t with $\sum_{i=1}^{m} R_i(t) < 1$, the non-wasting property would imply that all active jobs are finished. But then the edge e_{t+1} would not be part of C_k, yielding a contradiction. For the last time step t_2 of C_k we have $\sum_{i=1}^{m} R_i(t_2) \geq 0$. Since S is feasible and, w.l.o.g., does not use more of the resource than necessary, it follows that $\sum_{t=1}^{S} \sum_{i=1}^{m} R_i(t) = \sum_{i=1}^{m} \sum_{j=1}^{n_i} r_{ij}$. Let $e^{(k)}$ denote the last edge of C_k. Then we get:

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

□

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

Lemma 2.21. *Let* OPT *denote the minimal makespan of a given problem instance and remember that n denotes the maximum number of jobs any processor has to process. Given a balanced schedule S and its scheduling graph,* OPT *and n can be bounded by the inequalities*

$$\mathrm{OPT} \geq n \geq \sum_{k=1}^{N-1} \frac{|C_k|}{q_k} + \frac{|C_N|}{m}. \tag{2.7}$$

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

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

It remains to show that we have

$$\sum_{(i,j)\in C_k} \frac{1}{|M_j|} \geq \frac{|C_k|}{q_k} \tag{2.8}$$

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

Case 1: $n_{i_0}(t_0) > 1$
By applying Proposition 2.10, we get that all processors $i \in M_{j_0}$ are active at time step t_0. This yields $|M_{j_0}| \leq |e_{t_0}| = q_k$. Moreover, for a job $(i,j) \in C_k$, the minimality of j_0 gives us $|M_{j_0}| \geq |M_j|$. Combining both inequalities implies $|M_j| \leq q_k$. Applying this to the first part of Equation (2.8) eventually yields the desired inequality.

Case 2: $n_{i_0}(t_0) = 1$
In this case, (i_0, j_0) is the last job on processor i_0 at time step t_0. However, for any job $(i,j) \in C_k \setminus e_{t_0}$ we have $n_i(t_0) > 1$. Given such a job, let (i, j') be the job processed on i at time step t_0. Note that we have $j' < j$ and, thus, $M_j \subseteq M_{j'}$. By applying Proposition 2.10, we get that all $i' \in M_{j'}$ are active at time step t_0. Together with $M_j \subseteq M_{j'}$, this yields $|M_j| \leq q_k$. Thus, to prove Equation (2.8), it

only remains to show $\sum_{(i,j)\in e_{t_0}} {}^{1}\!/\!{}_{|M_j|} \geq \sum_{(i,j)\in e_{t_0}} {}^{1}\!/\!{}_{q_k} (= 1)$.

To this end, note that since C_k is not the last component, there exists at least one job $(i_1, j_1) \in e_{t_0}$ with $n_{i_1}(t_0) > 1$. Let this job be such that j_1 is minimal. Once more, by applying Proposition 2.10 we get that all $i \in M_{j_1}$ are active at time step t_0. Consider a job $(i,j) \in e_{t_0}$ with $i \in M_{j_1}$. If it is the last job on i (i.e., if $n_i(t_0) = 1$), we have $j = n_i$. Together with the definition of M_{j_1} we get $j = n_i \geq j_1$, yielding $|M_j| \leq |M_{j_1}|$. Similarly, if it is not the last job on i (i.e., if $n_i(t_0) > 1$), the minimality of j_1 gives us $|M_j| \leq |M_{j_1}|$. This yields the desired inequality as follows:

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

□

2.4.2 Deriving a $(2 - {}^{1}\!/\!{}_{m})$-Approximation

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

Theorem 2.22. *Consider a* CRSHARING *instance with unit size jobs and a feasible schedule S for it that is non-wasting, progressive, and balanced. Then S is a $(2 - {}^{1}\!/\!{}_{m})$-approximation with respect to the optimal makespan.*

Proof. In the following, let $\#_\varnothing := \sum_{k=1}^{N} {}^{\#_k}\!/\!{}_{N}$ denote the average number of edges in a component. Our proof uses two bounds on the approximation ratio. The first one follows easily from Lemma 2.20 and leads to a better approximation for instances with large $\#_\varnothing$. The second bound is much more involved and mainly based on Lemma 2.21. It yields a better approximation for instances with small $\#_\varnothing$. To get the first bound, we simply apply Lemma 2.20 and get

$$\frac{S}{\mathrm{OPT}} \leq \frac{\sum_{k=1}^{N} \#_k}{\sum_{k=1}^{N}(\#_k - 1)} = \frac{\#_\varnothing}{\#_\varnothing - 1}. \tag{2.9}$$

Let us now consider the second bound, based on Lemma 2.21. Our goal is to show that the inequality

$$\frac{S}{\mathrm{OPT}} \leq \frac{m \cdot \#_\varnothing}{\#_\varnothing + m - 1} \tag{2.10}$$

holds. Once this is proven, we can combine both bounds by realizing that the bound from Equation (2.9) is monotonously decreasing in $\#_\varnothing$ and the bound from Equation (2.10) is monotonously increasing in $\#_\varnothing$. Equalizing yields that their minimum's maximum is obtained at $\#_\varnothing = \frac{2m-1}{m-1}$, which results in an approximation ratio of $2 - {}^{1}\!/\!{}_{m}$.

The rest of this proof is geared towards proving Equation (2.10). We distinguish two cases. The first case covers the easier part, where we have $\mathrm{OPT} \geq n + 1$. That is, even an optimal solution cannot finish the jobs in n time steps. The second case, where we have $\mathrm{OPT} = n$, turns out to be more difficult to prove. While we can apply a similar analysis, we have to take more care when bounding our algorithm's progress in the first two time steps.

Case 1: $\mathrm{OPT} \geq n + 1$
Applying Lemma 2.21 to this case yields

$$\begin{aligned}\frac{S}{\mathrm{OPT}} &\leq \frac{\sum_{k=1}^{N} \#_k}{\sum_{k=1}^{N-1} \frac{|C_k|}{q_k} + \frac{|C_N|}{m} + 1} \\ &\leq \frac{N \cdot \#_\varnothing}{\sum_{k=1}^{N-1} \frac{\#_k + q_k - 1}{q_k} + \frac{\#_N + m - 1}{m}} \\ &\leq \frac{N \cdot \#_\varnothing}{\sum_{k=1}^{N} \frac{\#_k + m - 1}{m}} \leq \frac{m \cdot \#_\varnothing}{\#_\varnothing + m - 1}.\end{aligned} \tag{2.11}$$

Case 2: $\mathrm{OPT} = n$
If we apply the same analysis as in the first case, we will fall short of our desired approximation ratio. Surprisingly, it turns out to be sufficient to bound only the first two time steps more carefully. The idea of the following analysis is to consider the first two time steps of S and the remaining part of S separately. To this end, first note that we can assume, w.l.o.g., that $\#_1 > 1$ (i.e., the first two time steps belong to the same component). If this is not the case, our algorithm finishes all active jobs in the first time step and, thus, behaves optimally[2]. Consider the remaining jobs/workloads after the first two time steps. We can regard this as a subinstance of our original problem instance. Let S' denote the subschedule that results from restricting S to time steps $t \geq 3$. We use N', $\#'_k$, q'_k, and n' to refer to the corresponding properties of its scheduling graph $H_{S'}$. Note that we have $N' \geq N - 1$ (because of our assumption $\#_1 > 1$) as well as $N' \cdot \#'_\varnothing = N \cdot \#_\varnothing - 2$ (since exactly two time steps are missing in the subschedule). Moreover, we also have $n' = n - 2$. The inequality $n' \geq n - 2$ is obvious. For $n' \leq n - 2$, note that OPT must finish the jobs in the set $\{\, (i, 1) \mid n_i(1) \geq n - 1 \,\} \cup \{\, (i, 2) \mid n_i(1) \geq n \,\}$ during the first two time steps. Thus, the total resource requirement of these jobs is at most two. Since S is balanced, it will prioritize and, thus, finish these jobs in the first two time steps. Finally, we can bound our approximation ratio as follows (the first inequality applies Lemma 2.21 to S'):

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

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

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

This proves that Equation (2.10) also holds in this case.

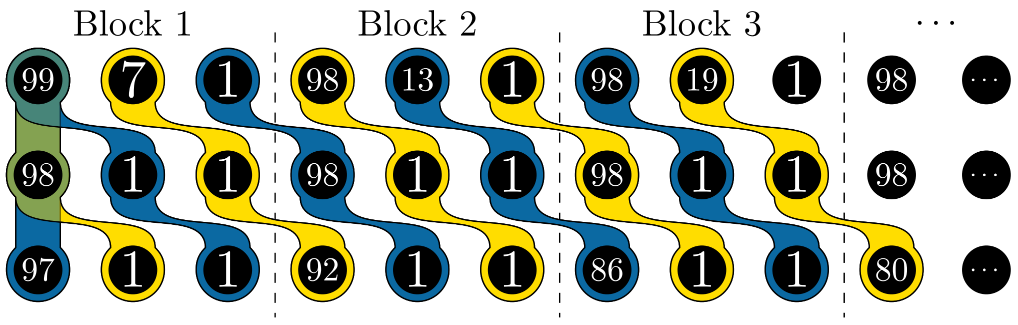
□

2.4.3 Tight Approximation Algorithm

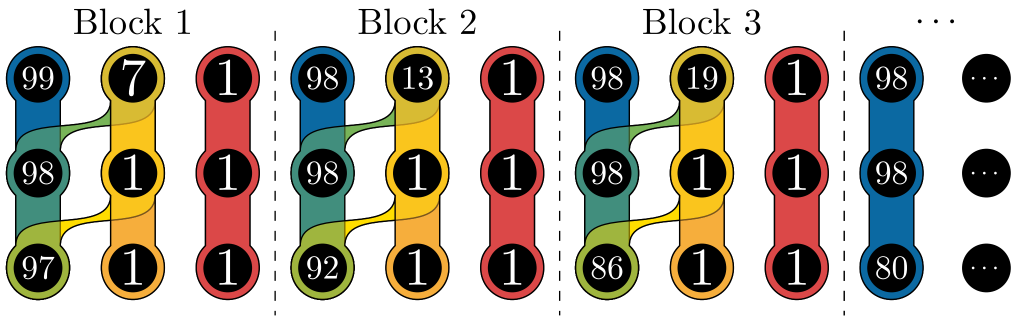
So far, we analyzed the quality of balanced schedules in general, but did not yet provide a concrete example of a corresponding algorithm. One of the most natural greedy algorithms schedules jobs by prioritizing processors with a higher number of remaining jobs and, in the case of a tie, by prioritizing jobs with larger remaining resource requirements. We name this algorithm GreedyBalance. In Section 2.4.2, we saw that balanced schedules and, as a consequence, the algorithm GreedyBalance yield a $(2 - 1/m)$-approximation for the CRSharing problem. Now we show that this approximation ratio is tight for GreedyBalance.

Theorem 2.23. *The* GreedyBalance *algorithm for the* CRSharing *problem with jobs of unit size has a worst-case approximation ratio of exactly* $2 - 1/m$.

Proof. Since GreedyBalance computes only balanced schedules, the upper bound follows immediately from Theorem 2.22. For the lower bound, consider a family of problem instances defined as follows: We define blocks of $m \times m$ jobs with resource requirements as described below. For the first block, let $r_{i1} := 1 - i \cdot \varepsilon$ for $i \in \{1, 2, \dots, m\}$, $r_{12} := 1 - \sum_{i=1}^{m}(1 - r_{i1}) + \varepsilon$, and $r_{i2} := \varepsilon$ for $i \in \{2, 3, \dots, m\}$. Moreover, define $r_{ij} := \varepsilon$ for all $i \in \{1, 2, \dots, m\}$ and $j \in \{3, 4, \dots, m\}$. This finishes the first $m \times m$-block of jobs. Having constructed the l-th block, we construct the next block, starting with its first column $j := l \cdot m + 1$. We define $r_{ij} := 1 - (m-1)\varepsilon$ for $i \in \{1, 2, \dots, m-1\}$ and $r_{mj} := 1 - \sum_{i'=1}^{m-1} r_{m-i', j-i'}$. For the second column of this block we set $r_{1,j+1} := 1 - \sum_{i=1}^{m}(1 - r_{ij}) + \varepsilon$, and $r_{i,j+1} := \varepsilon$ for $i \in \{2, 3, \dots, m\}$. To finish the block, we set $r_{ij'} := \varepsilon$ for all $i \in \{1, 2, \dots, m\}$ and $j' \in \{j+2, j+3, \dots, j+m-1\}$. We finish the construction once the next block contains jobs with negative resource requirements. Note that by choosing ε small enough, we can make this construction arbitrarily long. See Figure 2.5 for an illustration of this construction and the schedules produced by GreedyBalance and an optimal algorithm. Our construction is such that GreedyBalance needs exactly $2m - 1$ time steps per block: By balancing the number of remaining jobs, it is forced to work m time steps on a block's first column (which contains a total resource requirement of roughly m) before it can finish the remaining $m - 1$ columns of a block. In contrast, the optimal algorithm ignores any balancing issues, which allows it to exploit that all diagonals have a total resource requirement of 1. □



(a) An optimal schedule.



(b) Schedule computed by GreedyBalance.

Figure 2.5: Construction and schedules used in the proof of Theorem 2.23 for $m = 3$ and $\varepsilon = 0.01$. Node labels show the corresponding job's resource requirement in percent (e.g., $r_{12} = 0.07$). Note that the optimal schedule needs (essentially) m time steps to finish a block, while S needs $2m - 1$ time steps per block.

CHAPTER 3

Multiprocessor Scheduling with a Sharable Communication Channel

In "On-The-Fly Computing" [Hap+13; 17], one main idea is that future software-based IT services are automatically composed from base services traded on global markets. Thereby, the functionality of a service is provided by the interaction of smaller pieces of software resulting in the exchange of data during the execution. This strengthens the necessity of taking into account communication when designing scheduling algorithms that enable efficient execution of such software. It might even shift the focus from processing times to planning communication, particularly if the exchange of data rather than actual computations becomes the major bottleneck in a system.

These observations lead to a new scheduling problem that we study in this chapter. We are given a communication graph, where each connected component describes a service composed of jobs (base services) by identifying nodes with jobs and using weighted edges to model the required interjob communication of jobs. These edge weights can, for instance, be thought of as communication volume in bytes. Also, we are given a system comprised of m parallel, identical processors connected by a shared communication channel (e.g., a data bus) enabling communication between the processors and hence between jobs processed in parallel. Given that the available communication channel constitutes a scarce resource with bounded capacity (e.g., available data rate in bytes per second), a fundamental question arising in this setting is: how to assign jobs to processors and share the channel among them in order to minimize the time at which all jobs with their related communication demands are done and hence, to minimize the time until all services are completed.

We model this scheduling problem as a novel bin packing variant and propose and analyze approximation algorithms. In the following, we give a formal description of the studied problem. In Section 3.2, we study the computational hardness showing that the considered problem is (for the most relevant cases) NP-hard in the strong sense, even for a constant number of processors and a single path or a forest with arbitrary constant maximum degree. Consequently, we then focus on approximation algorithms and start by considering a simple NextFit strategy for graphs of degree two in Section 3.3. For trees and general graphs with arbitrary degree we provide a more complex approximation algorithm in Section 3.4. For an overview of our results see also Section 3.1.2.

3.1 Preliminaries

In the following we introduce the model and the notation used throughout the paper.

3.1.1 Model & Notation

We consider the following scheduling problem called Sic. Given a set of tasks $\{T_1, T_2, \dots, T_p\}$, each described by a connected, undirected graph $G_i = (V_i, E_i)$ on a set V_i of jobs together with a weight function $w : E_i \to]0, \infty[$. Each edge $\{u, v\} \in E_i$ represents the communication requirement (or communication demand) between jobs u and v. Additionally, we are given a set of m identical, parallel processors connected by a shared communication channel with capacity $C > 0$. Each processor can process at most one job per (discrete) time step while a job can be processed in several (not necessarily contiguous) time steps. Two jobs can communicate only when they are executed in parallel. Hence, in any time step t, at most m jobs can be processed and, additionally, a scheduler has to define how much capacity of the communication channel is allocated to pairs of jobs processed in t. Thereby the channel may not be overused, i.e., a capacity of at most C may be allocated to jobs per time step. As soon as for a pair of jobs with strictly positive communication demand the accumulated share of the channel it was assigned over time is at least its requirement $w(e)$, we call this edge to be completed. The objective is to find a schedule that minimizes makespan, i.e., the time until the last edge is completed.

Formally, the scheduling problem is defined as an equivalent bin packing formulation: Let $G = (\bigcup_i V_i, \bigcup_i E_i)$ be the (in general unconnected) *communication graph* consisting of the graphs G_i. In the bin packing formulation, each edge $e \in E$ corresponds to an item e with size $r_e := w(e)$. The goal is to pack all items into as few bins with capacity C as possible while allowing items to be arbitrarily split into parts and subject to the following constraints:

1. *Capacity Constraint:* Each bin may contain (parts of) items of an overall size of at most C,

2. *Edge Constraint:* Each bin may contain (parts of) items incident to at most m nodes in the underlying graph G.

In the rest of this chapter, we assume without loss of generality that $C = 1$. Observe that in terms of the original scheduling formulation each part of an item corresponds to one time step in which the corresponding jobs are scheduled and its item size represents the channel capacity assigned to this pair of jobs in this time step. The edge constraint respresents the fact that only m processors are available while the capacity constraint represents the available channel capacity. The number of bins then coincides with the number of time steps required to finish all tasks.

Since in the next section we show that Sic is NP-hard in general, even for a constant number of $m \geq 4$ processors and when G is a single tree, we focus on designing approximation algorithms. Remember that a polynomial-time algorithm A is called to have an *(absolute) approximation ratio* of α if, on any instance I, it holds $\frac{A(I)}{\mathrm{OPT}(I)} \leq \alpha$, where $A(I)$ and $\mathrm{OPT}(I)$ denote the number of bins used on instance I by algorithm A and by an optimal solution, respectively. A has an *asymptotic ratio* of α if $R^\infty \leq \alpha$, where $R^\infty := \lim_{k\to\infty} \sup_I \left\{ \frac{A(I)}{\mathrm{OPT}(I)} : \mathrm{OPT}(I) = k \right\}$.

A further notion we need in the following is the *arboricity* of a graph $\tilde{G} = (\tilde{V}, \tilde{E})$, denoted $\mathrm{arb}(\tilde{G})$. Let $X \subseteq \tilde{V}$ be a (sub-)set of nodes and $\tilde{E}_X \subseteq \tilde{E}$ be the set of edges induced by X. The arboricity is defined as $\mathrm{arb}(\tilde{G}) := \left\lceil \max_{X \subseteq \tilde{V}, |X| \geq 2} \frac{|\tilde{E}_X|}{|X|-1} \right\rceil$ and describes the minimum number of forests needed to cover the entire graph $\tilde{G}$. It is known that $\mathrm{arb}(\tilde{G})$ can be computed in polynomial time [GW92]. Furthermore, it is possible to compute a decomposition of $\tilde{G}$ into at most $\mathrm{arb}(\tilde{G})$ many forests and an additional graph of degree at most two in polynomial time (by applying a result from [KL11] with $\varepsilon = \frac{1}{\mathrm{arb}(\tilde{G})+2}$).

3.1.2 Contribution

We thoroughly study the complexity of Sic depending on the parameter m, the degree d of G and further structural properties of G. An overview of these results is given in Figure 3.1. For $m > 3$ the NP-hardness holds even if G is a single path, the most simple structure G can have.

	Constant Degree	**Variable Degree**
$m = 2$	Trivially in P	
$m = 3$	Forest: Exact Algorithm (Theorem 3.4) General Graph: NP-hard (Lemma 3.5)	NP-hard (Proposition 3.1)
$m > 3$	NP-hard (Section 3.2.2)	

Figure 3.1: Complexity results for different values of m and degree d of graph G.

We further present approximation algorithms for the cases where Sic is NP-hard. For G being a graph with maximum degree $d = 2$, we show that a simple NextFit strategy achieves an approximation ratio of $\frac{7}{3} + \frac{5}{6(m-1)}$ in Section 3.3. When G

becomes more complex but still is acyclic so that it is built by a set of trees (forest), we can asymptotically approximate an optimal packing by a ratio of $\min\{1.8, \frac{1.5m}{m-1}\} + 1$ as shown in Section 3.4.1. On the basis of our algorithm for forests, we then show how to handle arbitrary graphs G in Section 3.4.2. We assume that $\mathrm{arb}(G)/\mathrm{OPT} = o(1)$ and show an approximation ratio of $\min\{1.8, 1.5 \cdot {}^{m}/_{(m-1)}\} \cdot (\mathrm{arb}(G) + {}^{5}/_{3})$ for this general case. If one, however, wants to drop the aforementioned assumption, the approximation ratio only worsens by an additional summand of $2.5\mathrm{arb}(G)/\mathrm{OPT}$.

3.2 Complexity

First, note that our problem is trivial to solve for $m = 2$. In this case, each item has to be packed alone, hence packing all items into distinct bins is optimal.

For larger values of m, as a first observation and a direct corollary from NP-hardness of cardinality constrained bin packing with splittable items [Chu+06], which, for a cardinality constraint set to $m - 1$, is equivalent to Sic when the communication graph forms a star, we have the following proposition.

Proposition 3.1. *The* Sic *problem is strongly NP-hard for constant $m \geq 3$ processors and G being a single tree with degree d when d is part of the input.*

Despite this hardness result, it is interesting to study the question of whether the complexity changes when the degree d of the communication graph is fixed. We will see that the problem is in P for $m = 3$ when G is a forest, but not for arbitrary graphs with constant degree $d \geq 4$. More interestingly and suprisingly, the problem remains NP-hard for any G being a tree with constant degree d and constant $m \geq 4$.

3.2.1 Case $m = 3$ Processors

We first study the case where we have $m = 3$ processors and the underlying communication graph G is a forest with constant degree. Afterward, we consider the case of $m = 3$ processors and arbitrary complex graphs G with constant degree.

Exact Algorithm for Forests of Constant Degree. In this section, we use a similar representation for packings as Epstein and van Stee [ES07] used when they introduced a PTAS for cardinality constrained bin packing with splittable items. Here, a packing is represented by a graph where nodes correspond to items and edges correspond to bins. For a bin containing two item parts, there is an edge between the two items. If a bin contains only one item, there is a loop on that item. The following lemma can be adapted from Lemma 1 in [Chu+06].

Lemma 3.2. *Given a packing P with bins B for the communication graph (G, E), if the graph (E, B) representing the packing contains a cycle, there is a packing with the same number of bins and without any cycles.*

Proof. For the sake of completeness, we include the proof here. We start with a packing P for a communication graph (G, E) such that the underlying graph (E, B) contains a cycle. We consider an arbitrary cycle in B and remove it without increasing the number of bins and without creating a new cycle. Doing so repeatedly leads to a packing represented by a graph without cycles.

Now, let $\tilde{B} = \{B_1, \ldots, B_k\}$ be the set of bins (or edges) and $\tilde{E} = \{e_1, \ldots, e_k\}$ the set of items (or nodes) in this cycle. W.l.o.g., assume B_i contains a part of size r'_i from item e_i and of size r''_{i+1} from e_{i+1} for all $i \in \{1, \ldots, k-1\}$ and B_k contains a part of size r'_k from e_k and a part of size r''_1 from e_1. W.l.o.g., assume r'_1 is the smallest value out of all r'_i and r''_i. Repack all bins such that B_i contains a part of size $r'_i - r'_1$ from e_i and a part of $r''_i + r'_1$ from e_{i+1} for all $i \in \{1, \ldots, k-1\}$, and B_k contains a part of size $r'_k - r'_1$ from e_k and a part of size $r''_1 + r'_1$ from e_1. Feasibility follows from the minimality of r'_1 among the r'_i and r''_i, the number of bins remains the same and the cycle is broken between e_1 and e_2, that is, B_1 only contains parts of item e_2 and is not an edge in the underlying graph representation anymore. □

This lemma also directly implies the following corollary.

Corollary 3.3. *For constant $m = 3$ and any star communication graph, there exists an optimal packing where for each pair of items, parts of them are packed together in at most one bin.*

This corollary implies that in a star communication graph with degree d, the underlying graph representation of the packing is a forest with degree at most $d-1$ (as it only consists of d nodes).

We now provide an exact algorithm and a proof of its optimality. We assume G to be a single tree. However, this is without loss of generality as solving each individual tree of a forest optimally provides an optimal solution for the forest. This is true as no solution can pack any two items belonging to different trees into the same bin.

Theorem 3.4. *The* Sic *problem can be solved in polynomial time for constant $d \geq 2$, $m = 3$ and G a forest.*

Proof. Consider the algorithm in Listing 3.1. Informally, the algorithm performs a dynamic programming approach by proceeding from bottom to top and storing a set of candidate solutions for each level of the tree. To do so, it starts at the set of nodes with a distance of 1 to the closest leaf. For each node v, it considers the subtree rooted at v and generates all possible graph representations (of degree $m - 1 = 2$) without cycles where each item is split at most $d-1$ times. These representations induce a set of possible subsolutions for the subtree rooted at v, where for each subsolution the remaining space is filled with parts of the item e_v upwards from v. Using the condition in Line 16, only solutions that may be a subsolution of an optimal solution are stored. Once all nodes on a level are completed, the algorithm proceeds by doing the same for the next higher level. Now, as there are already different subsolutions for the lower level, all combinations of these subsolutions are

```
1  [$P[e]$: set of candidate packings for tree rooted at lower endpoint of e]
2  for all edges $e$ to a leaf node $P[e] := \{\{\}\}$ \EndFor\vspace{$-1mm$}
3  for $h$ \text{from} (height of $G = (V, E)$) \text{to} 1
4    for all nodes $v$ with depth $h - 1$
5      Let $E_v = \{e_1, \dots, e_k\}$ be the edges to the children of $v$
6      Let $e_v$ be the edge to the parent of $v$
7      for all $(B_1, \dots, B_k)$ with $(B_i \in P[e_i]\ \forall i)$
8        for all forests $(E_v, \tilde{E})$ with deg $\leq d$ representing a packing for $E_v$
9          $r'_{e_v} = r_{e_v}$
10         [Let $r'_{e_v}(B)$ be the remaining part of $r'_{e_v}$ after packing B]
11         Fill items induced by $E_v$ greedily starting from the leaves of $(E_v, \tilde{E})$ (for each tree separately),
12         according to the remaining item sizes $(r'_{e_1}(B_1), \dots, r'_{e_k}(B_k))$ into bins $\tilde{B}$
13         Fill bins from $\tilde{B}$ with only one item with additional parts of $e_v$; reduce $r'_{e_v}$ accordingly
14         Let $B := \tilde{B} \cup B_1 \cup \dots B_k$ be the bins in the induced packing
15         [If new packing is not dominated by an existing packing]
16         if $((|B| < |B'| \vee r'_v + |B| < r'_v(B') + |B'|) forall B' \in P[e_v])$
17           Let the set of possible packings $P[e_v] := P[e_v] \cup \{B\}$
18 Pack bins according to stored packing with minimal number of bins
```

Listing 3.1: Finding an optimal solution for $m = 3$ and constant degree d.

considered. Again, solutions that may be a subsolution of the optimal solution are stored for later iterations. After having considered all levels, the best packing of the full tree is returned.

From [ES07], we know that we can fill the bins greedily using the graph representation. Now, for any node v, any algorithm cannot pack items $e_1, \dots, e_k$ together with an item $e \notin \{e_1, \dots, e_k, e_v\}$. This implies that partial solutions for each of the item sets $\{e_1, \dots, e_k, e_v\}$ and $E \setminus \{e_1, \dots, e_k\}$ can be computed separately and combined subsequently. However, the remaining question is where item e_v is split, that is how much of the size of e_v is included in which solution. Hence, for each subtree rooted at v, we need to compute all solutions that are optimal for the subtree and use only a certain part of e_v. Fortunately, we have the following property:

Property. A solution B_1 *dominates* a solution B_2 if the number of bins in B_1 is at most as large as the number of bins in B_2 (i.e., $|B_1| \leq |B_2|$) and, additionally, $r'_v(B_1) - r'_v(B_2) \leq |B_2| - |B_1|$ holds.

In this case, in an optimal solution containing B_2 as an induced subsolution, B_2 can simply be replaced by B_1 together with $|B_2| - |B_1|$ bins, each packing a part of size 1 from the item e_v, thus reducing $r'_v(B_1) - r'_v(B_2)$ to at most 0 and ensuring feasibililty of the resulting solution. Also, there exists an optimal solution where the induced subpacking of items $\{e_1, \dots, e_k\}$ is packed together with parts of a size of at most $d - 1$ from e_v. This results in a difference of at most $d - 1$ between the number of bins of two stored solutions, as only non-dominated solutions are stored, hence resulting in at most d solutions to be stored. Otherwise, there must be an item e_i which is packed together with e_v twice, but due to Corollary 3.3, we know that any such packing can be modified such that this is not the case. Together with Lemma 3.2, we know that at least one of the graphs generated in Line 8 represents a subpacking for (the star subgraph) E_v that is induced by an optimal packing of the full graph. As our algorithm finds all candidate solutions for the set E_v that may end up to match the solution induced by the optimum, the optimality of the algorithm recursively follows.

Concerning the runtime, at most d solutions must be stored in Line 17 for each node. In turn, the loop in Line 7 is executed at most d^d times. Also, the number of graphs in Line 8 is definitely below 2^{d^2}. Polynomial runtime follows. $\square$

NP-Hardness for Graphs of Constant Degree. We show that for general graphs with constant degree $d \geq 4$, the problem is NP-hard when $m = 3$. To this end, we can easily reduce from the PartitionIntoTriangles problem: Given an undirected graph $G = (V, E)$, the question is whether there is a partitioning into 3-element sets $S_1, S_2, \ldots, S_{|V|/3}$ such that each S_i forms a triangle in G. This problem is proven to be NP-hard for graphs of (constant) degree $d \geq 4$ in [RKB13].

Lemma 3.5. *Sic is NP-hard for $m = 3$ and general graphs with constant degree $d \geq 4$.*

Proof. Given an instance I for PartitionIntoTriangles, construct an instance I' for Sic by assigning a weight of 1/3 to each edge. If I is a YES-instance, I' can be packed into $|V|/3$ many bins. If I is a NO-instance, more than $|V|/3$ bins are required.

$\square$

3.2.2 NP-Hardness for $m > 3$ Processors

We now study the complexity for $m > 3$ processors. We thereby focus on the NP-hardness when the underlying graph is a single path, yielding hardness results for the most basic case where only a single, most simply structured task is to be scheduled.

Theorem 3.6. *The Sic problem is strongly NP-hard for $d = 2$ and constant $m \geq 6$, even for a single path.*

Proof. We start with the 3-Partition problem with a restricted size of the elements which is defined as follows. Given a multiset $A = \{ a_1, \ldots, a_n \}$ of $n = 3k$ elements, a bound B with $B/4 < a_i < B/2 \ \forall\ i \in \{1, \ldots, n\}$ and $\sum_{a \in A} a = kB$, is there a partition into k sets $A_1, \ldots, A_k$ such that $|A_i| = 3$ and $\sum_{a \in A_i} a = B$ for all $i \in \{ 1, \ldots, k \}$?

Let our Sic instance consist of one path with $\ell + 1 := 3k \cdot (3m-2) + 2k \cdot (m-5) + 1$ nodes. We denote the edge between node i and $i+1$ by e_i, yielding $E = \{e_1, \ldots, e_\ell\}$ with sizes $\{r_1, \ldots, r_\ell\}$. Now let $r_{(i-1)\cdot(3m-2)+1} := \left(\frac{1}{2} + \frac{a_i}{2B}\right)\left(1 - \frac{m-5}{5m}\right) < \frac{3}{4}$ for all $i \in \{1, \ldots, 3k\}$, called *medium items*, and let $r_{(i-1)\cdot(3m-2)+m} = r_{(i-1)\cdot(3m-2)+\lfloor \frac{3}{2}m \rfloor} = r_{(i-1)\cdot(3m-2)+2m} = 1 - \frac{m-2}{5m} \ \forall\ i \in \{1, \ldots, 3k\}$, called *large items*. All other edges are assigned a size of $r_i = \frac{1}{5m}$, called *small items*. For a visualization, see Figure 3.2. Now is there a packing of a size of at most $11k$?

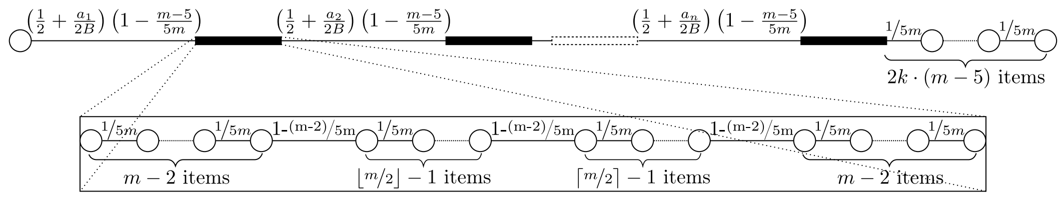

$(\frac{1}{2}+\frac{a_1}{2B})(1-\frac{m-5}{5m})$
$(\frac{1}{2}+\frac{a_2}{2B})(1-\frac{m-5}{5m})$
$(\frac{1}{2}+\frac{a_n}{2B})(1-\frac{m-5}{5m})$
$1/5m$
$2k\cdot(m-5)$ items
$1\text{-}(m\text{-}2)/5m$
$m-2$ items
$\lfloor m/2\rfloor-1$ items
$\lceil m/2\rceil-1$ items
$m-2$ items

Figure 3.2: Corresponding SIC instance for 3-PARTITION with input $\{a_1,\dots,a_n\}$.

In case that the 3-PARTITION instance is a YES-instance, we need to show that there is a corresponding packing with at most $11k$ bins for our SIC instance. Given a set A_i, the 3 (medium) items S_i derived from it can always be packed into two bins together with $m-5$ of the last $2k\cdot(m-5)$ items: that is, items $e_{3k\cdot(3m-2)+(2i-2)(m-5)+1},\dots,e_{3k\cdot(3m-2)+(2i-1)(m-5)}$ for the first bin, and items $e_{3k\cdot(3m-2)+(2i-1)(m-5)+1},\dots,e_{3k\cdot(3m-2)+2i(m-5)}$ for the second bin. That is, because the 3 items from S_i only use 4 of the allowed incident nodes in both bins, and the $m-5$ small items use $m-4$ of the allowed incident nodes. Also, we have $\sum_{r\in S_i} r+2\cdot\frac{m-5}{5m}=(\frac{3}{2}+\sum_{a\in A_i}\frac{a}{2B})(1-\frac{m-5}{5m})+2\cdot\frac{m-5}{5m}=2$ giving a valid packing if we split one of the medium items accordingly. Thus, we can pack all item sets S_i into two bins each, together with the last $2k\cdot(m-5)$ items, leading to $2k$ bins.

Observe that the $3m-3$ items in each block filling the gaps between two medium items (i.e., the rectangles in Figure 3.2) can be put into three bins: The first $m-1$ items of each block (i.e., $m-2$ small items and one large item) can be put into one bin, as the sum of their sizes is exactly 1 by construction. The same holds for the second and third item set of $m-1$ items, respectively.

More formally, for all $i\in\{1,\dots,3k\}$, the items $e_{(i-1)\cdot(3m-2)+2},\dots,e_{(i-1)\cdot(3m-2)+m}$ can be packed into one bin. The same property holds for the respective item sets $\{e_{(i-1)\cdot(3m-2)+m+1},\dots,e_{(i-1)\cdot(3m-2)+3m-2}\}$ for each $i\in\{1,\dots,3k\}$ as well as the item sets $\{e_{(i-1)\cdot(3m-2)+2m},\dots,e_{(i-1)\cdot(3m-2)+3m-2}\}$ for each $i\in\{1,\dots,3k\}$. This gives another $3\cdot 3k$ bins and all items are packed.

On the other hand, we show that if there is a packing with a size of at most $11k$, we show that the respective 3-PARTITION instance is a YES-Instance. In order to do so, we show the following properties in the given order:

(1) The capacity of each bin must be fully utilized.

(2) At least k medium or large items need to be split.

(3) In order to pack the last $2k\cdot(m-5)$ items, $4k$ additional separate components of the communication graph have to be packed together with them.

(4) Our $11k$ bins contain exactly $9k\cdot(m-1)+2k\cdot(m-3)$ item parts.

(5) The last $2k\cdot(m-5)$ items are packed into $2k$ bins containing exactly $m-5$ of these items and two medium item parts each.

(6) Exactly k of the $3k$ medium items are split, and they are split into exactly two parts each.

(7) The corresponding 3-Partition is a Yes-instance.

(1) The capacity must be fully utilized in any bin, because we have $\sum_{i=1}^{\ell} r_i = \sum_{i=1}^{3k}\left(\frac{1}{2}+\frac{a_i}{2B}\right)(1-\frac{m-5}{5m}) + 2k(m-5)\cdot\frac{1}{5m} + 3\cdot 3k\cdot\left((m-2)\cdot\frac{1}{5m}+\left(1-\frac{m-2}{5m}\right)\right) = \left(\frac{3k}{2}+\frac{kB}{2B}\right)(1-\frac{m-5}{5m}) + 2k\cdot\frac{m-5}{5m} + 3\cdot 3k = 11k$.

(2) Now we show that at least k medium or large items need to be split. There are $3\cdot 3k$ large items as well as $3k$ medium items. If less than k of these $12k$ items were split, there would be at least one bin fully containing two of these items. This is a contradiction, as all the item sizes are greater than $\frac{1}{2}$. Hence, there are at least $\ell + k$ item parts.

(3) We now concentrate on the $2k\cdot(m-5)$ last items, i.e., on the items $e_{3k\cdot(3m-2)+1},\ldots,e_{3k\cdot(3m-2)+2k\cdot(m-5)}$. Note that these items cannot be packed with a medium or a large item without using two components of the communication graph. However, if using two components, they contain at most $m-2$ items, which implies (as we always use the full capacity by (1)), that each bin containing one of these items also contains at least two medium or large items. As by construction, there are always at least $m-2$ edges between a medium and a medium or large item, and at least $\lfloor\frac{m}{2}\rfloor - 1$ edges between two large items, there are only two possibilities how to obtain this:

a) At most $m-5$ of the considered small items are combined with at least two further components of the communication graph, which contain exactly one medium or large item each.

b) At most $\left\lceil\frac{m-6}{2}\right\rceil \leq \frac{m-5}{2}$ of the considered small items are combined with only one further component of the communication graph, which contains exactly two large items.

Taken together, this implies that in order to pack all $2k\cdot(m-5)$ items, there have to be taken at least $2k\cdot 2 = 4k$ additional separate components from the communication graph.

(4) It follows that in our packing, the $11k$ bins can contain at most $11k\cdot(m-1) - 4k = 9k\cdot(m-1)+2k\cdot(m-3)$ item parts by (3). However, as we showed earlier in (2), the overall number of item parts is at least $\ell+k = 3k\cdot(3m-2)+2k\cdot(m-5)+k = 9k\cdot(m-1)+2k\cdot(m-3)$. Thus, both properties are tight and in all bins not containing any of the $2k\cdot(m-5)$ last items, exactly $m-1$ items must be packed. Now, this can only be done by using one (complete) large item and $m-2$ adjacent small items, as one medium item together with $m-2$ adjacent small items does not use the full capacity.

(5) Considering the last $2k\cdot(m-5)$ items again, note that all bins fulfilling Property a) from (3) now need to contain *exactly* $m-5$ of these small items as well as all bins fulfilling Property b) now need to contain *exactly* $\frac{m-5}{2}$ of these small

items. However, this implies that Property b) never happens. Otherwise, if there were $2j$ bins fulfilling Property b) (giving $2k - j$ bins fulfilling Property a)), each of them would make at least one large item incomplete. As this leads to exactly $2k - j$ bins containing $m - 3$ items and $2j$ bins containing $m - 2$ items, there must be exactly $9k - j$ bins containing $m - 1$ items. However, there are only at most $9k - 2j$ (complete) large items left, yielding $j = 0$.

(6) We now know that there are exactly $9k$ bins containing one (complete) large and $m-2$ small items each. Hence, there are exactly $2k$ bins where each bin contains $m - 5$ of the last $2k \cdot (m - 5)$ items and two parts of medium items. We also know that only medium items are split, thus exactly k of the medium items are split into exactly two parts each.

W.l.o.g., let $\tilde{E} = \{ e_1, \ldots, e_k \}$ be the (medium) items that are split. We observe that no two items from $\tilde{E}$ can be packed into the same bin. Otherwise, at least one of them uses a capacity of at least $\frac{1}{2}$ in that bin, hence a capacity of at most $\frac{1}{4}$ in the other bin, where the remaining part of the item is packed. Then, the capacity is not fully utilized in that other bin, as the other medium item (which has to exist by (5) and (6)) uses less than a capacity of $\frac{3}{4}$.

Now, for each medium item split into two parts, we know that exactly one complete other medium item is packed together with each part of it. For each split item, we build a set containing itself and the two medium items packed together with it. There are m such sets $S_1, \ldots, S_m$ with three elements each. Now the sum of the sizes of these three items is $2 - 2 \cdot \frac{m-5}{5m}$ (as both bins additionally contain $m - 5$ small items), implying $\sum_{r \in S_i} \left(\frac{1}{2} + \frac{r}{2B}\right) = 2$ which yields $\sum_{r \in S_i} r = B$. This is a 3-Partition. ☐

Corollary 3.7. *The* Sic *problem is strongly NP-hard for $d = 2$ and $m = 4$, even for a single path.*

Proof Sketch. We use the same reduction as in the proof of Theorem 3.6, but with medium item sizes $\frac{1}{2} + \frac{a_i}{2B}$ and without adding the last $2k(m - 5)$ auxiliary items. This implies removing step (3) and instead using the fact that any bin containing a medium item part can only be packed up to the full capacity using two separate components of the communication graph. As the medium items alone have an overall capacity of $2k$, it follows that at least $2k$ additional separate components need to be used, yielding exactly $9k \cdot (m - 1) + 2k \cdot (m - 2)$ item parts to be packed in step (4). However, this directly implies that at least $9k$ bins have to contain the full number of $m-1$ items, which (by (1)) is only possible using one large and $m-2$ adjacent small items. It follows that the remaining $2k$ bins contain the $3k$ medium items. Hence, the remaining part of the proof remains the same. ☐

Corollary 3.8. *The* Sic *problem is strongly NP-hard for $d = 2$ and $m = 5$, even for a single path.*

Proof Sketch. We modify the reduction of Corollary 3.7 by adding one additional small item adjacent to each medium item and reducing the medium item sizes by factor $(1 - \frac{1}{5 \cdot 5})$. Now three medium items are always packed together with two of these adjacent items into two bins. There are k remaining small items adjacent to medium items. In order for these to be packed, we add k small gadgets to the end of the path (instead of the $2k \cdot (m-5)$ small items from the proof of Theorem 3.6). These small gadgets consist of one *very large* item of size $1 - \frac{2}{5 \cdot 5}$ (in contrast to large items, which now have a size of $1 - \frac{3}{5 \cdot 5}$) and one small item. Two adjacent small gadgets are always separated by the usual rectangular gadgets from Figure 3.2. With a similar argument as (3) from the proof of Theorem 3.6, we now ask how to pack the small items adjacent to medium items and conclude that they either have to be packed together with medium items or (for the remaining k items) they have to be packed together with the newly introduced small gadgets. This concludes the proof for $m = 5$. □

Corollary 3.9. *The* Sic *problem is strongly NP-hard for constant $d \geq 2$ and constant $m \geq 4$, even for a single connected component.*

Proof Sketch. In order to achieve a higher degree than 2, we add a similar gadget to the (rectangular) gadgets with the large (and small) items from the original reduction in Theorem 3.6. This new gadget is visualized in Figure 3.3. To separate this gadget from the rest of the instance, we first add another rectangular gadget as used in the proof of Theorem 3.6 at the end of the path. The new part of the gadget (behind the rectangular gadget) consists of a star graph with many small and few large items to achieve the necessary degree. We fill it up with small items in a path towards the rectangular gadget. This is necessary to achieve an overall number of items divisible by $(m-1)$ (i.e., an overall number of small items divisible by $(m-2)$). Now the items from the newly introduced gadget have to be packed with one large item and $m-2$ small items into one bin each similar to the packing for the rectangular gadgets in Theorem 3.6. The remaining reduction is analogous. □

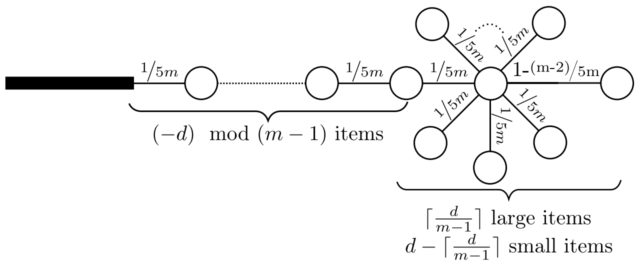
Proposition 3.10. *The* Sic *problem is strongly NP-hard for $d = 1$ and constant $m \geq 4$.*

Proof. The restrictions lead to having a set of single edges. Hence, similarly to Proposition 3.1, this problem is equivalent to bin packing with splittable items and cardinality constraint $\lfloor m/2 \rfloor$. This problem is already NP-hard for cardinality constraint at least 2, hence the claim follows. □

3.3 Communication Graphs of Degree Two

In this section, we analyze a simple greedy algorithm for instances with a communication graph of degree two.

First note that this problem can be solved efficiently for $m \leq 3$ similarly to the algorithm in Listing 3.1. For each path, the claim directly follows. For cycles, in order to find an optimal packing, we start the algorithm in Listing 3.1 at each node



$1/5m$
$1/5m$
$1/5m$
$1/5m$
$1/5m$
1-(m-2)/5m
$1/5m$
$1/5m$
$1/5m$
$(-d) \mod (m-1)$ items
$\lceil \frac{d}{m-1} \rceil$ large items
$d - \lceil \frac{d}{m-1} \rceil$ small items

Figure 3.3: Gadget to add for degree $d \geq 2$.

```
Split each cycle at an arbitrary node (duplicate node), obtaining a path
Connect all paths to one single path $V' = (G', E')$ by adding edges of size 0
Index items in order of path, $E' := \{ e_0, e_1, \dots \}$
$B$ := new empty bin
for all items $e_i \in E'$
  if $e_i$ does not fit into $B$ w.r.t.\ edge constraint
    $B$ := new empty bin
  if $e_i$ fits into $B$
    Pack $e_i$ into $B$
  else
    Pack as much of $e_i$ into $B$ as possible
    $B$ := new empty bin
    Pack remaining part of $e_i$ into $B$
```

Listing 3.2: Algorithm A_1 for instances of degree two.

of a cycle individually. By doing so, all possible points where the cycle may be split are considered and the optimal solution will be among those found.

For $m > 3$, we first remove cycles, then connect resulting paths arbitrarily such that we get one single path and finally use a straightforward adaption of the classical NextFit algorithm. The formal description of the algorithm A_1 is given in Listing 3.2. It starts by splitting up all cycles into paths. This is done at an arbitrary node: that is, after the split, two copies of this node (at each end of the path) are in the resulting graph. As a next step, the algorithm connects the ends of the set of paths by edges of size 0 such that the result is one single path. Note that after this, we have one path with $|E'| = |E| + (p-1)$ edges. The algorithm then dispatches the items in the order of the path starting at one of the nodes with degree one. If an item fits into the current bin while violating neither the capacity nor the edge constraint, it is placed in the current bin. If it does not fit into the current bin (since at least one of the constraints would be violated), as much of the item is placed in the current bin as possible and a new bin is opened in which the (remaing part of the) item is placed. That is, A_1 will always (except for the last bin) fully pack a bin or pack the maximum number of (parts of) items allowed in a bin.

We start by lower bounding the number of bins used by OPT. Let p_p be the number of paths and p_c be the number of cycles (implying $p = p_p + p_c$).

Lemma 3.11. *We can state that:*

1. $\mathrm{OPT} \geq \lceil \sum_{e \in E'} s_e \rceil$ *and*
2. $\mathrm{OPT} \geq \left\lceil \frac{|E|+p_p}{m} \right\rceil$.

Proof. The first inequality directly follows from the capacity constraint and the fact that items in $E' \setminus E$ have size 0 by construction.

For the second inequality, each of the p connected components of the input graph is either a path or a cycle by the degree constraint. If the input only consists of cycles, an optimal solution could potentially pack up to m items per bin. However, for each path, there is an overhead of one node, i.e., each path of length k is incident to $k+1$ nodes. Expressed differently, the upper bound of m items per bin is still valid if we close each path to a cycle. This gives the second bound. □

We next analyze the approximation ratio of A_1 and show that any solution is worse than the optimum by a factor only slightly larger than 7/3.

Lemma 3.12. *The algorithm A_1 uses at most*

$$A_1 \leq \frac{4m}{3m-3} \cdot \mathrm{OPT} + \left(1 - \frac{1}{2(m-1)}\right) \sum_{e \in E} r_e + 1$$

bins and runs in time $O(|V|)$.

Proof. For the sake of analysis, we partition the set of bins used by algorithm A_1 into the set B_1 containing those bins with a tight capacity constraint and B_2 containing those bins which are not full but have a tight edge constraint. We also define the following three sets of items on the basis of how they are packed in the solution given by algorithm A_1. E_1 contains those items from E' for which at least one part of the item is packed in a bin from B_1. In E_2 there are all items from E' for which at least one part of the item is packed in a bin from B_2, and finally $E_R := E' \setminus (E_1 \cup E_2)$ contains the remaining items. Note that in E_R there can only be items placed in the last bin that was opened, and that $E_1 \cap E_2$ need not be empty.

We now give a bound on the maximum number of bins used by A_1. We have

$$A_1 = |B_1| + |B_2| + \left\lceil \frac{|E_R|}{m-1} \right\rceil$$

since each bin either belongs to B_1 or B_2 or is the last bin and $E_R \neq \emptyset$. Additionally, since any bin belonging to B_1 only contains items from E_1, we have $|B_1| \leq \sum_{e \in E_1} r_e$. Any bin belonging to B_2 contains $m-1$ (parts of) items from E_2 and each item from E_2 is packed in at most one bin from B_2 and hence, $|B_2| = \frac{|E_2|}{m-1}$. Therefore, we have

$$A_1 = |B_1| + |B_2| + \left\lceil \frac{|E_R|}{m-1} \right\rceil \leq \sum_{e \in E_1} r_e + \frac{|E_2|}{m-1} + \left\lceil \frac{|E_R|}{m-1} \right\rceil .$$

Denoting the set of items belonging to $E_1 \cap E_2$ by $\tilde{E}$, we have $|E'| = |E_1| + |E_2| + |E_R| - |\tilde{E}|$. Also, $|E| \geq p_p + 3p_c$, because the smallest possible cycle is a triangle. Hence $|E'| = |E| + (p-1) < |E| + p_p + p_c \leq 4/3 \cdot |E| + 2/3 \cdot p_p \leq 4/3 \cdot (|E| + p_p)$. Together with the claim that $|\tilde{E}| \leq |E_1|/2$, we can conclude

$$\begin{aligned}
A_1 &\leq \sum_{e \in E_1} r_e + \frac{|E_2| + |E_R|}{m-1} + 1 \\
&\leq \frac{|E_1|}{2(m-1)} + \frac{|E_2| + |E_R|}{m-1} + \frac{2(m-1)-1}{2(m-1)} \sum_{e \in E} r_e + 1 \\
&\leq \frac{|E_1| + |E_2| + |E_R| - |\tilde{E}|}{m-1} + \frac{2 \cdot (m-1) - 1}{2(m-1)} \sum_{e \in E} r_e + 1 \\
&\leq \frac{4}{3} \cdot \frac{|E| + p_p}{m-1} + \left(1 - \frac{1}{2(m-1)}\right) \sum_{e \in E} r_e + 1 \\
&\leq \frac{4m}{3m-3} \cdot \mathrm{OPT} + \left(1 - \frac{1}{2(m-1)}\right) \sum_{e \in E} r_e + 1,
\end{aligned}$$

where we used the fact that $|E_1| \geq \sum_{e \in E_1} r_e$ and the aforementioned bound on A_1 in the first two estimations, the claimed bound on $|\tilde{E}|$ in the third, the inequalities claimed right before in the second last and the second bound on OPT in the last inequality.

Hence, it remains to prove the claim. Recall that $\tilde{E}$ contains the items that are in E_1 and in E_2, i.e., any item $e \in \tilde{E}$ is partly packed in a bin that is full and partly packed in a bin that is not full but has a tight edge constraint. By the definition of A_1, such an item $e \in \tilde{E}$ fulfills the condition that it is first *partly* packed in a bin B' belonging to B_1 and then a bin belonging to B_2 is opened to pack the remaining part of item e. Note that consequently B' contains a different item belonging to $E_1 \setminus \tilde{E}$. Hence, to any item $e \in \tilde{E}$ we can associate a different item $\bar{e} \in E_1 \setminus \tilde{E}$, proving the claim and concluding the proof.

The runtime is $O(|V|)$ as we essentially traverse the path once. □

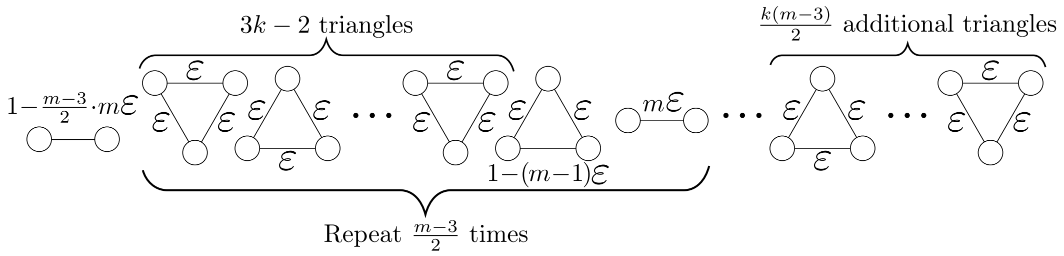
Corollary 3.13. *Algorithm A_1 has an asymptotic approximation ratio of at most* $\frac{7}{3} + \frac{5}{6(m-1)}$.

Proof. From Lemma 3.12 and using the bounds on OPT, we have

$$A_1 \leq \frac{4m}{3m-3} \cdot \mathrm{OPT} + \left(1 - \frac{1}{2(m-1)}\right) \sum_{e \in E} r_e + 1 \leq \left(\frac{7}{3} + \frac{5}{6(m-1)}\right) \mathrm{OPT} + 1.$$

□

We now show that our analysis of the approximation factor of A_1 is (almost) tight by giving an instance on which it obtains an approximation factor of at least $\frac{7}{3}\left(1 - \frac{1}{(6k-2)}\right)$.



$3k-2$ triangles
$\frac{k(m-3)}{2}$ additional triangles
$1-\frac{m-3}{2}\cdot m\varepsilon$
ε
$m\varepsilon$
$1-(m-1)\varepsilon$
Repeat $\frac{m-3}{2}$ times

Figure 3.4: Hard instance for algorithm A_1. Let $m = 12k - 3$ for some $k \in \mathbb{N}$.

Intuitively, this instance (cf. Figure 3.4) exploits the two different optimization goals, i.e., using the full capacity of a bin and using the full number of allowed adjacent nodes. The optimal algorithm chooses to skip certain elements in order to always use the full capacity. In contrast, the greedy algorithm always uses the full capacity (packing only two items) in every second bin and the full number of allowed adjacent nodes (using only low capacity) in the other bins. Additionally, this instance uses the fact that A_1 splits up all cycles to worsen the quality of A_1's solution. This gives an approximation factor of almost $7/3$. For $m \to \infty$ the upper as well as the lower bound converge to $7/3$.

Theorem 3.14. *There is an instance such that the algorithm A_1 performs by a factor of at least $\frac{7}{3}\left(1 - \frac{1}{(6k-2)}\right)$ worse than the optimum.*

Proof. Consider an instance as given in Figure 3.4 and let $\varepsilon > 0$ be sufficiently small and $m = 12k - 3$ for some $k \in \mathbb{N}$. The instance consists of a set of tasks that are mostly triangles with some additional edges with the following form: The leftmost item (single edge) has a size of $1 - \frac{m-3}{2} \cdot (m-1)\varepsilon$. It is repeatedly followed by the following set of tasks S: First, there is a set of $3k - 2$ triangles each containing three items with size ε. An additional triangle with two items of size ε and one item of size $1 - (m-1)\varepsilon$ follows. Finally, there is one edge of size $m\varepsilon$.

This set of tasks is repeated several times such that we obtain $(m-3)/2$ copies of S. Note that this number is integer by $m = 12k - 3$. Finally, $k(m-3)/2$ additional triangles follow, each containing three items of size ε.

On this instance, an optimal solution can pack the items of each subset S without the single item of size $m\varepsilon$ together with k additional triangles from the final set of triangles. Taken together, these are $(3k-2)\cdot 3 + k \cdot 3 + 2 = m - 1$ items of size ε, which fit exactly together with the one item of size $1-(m-1)\varepsilon$. Also, the packed items are incident to exactly $(3k-2)\cdot 3 + k\cdot 3 + 3 = 12k - 3$ nodes. As there are $\frac{m-3}{2}$ copies of S, $\frac{m-3}{2}$ bins are needed for the respective items. Additionally, the very first item is packed together with the $\frac{m-3}{2}$ items that were left over (each having a size of $m\varepsilon$). Overall, these are $\frac{m-3}{2} + 1 = \frac{m-1}{2}$ edges which can be packed into one bin. Hence, $\text{OPT} = \frac{m-3}{2} + 1 = \frac{m-1}{2}$.

In contrast, A_1 processes the nodes from left to right. In order to do so, it splits up the triangles and, thus, constructs a forth node for each triangle. We assume that the last triangle in each set S is split up such that it starts with the ε-edges and the last edge is the edge of size $1-(m-1)\varepsilon$. Hence, it packs the very first item together with all items from S except the last edge of the last triangle (size $1-(m-1)\varepsilon$) and the last edge (size $m\varepsilon$). By doing so, items in this bin are already incident to two nodes from the very first item and $(3k-2)\cdot 4+3 = 12k-5$ nodes from the other edges, hence these two items cannot be added. Afterward, these two items are packed in a second bin, leading to an unpacked part of the second item of size ε. This part serves as the first item packed together with items from the next copy of S. As this repeats $\frac{m-3}{2}$ times, A_1 needs $m-3$ bins for all the copies, leaving an unpacked part of size ε from the last item and the $\frac{k(m-3)}{2}$ additional triangles. Due to the cardinality constraint, these items will need at least another

$$\frac{(2+{}^{k(m-3)}/_{2}\cdot 4)-1}{m-1} = \frac{2k(m-3)+1}{m-1} > \frac{2k(m-3)}{m-1}$$

bins. Taking the upper and lower bound together yields

$$\begin{aligned}\frac{A_1}{\mathrm{OPT}} &\geq \frac{m-3+{}^{(2k(m-3))}/_{(m-1)}}{{}^{(m-1)}/_{2}}\\ &= \frac{6(2k-1)+{}^{(12k(2k-1))}/_{(12k-4)}}{6k-2}\\ &= \frac{6(2k-1)+{}^{3k}/_{(3k-1)}\cdot(2k-1)}{6k-2}\\ &\geq \frac{7(2k-1)}{(6k-2)}\\ &= \frac{7}{3}\left(1-\frac{1}{(6k-2)}\right).\end{aligned}$$

□

3.4 Communication Graphs with Arbitrary Degree

In this section, we study the case where G is a graph of arbitrary degree. We start with the case where G is a forest and then generalize our results to arbitrary graphs.

3.4.1 Packing Forests of Arbitrary Degree

In the following we propose an algorithm A_2, which provides $(\min\{1.8, \frac{1.5m}{m-1}\}+1)$-approximate solutions for instances described by a forest. Note that by splitting the tree at each node and losing a constant factor of 2, the problem can be reduced to cardinality constrained bin packing with splittable items. As there exists an EPTAS for this problem [ELS12], we can get a $2\cdot(1+\varepsilon)$-approximation by using it. However, to achieve an approximation factor of 2.5 for our problem, we need to set the ε from [ELS12] to at most $\frac{1}{36}$. The constant in the runtime is then around $\left(\frac{1}{\varepsilon^8}\right)^{\frac{1}{\varepsilon^2}}$, which is a very large number, rendering the usefulness rather questionable in practice. In the literature, the best algorithm of cardinality constrained bin packing with splittable items using a rather simple approach is NextFit with a tight approximation factor of $2 - \frac{1}{m}$, which would only yield a $2 \cdot (2 - \frac{1}{m})$-approximation. The advantage of our algorithm is its simplicity and low runtime.

Roughly speaking, our algorithm consists of two steps: In a first step (cf. Listing 3.3), A_2' computes a preliminary (generally infeasible) packing, which ignores the capacity constraints of bins. Then in a second step, we fix the preliminary packing by repacking parts of items that are packed in bins violating the capacity constraint. We use T_v to denote the nodes in the subtree rooted at node v and $|T|$ to denote the number of nodes in tree T.

Algorithm A_2' works as follows. In Line 1 to Line 8, we first identify those trees for which the number of its edges is $i \cdot (m-1)$ for some $i \in \mathbb{N}$, i.e., it contains $i \cdot (m-1)+1$ nodes. Such a tree can be packed into i bins such that each bin has a tight edge constraint. For all remaining trees we build as many pairs of trees as possible such that the overall number of edges per pair is $i \cdot (m-1)-1$ for some $i \in \mathbb{N}$, i.e., it once more contains $i \cdot (m-1)+1$ nodes. For ease of presentation each pair is combined to a new tree by adding an edge with weight 0 between their roots. Finally all remaining trees are combined to a single tree in the same way. Then in Line 10 to Line 23 each of the resulting trees is packed individually by processing its nodes from the leaves to the root of the tree. That is, the algorithm finds a node v with maximal depth such that the tree rooted in v has a size of at least m. Having found such a node, it packs sets of $m-1$ items into one bin, which implies an efficient utilization of the edge constraint. It proceeds this way until it reaches the root of the tree. After this step was performed for all trees, all yet unpacked items (E_2) are packed in a greedy way (Line 24).

For simplicity, denote as M_i' the set M_i where the virtual items are removed, i.e., we consider the original trees.

Also, for any packing P of a forest G, we introduce a *corresponding graph* $G_P = (V_P, E_P)$ as follows: Let V_P be the set of trees of G. For two trees $T_1, T_2 \in V_P$, let $\{T_1, T_2\} \in E_P$ if and only if there is a bin containing items from T_1 as well as T_2.

Definition 3.15. In a given packing P, a set of trees $\mathcal{T} = \{T_1, \ldots, T_i\}$ is *packed in conjunction* if $\mathcal{T}$ is a maximal connected component in the corresponding graph G_P.

Definition 3.16. A set of trees $\{T_1, \ldots, T_i\}$ is *packed perfectly* if it is packed in conjunction and the number of used bins is exactly $\frac{\sum_{\ell=1}^{i} |T_\ell| - 1}{m-1}$. This is also called a *perfect fit.*

In particular, for any perfect fit, we have $\sum_{\ell=1}^{i} |T_\ell| = 1 \mod (m-1)$. Note that no set of trees can be packed in conjunction using less bins than in a perfect fit. This is because, compared with the maximum number of $m-1$ items that can be packed in each bin, at least $i-1$ edges from the corresponding graph need to be additionally packed by the definition of packing in conjunction (i.e., for packed items from each additional tree in a bin, one item less may be packed).

```
$M_1 := \emptyset$, $M_2 := \emptyset$, $M_3 := \emptyset$
Add all trees $T$ with $|T| = 1 \mod m-1$ to $M_1$
for all trees $T$
  if $T \notin M_1 \cup M_2$, choose (if possible) a $T'$ such that $|T| + |T'| = 1 \mod m-1$
  Connect roots of $T$ and $T'$ by virtual edges with size 0
  Add resulting tree (root chosen arbitrarily) to $M_2$
Connect the roots of all remaining trees by virtual edges with size 0
Add resulting tree (root chosen arbitrarily) to $M_3$
$E_2 := \emptyset$
for all trees $T \in M_1 \cup M_2 \cup M_3$
  for $h$ from $T$ to 1
    for all nodes $v$ with depth $h$
      if $|T_v| \geq m$
        Let $C_v = \{v_1, \ldots, v_k\}$ be the children of $v$ s.t. $|T_{v_1}| \leq \ldots \leq |T_{v_k}|\}$
        $b := 1$
        for $i$ from 1 to $k$
          if $|T_{v_b}| + \ldots + |T_{v_i}| \geq m-1$
            Let $T'$ be the subtree induced by $\{v\} \cup T_{v_b} \cup \ldots \cup T_{v_i}$
            Pack $m-1$ adjacent items from $T'$ into new bin
            Put all remaining non-virtual items from $T'$ into $E_2$
            Remove $T'$ (except for $v$ if $i \neq k$) from $T$
            $b := i + 1$
  Add any remaining non-virtual items in $T$ to $E_2$
Greedily pack the items from $E_2$ into at most $\left\lceil \frac{|E_2|}{\lfloor m/2 \rfloor} \right\rceil$ bins
```

Listing 3.3: A_2' constructing a preliminary packing.

Hence, $|T_\ell| - 1$ being the number of items in tree T_ℓ, any packing needs at least $\frac{\left(\sum_{\ell=1}^{i}(|T_\ell|-1)\right)+(i-1)}{m-1} = \frac{\sum_{\ell=1}^{i}|T_\ell|-1}{m-1}$ bins. Thus, we also observe that any set of trees that is *not* packed perfectly uses at least $\frac{\sum_{\ell=1}^{i}|T_\ell|}{m-1}$ bins.

Observation 3.17. There are exactly $|M_1'|$ trees T with $|T| = 1 \mod m-1$. Hence, there can be at most $|M_1'|$ trees such that each of them is packed perfectly.

Observation 3.18. Any algorithm can construct at most $|M_2| = \frac{1}{2}|M_2'|$ pairs of trees T_1 and T_2 such that $|T_1| + |T_2| = 1 \mod m-1$. Thus, any algorithm can pack at most $\frac{1}{2}|M_2'|$ pairs of trees perfectly.

We now give two bounds on the optimal solution OPT.

Lemma 3.19. OPT *can be bounded by*

$$\mathrm{OPT} \geq \frac{|E| + \frac{1}{2}|M_2'| + \frac{2}{3}|M_3'|}{m-1}.$$

Proof. Consider an optimal packing. We decompose the input forest into a set of forests $\mathbf{T} := \{\mathcal{T}_1, \ldots, \mathcal{T}_\iota\}$ such that each $\mathcal{T} \in \mathbf{T}$ is packed in conjunction. Note that this decomposition is unique, as by definition of packing in conjunction it is equivalent to decomposing a graph into maximal connected components.

We denote the number of sets $\mathcal{T} \in \mathbf{T}$ with cardinality ℓ, i.e., those consisting of exactly ℓ trees, as k_ℓ, $\ell \in \mathbb{N}$. Also, we denote the edge set containing all items in the respective trees as E_ℓ, $\ell \in \mathbb{N}$. Finally, let $p_i = i \cdot k_i$ be the overall number of trees contained in a tree set of this type.

Now, if $k_1 > |M_1'|$, there must be at least $k_1 - |M_1'|$ trees that are not packed perfectly. Otherwise, we have a contradiction to Observation 3.17. This implies that OPT needs at least $\frac{|E_1|+\max\{0,k_1-|M_1'|\}}{m-1}$ bins to pack all items in E_1.

Further, if $k_2 > |M_2'|$, at least $k_2 - \frac{1}{2}|M_2'|$ of these components cannot be a perfect fit. Otherwise, we have a contradiction to Observation 3.18. Hence, OPT needs at least $\frac{|E_2|+k_2+\max\{0,k_2-\frac{1}{2}|M_2'|\}}{m-1}$ bins to pack all items in E_2.

For the remaining items, OPT needs at least

$$\frac{\sum_{i=3}^{\infty}(|E_i| + k_i \cdot (i-1))}{m-1} \geq \frac{\sum_{i=3}^{\infty}(|E_i| + p_i \cdot \frac{i-1}{i})}{m-1} \geq \frac{\sum_{i=3}^{\infty}(|E_i| + \frac{2}{3} \cdot p_i)}{m-1}$$

bins as it cannot pack these trees better than in a perfect fit.

Together, this gives

$$\begin{aligned}
(m-1)\mathrm{OPT} &\geq \sum_{i=1}^{\infty} |E_i| + k_2 + \frac{2}{3}\sum_{i=3}^{\infty} p_i + \max\{0, k_1 - |M_1'|\} \\
&\quad + \max\left\{0, k_2 - \frac{1}{2}|M_2'|\right\} \\
&= |E| + \frac{1}{2}p_2 + \frac{2}{3}(|M_1'| - p_1 + |M_2'| - p_2 + |M_3'|) \\
&\quad + \max\{0, p_1 - |M_1'|\} + \max\left\{0, \frac{1}{2}(p_2 - |M_2'|)\right\} \\
&\geq |E| + \frac{2}{3}|M_3'| + \max\left\{\frac{2}{3}(|M_1'| - p_1), \frac{1}{3}(p_1 - |M_1'|)\right\} \\
&\quad + \frac{1}{2}|M_2'| + \max\left\{\frac{1}{6}(|M_2'| - p_2), \frac{1}{3}(p_2 - |M_2'|)\right\} \\
&\geq |E| + \frac{1}{2}|M_2'| + \frac{2}{3}|M_3'|,
\end{aligned}$$

where the first inequality uses the above bounds, the first equality stems from $\sum_{i=1}^{\infty} p_i = |M_1'| + |M_2'| + |M_3'|$ and $p_i = ik_i$, the second inequality is by moving parts of the expression to the maxima, and the last inequality is by $\max\{x, -x\} \geq 0$.

□

Lemma 3.20. OPT *can be bounded by*

$$\mathrm{OPT} \geq \frac{|E| + |M_1'| + |M_2'| + |M_3'|}{m}.$$

Proof. The number of trees in the input instance is exactly $|M_1'| + |M_2'| + |M_3'|$. Each tree with k edges contains $k+1$ nodes. Hence, the overall number of nodes in the input forest is $|E| + |M_1'| + |M_2'| + |M_3'|$. As all nodes are adjacent to some edge and using the edge constraint, the claim directly follows. □

Corollary 3.21. OPT *can be bounded by*

$$\mathrm{OPT} \geq \max\left\{ \frac{|E| + \frac{1}{2}|M_2'| + \frac{2}{3}|M_3'|}{m-1}, \frac{|E| + |M_1'| + |M_2'| + |M_3'|}{m}, \sum_{e \in E} r_e \right\}.$$

Proof. The first two bounds follow from Lemmas 3.19 and 3.20. The last bound follows from the fact that the optimal solution cannot pack overfull bins. □

Lemma 3.22. A_2' *produces a preliminary packing with*

$$A_2' \leq \min\left\{1.8, \frac{1.5m}{m-1}\right\} \mathrm{OPT} + 2.5$$

bins. It runs in time $O(|V| \log |V|)$.

Proof. Let $E_1 := E' \setminus E_2$ where E' denotes the set of E together with the virtual items added in A_2'. We first show that $|E_1| + (m-1) \geq |E_2|$. Whenever items are added to E_2 in Line 20, a bin is packed with $m-1$ items not belonging to E_2 in Line 19. Additionally, in this case at most $m-1$ items are added to E_2 because of the following reasoning. Assume to the contrary that more than $m-1$ items are added. This could only happen if $|T_{v_b}| + \ldots + |T_{v_i}| \geq 2m-1$ holds in Line 17 at some point during the execution of the algorithm, since G' contains at most $|T_{v_b}| + \ldots + |T_{v_i}|$ items and $m-1$ items are packed in Line 19. In this case $|T_{v_i}| \geq m$ needs to hold. However, then there would have been an earlier iteration of the for-loop in Line 11 in which $|T_{v_i}|$ was removed or $|T_{v_i}|$ became smaller than m, which is a contradiction. Consequently, $|E_1| \geq |E_2|$ holds before the execution of Line 23 and then at most $m-1$ items may be added to E_2, yielding $|E_1| + (m-1) \geq |E_2|$.

Also, it is always possible to pack the items from E_2 in $\left\lceil \frac{|E_2|}{\lfloor m/2 \rfloor} \right\rceil \leq \left\lceil \frac{|E_2|}{(m-1)/2} \right\rceil$ bins in Line 23 since $\lfloor m/2 \rfloor$ items can be incident to at most m nodes and, thus, the edge constraint is met.

Therefore, we obtain

$$\begin{aligned} A_2' &\leq \left\lceil \frac{|E_1|}{m-1} \right\rceil + \left\lceil \frac{|E_2|}{(m-1)/2} \right\rceil \leq \frac{|E_1|}{m-1} + \frac{1.5|E_2|}{m-1} + \frac{0.5|E_2|}{m-1} + 2 \\ &\leq \frac{|E_1|}{m-1} + \frac{1.5|E_2|}{m-1} + \frac{0.5(|E_1| + (m-1))}{m-1} + 2 \leq \frac{1.5|E'|}{m-1} + 2.5. \end{aligned}$$

We have exactly $\frac{1}{2}|M_2'| + \max\{0, |M_3'| - 1\}$ virtual edges, hence

$$\begin{aligned} A_2' &\leq \frac{1.5(|E| + \frac{1}{2}|M_2'| + \max\{0, |M_3'| - 1\})}{m-1} + 2.5 \\ &\leq \frac{\frac{1.5\left(|E| + 1/2|M_2'| + \max\{0, |M_3'| - 1\}\right)}{m-1}}{\max\left\{ \frac{|E| + \frac{1}{2}|M_2'| + \frac{2}{3}|M_3'|}{m-1}, \frac{|E| + |M_1'| + |M_2'| + |M_3'|}{m} \right\}} \cdot \mathrm{OPT} + 2.5. \end{aligned}$$

For the two values in the denominator, we separately derive bounds. Regarding the first expression, we bound

$$\begin{aligned} A'_2 &\le \frac{\frac{1.5\left(|E|+\frac{1}{2}|M'_2|+\max\{0,|M'_3|-1\}\right)}{m-1}}{\frac{|E|+\frac{1}{2}|M'_2|+\frac{2}{3}|M'_3|}{m-1}} \cdot \mathrm{OPT} + 2.5 \qquad (3.1)\\ &\le \frac{1.5\left(|E|+\frac{1}{2}|M'_2|+\frac{1}{5}(|E|+\frac{1}{2}|M'_2|)+\frac{4}{5}|M'_3|\right)}{|E|+\frac{1}{2}|M'_2|+\frac{2}{3}|M'_3|} \cdot \mathrm{OPT} + 2.5\\ &= 1.8\mathrm{OPT} + 2.5, \end{aligned}$$

where the first inequality uses $|M'_3| \le |E| \le |E| + \frac{1}{2}|M'_2|$ and the second inequality is by factoring out $\frac{6}{5}$ in the numerator.

For the second bound, we have

$$\begin{aligned} A'_2 &\le \frac{\frac{1.5\left(|E|+1/2|M'_2|+|M'_3|\right)}{m-1}}{\frac{|E|+|M'_1|+|M'_2|+|M'_3|}{m}} \cdot \mathrm{OPT} + 2.5\\ &\le \frac{1.5m}{m-1} \cdot \mathrm{OPT} + 2.5, \end{aligned}$$

which yields

$$A'_2 \le \min\left\{1.8, \frac{1.5m}{m-1}\right\} \cdot \mathrm{OPT} + 2.5.$$

Concerning the runtime of A'_2 one can see that it can be implemented such that it runs in $O(|V|\log|V|)$ time. In a preprocessing step we can root the tree (if necessary) and compute the values $|T_{v_i}|$ and the depth of all nodes by applying a depth-first search, which takes $O(|V|)$ time. Then, we essentially visit each node twice (once in the two upper level loops and once in the loop in Line 16) and the overall runtime is dominated by sorting nodes in Line 14. Hence, A'_2 has a runtime of $O(|V|\log|V|)$ (as each node is only sorted once). □

Corollary 3.23. *For a single tree, A'_2 produces a preliminary packing with*

$$A'_2 \le 1.5\mathrm{OPT} + 2.5.$$

Proof. For a single tree, we have $|M'_2| = 0$ and $|M'_3| \le 1$. Using this in Inequality (3.1), the result directly follows.

□

Given a solution of A'_2, we can simply transform it into a feasible packing by reallocating (parts of) items into new bins such that no capacity constraint is violated. To this end, algorithm A_2 considers each overfull bin B and greedily takes (parts of) items of overall size one out of B and places them in new bins until B's capacity constraint is met. By $\mathrm{OPT} \ge \sum r_e$ from Corollary 3.21, we have the following theorem.

Theorem 3.24. *A_2 has an asymptotic approximation factor of 2.5 if the graph G is a single tree and $\min\{1.8, \frac{1.5m}{m-1}\} + 1$ if G is a forest.*

3.4.2 Packing General Graphs

Our algorithm for general graphs is based on a decomposition of graphs into forests, which are then packed using our algorithms presented before. Precisely, it decomposes G into forests $F_1, \dots, F_\ell$, $\ell \leq \text{arb}(G)$, and (possibly) an additional graph G' of degree two. As mentioned before in Section 3.1.1 this is possible in polynomial time. We then pack each forest F_i separately using algorithm A'_2 and (if it exists) we pack G' using algorithm A_1. Finally we transform the solution into a feasible packing as in A_2.

Theorem 3.25. *Algorithm A constructs a packing with at most*

$$\left(\left(\min\left\{1.8, \frac{1.5m}{m-1}\right\}\right) \cdot (arb(G)+1)+1\right) \cdot \text{OPT} + 2.5arb(G) + 1$$

bins. It has an asymptotic approximation ratio of at most

$$\min\left\{1.8, \frac{1.5m}{m-1}\right\} \cdot \left(arb(G) + \frac{5}{3}\right).$$

Proof. Transforming the solution of A'_2 to a feasible packing by reallocating (parts of) items into new bins such that no capacity constraint is violated leads to at most $\sum_{e \in \bigcup_{i=1}^{\ell} E_{F_i}} r_e$ additional bins. Let A'_1 be either A_1 for $m > 3$ or OPT for $m \leq 3$ (as this can be solved efficiently, see Section 3.3). From Lemmas 3.12 and 3.22, we have

$$\begin{aligned}
A &\leq A'_1 + A'_2 + \sum_{e \in \bigcup_{i=1}^{\ell} E_{F_i}} r_e \\
&\leq \left(\min\left\{\frac{16}{9}, \frac{4}{3} \cdot \frac{m}{m-1}\right\} \cdot \text{OPT} + \left(1 - \frac{1}{2(m-1)}\right) \sum_{e \in E_{G'}} r_e + 1\right) \\
&\quad + \left(\min\left\{1.8, \frac{1.5m}{m-1}\right\} \cdot \text{OPT} + 2.5\right) \text{arb}(G) + \sum_{e \in \bigcup_{i=1}^{\ell} E_{F_i}} r_e \\
&< \left(\min\left\{1.8, \frac{1.5m}{m-1}\right\} \cdot \text{OPT}\right) (\text{arb}(G)+1) \\
&\quad + \sum_{e \in \left(\bigcup_{i=1}^{\ell} E_{F_i}\right) \cup E_{G'}} r_e + 2.5\text{arb}(G) + 1 \\
&\leq \left(\left(\min\left\{1.8, \frac{1.5m}{m-1}\right\}\right) \cdot (\text{arb}(G)+1) + 1\right) \cdot \text{OPT} + 2.5\text{arb}(G) + 1
\end{aligned}$$

$$\leq \left(\left(\min\left\{1.8, \frac{1.5m}{m-1}\right\}\right) \cdot \left(\mathrm{arb}(G) + \frac{5}{3}\right)\right) \cdot \mathrm{OPT} + 2.5\mathrm{arb}(G) + 1,$$

where we use $\min\left\{\frac{16}{9}, \frac{4}{3} \cdot \frac{m}{m-1}\right\} \leq \min\left\{1.8, \frac{1.5m}{m-1}\right\}$ in the third inequality, the communication bound on OPT in the second last and $1.5 \leq \min\left\{1.8, \frac{1.5m}{m-1}\right\}$ in the last inequality.

□

CHAPTER 4

Multiprocessor Scheduling with a Sharable Resource

Multiprocessor scheduling is a classical resource allocation problem. In its simplest version, a computing system consisting of m identical processors has to execute n independent jobs of possibly different workloads. The objective is to find an assignment of jobs to processors that minimizes some quality of service measure such as the makespan (latest completion time of any job) or average completion time (the average time a job has to wait for its completion). Specific results differ widely depending on additional model parameters: Is preemption (pausing and resuming jobs) allowed? Can jobs be migrated from one to another processor? Is there any additional knowledge about the jobs (such as size, priority, or dependencies)? Leung [Leu04] gives a good overview of these and many more.

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

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

The first part of this chapter studies the above model for the objective of minimizing the makespan. We refer to this problem as ***S**hared Resource J**o**b-**S**cheduling* (SoS) (see Section 4.1.1 for the full, formal specification). In the second part, we extend this model to the setting of *composed services*, where the processors have to finish a set of *tasks* and each task itself consists of a set of jobs (each of which has its own resource requirement). A task is finished when all its jobs are finished. We aim at minimizing the average completion time of all tasks. This is a typical setting in cloud computing, where users submit applications (tasks) composed of many smaller parts (jobs) and require the output of all these parts. We refer to this setting as ***S**hared Resource T**a**sk-**S**cheduling* (SaS).

4.1 Preliminaries

In the following, we give an introduction to the model and summarize the results presented in this chapter.

4.1.1 Model & Notation

Consider a system of $m \in \mathbb{N}$ processors from the set $M := [m] = \{1, 2, \ldots, m\}$ and $n \in \mathbb{N}$ *jobs* from the set $J := [n]$. There is a *resource* that is to be shared by the processors. In each time step $t \in \mathbb{N}$, each processor i is assigned a share $R_i(t) \in [0, 1]$ of the resource. The resource may not be overused, such that we require $\sum_{i \in [m]} R_i(t) \leq 1$. Each processor can process at most one job per time step and each job can be processed by at most one processor. A job j has a *processing volume (size)* $p_j \in \mathbb{R}$ and a *resource requirement* $r_j > 0$. Note that we will assume $p_j \in \mathbb{N}$ for convenience throughout this paper, but all our results carry over to $p_j \in \mathbb{R}$ (see also the explanation below Equation (4.1)). Without loss of generality, we assume $r_1 \leq r_2 \leq \cdots \leq r_n$. The resource requirement specifies what portion of the resource is needed to finish one unit of a job's processing volume. More exactly, assume job j is processed by processor i during time step t. Then exactly $\min(R_i(t)/r_j, 1)$ units of j's processing volume are finished during that time step. A job is *finished* once all p_j units of its processing volume have been finished. Preemption and migration of jobs is not allowed. The objective is to find a *schedule* S (i.e., a resource and job assignment) having minimal *makespan* $|S|$ (the number of time steps until all jobs are finished). We refer to this problem as ***S**hared Resource J**o**b-**S**cheduling* (SoS).

As a special case, we sometimes consider $p_j = 1$ for all $j \in J$. We refer to this as the setting of jobs with unit size.

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

Recall that a polynomial-time algorithm A has an *(absolute) approximation ratio* of α if, on any instance I, the schedule S produced by A satisfies $|S|/|\mathrm{OPT}| \leq \alpha$, where OPT denotes an optimal solution for I. A has an *asymptotic ratio* of α if, on any instance, $|S| = \alpha \cdot |\mathrm{OPT}| + \mathrm{o}(|\mathrm{OPT}|)$.

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

$$|\mathrm{OPT}| \geq \max\left\{ \lceil s_0(J) \rceil, \frac{1}{m} \cdot \sum_{j \in J} \left\lceil \frac{s_j}{r_j} \right\rceil \right\}. \tag{4.1}$$

Note that these lower bounds on OPT remain valid if allowing $p_j \in \mathbb{R}$ and rescaling $p'_j := \lceil p_j \rceil$ and $r'_j := s_j/p'_j$, as this modification maintains the s_j and by $\lceil p'_j \rceil = \lceil p_j \rceil$ the bound in Equation (4.1) remains the same. Also, the lower bounds remain valid for the preemptive setting as they are only based on observations of the overall workload.

4.1.2 Contribution

We study a new scheduling model for a setting of parallel processors sharing a common scarce resource in terms of its complexity and approximations. Our model is an extension of a simpler variant studied in [Bri+14] and is closely related to a well-known bin packing problem [Chu+06]. Precisely, our results are as follows:

- We prove SoS and SaS to be NP-hard in the strong sense (Section 4.2).
- For SoS, we design and analyze a polynomial-time algorithm with an approximation ratio of $2 + 1/(m-2)$ for jobs of arbitrary size and (asymptotically)

$1 + 1/(m-1)$ for unit size jobs (Section 4.3). Our algorithm is based on the idea of a *maximal sliding window*: We order jobs by non-decreasing resource requirement and (for each time step) create a sliding window trying to find a subset of consecutive jobs such that $m-1$ of these jobs can be finished and the full resource can be used.

- Our algorithm implies the same (asymptotic) guarantee of $1 + 1/(k-1)$ for bin packing with splittable items and cardinality constraint k. Besides a known PTAS, which has a quite high runtime, the best known fast algorithm for this problem has an approximation ratio of $2 - 1/(k-1)$. For computing centers typically containing a huge amount of processors, this ratio approaches 2, whereas the ratio of our algorithm approaches 1.
- We generalize our algorithm to obtain an asymptotic approximation ratio of $2 + 4/(m-3)$ for SAS where unit size jobs are grouped into tasks and where we aim at minimizing the average completion time of all tasks (Section 4.4).

4.2 Complexity

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

Theorem 4.1. *The* SOS *problem with jobs of unit size is strongly NP-hard for* $m = 2$.

Proof. Bin packing with cardinality constraints and splittable items is equal to our setting with preemption. The NP-hardness of SOS, even for unit size jobs, can hence be shown similarly to the reduction found in [Chu+06]. For completeness sake and to show its adaptivity to our setting, the reduction is included here.

We start with the 3-PARTITION problem with a restricted size of the elements which is defined as follows. Given a multiset $S = \{ s_1, \dots, s_n \}$ of $n = 3m$ elements, a bound B with $B/4 < s_i < B/2$ for all s_i and $\sum s_i = mB$, is there a partition into m sets $S_1, \dots, S_m$ such that $|S_i| = 3$ and $\sum_{s \in S_i} s = B$ for all $i \in \{ 1, \dots, m \}$?

Let our SOS instance have n jobs $J = \{ j_1, \dots, j_n \}$ with resource requirement $r_j = \frac{1}{2} + \frac{s_j}{2B} < \frac{3}{4}$, where we abuse notation via $s_{j_i} = s_i$. We define J_i to contain the jobs induced by set S_i. Now is there a schedule with a makespan of at most $2m$?

In case that the 3-PARTITION instance is a YES-instance, we need to show that there is a corresponding schedule for our SOS instance. Given a set S_k, the three jobs derived from it can always be processed in at most two timesteps, because $\sum_{j \in J_k} r_j = \frac{3}{2} + \sum_{j \in J_k} \frac{s_j}{2B} = 2$. Finishing all job sets J_k in an arbitrary order yields a valid schedule with a makespan of $2m$.

On the other hand, we show that if there is a schedule with a makespan of at most $2m$, we can also generate a 3-PARTITION instance. Note that this is equivalent to showing that if the 3-PARTITION instance is a NO-instance, there is no schedule with makespan at most $2m$.

First observe that the resource must be fully utilized in any timestep, because $\sum_{j\in J} r_j = \sum_{j\in J}\left(\frac{1}{2}+\frac{s_j}{2B}\right) = \frac{n}{2}+\frac{mB}{2B} = 2m$.

Then, any job must be processed during at most two timesteps. If this were not the case, due to $r_j < \frac{3}{4}$, there is a timestep t where the share of the resource of job j is less than $\frac{1}{4}$. However, since for the job j' on the second machine we have $r_{j'} < \frac{3}{4}$, the resource at this timestep cannot be fully used.

Now there must be exactly m jobs processed during two timesteps. This is because if there were more than m jobs with this property, we would have more than $4m$ job parts yielding a makespan of more than $2m$. On the other hand, if there were less than m jobs with this property, there is at least one timestep where only one job is processed, which is a contradiction to fully utilizing the resource in every time step.

W.l.o.g., let $\tilde{J} = \{\, j_1, \dots, j_m \,\}$ be the jobs processed during two timesteps. We observe that no two jobs from $\tilde{J}$ can be processed at the same timestep. Otherwise, at least one of them uses a resource of at least $\frac{1}{2}$ in that timestep, hence a resource of at most $\frac{1}{4}$ in the adjacent timestep. Then, the resource is not fully utilized in that adjacent timestep, as the other job uses less than a resource of $\frac{3}{4}$.

Now, for each job split into two parts, we know that exactly one full other job is processed together with each part of it. For each split job, we build a set together with these two jobs processed together with it. There are m such sets $J_1, \dots, J_m$ with three elements each. The sum of the resource requirements of these three jobs is 2, implying $\sum_{j\in J_i}\left(\frac{1}{2}+\frac{s_j}{2B}\right) = 2$ which yields $\sum_{j\in J_i} s_j = B$. This is a 3-Partition. $\square$

Note that the hardness of the general SoS problem (with jobs of arbitrary size) directly follows. This also holds for the SaS problem, as it contains the SoS problem with unit size jobs as a special case.

As stated before, there is a PTAS [ES07] for bin packing with cardinality constraints and splittable items if the cardinality constraint (corresponding to the number of processors in our model) is in $\mathrm{o}(n)$. This bin packing variant is similar to the unit size version of our problem, but with preemption. However, this PTAS can be adapted easily to the setting without preemption by restricting the set of solutions to non-preemptive schedules. For unit size jobs, this implies a better approximation ratio than our algorithm in Section 4.3, but at the cost of very high runtime.

4.3 Approximation Algorithm

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

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

Definition 4.2 (Job Window)**.** A subset of unfinished jobs $W \subseteq J(t-1)$ is called a *job window* for time step t if

1. $j_1, j_2 \in W \Rightarrow J(t-1) \cap \{ j_1, j_1+1, \dots, j_2 \} \subseteq W$,
2. $r(W \setminus \{ \max W \}) < 1$,
3. $|\{ j \in W \mid q_j(t-1) > 0 \}| \leq 1$, and
4. $j \in J(t-1) \setminus W \Rightarrow s_j(t-1) = s_j$.

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

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

In other words, a window W (of size $\leq m$) is a set of consecutive jobs (Property 1) such that we can assign all but the rightmost job their full resource requirements (Property 2). Moreover, W contains all started jobs and at most one of these is fractured (Properties 3 and 4). To be k-maximal, a window of a size of at most k must contain either exactly k jobs or lie at the left border, and either utilize the full resource or lie at the right border (Properties 5 and 6).

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

- During any time step t, it processes jobs from an $(m-1)$-maximal window $W_t \subseteq J(t-1)$ (Lemma 4.8).
- If the window W_t is at the left border of the remaining jobs (i.e., $L_t(W_t) = \emptyset$), then this remains true for all $W_{t'}$ with $t' > t$ (Lemma 4.91).
- If the window W_t is at the right border of the remaining jobs (i.e., $R_t(W_t = \emptyset)$), then this remains true for all $t' > t$ (Lemma 4.92).

Note that if W_t is *not* at the left border of the remaining jobs, Properties 2 and 5 of Definition 4.2 imply that we can assign the resource such that at least $m-2$ jobs (all of W except for $\max W_t$) receive their full resource requirement r_j during time

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

Listing 4.1: Approximation algorithm for SoS.

step t. Similarly, if W_t is not at the right border of the remaining jobs, Property 6 implies that we can utilize the full resource during time step t.

Consider the first time step T such that $L_T(W_T) \cup R_T(W_T) = \emptyset$. In particular, W_T contains all remaining jobs. It is not hard to see that these can be finished by our algorithm in $|\text{OPT}|$ time steps. On the other hand, up to time step T the three key properties and the above observations imply that in each time step either at least $m-2$ jobs receive their full resource requirement or the full resource is utilized. In the former case, the lower bound from Equation (4.1) implies $T \leq \frac{m}{m-2} \cdot |\text{OPT}|$. In the latter case, the same bound implies $T \leq |\text{OPT}|$. Together, this yields an approximation ratio of at most $\frac{m}{m-2} + 1 = 2 + \frac{m-2}{m}$.

A slightly more careful but similar analysis yields Theorem 4.4. We proceed to describe our algorithm. Afterward we show that the three key properties hold and formalize the above argument.

4.3.1 Algorithm Description

In the following we describe our algorithm. The corresponding pseudocode can be found in Listing 4.1 (with some auxiliary procedures outsourced to Listing 4.2). If not explicitly stated otherwise, references to lines refer to Listing 4.1. Note that the implementation as shown in Listing 4.1 has only pseudo-polynomial runtime. It is not hard to adapt it such that it yields polynomial runtime; we describe how to do that in the proof of Theorem 4.4.

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

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

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

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

Listing 4.2: Auxiliary procedures. The parameters $size$ and R are only to facilitate the algorithm from Section 4.4. In this section, we call these only with $size = m - 1$ and $R = 1$.

then right, and finally move it as far to the right as possible. This way, W becomes $(m-1)$-maximal for this time step.

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

Case 1: $r(W \setminus F) \geq 1$
Note that $\iota \neq \max W$, as otherwise Property 2 of Definition 4.2 violates the case assumption. Each job $j \in W$ except for ι and $\max W$ receives its full resource requirement r_j. Job ι receives resource $q_\iota(t-1)$. Any remaining resource is assigned to $\max W$.

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

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

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

resource). This case might leave us with some unecessarily wasted resource (if $s_\iota(t-1) = \varepsilon \ll r_\iota$ and $R_t(W) \neq \emptyset$). If so, we finish ι and use the (so far unused) m-th processor to start a new job. We gather this discussion in the following observation.

Observation 4.3. Given an $(m-1)$-maximal window W for the current time step, Lines 7 to 20 compute a resource assignment for jobs in W such that at least $|W|-1$ jobs $j \in W$ receive their full resource requirement r_j, at most one job is fractured after this time step, and at most $|W|$ jobs are started (and not finished) after this time step.

4.3.2 Analysis

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

Theorem 4.4. *The algorithm from Listing 4.1 generates a schedule S with approximation ratio $2 + \frac{1}{m-2}$. If jobs have unit size, we get the stronger guarantee $|S| \leq (1 + \frac{2}{m-2}) \cdot |\mathrm{OPT}| + 1$. The algorithm can be implemented with a runtime of $\mathrm{O}((m+n) \cdot n)$.*

It is not hard to see that for jobs of unit size, a minor algorithm modification avoids to reserve the m-th processor: If jobs have unit size, we have $s_j = r_j$ for all $j \in J$. Note that there will be always at most one started (and thus at most one fractured) job: Indeed, by the while-loops of the auxiliary procedures, the window can contain at most one job with $s_j = r_j > 1$ (this will be $\max W$). Since for all jobs $j \in W \setminus \{ \max W \}$ we have $s_j = r_j \leq 1$ and $r(W \setminus \{ \max W \}) < 1$ (Property 2 of Definition 4.2), such j will be finished in the current time step. We can treat the only started job ι in step t as a job with resource requirement $s_\iota(t-1)$ and reorder the jobs accordingly. The next time step will either finish ι or it will once more be the only started job. This modification does not need the reserved processor, so we can use m-maximal instead of $(m-1)$-maximal windows, improving the approximation factor for unit size jobs from $\frac{m}{m-2} = 1 + \frac{2}{m-2}$ to $\frac{m}{m-1} = 1 + \frac{1}{m-1}$. The analysis is analogous to the one given below for the unmodified algorithm.

We now start to provide tools for the proof of Theorem 4.4. We start with some auxiliary claims and then prove the above-mentioned key properties in Lemmas 4.8 and 4.9

Claim 4.5. *If Properties 1 to 4 from Definition 4.2 hold for W right before we call the auxiliary procedures, then they also hold at any later point in this time step.*

Proof. Property 1 holds since jobs are added one by one at the left/right borders (Lines 3 and 8 in Listing 4.2) or one job is removed at the left border and another added at the right border (Line 13). Property 2 is enforced by the while-loops' conditions. Property 3 holds since only unstarted (and thus unfractured) jobs are added to W. Finally, Property 4 holds since the while-loop in Line 12 of Listing 4.2 ensures that no started jobs are removed. □

Claim 4.6. *If $W = \emptyset$ after Line 2 of Listing 4.1 in time step t and no job in $J(t-1)$ is started, then W is an $(m-1)$-maximal window when `MoveWindowRight` exits.*

Proof. We have $W = \emptyset$ right before the auxiliary procedures are called. In particular, W is a (trivial) window for time step t. We apply Claim 4.5 to get that Properties 1 to 4 of Definition 4.2 hold when `MoveWindowRight` exits. Since the while-loops ensure that the window size is at most $m-1$, it remains to show that Properties 5 and 6 hold after the auxiliary procedures.

For Property 5, note that $L_t(W) = L_t(\emptyset) = \emptyset$. Thus, procedure `GrowWindowLeft` exits immediately, leaving $W = \emptyset$. If `GrowWindowRight` exits because of $|W| = m-1$, Property 5 holds (and remains true since `MoveWindowRight` does not change the size of W). Otherwise, if `GrowWindowRight` exits because the condition "$r(W) < 1 \wedge R_t(W) \neq \emptyset$" is violated, `MoveWindowRight` exits immediately for the same reason. But then, we still have $\min J(t-1) \in W$ (impplying $L_t(W) = \emptyset$) and Property 5 holds.

For Property 6, note that `MoveWindowRight` cannot exit because of the condition "$s_{\min W} = s_{\min W}(t-1)$" (there are no started jobs). Thus, it can only exit because one of the other two conditions is violated, which immediately implies Property 6. ☐

Claim 4.7. *If $W \neq \emptyset$ after Line 2 of Listing 4.1 in time step t and the window $\tilde{W}$ computed in the previous time step was $(m-1)$-maximal, then W is a $(m-1)$-maximal window when `MoveWindowRight` exits.*

Proof. We have $W = \tilde{W} \cap J(t-1)$ right before the auxiliary procedures are called. $\tilde{W}$ was a maximal window, and removing finished jobs cannot violate Properties 1 to 4 of Definition 4.2. We apply Claim 4.5 to get that Properties 1 to 4 hold when `MoveWindowRight` exits. It remains to show that Properties 5 and 6 hold after the auxiliary procedures.

For Property 5, we first show that it holds after `GrowWindowLeft`. When we call `GrowWindowLeft` for window W, note that $L_t(W) = L_{t-1}(\tilde{W})$. Thus, if $L_{t-1}(\tilde{W}) = \emptyset$, Property 5 holds trivially after `GrowWindowLeft` (the while-loop exits immediately because of the condition "$L_t(W) \neq \emptyset$"). If $L_{t-1}(\tilde{W}) \neq \emptyset$, since Property 5 holds for window $\tilde{W}$, we have $|\tilde{W}| = m-1$. Note that for all $j \in L_t(W) = L_{t-1}(\tilde{W})$ and $j' \in \tilde{W}$ we have $r_j \leq r_{j'}$ (by the job ordering). This implies that we cannot violate condition "$r(W) < 1$" of the while-loop of `GrowWindowLeft` before adding $|\tilde{W}| - |W|$ jobs. Moreover, we cannot violate "$|W| \leq m-1$" before adding $|\tilde{W}| - |W|$ jobs (since $|W| + (|\tilde{W}| - |W|) = |\tilde{W}| \leq m-1$). Thus, `GrowWindowLeft` adds at least $\min\{\,|L_t(W)|, |\tilde{W}| - |W|\,\}$ jobs to W. If the minimum equals $|L_t(W)|$ we added all jobs left of W and Property 5 holds. If the minimum equals $|\tilde{W}| - |W|$, Property 5 holds since the resulting window has a size of at least $|W| + (|\tilde{W}| - |W|) = |\tilde{W}| = m-1$.

So Property 5 holds for W right before `GrowWindowRight`. We show that it still holds after procedure `MoveWindowRight`. The statement is trivial if $|W| = m-1$ (both procedures do not decrease W). Otherwise, we use that W has Property 5 to get $L_t(W) = \emptyset$. If `GrowWindowRight` exits because the condition

"$r(W) < 1 \land R_t(W) \neq \emptyset$" got violated, `MoveWindowRight` exits immediately for the same reason, leaving $L_t(W) = \emptyset$. Otherwise, if `GrowWindowRight` exits because of $|W| = m - 1$, this is maintained by `MoveWindowRight` and, thus, Property 5 holds after `MoveWindowRight`.

It remains to prove that Property 6 holds after procedure `MoveWindowRight`. Consider the conditions of the while-loop in Line 12 of Listing 4.2. Property 6 holds if the while-loop exits because the condition "$r(W) < 1 \land R_t(W) \neq \emptyset$" got violated. So assume it exits only because of the condition "$s_{\min W} = s_{\min W}(t - 1)$". At that moment, we have a window W with $r(W) < 1$, $R_t(W) \neq \emptyset$, and $s_{\min W} > s_{\min W}(t - 1)$. The first two imply that $|W| = m - 1$, since otherwise `GrowWindowRight` would not have exited. The inequality $s_{\min W} > s_{\min W}(t - 1)$ implies that job $\min W$ is already started, so it must have been in the last time step's window $\tilde{W}$. Now, since W has maximal size $m - 1$ and its leftmost job was also in $\tilde{W}$, we get $r(\tilde{W}) \leq r(W) < 1$ as well as $R_t(W) \subseteq R_{t-1}(\tilde{W})$. But since $\tilde{W}$ had Property 6, we know $R_{t-1}(\tilde{W}) = \emptyset$. Together, $R_t(W) = \emptyset$, a contradiction. □

Lemma 4.8. *Fix $t \in \mathbb{N}_0$ and consider the job window W processed during time step t. Then W is an $(m - 1)$-maximal window for time step t.*

Proof. We prove the statement inductively. In the first time step $t = 1$, we start with $W = \emptyset$ (initialization by the for-loop) and no job has been started. We apply Claim 4.6 to get that W is an $(m-1)$-maximal window after the auxiliary procedures. For $t > 1$ we either have $W = \emptyset$ or $W \neq \emptyset$ after Line 2 of Listing 4.1. In the former case, we once more apply Claim 4.6. In the latter case, we apply Claim 4.7. In both cases, we get that W is an $(m - 1)$-maximal window after the auxiliary procedures, proving the desired statement. □

Lemma 4.9. *Let $\tilde{W} \subseteq J(t-2)$ and $W \subseteq J(t-1)$ be the $(m-1)$-maximal windows processed during time step $t - 1$ and t, respectively. Then*

1. $L_{t-1}(\tilde{W}) = \emptyset \Rightarrow L_t(W) = \emptyset$ *and*
2. $R_{t-1}(\tilde{W}) = \emptyset \Rightarrow R_t(W) = \emptyset \land r(W) \leq r(\tilde{W})$.

Proof. For Statement 1, note that W starts out as $\tilde{W} \cap J(n - 1)$ in time step t. Since $L_t(W) = L_{t-1}(\tilde{W}) = \emptyset$, we only add jobs from $R_t(W) = R_{t-1}(\tilde{W})$. All these jobs have a larger resource requirement than any job in $\tilde{W}$. As a consequence, after `GrowWindowRight` we have $|W| \leq |\tilde{W}|$. If $|W| < |\tilde{W}| \leq m - 1$, `MoveWindowRight` exits immediately and we have $L_t(W) = \emptyset$. Otherwise, if $|W| = |\tilde{W}|$ after `GrowWindowRight`, we must have $r(W) \geq r(\tilde{W})$ and $R_t(W) \subseteq R_{t-1}(\tilde{W})$. Since $\tilde{W}$ is $(m-1)$-maximal in time step $t-1$, this implies either $r(W) \geq 1$ or $R_t(W) = \emptyset$, such that `MoveWindowRight` exits immediately and leaves $L_t(W) = \emptyset$. This proves Statement 1. The first part of Statement 2 follows analogously. The second part holds either since $|W| = |\tilde{W}| = m - 1$ and jobs that were finished in $\tilde{W}$ are exchanged for jobs with at most the same resource requirement, or since $W \subseteq \tilde{W}$ (if $L_{t-1}(\tilde{W}) = \emptyset$). □

With these lemmas, we are ready to prove Theorem 4.4.

Proof of Theorem 4.4. We consider the schedule S produced by our algorithm from Listing 4.1. By Lemma 4.8, the jobs processed during each time step $t \in \mathbb{N}$ are contained in a maximal window W_t for time step t. We define $T_L := \min\{\, t \in \mathbb{N} \mid |W_t| < m-1 \,\}$ and, similarly, $T_R := \min\{\, t \in \mathbb{N} \mid r(W_t) < 1 \,\}$. By Properties 5 and 6 of Definition 4.2 and Lemma 4.9 we have $L_t(W_t) = R_t(W_t) = \emptyset$ and $r_t(W_t) < 1$ for all $t \geq \max\{\, T_L, T_R \,\} =: T$. In particular, the former implies $W_t = J(t-1)$ for all $t \geq T$. Combining these insights we get that for each $t \geq T$, each of the at most $|W_t| \leq |W_T| < m-1$ remaining jobs gets its full resource requirement. Thus, each $j \in W_T$ is finished after exactly $\lceil s_j(T-1)/r_j \rceil$ additional time steps. Let $p := \max\{\, s_j(T-1)/r_j \mid j \in W_T \,\}$. Note that $|S| = T - 1 + \lceil p \rceil$. We distinguish two cases:

Case 1: $T = T_L$

For each $t < T$ we have $|W_t| = m-1$. Thus, by Observation 4.3, at least $|W_t| - 1 = m-2$ jobs $j \in W_t$ receive their full resource requirement r_j. Remember that $p_j = s_j/r_j$. An average argument gives

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

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

$$\begin{aligned} |S| = T - 1 + \lceil p \rceil &\leq |\mathrm{OPT}| \cdot \frac{m}{m-2} - \frac{\lceil p \rceil}{m-2} + \lceil p \rceil \\ &\leq |\mathrm{OPT}| \cdot \left(\frac{m}{m-2} + 1 - \frac{1}{m-2} \right) \\ &= |\mathrm{OPT}| \cdot \left(2 + \frac{1}{m-2} \right). \end{aligned}$$

Case 2: $T = T_R$

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

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

For the runtime, first note that the implementation given in Listing 4.1 has actually pseudo-polynomial runtime (it depends on the sum $\sum_{j \in J} p_j$, since each job j needs a dedicated processor for at least p_j time steps). However, note that if no job is finished in the current time step, the maximal window in the next step will be identical to the current maximal window. With this observation, we can calculate via a simple linear equation after how many step with the current maximal window the first job in the window will be finished. This allows us to “skip” time steps where no job is finished. Thus, given the maximal window in a time step t, we go over the $\mathrm{O}(m)$ jobs in the window and find the first one(s) that will be finished under the current resource assignment. To compute the next maximal window, we remove the finished jobs and grow the window left/right. This can be computed in time $\mathrm{O}(|W|) = \mathrm{O}(m)$ (each adding/removal can be implemented trivially in constant time using doubly linked lists). Then we move the window up to n steps to the right, which can be done in time $\mathrm{O}(n)$. Since this always eliminates at least one job from the old maximal window, this repeats at most $\mathrm{O}(n)$ times, yielding a total runtime of $\mathrm{O}(n \cdot (m+n))$. □

As the lower bounds on OPT are still valid for the preemptive setting (see description below Equation (4.1)), and the upper bounds of the algorithm obviously do not increase by allowing preemption, our results for unit size jobs carry over to bin packing with cardinality constraints and splittable items. Our algorithm scales well with the number of processors in contrast to existing simple (i.e., fast) algorithms, but (obviously) does not reach the approximation ratio of the existing EPTAS [ELS12]. Note that in the following corollary, k denotes the cardinality constraint as this is common notion in the related literature.

Corollary 4.10. *Our results give an algorithm for bin packing with cardinality constraints and splittable items [Chu+06] with asymptotic approximation ratio* $1 + {}^1\!/_{(k-1)}$ *and runtime* $\mathrm{O}((k+n)n)$.

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

The next results also follow almost directly from the results in this chapter. The first result improves upon the prior approximation ratio for graphs with tree structure, the second considers general graphs.

Corollary 4.11. *A simple variant of our algorithm results in an approximation guarantee of* $2 + \frac{2}{m-1}$ *for the model from Chapter 3 if the input graph has the structure of a forest.*

Proof. We start with an arbitrary forest and consider each edge separately. As each forest with n nodes consists of at most $2n-1$ edgest (in case the forest is a tree), and any optimal algorithm could schedule at most $n-1$ edges at once (due to the forest structure), we lose at most a factor of $\frac{2n-1}{n-1} \leq 2$ by considering each edge separately. Our algorithm from this chapter has an approximation guarantee of $1 + \frac{1}{m-1}$, resulting in an overall approximation ratio of at most $2 \cdot \left(1 + \frac{1}{m-1}\right)$. □

For the second result, recall that the arboricity $\mathrm{arb}(G)$ of a graph G denotes the minimum number of forests into which a graph can be decomposed. Note that the approximation ratio of the following corollary is better than the results from Chapter 3 if m is large and $\mathrm{arb}(G)$ is small, that is, $\mathrm{arb}(G) < 5$. For small m (that is, $\min\left\{1.8, \frac{1.5m}{m-1}\right\} = 1.8$), it only improves upon the prior result if $\mathrm{arb}(G)$ is even smaller, the threshold depending on the exact value of m.

Corollary 4.12. *Our algorithm can easily be modified to guarantee an approximation ratio of* $({}^{2m}/_{(m-1)})\mathit{arb}(G)$ *for the model from Chapter 3, resulting in an overall approximation ratio of* $\min\left\{\min\left\{1.8, \frac{1.5m}{m-1}\right\} \cdot \left(\mathit{arb}(G) + \frac{5}{3}\right), (\frac{2m}{m-1})\mathit{arb}(G)\right\}$

Proof. We start with an arbitrary graph and consider each edge separately. We know that the graph can be decomposed into $\mathrm{arb}(G)$ many forests. As each forest with k nodes consists of at most $2k-1$ edges (in case the forest is a tree), it also follows that each graph of size n consists of at most $2\mathrm{arb}(G)n$ single edges. Therefore, using our algorithm for unit size jobs on the resulting set of edges yields an approximation ratio of $(2\mathrm{arb}(G)) \cdot (1 + {}^{1}/_{(m-1)}) = {}^{2m}/_{(m-1)} \cdot \mathrm{arb}(G)$. □

4.4 The Shared Resource Task-Scheduling Problem

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

We now consider the model where a set of tasks needs to be executed and where each task consists of multiple unit size jobs, i.e., given a task set $\mathcal{T} = \{ T_1, \dots, T_k \}$ each containing a set of jobs $T_i = \{ j_{i1}, \dots, j_{in_i} \}$ with $p_{ik} = 1$ for all i, k. The objective is to minimize the average completion time, where the completion time f_i of a task T_i denotes the time the last job of this task is finished, i.e., $f_i := \max\{t : s_j(t-1) > 0 \text{ for some } j \in T_i\}$. Note that this equals the objective of minimizing the sum of completion times, which we will do throughout this section.

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

4.4.1 Prerequisites

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

We begin with the tasks that have high resource requirements. Here, the available resource is R instead of 1 as in the previous section. Note that the auxiliary procedures called in the algorithms in Listing 4.3 (Lines 7 to 9) and Listing 4.4 (Lines 8 to 10) are applied only to the currently considered task instead of the whole set of jobs.

```
for $(t, S, W, i) \leftarrow (1, \emptyset, \emptyset, 1)$; $T(t-1) \neq \emptyset$; $t \leftarrow t+1$:
    $m' \leftarrow m$
    while $(\tilde{r}(S) + \tilde{r}(J_i(t-1)) \leq 1)$:
        $S \leftarrow S \cup T_i$; $i \leftarrow i+1$; $m' \leftarrow m' - |J_i(t-1)|$
        process all jobs in $T_i$ with their full resource requirement
    $W \leftarrow W \cap J_i(t-1)$
    $W \leftarrow$ GrowWindowLeft($W$, $t$, $m'$, $1 - \tilde{r}(S)$)
    $W \leftarrow$ GrowWindowRight($W$, $t$, $m'$, $1 - \tilde{r}(S)$)
    $W \leftarrow$ MoveWindowRight($W$, $t$, $1 - \tilde{r}(S)$)

    if $\exists$ fractured job $\iota \in W$:  $F \leftarrow \{ \iota \}$
    else:  $F \leftarrow \emptyset$
    process each job $j \in W \setminus (F \cup \{ \max W \})$ with resource $r_j$
    if $F = \{ \iota \}$:
        process job $\iota$ with resource $q_\iota(t)$
    process job $\max W$ with the remaining resource
```

Listing 4.3: Algorithm for task set $\mathcal{T}_1$.

Lemma 4.13. *For a set of tasks $\mathcal{T} = \{ T_1, \ldots, T_k \}$ with*

$$\frac{r(T)}{|T|} > \frac{R}{(m-1)}$$

for all $T \in \mathcal{T}$, the algorithm in Listing 4.3 computes a schedule such that the completion time f_i of task T_i is

$$f_i \leq \left\lceil \frac{\sum_{l=1}^{i} r(T_l)}{R} \right\rceil .$$

Proof. The algorithm in Listing 4.3 processes tasks by increasing index and proceeds according to the algorithm in Listing 4.1 and Section 4.3 separately for each task.

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

Case 1: First consider the case in which there is no transition between tasks in the current time step $t+1$. As we have $\frac{\tilde{r}(J_i(t))}{|J_i(t) \setminus F|} \geq \frac{R}{m-1}$ and $|F| \leq 1$, the algorithm always finds an m-maximal window using the full resource R in time step $t+1$. By Property 5 from Section 4.3, we have that (i) the windows has size m or (ii) $L_t(W) = \emptyset$. In case (i), $\tilde{r}(J_i(t))$ is reduced by R and $|J_i(t) \setminus F|$ by at least $m-1$, thus $\frac{\tilde{r}(J_i(t+1))}{R} = \frac{\tilde{r}(J_i(t))-R}{R} \geq \frac{|J_i(t) \setminus F|-(m-1)}{m-1} \geq \frac{|J_i(t+1) \setminus F|}{m-1}$. In case (ii), the jobs from $J_i(t)$ with the smallest resource requirement are finished, thus the ratio $\frac{\tilde{r}(J_i(t))}{|J_i(t) \setminus F|} \geq \frac{R}{m-1}$ can only increase. The claim follows.

Case 2: Now consider the case that there is a transition between tasks. That is, there is an arbitrary number of tasks that is finished in Line 3 of Listing 4.3.

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

Listing 4.4: Algorithm for task set $\mathcal{T}_2$.

Those tasks used $m - m'$ processors, hence at least $m - m' - 1$ processors were occupied with full jobs. By induction hypothesis and by the average size of jobs in task set $\mathcal{T}$, at least a resource of $\frac{m-m'-1}{m-1} \cdot R$ was used. Hence, the resource available to the sliding window determined in Lines 7 to 9 is at most $\frac{m'}{m-1} \cdot R$. By Lemma 4.8, we conclude that we computed an m'-maximal window. Now we either have (a) $|W| = m'$ or (b) $|W| < m'$. In case (a), $\tilde{r}(W)$ was reduced by at most $\frac{m'}{m-1} \cdot R$, whereas $|T_i \setminus F|$ was reduced by exactly $|W| = m'$. Hence $\frac{\tilde{r}(J_i(t+1))}{R} \geq \frac{\tilde{r}(J_i(t)) - m'R/(m-1)}{R} \geq \frac{|J_i(t) \setminus F| - m'}{m-1} = \frac{|J_i(t+1) \setminus F|}{m-1}$ in case (b) with $|W| < m'$, we know $L_t(W) = \emptyset$ by Property 5 from Section 4.3, implying that the smallest jobs of the new task were executed. The claim follows, as the average size of jobs in each task is at least $R/m-1$.

□

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

Lemma 4.14. *For a set of tasks $\mathcal{T} = \{T_1, \ldots, T_k\}$ with*

$$\frac{r(T)}{|T|} \leq \frac{R}{(m-1)}$$

for all $T \in \mathcal{T}$, the algorithm in Listing 4.4 computes a schedule such that the completion time f_i of task T_i is

$$f_i \leq \left\lceil \frac{\sum_{l=1}^{i} |T_i|}{m-1} \right\rceil.$$

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

(i) T_i is finished at $f_i \leq \lceil t_i \rceil$.

(ii) The number of processors occupied by tasks $T_1 \ldots, T_i$ in time step $\lceil t_i \rceil$ is at most $m_i := (t_i - (\lceil t_i \rceil - 1))\,(m-1)$.

(iii) Tasks $T_1 \ldots, T_i$ occupy at most a resource of $m_i \cdot \frac{R}{m-1}$ in time step $\lceil t_i \rceil$.

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

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

Case 2: In the case that T_{i+1} is not finished in time step f_i, we will start task T_{i+1} with $m' \geq m - m_i$ processors and allow a resource of at most $(m'-1) \cdot \frac{R}{m-1}$ in this time step (and the full resource in any following non-transitional time step). Since the average resource per full non-fractured job in our sliding window (Lines 8 to 10) is $\frac{R}{m-1}$, we get an analogue statement to Lemma 4.9 from Section 4.3. That is, we will (a) finish $m'-1$ jobs in f_i and $m-1$ jobs in any time step $t \in (f_i, f_{i+1})$ or (b) use the full resource in any time step $t \in (f_i, f_{i+1})$.

In case (a), we have

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

which yields (i). For (ii), the number of occupied processors in time step $\lceil t_{i+1} \rceil$ is at most $\sum_{k=1}^{i+1} |T_k| - (\lceil t_{i+1} \rceil - 1) \cdot (m-1) = m_{i+1}$. For (iii), observe that the average resource of the window is non-increasing by Lemma 4.9, Statement 2. In particular, the resource used at time $\lceil t_{i+1} \rceil$ is at most $m_{i+1} \cdot \frac{R}{m-1}$.

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

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

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

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

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

□

We now give bounds for the optimal algorithm.

Lemma 4.15. *The sum of completion times of the optimal solution can be bounded as follows.*

1. *Given a set of tasks $\mathcal{T} = \{ T_1, \dots, T_k \}$ with $R_l \leq R_{l+1}$ for all l, we have* $\mathrm{OPT}_{\mathcal{T}} \geq \sum_{i=1}^{k} \left\lceil \sum_{l=1}^{i} R_l \right\rceil$.
2. *Given a set of tasks $\mathcal{T} = \{ T_1, \dots, T_k \}$ with $|T_l| \leq |T_{l+1}|$ for all l, we have* $\mathrm{OPT}_{\mathcal{T}} \geq \sum_{i=1}^{k} \left\lceil \sum_{l=1}^{i} \frac{|T_k|}{m} \right\rceil$.

Proof. We first prove Property 1. As the optimal solution cannot overuse the resource, there is obviously an order $\mathcal{T} = \{ T_{\sigma_1}, \dots, T_{\sigma_l} \}$ such that $\mathrm{OPT}_{\mathcal{T}} \geq \sum_{i=1}^{k} \left\lceil \sum_{l=1}^{i} R_{\sigma_l} \right\rceil$. We denote the bounds on the completion times as

$$f_i := \left\lceil \sum_{l=1}^{i} R_l \right\rceil, \quad f'_i := \left\lceil \sum_{l=1}^{i} R_{\sigma_l} \right\rceil.$$

We prove $f_i \leq f'_i$ for all i which directly implies Property 1. Let i be arbitrary. Assume $f_i > f'_i$. Then $\left\lceil \sum_{l=1}^{i} R_l \right\rceil > \left\lceil \sum_{l=1}^{i} R_{\sigma_l} \right\rceil$, hence $\sum_{l=1}^{i} R_l > \sum_{l=1}^{i} R_{\sigma_l}$. This is a contradiction to $R_l \leq R_{l+1}$ for all l.

For Property 2, as the optimal solution cannot finish more than m jobs per time step, there is an order $\mathcal{T} = \{ T_{\sigma_1}, \dots, T_{\sigma_l} \}$ such that $\mathrm{OPT}_{\mathcal{T}} \geq \left\lceil \sum_{l=1}^{i} \frac{|T_{\sigma_l}|}{m} \right\rceil$. Now denote $f_i := \left\lceil \sum_{l=1}^{i} \frac{|T_l|}{m} \right\rceil$, $f'_i := \left\lceil \sum_{l=1}^{i} \frac{|T_{\sigma_l}|}{m} \right\rceil$. Assume $f_i > f'_i$ for some i. Hence $\sum_{l=1}^{i} |T_l| > \sum_{l=1}^{i} |T_{\sigma_l}|$, which is a contradiction. □

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

Lemma 4.16. *Given $z \in \mathbb{N}_{\geq 3}$ and $\{ x_1, \dots, x_k \} \in \textsc{RoundRobin}_{\geq 1/z}$ such that $x_i + 1/z \leq x_{i+1}$ for all $i \in \{ 1, \dots, k-1 \}$, there is a $q \in \mathbb{N}_0$ such that*

$$\sum_{i=1}^{k} \left(\left\lceil \frac{z}{\lfloor (z-1)/2 \rfloor} \cdot x_i \right\rceil - \frac{z}{\lfloor (z-1)/2 \rfloor} \cdot \lceil x_i \rceil \right) \leq q \tag{4.2}$$

and

$$\sum_{i=1}^{k} \lceil x_i \rceil \geq \frac{2}{3} (\sqrt{q} - 2)^3 + (k - q). \tag{4.3}$$

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

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

for all $i \in \{ 1, \dots, k \}$. We choose $q = |E_{>0}|$, so Equation (4.2) follows. We further show that

$$x_i \in \left[l, \frac{\lfloor (z-1)/2 \rfloor}{z} \cdot \left(\left\lceil \frac{(l+1) \cdot z}{\lfloor (z-1)/2 \rfloor} \right\rceil - 1 \right) \right] \text{ for } l \in \mathbb{N}_0 \tag{4.4}$$

implies $err_i \leq 0$. First note that Property 4.4 implies $\lceil x_i \rceil \leq l + 1$. We upper bound

$$\begin{aligned} \left\lceil \frac{z}{\lfloor (z-1)/2 \rfloor} \cdot x_i \right\rceil &\leq \left\lceil \frac{(l+1) \cdot z}{\lfloor (z-1)/2 \rfloor} \right\rceil - 1 \\ &\leq \frac{(l+1) \cdot z}{\lfloor (z-1)/2 \rfloor} = \frac{z}{\lfloor (z-1)/2 \rfloor} \cdot \lceil x_i \rceil . \end{aligned} \tag{4.5}$$

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

$$\left(\frac{\lfloor (z-1)/2 \rfloor}{z} \cdot \left(\left\lceil \frac{l \cdot z}{\lfloor (z-1)/2 \rfloor} \right\rceil - 1 \right), l \right) \tag{4.6}$$

for some $l \in \mathbb{N}$ since otherwise $err_i \leq 0$ by Inequality (4.5). The length of each such interval can be upper bounded by

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

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

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

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

4.4.2 Approximation Algorithm

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

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

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

Lemma 4.17. *Scheduling $\mathcal{T}_1$ using the algorithm in Listing 4.3 with $\lfloor \frac{m}{2} \rfloor$ processors and a resource of $R = \frac{\lfloor m/2 \rfloor - 1}{m-1} < \frac{1}{2}$, there is a $q_1 \in \mathbb{N}_0$ such that the sum of completion times is at most*

$$\left(2 + \frac{4}{m-3}\right) \mathrm{OPT}_{\mathcal{T}_1} + q_1$$

and

$$\mathrm{OPT}_{\mathcal{T}_1} \geq \frac{2}{3}(\sqrt{q_1} - 2)^3 + (k_1 - q_1). \tag{4.7}$$

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

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

by construction of $\mathcal{T}_1$. Assume the tasks $\mathcal{T}_1 = \{ T_1, \ldots, T_{k_1} \}$ are ordered by non-decreasing overall resource requirement (i.e., $r(T_1) \leq r(T_2) \leq \cdots \leq r(T_{k_1})$). Applying Lemma 4.13, we know that the full resource of $\frac{\lfloor m/2 \rfloor - 1}{m-1}$ is used in every time step. Hence, the tasks are scheduled such that the sum of their completion times is

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

From Lemma 4.15, Property 1, we have

$$\mathrm{OPT}_{\mathcal{T}_1} \geq \sum_{i=1}^{k_1} \left\lceil \sum_{l=1}^{i} r(T_j) \right\rceil .$$

Now, using Lemma 4.16 with $x_i := \sum_{l=1}^{i} r(T_j)$ and $z := m - 1$, we conclude that there is a $q_1 \in \mathbb{N}_0$ such that

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

as well as Inequality (4.7), which proves the claim. □

Lemma 4.18. *Scheduling $\mathcal{T}_2$ using the algorithm in Listing 4.4 with $\lceil \frac{m}{2} \rceil$ processors and a resource of $R = \frac{1}{2}$, there is a $q_2 \in \mathbb{N}_0$ such that the sum of completion times is at most*

$$\left(2 + \frac{4}{m-2}\right) \mathrm{OPT}_{\mathcal{T}_2} + q_2$$

as well as

$$\mathrm{OPT}_{\mathcal{T}_2} \geq \frac{2}{3}(\sqrt{q_2}-2)^3 + (k_2 - q_2). \tag{4.8}$$

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

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

by construction of $\mathcal{T}_2$. Assume the tasks $\mathcal{T}_2 = \{ T_1, \ldots, T_{k_2} \}$ are ordered by non-decreasing number of jobs (i.e., $|T_1| \leq |T_2| \leq \cdots \leq |T_{k_2}|$). By Lemma 4.14, we have

$$S_{\mathcal{T}_2} = \sum_{i=1}^{k_2} \left\lceil \frac{|T_i|}{\lceil m/2 \rceil - 1} \right\rceil .$$

From Lemma 4.15, Property 2 we have

$$\mathrm{OPT}_{\mathcal{T}_2} \geq \sum_{i=1}^{k_2} \left\lceil \frac{|T_i|}{m} \right\rceil .$$

Now, observing $\lceil m/2 \rceil = \lfloor (m+1)/2 \rfloor$ and using Lemma 4.16 with $x_i := \frac{|T_i|}{m}$ and $z := m$, we conclude that there is a $q_2 \in \mathbb{N}_0$ with

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

as well as Inequality (4.8). □

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

Lemma 4.19. *Given $q_1, q_2, k_1, k_2 \in \mathbb{N}_0$ and $k \in \mathbb{N}$ such that $q_1 + q_2 \leq k$. Then*

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

with respect to k.

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

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

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

Hence

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

The claim follows. □

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

Theorem 4.20. *Splitting up $\mathcal{T}$ into task sets $\mathcal{T}_1$ and $\mathcal{T}_2$ and scheduling them separately with the algorithms from Listing 4.3 and Listing 4.4 results in a sum of completion times of* $((2+4/(m-3))+\mathrm{o}(1))\cdot\mathrm{OPT}$.

Proof. By $S = S_{\mathcal{T}_1}+S_{\mathcal{T}_2}$ and $\mathrm{OPT}=\mathrm{OPT}_{\mathcal{T}_1}+\mathrm{OPT}_{\mathcal{T}_2}$ as well as Lemma 4.17 and Lemma 4.18, there are $q_1, q_2 \in \mathbb{N}_0$ such that

$$\begin{aligned}
S &\leq \left(2+\frac{4}{m-3}\right)\mathrm{OPT}_{\mathcal{T}_1}+q_1+\left(2+\frac{4}{m-2}\right)\mathrm{OPT}_{\mathcal{T}_2}+q_2\\
&\leq \left(2+\frac{4}{m-3}\right)(\mathrm{OPT}_{\mathcal{T}_1}+\mathrm{OPT}_{\mathcal{T}_2})+q_1+q_2
\end{aligned}$$

and

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

Dividing S by OPT, using these inequalities, and applying Lemma 4.19 completes the proof. □

If we denote the total number of jobs by $n=\sum_{i=1}^{k} n_i$ and by applying the same arguments as in the proof of Theorem 4.4, we get a bound on the runtime.

Corollary 4.21. *Splitting up $\mathcal{T}$ into task sets $\mathcal{T}_1$ and $\mathcal{T}_2$ and scheduling them separately with the algorithms from Listing 4.3 and Listing 4.4 can be implemented with a runtime of* $\mathrm{O}((m+n)\cdot n)$.

CHAPTER 5

Scheduling with a Bounded Speed Limit and Variable Energy Costs

Energy has long since been recognized as one of the most important factors concerning the profitability of modern data centers [BH07]. In fact, energy efficiency has risen to be a major factor in the design and development of most technical systems, ranging from the above-mentioned data centers, over the embedded systems in our homes ("Internet of Things"), to the mobile devices everybody carries around. There are efforts on a multitude of levels to make such systems more energy (cost) efficient, in order to reduce the energy footprints of all those gadgets and to let our mobile devices run longer. On an algorithmic level, these efforts focus largely on a technique called *speed scaling*, also known as *dynamic voltage scaling* [Alb10]. It describes the ability of a device to adapt its speed, and thus energy consumption, to the current requirements. It is one of the most prominent energy saving techniques, and most modern systems support it in one way or another. This chapter joins a line of research that models different incarnations of this technique and designs provably efficient algorithms (see [Alb11] for a survey). The first theoretical study of speed scaling models is due to Yao et al. [YDS95]. The authors modeled the power consumption of a processor running at speed s by a *power function* $P(s) = s^{\alpha}$. Here, α is a device-dependent constant that is approximately 3 in practice [Bro+00]. Yao et al. designed a polynomial-time algorithm to compute an energy-minimal schedule for a given number of jobs, each with its own release time, deadline, and workload. We consider a model variant of [YDS95] with the following additional characteristics:

Dynamic Speed Limits: Most results adopt the unbounded speed model of [YDS95], where the processing speed s can be arbitrarily large. In practice, however, there are limits to the speed, and they no longer stem solely from the (static) maximum processor frequency. Instead, as devices become smaller and more sensitive to environmental conditions such as temperature and humidity, speed limits become highly dynamic. For example, failures of air conditioning, broken fans, or airflow problems can cause severe temperature fluctuations, requiring a temporary slow down of processors [ITW14]. Other sources of dynamic speed limits are voltage fluctuations as they occur in solar-powered devices, for example.

Dynamic Electricity Costs: A second, often neglected model constraint, are dynamic electricity costs. In particular for data centers, energy minimization aims at cost reduction. But often, algorithm design assumes energy costs to be uniform over time. However, electricity providers increasingly adopt time-dependent tariff policies. In fact, most providers already offer heavily discounted rates during off peak times, for example at night or before noon. While such cost changes are not as frequent and dynamic as the aforementioned changes of the maximum speed, they can have a huge impact on the operating costs.

We consider the problem of minimizing the total (energy) costs in a system with these characteristics. Note that while problems motivated by dynamic environments often call for a consideration as an online problem, similar to [YDS95] this chapter concentrates on the offline optimization problem as a first step to solve this variant. Also, previous work has shown that offline algorithms can be an integral part of online algorithms [BKP07].

5.1 Preliminaries

We formally introduce the model in Section 5.1.1. We then proceed by giving an overview of our contribution in Section 5.1.2

5.1.1 Model & Notation

We consider the scheduling of n jobs $J \coloneqq \{1, 2, \dots, n\}$ on a single, speed-scalable processor. Here, speed-scalable means that the processor's speed $s \in \mathbb{R}_{\geq 0}$ is controlled by the scheduler. The power consumption is modeled by a *power function* $P\colon \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}, s \mapsto s^{\alpha}$. That is, while running at speed s, energy is consumed at a rate of $P(s) = s^{\alpha}$. The constant $\alpha > 1$ is called the *energy exponent*. This assumption is common in the speed scaling literature as the power consumption of CMOS devices can roughly be estimated by s^3 and CMOS devices will presumably remain the dominant technology in the near future [BKP07]. Technically, our work could be generalized to convex power functions with invertible P', but at the cost of intuition in the properties of our algorithm and schedule, in particular during the computation of water levels (cf. Section 5.2) in Section 5.3.1.

In addition to these classical speed scaling properties, we have the constraint that the maximum speed at time t is bounded. We model this constraint via a *maximum speed function* $s_{\max}\colon \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$. Further, there is a cost factor associated with every timepoint $t \in \mathbb{R}_{\geq 0}$, specifying the cost per unit of energy. The cost factor is modeled via a *cost factor function* $c\colon \mathbb{R}_{\geq 0} \to \mathbb{R}_{>0}$.

Each job $j \in J$ comes with a *release time* $r_j \in \mathbb{R}_{\geq 0}$, a *deadline* $d_j \in \mathbb{R}_{\geq 0}$, and a *workload* $w_j \in \mathbb{R}_{\geq 0}$. For each time $t \in \mathbb{R}_{\geq 0}$, a *schedule* S must decide which job to process at what speed. Preemption is allowed, so that a job may be suspended and resumed later on. We model a schedule S by a *speed function* $s\colon \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$ and a *scheduling policy* $\mathcal{J}\colon \mathbb{R}_{\geq 0} \to J$. Here, $s(t)$ denotes the speed at time t, and $\mathcal{J}(t)$ the job that is scheduled at time t. A *feasible* schedule must finish all jobs within their release time/deadline intervals $[r_j, d_j)$ without exceeding the maximum speed function $s_{\max}$. More formally, we require $s(t) \leq s_{\max}(t)$ for all $t \in \mathbb{R}_{\geq 0}$ and $\int_{\mathcal{J}^{-1}(j)\cap[r_j,d_j)} s(t)\,\mathrm{d}t \geq w_j$ for all $j \in J$. If, additionally, all workloads are met exactly (i.e., $\int_{\mathcal{J}^{-1}(j)\cap[r_j,d_j)} s(t)\,\mathrm{d}t = w_j$ for all $j \in J$), we call the schedule *non-wasting*. The total energy consumption of a schedule S is given by $\int_0^\infty P(s(t))\,\mathrm{d}t$, and its total energy cost by $E(s) \coloneqq \int_0^\infty c(t) \cdot P(s(t))\,\mathrm{d}t$. For technical reasons, we restrict ourselves to functions in C_{pr} (i.e., to functions that are right-continuous with finitely many discontinuities), which covers all practically relevant schedules. Our goal is to find a feasible schedule of minimum cost. In the rest of this paper, we refer to this scheduling problem as `ContBERS` (**Cont**inuous **B**ounded Speed & **E**lectricity **R**ates **S**cheduling).

Computational Model We assume oracle access to the functions $s_{\max}$ and c. Similarly, we assume access to basic function calculus such as taking the min of two functions or computing integrals (cf. Section 5.3.1). This is in accord with standard speed-scaling literature (e.g., [AF07; BCP13]) where one needs the ability to, for example, solve equations involving high degree polynomials.

5.1.2 Contribution

Theoretical algorithm design for speed scaling problems tends to consider discretized versions of problems, as our tools in the discrete realm are often better developed or understood. Extending [YDS95] to also feature the aforementioned continuous characteristics of "Dynamic Speed Limits" and "Dynamic Electricity Costs" makes for a good example in which discretization results in a burdensome quantity of variables and constraints. On the basis of this problem we demonstrate that a more direct approach via tools from variational calculus not only leads to a very concise formulation and analysis, but also avoids the "explosion of variables/constraints" that often comes with discretizing [Ant+14].

Although specific calculus of variations tools have been used before in the speed scaling literature [BKP07; Tha13], they were merely used as tools for analyzing a discrete problem. In contrast, we derive combinatorial optimality characteristics for a continuous problem. More specifically, our approach is based on formulating the problem, as well as designing and analyzing the algorithm in a continuous fashion, which helps us provide a quite concise and simple correctness proof.

It should be noted that because practical implementations of our approach would typically rely on numerical (i.e., discrete) function calculus, it is natural that our algorithm turns out to be similar to existing ones for discrete versions of the problem.

However, our focus here is not to provide a better practical implementation, but rather to simplify (at least in some settings) a rigorous design and analysis. We believe that using the available mathematical tools from calculus of variations to directly formulate, analyze, and design algorithms in a continuous setting yields a holistic approach that might help in solving some longstanding open problems, such as [Ant+14; Bar+13] for arbitrary continuous speeds.

5.2 Balance for Optimality

This section is dedicated to proving the following theorem.

Theorem 5.1. *A feasible schedule is optimal if and only if it is both non-wasting and work-balanced.*

Here, being *work-balanced* is a natural structural property, which we formally introduce in Section 5.2.3. For now, think of schedules that distribute the jobs' workload "as evenly as possible" while taking constraints (e.g., the release times/deadlines or the speed limits) and cost factors into account.

Being work-balanced is a natural condition, and similar structural properties have been exploited for a variety of problems to study and compute optimal or approximate solutions. Examples include the original speed-scaling algorithm YDS [BKP07] or the study of equilibria in resource selection games [GT14]. Another related property is used in the standard approximation algorithm for metric facility location [JV01], which carefully controls the contribution of the different clients to the costs of opening facilities. The common ground of these properties is that they can be derived using a linear or convex program and duality theory and, by controlling how much different elements (jobs, clients) contribute to the solution, yield a corresponding *primal-dual algorithm*. The basic ingredients to analyze the solution quality of such an approach are the KKT conditions known from convex programming [BV04]. Unfortunately, this approach does not work in our setting. Although the considered optimization problem is convex, the general maximum speed restriction leads to infinitely many variables/constraints: for each time $t \in \mathbb{R}_{\geq 0}$ the speed must not exceed $s_{\max}(t)$. On the basis of the theory of variational calculus [Smi98], we can still approach the `ContBERS` problem by similar means.

Overview We continue in Section 5.2.1 with a presentation of the `ContBERS` problem viewed as an optimization problem with an infinite number of constraints. Afterward, Section 5.2.2 provides a framework to characterize optimal solutions for problems of a more general form. Finally, Section 5.2.3 applies this framework to prove Theorem 5.1.

5.2.1 Scheduling via Variational Calculus

Optimization problems with infinitely many variables/constraints can be modeled via function variables. In the case of the `ContBERS` problem, one can think of the schedule's speed function $s\colon \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$ as a variable that has to fulfill the constraint $s(t) \leq s_{\max}(t)$ at any time $t \geq 0$. Remember that the costs of a speed function s are given by $E(s) = \int_0^\infty c(t)P(s(t))\,\mathrm{d}t \in \mathbb{R}$. In other words, E is a function that maps a speed function s to a real cost value $E(s)$. Functions mapping other functions to real values are called *functionals* [Smi98]. We seek a speed function s that minimizes the functional E under the constraints that the maximum speed is never exceeded and that all jobs are finished. Since we also need a scheduling policy (to decide which job to run when), we actually search n speed functions $s_j\colon \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$ telling us when and how to run j. The set of candidate functions is

$$\mathcal{S}_j := \{\, f\colon \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0} \mid f \in C_{pr} \wedge \forall x \notin [r_j, d_j)\colon f(x) = 0 \,\}\,. \tag{5.1}$$

Let $\mathcal{S} := \prod_{j \in J} \mathcal{S}_j$. To improve readability, we slightly abuse notation by using s for an element of $\mathcal{S}$ (i.e., a vector of the n different s_j's) as well as for the speed function of the schedule (i.e., the sum of the n different s_j's). We can formulate our optimization problem as the (infinite) mathematical program (SP) shown below.

$$\begin{aligned} \min_{s \in \mathcal{S}} \quad & E\left(\textstyle\sum_{j \in J} s_j\right) & & \\ \text{s.t.} \quad & \textstyle\sum_{j \in J} s_j(t) \leq s_{\max}(t) & \forall t \geq 0 \qquad & (5.2) \\ & \textstyle\int_{r_j}^{d_j} s_j(t)\,\mathrm{d}t \geq w_j & \forall j \in J \qquad & (5.3) \end{aligned}$$

An optimal solution minimizes the energy costs for the speed function $\sum_{j \in J} s_j$ without exceeding the maximum speed (Constraint (5.2)) and finishes all jobs (Constraint (5.3)). Note that we do not require the s_j to have pairwise disjoint supports. In other words, the resulting schedule might run two jobs at the same time. Omitting this requirement is without loss of generality, as we can show how to transform such schedules to obtain pairwise disjoint supports.

Lemma 5.2 (EDF Schedule)**.** *Consider an arbitrary feasible solution $s \in \mathcal{S}$ to the optimization problem (SP). Then there exists a feasible solution $s' \in \mathcal{S}$ with $E(s') \leq E(s)$ and the property $\forall j_1, j_2 \in J, t \in \mathbb{R}_{\geq 0}\colon s'_{j_1}(t) \cdot s'_{j_2}(t) > 0 \implies j_1 = j_2$.*

Proof. We transform solution s to a solution s' by employing *Earliest Deadline First* (EDF) scheduling. Intuitively, at every timepoint we run only the task that has the earliest deadline among all available tasks.

Let the indices of the jobs be ordered according to their deadline and assume, w.l.o.g., that no two jobs share the same deadline. We first transform s to a solution $\hat{s}$ such that for every job j, $\hat{s}$ processes exactly w_j workload (i.e., a non-wasting schedule). To do this we identify for each job j the earliest timepoint t_j such that $\int_{r_j}^{t_j} s_j(t)\,\mathrm{d}t = w_j$. Note that by constraint (5.3) we have that $t_j \leq d_j$. We then set $\hat{s}_j(t) = 0$ for all $t \geq t_j$ and $\hat{s}_j(t) = s_j(t)$ for all $t < t_j$. Note that $E(\hat{s}) \leq E(s)$ holds.

We next transform $\hat{s}$ into s'. To this end, set $t^* := \min_j r_j$ and let $\hat{w}_j := w_j$ denote the *remaining workload* of j, for all $j \in J$. We identify the job i that has the earliest deadline among all jobs j released by time t^* and with nonzero $\hat{w}_j$. Now set

t_i such that $\int_{t^*}^{t_i} \sum_j s_j(t)\,\mathrm{d}t = \hat{w}_i$. That is, the point at which i would be finished if it were exclusively processed by schedule s. If there is no release time in $[t^*, t_i)$, we set $s'_i(t) := \sum_j s_j(t)$ for every $t \in [t^*, t_i)$ and $s'_j(t) := 0$ for all other jobs. Then t^* is updated to t_i and $\hat{w}_i$ to 0. Otherwise, let r_k be the earliest release time in $[t^*, t_i)$. We set $s'_i(t) := \sum_j s_j(t)$ for every $t \in [t^*, r_k)$ and $s'_j(t) = 0$ for all other jobs. Finally we update $\hat{w}_i$ to $\hat{w}_i - \int_{t^*}^{r_k} s_i(t)\,\mathrm{d}t$ and t^* to r_k. We repeat the above step until all $\hat{w}_j$'s are set to 0.

The above transformation terminates, because in each iteration (new t^*) we make progress: either we move to the next release time, or one of the $\hat{w}_j$'s is set to zero. Also note that $E(\hat{s}) = E(s')$. This immediately follows by the fact that by the way the transformation is defined: for any t, $\sum_j \hat{s}_j(t) = \sum_j s'_j(t)$ holds. Further, since at every timepoint, t there exists at most one j such that $s'_j(t) > 0$, s' satisfies the property stated in the lemma. However, it is not immediately obvious that s' is feasible. We continue to show this. By the above transformation, we have for any j that $s'_j(t) = 0$ for all $t \in [r_{\min}, r_j)$. It remains to show that the total workload of each job is processed before its deadline. More formally, we must have $\int_0^{d_j} s'_j(t)\,\mathrm{d}t = w_j$ for all jobs j. (The fact that $s'_j(t) = 0$ for $t \in (d_j, d_{\max})$ then follows by the definition of the transformation). To this end, define for any job j, any timepoint t, and any solution s the value $F(t, j, s) := \int_0^t \sum_{i \le j} s_i(x)\,\mathrm{d}x$. Intuitively, $F(t, j, s)$ denotes the total workload of jobs with a deadline of at most d_j that s has finished by timepoint t. By Constraint (5.3) of (SP) and the first part of the transformation we have $F(d_j, j, \hat{s}) = \sum_{i=1}^{j} w_i$ for any job j.

We now show that for any j and t, we have the inequality $F(t, j, s') \ge F(t, j, \hat{s})$. That is, s' finishes as least as much workload of jobs with deadline at most d_j by time t as $\hat{s}$. Indeed, assume that this is not the case and let t' be the first time such that $F(t', j, s') < F(t', j, \hat{s})$. In combination with the fact that s' satisfies the property stated in the lemma, this implies that at t' we have $\sum_{i=1}^{j} \hat{s}_i(t') > 0$ while $\sum_{i=1}^{j} s'_i(t') = 0$. However, by the definition of s', $\sum_{i=1}^{j} s'_i(t') = 0$ can only hold when all jobs with a release time $\le t'$ and a deadline $\le d_j$ are fully processed. This contradicts the assumption $F(t', j, s') < F(t', j, \hat{s})$, since $\hat{s}$ processes exactly w_j units for each job j. Thus, we must have $F(t, j, s') \ge F(t, j, \hat{s})$ for all t and j and, in particular, $F(d_j, j, s') \ge F(d_j, j, \hat{s})$ for all j. The lemma follows since $F(d_j, j, \hat{s}) = \sum_{i=1}^{j} w_i$. □

5.2.2 Characterizing Optimal Solutions

In the following, we formulate a more general optimization problem and derive (rather abstract) optimality conditions that can be viewed as an extended version of the KKT conditions. We will see in Section 5.2.3 how to apply these conditions to the `ContBERS` problem.

Let $N, m, n \in \mathbb{N}$. We consider an optimization problem for functionals over the set $\mathcal{F} := \prod_{j=1}^{N} \mathcal{F}_j$ and $m + n$ constraints, where for intervals I_j with $j \in \{1, \dots, N\}$, we define $\mathcal{F}_j := \{ g \colon \mathbb{R} \to \mathbb{R} \mid g \in C_{pr} \land \forall x \notin I_j \colon g(x) = 0 \}$. The j-th component of $f \in \mathcal{F}$ therefore is a right-continuous function g with finitely many discontinuities, and $g|_{\mathbb{R}\setminus I_j} = 0$. We also view the vectors $f \in \mathcal{F}$ as vector-valued functions $f\colon \mathbb{R} \to \mathbb{R}^N$.

We have an *objective function* $L\colon \mathbb{R} \times \mathbb{R}^N \to \mathbb{R}$ as well as two types of *constraint functions* $G_k, H_l\colon \mathbb{R} \times \mathbb{R}^N \to \mathbb{R}$ for $k \in \{1, 2, \dots, m\}$ and $l \in \{1, 2, \dots, n\}$. All these functions are assumed to be piecewise differentiable and convex in their second argument. For example, $G_k(x, y)$ with $x \in \mathbb{R}$ and $y \in \mathbb{R}^N$ is piecewise continuously differentiable and convex in y. We write ∇L (and similar for the other functions) to refer to the gradient of L taken with respect to the components of the second argument $y \in \mathbb{R}^N$ and $\nabla_j L$ for the j-th component of L's gradient. Let I be any interval in $\mathbb{R}$. The considered general optimization problem (GP) is

$$\begin{aligned} \min_{f\in\mathcal{F}} \quad & \int_I L(x, f(x))\,\mathrm{d}x \\ \text{s.t.} \quad & G_k(x, f(x)) \le 0 && \forall x \in I, k \in \{1, 2, \dots, m\} && \text{(I)} \\ & \int_I H_l(x, f(x))\,\mathrm{d}x \le 0 && \forall l \in \{1, 2, \dots, n\} && \text{(II)} \end{aligned}$$

Here, constraints of type (I) represent local constraints that hold at any point in time t (e.g., restricted processor speed). Constraints of type (II) represent global constraints that hold for some kind of volume (e.g., finished workload of a job). The following theorem provides sufficient optimality conditions for solutions of (GP).

Theorem 5.3 (Extended KKT conditions)**.** *Assume that $f \in \mathcal{F}$ is a feasible solution for (GP) with finite solution value. Furthermore, assume that there exist functions $\lambda_k\colon I \to \mathbb{R}_{\ge 0}$, $\lambda_k \in C_{pr}$, and constants $\mu_l \in \mathbb{R}_{\ge 0}$ such that the following properties hold:*

1. *For all $j \in \{1, 2, \dots, N\}$ and $x \in I_j$, we have*

$$\nabla_j L(x, f(x)) + \sum_{k=1}^{m} \lambda_k(x) \cdot \nabla_j G_k(x, f(x)) + \sum_{l=1}^{n} \mu_l \cdot \nabla_j H_l(x, f(x)) = 0. \quad (5.4)$$

2. *For all $k \in \{1, 2, \dots, m\}$ and $x \in I$, we have $\lambda_k(x) \cdot G_k(x, f(x)) = 0$.*
3. *For all $l \in \{1, 2, \dots, n\}$, we have $\mu_l \cdot \int_I H_l(x, f(x))\,\mathrm{d}x = 0$.*

Then f is an optimal solution to (GP).

Similar conditions have been used before (see for example [Lue69]). For a restricted class of these problems, such conditions were even given in a quite similar form under the term *continuous time optimization*: Hanson and Mond [HM68] consider a class of nonlinear programming problems in the continuous variant and present KKT-like conditions for the adapted problems. In their variant, the equivalent of our $f(x)$ appears only linearly in the constraints. Farr and Hanson [FH74] extend the prior results to nonlinear constraints, but the equivalent to x and $f(x)$ still appear in separate functions in their work and hence cannot depend arbitrarily on each other.

In the following we give a self-contained proof of the extended KKT conditions.

Getting Rid of the Constraints Using *Lagrange multipliers* $\lambda_k\colon I \to \mathbb{R}_{\geq 0}$, $\lambda_k \in C_{pr}$, $k \in \{1, 2, \dots, m\}$, and $\mu_l \in \mathbb{R}_{\geq 0}$, $l \in \{1, 2, \dots, n\}$, we can define the functional

$$\Lambda(f, \lambda, \mu) := \sum_{k=1}^{m} \int_I \lambda_k(x) \cdot G_k(x, f(x))\,\mathrm{d}x + \sum_{l=1}^{n} \mu_l \cdot \int_I H_l(x, f(x))\,\mathrm{d}x, \tag{5.5}$$

where $\lambda = (\lambda_1, \dots, \lambda_m)$ and $\mu = (\mu_1, \dots, \mu_n)$. This is the so-called *Lagrangian.* By construction, we have $\Lambda \leq 0$ if f satisfies the constraints of (GP) (independently of λ and μ). This can be used to prove the following result known from duality theory [BV04; Smi98]:

Lemma 5.4. *Fix λ and μ, and consider an optimal solution $\tilde{f} \in \mathcal{F}$ to the minimization problem (LGR) given as*

$$\min_{f \in \mathcal{F}} D(f), \text{ where } D(f) := \int_I L\,(x, f(x))\,\mathrm{d}x + \Lambda(f, \lambda, \mu). \tag{5.6}$$

Assume that λ, μ, and $\tilde{f}$ satisfy Properties 2 and 3 of Theorem 5.3. If, additionally, $\tilde{f}$ fulfills the constraints of (GP), then $\tilde{f}$ is an optimal solution to (GP).

Proof. For such λ, μ, and $\tilde{f}$, we have $\Lambda(\tilde{f}, \lambda, \mu) = 0$. Thus, when comparing $\tilde{f}$ to an arbitrary feasible solution f of (GP), we get

$$\begin{aligned}\int_I L\left(x, \tilde{f}(x)\right)\mathrm{d}x &= \int_I L\left(x, \tilde{f}(x)\right)\mathrm{d}x + \Lambda\left(\tilde{f}, \lambda, \mu\right)\\ &\leq \int_I L\,(x, f(x))\,\mathrm{d}x + \Lambda\,(f, \lambda, \mu)\\ &\leq \int_I L\,(x, f(x))\,\mathrm{d}x.\end{aligned}$$

The last inequality holds because f is a feasible solution to (GP), which implies $\Lambda(f, \lambda, \mu) \leq 0$. This proves the lemma's statement. □

Lemma 5.4 says that in order to solve the minimization problem (GP) with its constraints, it is sufficient to solve (LGR) (which does not have constraints) for arbitrary, fixed dual variables, *but only if we can guarantee that the constraints are fulfilled.* This seems of small help, and in general such a solution might actually not exist. However, the dual variables give us an extra degree of freedom, and convexity ensures that our problem is “well-behaved”. Thus, our strategy is to find dual variables such that the optimal solution for (LGR) adheres to the constraints of (GP).

Convexity of (LGR) In order to solve (LGR), we first observe that the set $\mathcal{F}$ over which we optimize is convex. That is, for any two functions $f, g \in \mathcal{F}$ and $t \in [0,1]$ we have $(1-t)\cdot f + t \cdot g \in \mathcal{F}$. Another useful observation is that the objective D of (LGR) is convex (as a functional over $\mathcal{F}$). To see this, remember that L, G_k, and H_l are convex in their second argument. Moreover, D can be rewritten as

$$\begin{aligned} D(f) &= \int_I L\left(x, f(x)\right) \mathrm{d}x + \Lambda(f, \lambda, \mu) \\ &= \int_I \left(L(x, f(x)) + \sum_{k=1}^{m} \lambda_k(x) \cdot G_k(x, f(x)) + \sum_{l=1}^{n} \mu_l \cdot H_l(x, f(x)) \right) \mathrm{d}x \end{aligned}$$

$=: \int_I \tilde{L}(x, f(x), \lambda(x), \mu)\, \mathrm{d}x$, for a suitably defined $\tilde{L} \colon \mathbb{R} \times \mathbb{R}^N \times \mathbb{R}^m \times \mathbb{R}^n \to \mathbb{R}$. The function $\tilde{L}$ is (as a positive sum of convex functions) convex in its second argument. Using monotonicity and linearity of the integration operator, we can prove the convexity of D:

$$\begin{aligned} D\big(tf + (1-t)g\big) &= \int_I \tilde{L}(x, tf(x) + (1-t)g(x), \lambda(x), \mu)\, \mathrm{d}x \\ &\le \int_I \Big(t\tilde{L}(x, f(x), \lambda(x), \mu) + (1-t)\tilde{L}(x, g(x), \lambda(x), \mu) \Big)\, \mathrm{d}x \\ &= t \int_I \tilde{L}(x, f(x), \lambda(x), \mu)\, \mathrm{d}x + (1-t) \int_I \tilde{L}(x, g(x), \lambda(x), \mu)\, \mathrm{d}x \\ &= tD(f) + (1-t)D(g). \end{aligned}$$

Optimality Condition for (LGR) With D being convex, we can use the property that any local optimum is also globally optimal (cf. [KM11, Chap. 3]). Local optima can be characterized via their derivatives. Consider the (one-sided) directional derivatives $\delta_+ D(f, v) := \lim_{\varepsilon \to 0^+} \frac{D(f+\varepsilon v) - D(f)}{\varepsilon}$ of D at $f \in \mathcal{F}$ in any direction v with $f + v \in \mathcal{F}$. By convexity, a solution $f \in \mathcal{F}$ to (LGR) is (globally) optimal if and only if

$$\delta_+ D(f, v) \ge 0 \qquad \forall v \colon f + v \in \mathcal{F}. \tag{$\heartsuit$}$$

With this, we are now finally ready to prove Theorem 5.3.

Proof of Theorem 5.3. Assume we have functions $\lambda_k \colon I \to \mathbb{R}_{\ge 0}$ and constants $\mu_l \in \mathbb{R}_{\ge 0}$ as in Theorem 5.3, so that the properties 1 to 3 hold for a feasible solution $f \in \mathcal{F}$ to (GP). Remember that L, G_k, and H_l are piecewise differentiable, implying that $\tilde{L}$ is piecewise differentiable. Similarly to the other functions, we write $\nabla \tilde{L}$ to denote $\tilde{L}$'s gradient taken with respect to the components of the second argument $y \in \mathbb{R}^N$. Then, by Leibniz and chain rule, we can write the directional derivative of

Equation (♡) as

$$\begin{aligned}\delta_+ D(f,v) &= \frac{\mathrm{d}}{\mathrm{d}\varepsilon}\int_I \tilde{L}(x, f(x)+\varepsilon v(x), \lambda(x), \mu)\,\mathrm{d}x\big|_{\varepsilon=0}\\ &= \int_I \frac{\mathrm{d}}{\mathrm{d}\varepsilon}\tilde{L}(x, f(x)+\varepsilon v(x), \lambda(x), \mu)\big|_{\varepsilon=0}\,\mathrm{d}x = \int_I \Big\langle \nabla\tilde{L}(x, f(x), \lambda(x), \mu), v(x)\Big\rangle\,\mathrm{d}x\end{aligned} \tag{5.7}$$

Note that it may be necessary to split the integral for this operation as the involved functions are only *piecewise* differentiable. Now, by Property 1 of Theorem 5.3, we have that component j of $\nabla\tilde{L}(x, f(x), \lambda(x), \mu)$ is equal to zero whenever $x \in I_j$. On the other hand, whenever $x \notin I_j$, we must have $v_j(x) = 0$ as otherwise $f_j(x) + v_j(x) \neq 0$, contradicting the fact that $f + v \in \mathcal{F}$. The integrand of Equation (5.7) thus vanishes, and $\delta_+ D(f,v) = 0$ for all directions v with $f + v \in \mathcal{F}$. This implies optimality of f for the optimization problem (LGR). As λ, μ, and f satisfy Properties 2 and 3 of Theorem 5.3, we can apply Lemma 5.4 to show that f is an optimal solution of (GP). □

5.2.3 Extracting Structural Properties

We rewrite the mathematical program (SP) from Section 5.2.1 such that it has the form of the general mathematical program (GP) from Section 5.2.2. To this end, let T be the latest deadline and set $I := [0, T)$. We get the following convex problem:

$$\begin{aligned}\min_{s\in\mathcal{S}} \quad & E(s) & & \\ \text{s.t.} \quad & s(t) - s_{\max}(t) \le 0 & \forall t \ge 0 \quad & (5.8)\\ & \int_I \frac{w_j}{T} - s_j(t)\,\mathrm{d}t \le 0 & \forall j \in J \quad & (5.9)\\ & -s_j(t) \le 0 & \forall j \in J, t \ge 0 \quad & (5.10)\end{aligned}$$

Theorem 5.3 gives us a continuous version of the KKT conditions, which we can apply to extract a nice and combinatorial optimality condition for our problem. Note that the Constraints (5.8) and (5.10) translate to inequality constraints of type (I), whereas Constraint (5.9) corresponds to an inequality constraint of type (II).

Extended KKT Conditions for `ContBERS` We now introduce a dual variable $\lambda\colon I \to \mathbb{R}_{\ge 0}$ for Constraint (5.8), dual variables $\mu_j \in \mathbb{R}_{\ge 0}$ for Constraint (5.9), and dual variables $\gamma_j\colon I \to \mathbb{R}_{\ge 0}$ for Constraint (5.10). Then the extended KKT conditions are:

1. Extended Stationarity: For all $j \in J$ and $t \in [r_j, d_j)$, the expression

   $$\nabla_j\left(c(t)P(s(t))\right) + \lambda(t)\cdot\nabla_j(s(t) - s_{\max}(t)) - \sum_{j'\in J}\gamma_{j'}(t)\cdot\nabla_j s_{j'}(t) + \sum_{j'\in J}\mu_{j'}\cdot\nabla_j\left(\frac{w_{j'}}{T} - s_{j'}(t)\right)$$

   equals zero. Recall that ∇_j denotes the j-th component of the gradient of the *second* argument (where the arguments are t and $s(t)$) and, therefore, the partial derivative with respect to $s_j(t)$. Hence, the equality above is equivalent to

$$c(t)P'(s(t)) + \lambda(t) - \gamma_j(t) - \mu_j = 0. \tag{5.11}$$

2. Continuous complementary slackness conditions:

$$\lambda(t) \cdot \big(s(t) - s_{\max}(t)\big) = 0 \qquad \forall t \in I, \tag{5.12}$$

$$\gamma_j(t) \cdot s_j(t) = 0 \qquad \forall j \in J, t \in I. \tag{5.13}$$

3. Discrete complementary slackness conditions:

$$\mu_j \cdot \left(\int_I \frac{w_j}{T} - s_j(t)\,\mathrm{d}t\right) = 0 \qquad \forall j \in J. \tag{5.14}$$

Characterizing Optimality We now use the above stated extended KKT conditions to characterize optimal solutions for our ContBERS scheduling problem. Using the jobs' release times and deadlines, we partition the time horizon into m non-overlapping, consecutive time intervals $T_i := [t_i, t_{i+1})$, $i \in \{1, 2, \dots, m\}$, where t_i is the i-th point in the set $\{r_j, d_j \mid j \in J\}$. Note that $m \leq 2n - 1$. We call T_i the i-th *atomic interval* and use $J(i) := \{j \in J \mid T_i \subseteq [r_j, d_j)\}$ to denote the set of jobs that are *active* in T_i (i.e., that may be scheduled in T_i).

The resulting scheduling condition is essentially a generalization of the well-known optimality condition for the classical speed-scaling model from Yao et al. [YDS95]. There, an important property of optimal schedules is that during the lifetime of a job j, speed never drops below the speed s_j used to process j. For our setting, we need a more general and complex optimality condition. In the following, we provide such a property (Definition 5.6) and prove that it characterizes optimal schedules (by proving Theorem 5.1) using our extended KKT conditions.

Definition 5.5 (Work-Transferable)**.** For a given schedule and two atomic intervals T_i and $T_{i'}$, the *work-transferable* relation $i \rightarrow i'$ holds if there exists a job $j \in J(i) \cap J(i')$ with $\int_{t_i}^{t_{i+1}} s_j(t)\,\mathrm{d}t > 0$. Furthermore, let $\twoheadrightarrow$ be the reflexive transitive closure of $\rightarrow$.

Definition 5.6 (Work-Balanced)**.** We say that a schedule is *work-balanced* if there are constants $s_i \in \mathbb{R}$ for $i \in \{1, \dots, m\}$ so that 1. for any fixed atomic interval T_i the speed $s(t) \in \mathbb{R}$ at time $t \in [t_i, t_{i+1})$ is $\min(s_{\max}(t), c(t)^{-\frac{1}{\alpha-1}} \cdot s_i)$ and 2. for any two atomic intervals T_i and $T_{i'}$ with $i \twoheadrightarrow i'$, we have that $s_i \leq s_{i'}$.

To get an intuition, assume $c(t)$ to be constant in each atomic interval. Then, the first property implies that, unless $s_{\max}$ forces us to run slower, we run at a constant speed in each atomic interval (which can be different for each atomic interval). The second property says that workload can only be transferred to intervals of higher speed (which would increase the cost). For non-constant $c(t)$, we have to weight the speed suitably.

To ease the further discussion, we slightly abuse notation by extending the work-transferable relation "$\twoheadrightarrow$" to timepoints and jobs. More specifically, for an atomic interval T_i and a time $t \in \mathbb{R}_{\geq 0}$ we write $i \twoheadrightarrow t$ if for the atomic interval $T_{i'}$ with $t \in T_{i'}$ we have $i \twoheadrightarrow i'$. Similarly, for an atomic interval T_i and a job $j \in J$ we write $i \twoheadrightarrow j$ if there is an atomic interval $T_{i'}$ in which j is processed and for which $i \twoheadrightarrow i'$.

Using these definitions, we have everything needed for the formal proof of Theorem 5.1 (as stated at the beginning of Section 5.2). Our analysis uses the following simple observation:

Observation 5.7. Consider a feasible schedule, an atomic interval i, a job j, and a time $t_0 \in [r_j, d_j)$. Then we have

1. $\{\, i \mid i \twoheadrightarrow j \,\} \subseteq \{\, i \mid i \twoheadrightarrow t_0 \,\}$ and
2. if $s_j(t_0) > 0$, then $\{\, i \mid i \twoheadrightarrow t_0 \,\} = \{\, i \mid i \twoheadrightarrow j \,\}$.

We also need the following auxiliary lemma to characterize optimality via the above notions.

Lemma 5.8. *Assume that for a feasible schedule S there exist two timepoints $t_1 \in T_i$ and $t_2 \in T_{i'}$ such that*

- $i \twoheadrightarrow i'$,
- $c(t_1)s(t_1)^{\alpha-1} > c(t_2)s(t_2)^{\alpha-1}$, *and*
- $s(t_2) < s_{\max}(t_2)$.

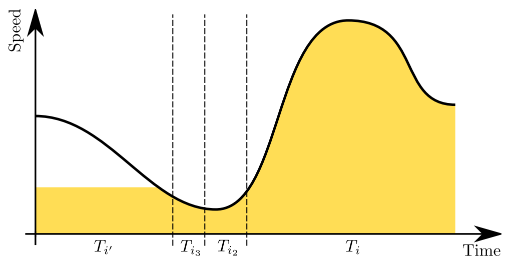
Then S cannot be optimal.

Proof. To prove the lemma, we transform S to $\tilde{S}$, such that $\tilde{S}$ is feasible and $E(s) > E(\tilde{s})$. The transformation is as follows:

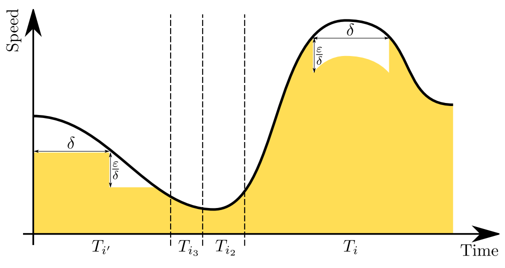
By the right-continuity of the involved functions, there exist intervals $I_i := [t_1, t_1 + \varepsilon) \subseteq T_i$ and $I_{i'} := [t_2, t_2 + \varepsilon) \subseteq T_{i'}$ for some $\varepsilon > 0$ such that $\min_{t \in I_i} c(t)s(t)^{\alpha-1} > \max_{t \in I_{i'}} c(t)s(t)^{\alpha-1}$ and $s(t') < s_{\max}(t')$, for all $t' \in I_{i'}$.

By the definition of "$\twoheadrightarrow$", there exists a sequence of atomic intervals $T_i = T_{i_1}, T_{i_2}, \dots, T_{i_l} = T_{i'}$, such that for each $y \in \{1, \dots, l-1\}$ there holds $i_y \rightarrow i_{y+1}$. In other words, for every $y \in \{0, \dots, l-1\}$, there exists a j_y such that $j_y \in J(i_y) \cap J(i_{y+1})$, and $\int_{t_{i_y}}^{t_{i_y+1}} s_{j_y}(t)\,\mathrm{d}t > 0$. Consecutively, for every such y we reduce the load of job j_y in the atomic interval T_{i_y} by $\delta > 0$, and increase the load of job j_y in $T_{i_{y+1}}$ by δ. At the same time we decrease the speed in I_i by δ/ε and increase the speed in $I_{i'}$ by δ/ε. It is easy to see that by choosing δ small enough, the resulting schedule $\tilde{S}$ is feasible (although during the above procedure it may have been infeasible at times), and that $\min_{t \in I_i} c(t)\tilde{s}(t)^{\alpha-1} > \max_{t \in I_{i'}} c(t)\tilde{s}(t)^{\alpha-1}$ still holds. Further, note that the cost only changes in the intervals I_i and $I_{i'}$.

Figure 5.1 visualizes the process described above for the simplified version with constant energy costs $c(t)$.



(a) Speed levels before moving workload.



(b) Speed levels after moving workload.

Figure 5.1: Energy decrease when moving workload for the simplified problem with constant energy costs. The black line denotes the upper speed limit, whereas the level of the yellow area denotes the current speed level of the schedule.

In I_i the energy cost decreases by:

$$\begin{aligned}
&\int_{I_i} c(t)\left(P(s(t)) - P(\tilde{s}(t))\right) \mathrm{d}t \\
&\qquad \geq \int_{I_i} c(t) \left(\frac{\delta}{\varepsilon} P'(\tilde{s}(t))\right) \mathrm{d}t \\
&\geq \int_{I_i} \frac{\delta}{\varepsilon} \min_{t \in I_i} \left(\alpha c(t) \tilde{s}(t)^{\alpha-1}\right) \mathrm{d}t \\
&\qquad = \delta\alpha \cdot \min_{t \in I_i} \left(c(t)\tilde{s}(t)^{\alpha-1}\right),
\end{aligned}$$

where the first inequality follows by the convexity of the power function.

On the other hand, by a similar calculation, the energy cost in $I_{i'}$ increases by at most:

$$\delta\alpha \cdot \max_{t \in I_{i'}} \left(c(t)\tilde{s}(t)^{\alpha-1}\right).$$

Since we chose δ so that $\min_{t \in I_i} \left(c(t)\tilde{s}(t)^{\alpha-1}\right) > \max_{t \in I_{i'}} \left(c(t)\tilde{s}(t)^{\alpha-1}\right)$ still holds, the lemma follows. □

We are now ready to prove our characterization of optimal schedules stated in Theorem 5.1.

Proof of Theorem 5.1. We start with the proof that being non-wasting and work-balanced is sufficient for optimality. Afterward, we show the necessity of both properties.

"⇐": Any feasible schedule S defines function variables s_j for a feasible solution. Here, $s_j(t)$ denotes the processing speed of job j at time t. In the following we set the dual variables and show that they satisfy the extended KKT conditions.

$$\begin{aligned}
\lambda(t) &:= \sup_{t_0 \in T_k, k \twoheadrightarrow t} \big(c(t_0)P'(s(t_0))\big) - c(t)P'(s(t)) && \forall t \in [0,T) \\
\mu_j &:= \sup_{t_0 \in T_k, k \twoheadrightarrow j} \big(c(t_0)P'(s(t_0))\big) && \forall j \in J \\
\gamma_j(t) &:= \lambda(t) - \mu_j + c(t)P'(s(t)) && \forall j \in J, t \in [r_j, d_j)
\end{aligned}$$

Moreover, we set $\gamma_j(t) := 0$ for $t \notin [r_j, d_j)$. We first need to show that these variables are dual-feasible (i.e., non-negative). We start with $\lambda(t)$. Consider the atomic interval T_i with $t \in T_i$. We obviously have $i \twoheadrightarrow t$, causing the supremum to consider t itself. Thus $\lambda(t)$ cannot be negative. The non-negativity of μ_j follows immediately from S being a feasible schedule. Because of this, there is an atomic interval T_i in which j is processed at some $t_0 \in T_i$ with speed $s(t_0) > 0$. For this atomic interval we have $i \twoheadrightarrow j$. Finally, the non-negativity of $\gamma_j(t)$ for any $j \in J$ and $t \in [r_j, d_j)$ follows immediately from Observation 5.71.

It remains to prove that the extended KKT conditions hold. The first condition, Equation (5.11), holds by definition of $\gamma_j(t)$. For Equation (5.13), fix $j \in J$, $t \in I$ and assume $s_j(t) > 0$. Then we must have $t \in [r_j, d_j)$. By applying Observation 5.7, Property 2 we get $\{\, i \mid i \twoheadrightarrow t \,\} = \{\, i \mid i \twoheadrightarrow j \,\}$. This implies the equality of the supremum expressions in the definition of $\lambda(t)$ and μ_j and, thus, $\gamma_j(t) = 0$. Now look at Equation (5.12) for some fixed $t \geq 0$ with i such that $t \in T_i$ and assume $s(t) < s_{\max}(t)$. By definition of the work-balanced property, we must have $\sup_{t_0 \in T_i} \Big(c(t_0)^{1/(\alpha-1)} s(t_0)\Big) \leq s_i = c(t)^{1/(\alpha-1)} s(t)$. Moreover, any k with $k \twoheadrightarrow i$ satisfies $s_k \leq s_i$, which yields $\sup_{t_0 \in T_k} \Big(c(t_0)^{1/(\alpha-1)} s(t_0)\Big) \leq c(t)^{1/(\alpha-1)} s(t)$. By rearranging, we get $\sup_{t_0 \in T_k} \Big(c(t_0) s(t_0)^{\alpha-1}\Big) \leq c(t)s(t)^{\alpha-1}$. Since we have shown that $\lambda(t)$ cannot be negative, this yields $\lambda(t) = 0$. Finally, Equation (5.14) follows because S is non-wasting, which gives us $w_j = \int_{i \in I} s_j(t)\,\mathrm{d}t \; \forall j \in J$.

"$\Rightarrow$": First, we show that any optimal schedule S is work-balanced.

For every atomic interval T_ℓ, let $t_\ell := \arg\max_{t \in T_k, k \twoheadrightarrow \ell} s(t)c(t)^{\frac{1}{\alpha-1}}$, and $s_\ell := s(t_\ell)c(t_\ell)^{\frac{1}{\alpha-1}}$. For the sake of contradiction, assume that these s_ℓ's do not satisfy property (a) of work-balanced schedules (i.e., there exists some interval T_ℓ and $t^* \in T_\ell$ so that $s(t^*) \neq \min(s_{\max}(t^*), c(t^*)^{-\frac{1}{\alpha-1}} \cdot s_\ell)$). Then it must be the case that $s(t^*) < s_{\max}(t^*)$, since $s(t^*) > s_{\max}(t^*)$ would contradict the feasibility of S, and $s(t^*) = s_{\max}(t^*)$ would imply $c(t^*)^{-\frac{1}{\alpha-1}} \cdot s_\ell < s(t^*)$ and thus contradict our choice of s_ℓ. Therefore we have $s(t^*) < s_{\max}(t^*)$ and $c(t^*)^{-\frac{1}{\alpha-1}} \cdot s_\ell \geq s(t^*)$. In fact, even the strict inequality $c(t^*)^{-\frac{1}{\alpha-1}} \cdot s_\ell > s(t^*)$ must hold, since equality would contradict the definition of t^*. Hence, all the properties of Lemma 5.8 are satisfied with $t_1 = t_\ell$ and $t_2 = t^*$, contradicting

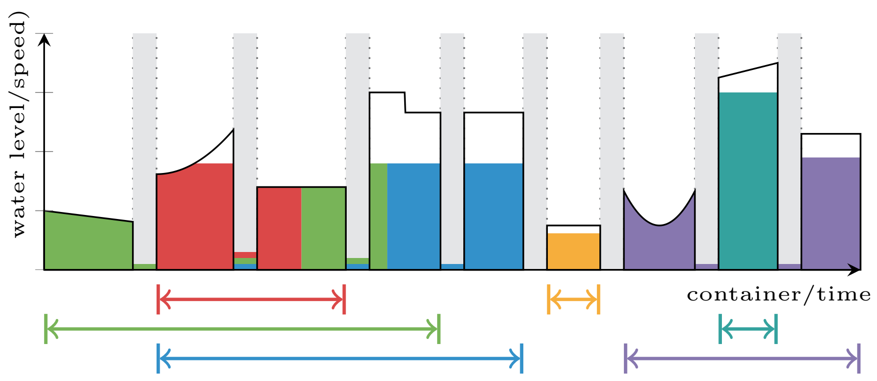


Figure 5.2: Water-filling: Atomic interval containers whose upper borders represent the maximum speed function $s_{\max}$. This example is for constant energy costs and shows a work-balanced schedule.

the optimality of S.

Property (b) of work-balanced schedules follows directly from the transitivity of $\twoheadrightarrow$ and the fact that s_i is defined as $\max_{t \in T_k, k \twoheadrightarrow i} s(t)c(t)^{\frac{1}{\alpha-1}}$.

Finally, assume S is optimal and not non-wasting. Obviously, we can uniformly decrease the speed for jobs with $w_j > \int_I s_j(t)\,\mathrm{d}t$ which leads to a lower energy cost and a contradiction.

□

5.3 Exact Polynomial-Time Algorithm

This section states our algorithm (Section 5.3.1) and proves both its correctness (via the work-balanced property, Theorem 5.10) and runtime bound (Theorem 5.11).

Overview Our algorithm can be seen as pouring a liquid (workload of the jobs) into a number of connected containers (atomic intervals). The upper border of these containers is given by the maximum speed function $s_{\max}$, and neighboring containers are connected with valves. Pouring liquid into the containers causes the water levels to rise evenly among all non-full containers, while the valves ensure that the workload of a job does not leave its release-deadline interval. The process is stopped when all the liquid has been poured. The water level essentially corresponds to the speed used in the atomic interval. Figure 5.2 illustrates this intuition for constant energy costs.

If we consider dynamic electricity rates, the situation becomes more complicated. Here, the energy costs at time t can be interpreted as changing the liquids density over time. The water levels no longer correspond immediately to job speeds. Instead, a job's speed at time t is essentially given by its water level times the density factor at time t. We note that water-filling is a natural way of viewing primal-dual algorithms (see, e.g., water-filling algorithms in [BV04, Chap. 5]).

5.3.1 Algorithm Description

Our algorithm works in rounds. In the first round, we find the set of consecutive atomic intervals $T_{i_1}, T_{i_1+1}, \dots, T_{i_2}$ that require the "highest water level". This fixes the schedule for T_{i_1} to T_{i_2}. We then remove these atomic intervals and the scheduled jobs from the input, adapt the remaining jobs' release and deadlines, and start over again. We continue by formally defining *water levels* and describe more exactly how the algorithm computes a schedule in each round. See Listing 5.1 for the corresponding pseudocode.

Computing Water Levels Consider a collection (union) I of atomic intervals. In the following, one can mostly think of I as a union of consecutive atomic intervals. However, for our proofs we also need to cover the case that I contains "holes" (i.e., a union of nonconsecutive atomic intervals). Define the set $J(I) :=$ $\{\, j \in J \mid r_j \in I, d_j \in \overline{I} \,\}$ of jobs whose release times and deadlines are contained in I and the closure of I, respectively. Moreover, let $W(I) := \sum_{j \in J(I)} w_j$ denote the total workload of these jobs. For a time $t \in \mathbb{R}_{\geq 0}$ let $\phi(t) := c(t)^{-1/\alpha - 1}$ denote the *density factor at time t*. We define the *water level* $\rho(I) \in \mathbb{R}_{\geq 0}$ of I as the solution to the equation

$$W(I) = \int_I \min\bigl(\phi(t) \cdot \rho(I), s_{\max}(t)\bigr) \,\mathrm{d}t. \tag{5.15}$$

Equation (5.15) has a solution if and only if $W(I) \leq \int_I s_{\max}(t) \,\mathrm{d}t$. If this inequality is strict, the solution is unique. If it is an equality, we agree on $\rho(I) = \sup_{t \in I} \frac{s_{\max}(t)}{\phi(t)}$. If there is no solution to Equation (5.15), we define $\rho(I) := \infty$.

Note that the computability of $\rho(I)$ depends not only on the ability to compute the involved integrals. Rather, one also must be able to solve an *integral equation* involving $s_{\max}$ and c. This is possible for practically relevant functions but can be nontrivial depending on $s_{\max}$ and c (e.g., for high-degree polynomials). In such cases, one can use numerical methods such as binary search. Since our focus lies on the combinatorial scheduling aspect and continuity of the involved functions, we assume that $\rho(I)$ can be computed efficiently.

From Water Levels to Schedules We describe the algorithm in an iterative way. This gives not the most efficient implementation but simplifies the analysis. Our algorithm iteratively computes a schedule for a subset of jobs and removes these from the input, creating a new subinstance of the original problem. This is then solved in the next iteration.

Set $I_0 := \emptyset$ and consider an iteration $k \geq 1$. We first find indices i_{1k} and i_{2k} for which the water level $\rho_k := \rho(I_k)$ of $I_k := \bigl(\bigcup_{i=i_{1k}}^{i_{2k}} T_i\bigr) \setminus \bigl(\bigcup_{k'<k} I_{k'}\bigr)$ is maximal. If this water level is ∞, the problem instance is infeasible. Otherwise, we can schedule all

```
$A \coloneqq \{ 1, 2, \dots, m \}$                         {remaining atomic interval indices}
$B \coloneqq \emptyset$, $\mathcal{J} \coloneqq J$           {removed atomic interval indices & remaining jobs}
while $\mathcal{J} \neq \emptyset$:
    for each pair $i_1 \leq i_2$ from $A$: compute water level $\rho(i_1, i_2) \coloneqq \rho\big(\bigcup_{i \in [i_1, i_2] \setminus B} T_i\big)$
    find maximum water level $\rho_k \coloneqq \rho(i_{1k}, i_{2k}) = \max_{i_1, i_2} \rho(i_1, i_2)$
    $I_k \coloneqq \{ i \in \mathbb{N} \mid i_{1k} \leq i \leq i_{2k}, i \notin B \}$          {atomic intervals of this iteration}
    if $\rho_k = \infty$: return infeasible                              { feasibility check}
    set $s_k(t) \coloneqq \min\big(\phi(t) \cdot \rho_k, s_{\max}(t)\big)$   {speed to be used in atomic intervals of $I_k$}
    $A \coloneqq A \setminus I_k$, $B \coloneqq B \cup I_k$, $\mathcal{J} \coloneqq \mathcal{J} \setminus J(I_k)$
    for all $j \in \mathcal{J}$: update release times and deadlines
```

Listing 5.1: Primal-dual algorithm for the `ContBERS` problem. It returns the speed functions s_k to be used during the atomic intervals I_k of iteration k. To keep the pseudocode simple, we define the interval collections I_k as index sets instead of the actual unions of atomic intervals.

jobs $J_k \coloneqq J(I_k)$ during I_k by using the EDF (earliest deadline first) scheduling policy and the speed function $s_k(t) \coloneqq \min(\phi(t) \cdot \rho_k, s_{\max}(t))$ during I_k. At the end of iteration k, we remove the scheduled jobs J_k and the time subset I_k from the input. This entails updating any remaining release time $r_j \in I_k$ to $\min \{ t \geq r_j \mid t \notin \bigcup_{k' \leq k} I_{k'} \}$ and any remaining deadline $d_j \in \overline{I_k}$ to $\sup \{ t \leq d_j \mid t \notin \bigcup_{k' \leq k} \overline{I_{k'}} \}$.

5.3.2 Correctness & Runtime

Before we state and prove our main result, we give an auxiliary lemma, showing that the water levels computed by our algorithm are monotonously decreasing.

Lemma 5.9. *The algorithm's water levels ρ_k are monotonously decreasing in k.*

Proof. Assume this is not true, so there is a minimal k such that $\rho_k < \rho_{k+1}$. First note that there are $u_1 < u_2$ and $v_1 < v_2$ with $I_k = [u_1, u_2) \setminus \bigcup_{k' < k} I_{k'}$ and $I_{k+1} = [v_1, v_2) \setminus \bigcup_{k' < k+1} I_{k'}$. We consider two cases:

Case 1: $[u_1, u_2] \cap [v_1, v_2] = \emptyset$

Note that, in this case, job removals and changes to release times or deadlines from iteration k cannot affect the job set $J(I_{k+1})$ in iteration $k + 1$. But then, since our algorithm also considered I_{k+1} in iteration k, it would have computed the same water level $\rho_{k+1} > \rho_k$ for I_{k+1} in this iteration and chosen it instead of I_k, which is a contradiction.

Case 2: $[u_1, u_2] \cap [v_1, v_2] \neq \emptyset$

Consider the interval $I \coloneqq I_k \cup I_{k+1}$. Because of $[u_1, u_2] \cap [v_1, v_2] \neq \emptyset$, our algorithm did consider I during iteration k. Moreover, note that $J(I)$ (in iteration k) contains both the job set $J(I_k)$ from iteration k and the job set $J(I_{k+1})$ from iteration $k + 1$. Together with the definition of water levels, we have

$$\begin{aligned} W(I) \geq & W(I_k) + W(I_{k+1}) = \int_{I_k} \min(\phi(t) \cdot \rho_k, s_{\max}(t))\,\mathrm{d}t \\ & + \int_{I_{k+1}} \min(\phi(t) \cdot \rho_{k+1}, s_{\max}(t))\,\mathrm{d}t > \int_{I} \min(\phi(t) \cdot \rho_k, s_{\max}(t))\,\mathrm{d}t. \end{aligned}$$

Since the function $x \mapsto \int_I \min(\phi(t) \cdot x, s_{\max}(t))\,\mathrm{d}t$ is non-decreasing, I's water level in iteration k must be larger than ρ_k, yielding a contradiction to our algorithm's choice.

□

Given this lemma, we can now prove the correctness of our algorithm.

Theorem 5.10 (Correctness)**.** *Consider an instance of the* `ContBERS` *problem. If there exists a feasible solution, our algorithm returns a work-balanced and non-wasting schedule. In particular, the returned schedule is optimal.*

Proof. Assume there is a feasible solution to the given instance. We first show that our algorithm returns a non-wasting schedule. Afterward, we show that this schedule is also work-balanced (implying its optimality by Theorem 5.1).

For the first iteration's water level, we have $\rho_1 < \infty$. Otherwise, even running all the time at maximum speed would not finish all jobs in J_1, causing any schedule to be infeasible. Moreover, it is easy to see that EDF together with the speed function s_1 on I_1 yields a feasible (and non-wasting) schedule for the jobs J_1. This is because, by Equation (5.15), EDF with speed function s_1 exactly finishes the *workload* of all jobs within I_1 (i.e., when ignoring release times and deadlines). If this schedule is infeasible, there must be an $I'_1 \subset I_1$ with $W(I'_1) > \int_{I'_1} \min(\phi(t) \cdot \rho_1, s_{\max}(t))\,\mathrm{d}t$. But then, since $x \mapsto \int_{I'_1} \min(\phi(t) \cdot x, s_{\max}(t))\,\mathrm{d}t$ is non-decreasing and continuous, we get $\rho(I'_1) > \rho_1$, contradicting the maximality of ρ_1. Now consider a later iteration k and assume we found a feasible subschedule in the previous iteration $k-1$. We immediately get $\rho_k < \infty$ by Lemma 5.9 ($\rho_k = \infty$ would contradict $\rho_k \leq \rho_{k-1} < \infty$). The feasibility and non-wasting property of EDF with speed function s_k in I_k follows by the same argument as for the first iteration.

We continue to show that the algorithm computes a work-balanced schedule. To this end, we show that the constants s_i from Definition 5.6 are given by the water levels ρ_k with $T_i \subseteq I_k$. The first part of this definition is obviously met, as it corresponds exactly to our definition of water levels and the speed functions in I_k. For the second part, note that we need only to consider two atomic intervals $T_i \subseteq I_k$ and $T_{i'} \subseteq I_{k'}$ from *different iterations* $k < k'$ (if they are from the same iteration, their water levels match, such that the definition's second part holds trivially). By construction of the algorithm, we cannot have $i \twoheadrightarrow i'$: no j scheduled in T_i is active outside of $\bigcup_{k'' \leq k} I_{k''}$, and the same holds for any j scheduled in $\bigcup_{k'' \leq k} I_{k''}$ (and thus, no such job is active in $I_{k'}$). On the other hand, if $i' \twoheadrightarrow i$, the second part of Definition 5.6 holds as $\rho_{k'} \leq \rho_k$ by Lemma 5.9. □

Note that the runtime of any algorithm for the `ContBERS` problem inherently depends on the ability to perform advanced computations on continuous functions. Depending on $s_{\max}$ and the cost factor function c, relying on numerical methods might even be unavoidable. Since we are interested in the scheduling aspect of the model, the following runtime discussion assumes computations (such as integrals, solving equations, taking the minimum etc.) involving continuous functions can be performed in constant time. As noted earlier, our iterative implementation is not the most efficient one, but it is convenient for our correctness analysis. A rather simple improvement can be achieved by precomputing the water levels for all pairs i_1 and i_2 of atomic intervals beforehand and merely updating these values at the end of each iteration. This immediately yields the same cubic runtime as known from the original YDS algorithm [YDS95][1]:

Theorem 5.11 (Runtime)**.** *The `ContBERS` problem can be solved in time* $\mathrm{O}\left(n^3\right)$.

[1]There are improved implementations of YDS with runtime $\mathrm{O}\left(n^2 \log n\right)$ [LYY06] and $\mathrm{O}\left(n^2\right)$ [LYY17].

CHAPTER 6

Conclusion & Outlook

The main focus of this thesis lies on models where one common resource is shared by a number of processors. In the following, I discuss the results and future directions for each chapter as well as possible extensions for the whole research field. I also briefly comment on possible online variants of the considered models.

Assigning a Sharable Resource in a Multiprocessor System. In Chapter 2, the assignment of the resource to different processors is considered. The assignment of the jobs to the processors and their order is assumed to be already fixed. Even for unit size jobs, this problem turns out to be NP-hard in the number of processors. However, if the number of processors is constant, the problem can be solved optimally in polynomial time. The respective algorithm merely proves that such solutions exist, but it is by no means practical as its runtime is roughly $\mathrm{O}\left(n^{2(m+1)^2}\right)$. Thus, for $m = 2$, the runtime of the general algorithm is already $\mathrm{O}\left(n^{18}\right)$ and for $m = 3$ even $\mathrm{O}\left(n^{32}\right)$. While for two processors an exact quadratic-time algorithm is presented, it remains an open question whether there are algorithms that have a similar runtime (such as $\mathrm{O}\left(n^m\right)$) for $m \geq 3$. Consequently, a linear-time approximation algorithm with a worst-case approximation ratio of $2 - 1/m$ is provided for an arbitrary number of processors m.

Restricting the analysis to unit size jobs, no analytical results for jobs of arbitrary sizes[1] are given, but I conjecture that the results should be transferable. However, extending the analysis turns out to be non-trivial. In particular the scheduling of (hyper-)graphs cannot, with their current definition, capture such problem instances. And yet, intuition suggests that one should be able to extend the definitions and

[1] One could also consider resource requirements > 1. However, the most natural extension of the considered model can easily be shown to reduce to non-unit size jobs with resource requirements ≤ 1 (rescale jobs with resource requirement $r > 1$ and workload p such that it has resource requirement $1/r \cdot r = 1$ and workload $r \cdot p$).

find similar structural properties for arbitrary job sizes.

Regarding online variants of this problem, the most natural model seems to be to assume that for each processor, only the next (i.e., current) job in the queue is visible and all jobs behind this job are yet unknown. Assuming this model, there is the following simple lower bound proving that any online algorithm cannot achieve a better competitive ratio than $2 - 1/m$. Given m^2 *large* jobs with a resource requirement of $1 - \varepsilon$, the adversary distributes them evenly among the m processors, resulting in m large jobs on each processor. If ε is chosen sufficiently small, this implies that any algorithm needs m^2 time steps until all these jobs are finished. The adversary then adds $m \cdot (m - 1)$ *small* jobs with a resource requirement of ε to the processor whose workload is finished last, resulting in an overall makespan of $m^2 + m(m - 1)$ for the algorithm's schedule. In contrast, the optimal (offline) algorithm prioritizes the processor containing the small jobs and executes them together with the large jobs, leading to an overall makespan of $m + m(m - 1) = m^2$. The lower bound of $2 - 1/m$ follows.

Note that the simple round robin algorithm introduced in Chapter 2 also works online and achieves an approximation ratio of 2. On the other hand, the introduced approximation algorithm with guarantee $2 - 1/m$ cannot be adapted to be used online, as it incorporates the length of the queues. I suspect that the best way to proceed to possibly find a tight online algorithm is to try to adapt the round robin algorithm. More precisely, it currently works in phases, finishing only one *column* of the jobs at a time. This simplifies the analysis, but may result in an inferior approximation guarantee compared to a round robin approach where the next job is started right away if a part of the resource is remaining. For a slightly modified variant of the tight example in Figure 2.3 (the variant making it harder for the improved round robin algorithm) as well as the lower bound above, at least, such a round robin approach results in the desired $(2 - 1/m)$-approximation, indicating that this guarantee may also hold in general.

Multiprocessor Scheduling with a Sharable Communication Channel. In the model from Chapter 3, composed services have to be scheduled, whereas the communication demand is assumed to be the bottleneck of the system. In order to complete a set of jobs, the communication demand on their interconnecting edges has to be processed. Similar to the prior model, the analysis is limited to unit size items: that is, a single edge can be completed in one time step as long as the full communication demand is supplied. However, the communication requirement for each edge is allowed to be arbitrarily high, implying that the demand of a single edge may exceed the capacity of the communication channel and making it impossible to fulfill the full communication requirement of such an edge in a single time step. At first sight, this seems natural, as the only requirement that has to be fulfilled is actually the communication demand. Hence, there is no reason why it should not be possible to complete the whole communication requirement in a single time step as long as its full requirement does not exceed the capacity of the channel.

On the other hand, computational limits on the nodes could make it impossible to process the amount of data of a single edge at once, implying that only a part of the overall communication demand can be finished in a single time step. This results in the necessity to introduce a second parameter, the size of an edge. This is the number of time steps this edge needs to receive its full communication demand in order to be completed (thus resulting in a similar two-dimensional problem as in the model of Chapter 4).

However, as our algorithms rely on the fact that any item can be packed in a single bin, I suspect that the extension to non-unit size items is non-trivial. But for the cases where Chapter 4 improves upon the results of this chapter, the same method can naturally be used to derive guarantees for the non-unit size case as well: The graph is split into single edges and each edge is scheduled separately.

For online variants of this problem, a reasonable model is to introduce release times. The most interesting variant seems to be that tasks arrive at different times, but all parts of a task (i.e., the items in the underlying bin packing representation) still arrive at the same time (thus no new communication requirements between already present nodes are added during the execution). In turn, it also makes sense to modify the optimization goal, for example to minimize the average flow time. This variant differs a lot from the model considered in Chapter 3 and, personally, I think that it may be impossible to find an $O(1)$-competitive online algorithm. As this model is a variant of the model in Chapter 4, but with more constraints, I will discuss the reasoning behind this hypothesis in more detail in the next paragraph.

Multiprocessor Scheduling with a Sharable Resource. In contrast to the models from Chapters 2 and 3, arbitrarily sized jobs are considered in Chapter 4. However, in doing so, the approximation ratio increases. While this is to be expected, I am confident that there is room to improve the ratio. Omitting the resource, the sorted list scheduling algorithm yields a ratio of roughly $4/3$ (cf. Section 1.4). Recall that this algorithm simply creates a list sorted by non-increasing job size and greedily schedules these jobs on the processor with the lowest load currently available. This algorithm yields a much better guarantee than the achieved approximation of roughly 2. The difficulty lies in combining the two approaches of prioritizing the larger jobs and at the same time ensuring that resource and parallelism both remain high. A reasonable approach seems to be to introduce some kind of trade-off between these two properties. But while I still suspect that this approach could be successful, a working approach together with the proper analysis seems to be hard to find.

Also, the procedure for the task scheduling problem may be improved by modifying it with respect to how the set of tasks is divided into two sets. Currently, the scheduler does not take into account how many tasks are restricted by the overall resource, and how many of them are restricted by the number of jobs that can be executed at the same time. Choosing the partition of the tasks into the two task sets using this information cleverly may lead to a better approximation. The objective being the average completion time, however, complicates analyzing an arbitrary trade-off.

That is, in case the workload of one of the task sets (say $\mathcal{T}_1$) is small, an optimal algorithm could still prioritize the few tasks (with small workload) in $\mathcal{T}_1$. But a different partition supplying $\mathcal{T}_1$ with reduced resources forces the algorithm to stretch these short tasks, hence drastically increasing the cost.

Considering online variants of this problem, a natural extension arises in adding release times to the jobs (see also prior paragraph). In doing so, a new optimization goal such as average flow time should be defined. This modification results in a significantly different model, and I hypothesize that an $\mathrm{O}(1)$-competitive online algorithm for this problem does not exist. I believe so because errors made at some point in time affect the full future timeline.

Regarding polynomial-time online algorithms, an indication that such an algorithm does not exist stems from the following lower bound for the offline version of average flow time scheduling with job lengths and without resource constraints [LR07]: The authors prove that there is no polynomial-time (offline) algorithm approximating the problem within a factor of $\mathrm{O}\left(n^{1/3-\varepsilon}\right)$ for some $\varepsilon > 0$ unless $P = NP$. However, I was not able to adjust this bound such that it carries over to the unit size model from Chapter 4 with additional release times (whereas it trivially applies to the non-unit size model with additional release times). This is because when scheduling multiple jobs with a high resource requirement at the same time, one can stretch and compress the jobs on the different processors, whereas this is not possible for jobs that have a specified length.

On the other hand, I was confident for a while that a *resource augmentation* in the form of one additional processor renders the suggested sliding window algorithm optimal for the online problem, at least for the unit size variant. The idea is to allow the online algorithm to use $m+1$ instead of m processors and to reserve one processor for the leftovers of prior jobs. However, it turns out that this is not sufficient, so the algorithm used can at least not directly be applied for the online problem with a resource augmentation of only one processor. The problem regarding the adaptability of the algorithm is that the chosen jobs in the sliding window depend on the remaining job(s), as the underlying model is non-preemptive and jobs, once started, cannot be removed from the system. This may result in the algorithm choosing the wrong jobs and being stuck with jobs with a large resource requirement, resulting in a suboptimal solution. For the interested reader, the following example illustrates this problem in more detail (a resource augmentation of $m+1$ over m processors is assumed).

Given one job with a resource requirement of $\frac{1}{m} - \varepsilon$, $m-1$ *small* jobs with a resource requirement of $\frac{1}{m}$ and one *large* job of resource requirement $1-(m-1)\varepsilon$. No matter how the sliding window is chosen (as far as possible to the left or to the right), the algorithm always finishes all except the last job. Also, it starts the last job and finishes a fraction of ε of it. In contrast, the optimal algorithm only finishes the first m jobs. In each of the next m steps, add one *tiny* job with a resource requirement of $\frac{1}{2m}$, $m-1$ small jobs with a resource requirement of $\frac{1}{m}$ and one *medium* job with a resource requirement of $\frac{3}{2m}$. As the large job has been started in the first step of ALG, the sliding window of ALG always leads to the execution of the tiny and small jobs. In contrast, OPT finishes one tiny job, $m - 2$ small jobs and one medium job. Then, after these m steps, ALG has left m jobs with a resource requirement of $\frac{3}{2m}$ and one job with a resource requirement of $\frac{1}{2} - m\varepsilon$. On the other hand, OPT has left m jobs with a resource requirement of $\frac{1}{m}$ and one job with a resource requirement of $1 - (m - 1)\varepsilon$. In this step, no additional jobs are added. Hence, OPT can finish all m jobs with a resource requirement of $\frac{1}{m}$, whereas ALG can only finish $\frac{2m}{3}$ jobs with a resource requirement of $\frac{3}{2m}$. In the last time step, ALG has $\frac{m}{3}$ jobs with a resource requirement of $\frac{3m}{2}$ and one job with a resource requirement of $\frac{1}{2} - m\varepsilon$ left. OPT has left just one job with a resource requirement of $1 - (m - 1)\varepsilon$. Finally, $m - 1$ jobs with a resource requirement of ε are added. OPT will finish all remaining jobs. ALG has $\frac{4m}{3}$ jobs left, which cannot be finished in one time step.

Note that this example does not give insights whether the algorithm achieves a constant competitive ratio under resource augmentation. Also, it does not imply that a larger resource augmentation does not result in an optimal solution. Further, a modification of the algorithm or a completely different algorithm may lead to better results, but as stated earlier I suspect that an $\mathrm{O}(1)$-competitive online algorithm without resource augmentation is not possible.

General Extensions of Scheduling with Scarce Resources. The problem variants from Chapters 2 to 4 only deal with discrete time models, both because it facilitates the analysis and because it fits well in typical implementations of real-world schedulers (which are usually called at regular time intervals [Bła+07]). Nevertheless, it seems an intriguing question to consider these models in a more sophisticated, continuous setting where the scheduler can act at arbitrary times. Also, these chapters mostly focus on practically viable algorithms, that is, simple algorithms with a short runtime. I strongly believe that the results could be improved with respect to the approximation ratio by allowing arbitrary polynomial runtimes and considering polynomial-time approximation schemes. As indicated before, for the unit size variant of the job scheduling model in Chapter 4, a PTAS already exists [ES07], and adopting it to jobs of arbitrary size should be possible by additionally rounding job lengths to multiples of $\varepsilon^2\mathrm{OPT}/m$, removing long jobs like Epstein and Stee [ES07], removing resource intensive jobs, grouping short jobs, and slightly modifying the grouping and rounding steps with respect to resource requirement.

Scheduling with a Bounded Speed Limit and Variable Energy Costs. Chapter 5 deals with energy-efficient scheduling with variable energy prices and upper speed limits. Also, new combinatorial optimality conditions are derived that will hopefully be useful beyond the scope of this work. As these conditions are a variant of the KKT conditions that can be applied on continuous optimization problems (with variables from a function space), it should be particularly interesting for extensions of discrete problems where the KKT conditions could be used to prove optimality. In using the proposed extended KKT conditions and adapting the algorithms accordingly, it should be possible to prove the optimality of these algorithms for continuous variants of interesting discrete problems. However, note that a slightly less general variant of these conditions was already stated earlier [FH74] and to the best of my knowledge, there are barely applications. As many problems in computer science are still discrete by nature, we also have not yet succeeded in finding relevant problems to which our framework can be directly applied. Nevertheless, I see potential in these continuous optimization problems, but it may turn out to be more fruitful to look for problems that are continuous by nature.

Regarding online variants of this problem, the main issue lies in the upper speed limit, which may force an online algorithm to produce an infeasible solution. That is, if an online algorithm runs at full speed (i.e., right at the speed limit) even though there is plenty of time for the currently available jobs, the competitive ratio can be arbitrarily bad. On the other hand, if an online algorithm does not run at full speed in the same setting, an adversary can add enough jobs with high workload such that the only feasible solution would have been to run with full speed over the full timeline. In doing so, such a solution is rendered infeasible. Hence, for future online variants of this problem, the adversary has to be restricted in a reasonable way.

Combining Scarce Resources and Energy Efficiency. At the intersection of Chapters 2 to 4 and Chapter 5, a very interesting model arises in assuming a shared energy source in a computing center. That is, the resource in form of available energy is shared among the m processors and it is the goal of the scheduler to generate a schedule that consumes as little energy as possible and at the same time does not violate the power limit. However, the additional complexity arising by adding multiple processors somehow seems to be higher if the overall energy is (dynamically) restricted than in the original YDS model. That is, we were not able to adapt the approaches that were used to extend the YDS algorithm to multiple processors (i.e., without an energy limit) to the model with energy limits. Nevertheless, I suspect that this problem can still be solved optimally in polynomial time, yielding an enthralling open question.

List of Figures

Bibliography

[17] *Collaborative Research Centre 901 - On-The-Fly-Computing.* `http://sfb901.uni-paderborn.de`. 2017.

[AF07] Susanne Albers and Hiroshi Fujiwara. “Energy-Efficient Algorithms for Flow Time Minimization”. In: *ACM Transactions on Algorithms* vol. 3, no. 4 (2007).

[Alb10] Susanne Albers. “Energy-Efficient Algorithms”. In: *Communications of the ACM* vol. 53, no. 5 (2010), pp. 86–96.

[Alb11] Susanne Albers. “Algorithms for Dynamic Speed Scaling”. In: *Proceedings of the 28th International Symposium on Theoretical Aspects of Computer Science (STACS).* 2011, pp. 1–11.

[Alo+98] Noga Alon, Yossi Azar, Gerhard J. Woeginger, and Tal Yadid. “Approximation Schemes for Scheduling on Parallel Machines”. In: *Journal of Scheduling* vol. 1, no. 1 (1998), pp. 55–66.

[Alt+17] Ernst Althaus, André Brinkmann, Peter Kling, Friedhelm Meyer auf der Heide, Lars Nagel, Sören Riechers, Jiří Sgall, and Tim Süß. “Scheduling Shared Continuous Resources on Many-Cores”. In: *Journal of Scheduling* (2017). DOI: `10.1007/s10951-017-0518-0`.

[Ant+14] Antonios Antoniadis, Neal Barcelo, Mario E. Consuegra, Peter Kling, Michael Nugent, Kirk Pruhs, and Michele Scquizzato. “Efficient Computation of Optimal Energy and Fractional Weighted Flow Trade-off Schedules”. In: *Proceedings of the 31st International Symposium on Theoretical Aspects of Computer Science (STACS).* 2014, pp. 63–74.

[Ant+17] Antonios Antoniadis, Peter Kling, Sebastian Ott, and Sören Riechers. “Continuous Speed Scaling with Variability: A Simple and Direct Approach”. In: *Theoretical Computer Science* vol. 678 (2017), pp. 1–13.

[Ban+08] Nikhil Bansal, David P. Bunde, Ho-Leung Chan, and Kirk Pruhs. “Average Rate Speed Scaling”. In: *Proceedings of the 8th Latin American Symposium on Theoretical Informatics (LATIN).* Springer. 2008, pp. 240–251.

[Ban+09] Nikhil Bansal, Ho-Leung Chan, Kirk Pruhs, and Dmitriy Katz. “Improved Bounds for Speed Scaling in Devices Obeying the Cube-Root Rule”. In: *Proceedings of the 36th International Colloquium on Automata, Languages and Programming (ICALP).* Springer, 2009, pp. 144–155.

[Ban+11] Nikhil Bansal, David P. Bunde, Ho-Leung Chan, and Kirk Pruhs. “Average Rate Speed Scaling”. In: *Algorithmica* vol. 60, no. 4 (2011), pp. 877–889.

[Bar+13] Neal Barcelo, Daniel Cole, Dimitrios Letsios, Michael Nugent, and Kirk Pruhs. “Optimal Energy Trade-off Schedules”. In: *Sustainable Computing: Informatics and Systems* vol. 3, no. 3 (2013), pp. 207–217.

[BCP09] Nikhil Bansal, Ho-Leung Chan, and Kirk Pruhs. “Speed Scaling with a Solar Cell”. In: *Theoretical Computer Science* vol. 410, no. 45 (2009), pp. 4580–4587.

[BCP13] Nikhil Bansal, Ho-Leung Chan, and Kirk Pruhs. “Speed Scaling with an Arbitrary Power Function”. In: *ACM Transactions on Algorithms* vol. 9, no. 2 (2013).

[Bem+17] Pascal Bemmann, Felix Biermeier, Jan Bürmann, Arne Kemper, Till Knollmann, Steffen Knorr, Nils Kothe, Alexander Mäcker, Manuel Malatyali, Friedhelm Meyer auf der Heide, Sören Riechers, Johannes Schaefer, and Jannik Sundermeier. “Monitoring of Domain-Related Problems in Distributed Data Streams (to appear)”. In: *Proceedings of the 24th International Colloquium on Structural Information and Communication Complexity (SIROCCO).* Springer, 2017.

[BH07] Luiz André Barroso and Urs Hölzle. “The Case for Energy-Proportional Computing”. In: *Computer* vol. 40, no. 12 (2007), pp. 33–37.

[BKP07] Nikhil Bansal, Tracy Kimbrel, and Kirk Pruhs. “Speed Scaling to Manage Energy and Temperature”. In: *Journal of the ACM* vol. 54, no. 1 (2007), 3:1–3:39.

[Bła+07] Jacek Błażewicz, Klaus H. Ecker, Erwin Pesch, Günter Schmidt, and Jan Weglarz. *Handbook on Scheduling: From Theory to Applications.* Springer, 2007.

[BLK83] J. Blazewicz, J.K. Lenstra, and A.H.G.Rinnooy Kan. “Scheduling Subject to Resource Constraints: Classification and Complexity”. In: *Discrete Applied Mathematics* vol. 5, no. 1 (1983), pp. 11–24.

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

[Bro+00] David M. Brooks, Pradip Bose, Stanley E. Schuster, Hans Jacobson, Prabhakar N. Kudva, Alper Buyuktosunoglu, John-David Wellman, Victor Zyuban, Manish Gupta, and Peter W. Cook. “Power-Aware Microarchitecture: Design and Modeling Challenges for Next-Generation Microprocessors”. In: *IEEE Micro* vol. 20, no. 6 (2000), pp. 26–44.

[BV04] Stephan P. Boyd and Lieven Vandenberghe. *Convex Optimization.* Cambridge University Press, 2004.

[Cha+09] Ho-Leung Chan, Joseph Wun-Tat Chan, Tak Wah Lam, Lap-Kei Lee, Kin-Sum Mak, and Prudence W. H. Wong. “Optimizing Throughput and Energy in Online Deadline Scheduling”. In: *ACM Transactions on Algorithms* vol. 6, no. 1 (2009).

[Chu+06] Fan Chung, Ronald Graham, Jia Mao, and George Varghese. “Parallelism versus Memory Allocation in Pipelined Router Forwarding Engines”. In: *Theory of Computing Systems* vol. 39, no. 6 (2006), pp. 829–849.

[CO02] K.G. Coffman and A.M. Odlyzko. “Internet Growth: Is There a ”Moore’s Law” for Data Traffic?” English. In: *Handbook of Massive Data Sets*. Vol. 4. Massive Computing. Springer, 2002, pp. 47–93.

[Dós+13] György Dósa, Rongheng Li, Xin Han, and Zsolt Tuza. “Tight absolute bound for First Fit Decreasing bin-packing: FFD(l) $\leq$ 11/9 OPT(L) + 6/9”. In: *Theoretical Computer Science* vol. 510 (2013), pp. 13–61.

[Dre+15] Maximilian Drees, Matthias Feldotto, Sören Riechers, and Alexander Skopalik. “On Existence and Properties of Approximate Pure Nash Equilibria in Bandwidth Allocation Games”. In: *Proceedings of the 8th International Symposium on Algorithmic Game Theory (SAGT).* Springer, 2015, pp. 178–189.

[Dre+17] Maximilian Drees, Matthias Feldotto, Sören Riechers, and Alexander Skopalik. “Pure Nash Equilibria in Restricted Budget Games”. In: *Proceedings of the 23rd International Computing and Combinatorics Conference (COCOON).* Springer, 2017, pp. 175–187.

[DRS14] Maximilian Drees, Sören Riechers, and Alexander Skopalik. “Budget-Restricted Utility Games with Ordered Strategic Decisions”. In: *Proceedings of the 7th International Symposium on Algorithmic Game Theory (SAGT).* Springer. 2014, pp. 110–121.

[EL10] Leah Epstein and Asaf Levin. “AFPTAS Results for Common Variants of Bin Packing: A New Method for Handling the Small Items”. In: *SIAM Journal on Optimization* vol. 20, no. 6 (2010), pp. 3121–3145.

[ELS12] Leah Epstein, Asaf Levin, and Rob van Stee. “Approximation Schemes for Packing Splittable Items with Cardinality Constraints”. In: *Algorithmica* vol. 62, no. 1-2 (2012), pp. 102–129.

[ES07] Leah Epstein and Rob van Stee. “Approximation Schemes for Packing Splittable Items with Cardinality Constraints”. In: *Proceedings of the 5th International Workshop on Approximation and Online Algorithms (WAOA)*. Springer, 2007, pp. 232–245.

[ES11] Leah Epstein and Rob van Stee. “Improved Results for a Memory Allocation Problem”. In: *Theory of Computing Systems* vol. 48, no. 1 (2011), pp. 79–92.

[Fan+15] Kan Fang, Nelson A Uhan, Fu Zhao, and John W Sutherland. “Scheduling on a Single Machine under Time-of-Use Electricity Tariffs”. In: *Annals of Operations Research* (2015), pp. 1–29.

[FH74] William H Farr and Morgan A Hanson. “Continuous Time Programming with Nonlinear Constraints”. In: *Journal of Mathematical Analysis and Applications* vol. 45, no. 1 (1974), pp. 96–115.

[GG75] Michael R Garey and Ronald L. Graham. “Bounds for Multiprocessor Scheduling with Resource Constraints”. In: *SIAM Journal on Computing* vol. 4, no. 2 (1975), pp. 187–200.

[Gra69] Ronald L. Graham. “Bounds on Multiprocessing Timing Anomalies”. In: *SIAM Journal on Applied Mathematics* vol. 17, no. 2 (1969), pp. 416–429.

[GT14] Yannai A Gonczarowski and Moshe Tennenholtz. *Noncooperative Market Allocation and the Formation of Downtown.* Center for the Study of Rationality, 2014.

[GW92] Harold N. Gabow and Herbert H. Westermann. “Forests, Frames, and Games: Algorithms for Matroid Sums and Applications”. In: *Algorithmica* vol. 7, no. 1 (1992), pp. 465–497.

[Hap+13] Markus Happe, Friedhelm Meyer auf der Heide, Peter Kling, Marco Platzner, and Christian Plessl. “On-The-Fly Computing: A Novel Paradigm for Individualized IT Services”. In: *Proceedings of the 16th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC)*. IEEE, 2013, pp. 1–10.

[HM68] Morgan A Hanson and Bertram Mond. “A Class of Continuous Convex Programming Problems”. In: *Journal of Mathematical Analysis and Applications* vol. 22, no. 2 (1968), pp. 427–437.

[HS76] Ellis Horowitz and Sartaj Sahni. “Exact and Approximate Algorithms for Scheduling Nonidentical Processors”. In: *Journal of the ACM* vol. 23, no. 2 (1976), pp. 317–327.

[IP05] Sandy Irani and Kirk Pruhs. “Algorithmic Problems in Power Management”. In: *SIGACT News* vol. 36, no. 2 (2005), pp. 63–76.

[ITW14] ITWatchDogs. “Are Heat & Humidity Hurting Your Equipment?” In: *Processor* vol. 36, no. 20 (2014).

[JJL07] Adam Janiak, Władysław Janiak, and Maciej Lichtenstein. “Resource Management in Machine Scheduling Problems: A Survey”. In: *Decision Making in Manufacturing and Services* vol. 1, no. 12 (2007), pp. 59–89.

[JMR16] Klaus Jansen, Marten Maack, and Malin Rau. “Approximation Schemes for Machine Scheduling with Resource (In-)Dependent Processing Times”. In: *Proceedings of the 27th ACM-SIAM Symposium on Discrete Algorithms (SODA).* Society for Industrial and Applied Mathematics. 2016, pp. 1526–1542.

[Joh54] S. M. Johnson. “Optimal Two- and Three-Stage Production Schedules with Setup Times Included”. In: *Naval Research Logistics Quarterly* vol. 1, no. 1 (1954), pp. 61–68.

[Józ+00] Joanna Józefowska, Marek Mika, Rafał Różycki, Grzegorz Waligóra, and Jan Weglarz. “Solving the Discrete-Continuous Project Scheduling Problem via its Discretization”. In: *Mathematical Methods of Operations Research* vol. 52, no. 3 (2000), pp. 489–499.

[Józ+02] Joanna Józefowska, Marek Mika, Rafał Różycki, Grzegorz Waligóra, and Jan Weglarz. “A Heuristic Approach to Allocating the Continuous Resource in Discrete-Continuous Scheduling Problems to Minimize the Makespan”. In: *Journal of Scheduling* vol. 5, no. 6 (2002), pp. 487–499.

[Józ+99] Joanna Józefowska, Marek Mika, Rafal Różycki, Grzegorz Waligóra, and Jan Weglarz. “Discrete-Continuous Scheduling to Minimize the Makespan for Power Processing Rates of Jobs”. In: *Discrete Applied Mathematics* vol. 94, no. 1 (1999), pp. 263–285.

[JV01] Kamal Jain and Vijay V. Vazirani. “Approximation Algorithms for Metric Facility Location and k-Median Problems using the Primal-Dual Schema and Lagrangian Relaxation”. In: *Journal of the ACM* vol. 48, no. 2 (2001), pp. 274–296.

[JW96] Joanna Józefowska and Jan Weglarz. “Discrete-Continuous Scheduling Problems – Mean Completion Time Results”. In: *European Journal of Operational Research* vol. 94, no. 2 (1996), pp. 302–309.

[JW98] Joanna Józefowska and Jan Weglarz. “On a Methodology for Discrete-Continuous Scheduling”. In: *European Journal of Operational Research* vol. 107, no. 2 (1998), pp. 338–353.

[Kis05] Tamás Kis. “A Branch-and-Cut Algorithm for Scheduling of Projects with Variable-Intensity Activities”. In: *Mathematical Programming* vol. 103, no. 3 (2005), pp. 515–539.

[KK82] Narendra Karmarkar and Richard M. Karp. “An Efficient Approximation Scheme for the One-Dimensional Bin-Packing Problem”. In: *Proceedings of the 23rd IEEE Symposium on Foundations of Computer Science (FOCS).* IEEE, 1982, pp. 312–320.

[KL11] Tamás Király and Lap C. Lau. “Degree Bounded Forest Covering”. In: *Proceedings of the 15th International Conference on Integer Programming and Combinatorial Optimization (IPCO).* 2011, pp. 315–323.

[Kli+17] Peter Kling, Alexander Mäcker, Sören Riechers, and Alexander Skopalik. “Sharing is Caring: Multiprocessor Scheduling with a Sharable Resource”. In: *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA).* ACM, 2017, pp. 123–132.

[KM11] Peter Kosmol and Dieter Müller-Wichards. *Optimization in Function Spaces: With Stability Considerations in Orlicz Spaces.* De Gruyter, 2011.

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

[KT11] Iordanis Koutsopoulos and Leandros Tassiulas. “Control and Optimization Meet the Smart Power Grid: Scheduling of Power Demands for Optimal Energy Management”. In: *Proceedings of the 2nd International Conference on Energy-Efficient Computing and Networking (e-Energy).* 2011, pp. 41–50.

[Leu04] Joseph Y-T. Leung. *Handbook of Scheduling: Algorithms, Models, and Performance Analysis.* Chapman & Hall/CRC, 2004.

[Li+15] Shouwei Li, Alexander Mäcker, Christine Markarian, Friedhelm Meyer auf der Heide, and Sören Riechers. “Towards Flexible Demands in Online Leasing Problems”. In: *Proceedings of the 20th International Computing and Combinatorics Conference (COCOON).* Springer, 2015, pp. 277–288.

[Li+17] Shouwei Li, Alexander Mäcker, Christine Markarian, Friedhelm Meyer auf der Heide, and Sören Riechers. “Towards Flexible Demands in Online Leasing Problems (to appear)”. In: *Algorithmica* (2017).

[Li11] Minming Li. “Approximation Algorithms for Variable Voltage Processors: Min Energy, Max Throughput and Online Heuristics”. In: *Theoretical Computer Science* vol. 412, no. 32 (2011), pp. 4074–4080.

[LLP05] Joseph YT Leung, Haibing Li, and Michael Pinedo. “Order Scheduling Models: An Overview”. In: *Multidisciplinary Scheduling: Theory and Applications.* Springer, 2005, pp. 37–53.

[LR07] Stefano Leonardi and Danny Raz. “Approximating Total Flow Time on Parallel Machines”. In: *Journal of Computer and System Sciences* vol. 73, no. 6 (2007), pp. 875–891.

[Lue69] David G. Luenberger. *Optimization by Vector Space Methods.* John Wiley & Sons, Inc., 1969.

[LYY06] Minming Li, Andrew C. Yao, and Frances F. Yao. "Discrete and Continuous Min-Energy Schedules for Variable Voltage Processors". In: *Proceedings of the National Academy of Sciences of the United States of America (PNAS)* vol. 103, no. 11 (2006), pp. 3983–3987.

[LYY17] Minming Li, Frances F. Yao, and Hao Yuan. "An $O(n^2)$ Algorithm for Computing Optimal Continuous Voltage Schedules". In: *Proceedings of the 14th Conference on Theory and Applications of Models of Computation (TAMC).* Springer, 2017, pp. 389–400.

[Mäc+15] Alexander Mäcker, Manuel Malatyali, Friedhelm Meyer auf der Heide, and Sören Riechers. "Non-Preemptive Scheduling on Machines with Setup Times". In: *Proceedings of the 14th International Symposium on Algorithms and Data Structures (WADS).* Springer, 2015, pp. 542–553.

[Mäc+16] Alexander Mäcker, Manuel Malatyali, Friedhelm Meyer auf der Heide, and Sören Riechers. "Cost-Efficient Scheduling on Machines from the Cloud". In: *Proceedings of the 10th Annual International Conference on Combinatorial Optimization and Applications (COCOA).* Springer, 2016, pp. 578–592.

[Mäc+17a] Alexander Mäcker, Manuel Malatyali, Friedhelm Meyer auf der Heide, and Sören Riechers. "Cost-Efficient Scheduling on Machines from the Cloud". In: *Journal of Combinatorial Optimization* (2017). DOI: `10.1007/s10878-017-0198-x`.

[Mäc+17b] Alexander Mäcker, Manuel Malatyali, Friedhelm Meyer auf der Heide, and Sören Riechers. "Non-Clairvoyant Scheduling to Minimize Max Flow Time on a Machine with Setup Times (to appear)". In: *Proceedings of the 15th Workshop on Approximation and Online Algorithms (WAOA).* Springer, 2017.

[Moo65] Gordon E. Moore. "Cramming More Components onto Integrated Circuits". In: *Electronics* vol. 38, no. 8 (1965), pp. 114–117.

[NW15] Martin Niemeier and Andreas Wiese. "Scheduling with an Orthogonal Resource Constraint". In: *Algorithmica* vol. 71, no. 4 (2015), pp. 837–858.

[PS09] C. N. Potts and V. A. Strusevich. "Fifty Years of Scheduling: A Survey of Milestones". In: *Journal of the Operational Research Society* vol. 60, no. 1 (2009), S41–S68.

[RKB13] Johan M.M. van Rooij, Marcel E. van Kooten Niekerk, and Hans L. Bodlaender. "Partition Into Triangles on Bounded Degree Graphs". In: *Theory of Computing Systems* vol. 52, no. 4 (2013), pp. 687–718.

[Smi98] Donald. R. Smith. *Variational Methods in Optimization.* Dover Publications, 1998.

[Tha13] Nguyen Kim Thang. “Lagrangian Duality in Online Scheduling with Resource Augmentation and Speed Scaling”. In: *Proceedings of the 21st European Symposium on Algorithms (ESA)*. 2013, pp. 755–766.

[VL81] Wenceslas F. de la Vega and George S. Lueker. “Bin Packing Can Be Solved within $1+\varepsilon$ in Linear Time”. In: *Combinatorica* vol. 1, no. 4 (1981), pp. 349–355.

[Wal11] Grzegorz Waligóra. “Heuristic Approaches to Discrete-Continuous Project Scheduling Problems to Minimize the Makespan”. In: *Computational Optimization and Applications* vol. 48, no. 2 (2011), pp. 399–421.

[Weg+11] Jan Weglarz, Joanna Józefowska, Marek Mika, and Grzegorz Waligóra. “Project Scheduling with Finite or Infinite Number of Activity Processing Modes – A Survey”. In: *European Journal of Operational Research* vol. 208, no. 3 (2011), pp. 177–205.

[YDS95] Frances Foong Yao, Alan J. Demers, and Scott Shenker. “A Scheduling Model for Reduced CPU Energy”. In: *Proceedings of the 36th IEEE Symposium on Foundations of Computer Science (FOCS)*. 1995, pp. 374–382.

[Zhu+12] Sergey Zhuravlev, Juan Carlos Saez, Sergey Blagodurov, Alexandra Fedorova, and Manuel Prieto. “Survey of Scheduling Techniques for Addressing Shared Resources in Multicore Processors”. In: *ACM Computing Surveys* vol. 45, no. 1 (2012), 4:1–4:28.