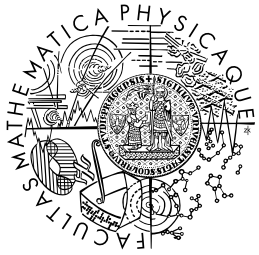

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Štěpán Šimsa

Online scheduling of multiprocessor jobs with preemption

Computer Science Institute of Charles University

Supervisor of the master thesis:	prof. RNDr. Jiří Sgall, DrSc.
Study programme:	Computer Science
Study branch:	Discrete Models and Algorithms

Prague 2018

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ..... signature of the author

I would like to thank my supervisor, Jiří Sgall, for the amount of time he gave me. Both throughout the last two years during (more or less) regular sessions and especially for the time when writing this thesis when he responded very quickly with reviews on developing parts of the thesis and with answer to all my questions.

I would also like to thank my fiancée, Bára Dolejší, for her support, care and understanding when I had to put all my focus on writing up this thesis.

Last but not least, I would like to thank my family, especially my parents, for my upbringing in a scientific environment, leading me to this point of my life.

Title: Online scheduling of multiprocessor jobs with preemption

Author: Štěpán Šimsa

Institute: Computer Science Institute of Charles University

Supervisor: prof. RNDr. Jiří Sgall, DrSc., Computer Science Institute of Charles University

Abstract: The thesis is devoted to the problem of online preemptive scheduling of multiprocessor jobs. It gives a summary of previous work on this problem. For some special variants of the problem, especially if we restrict the sizes of jobs to one and two, new results are given, both in the terms of lower bounds and in the terms of competitive algorithms. A previously published lower bound is showed to be computed incorrectly and it is replaced by a correct lower bound in this thesis. An algorithm is presented for the special case of four processors and sizes of jobs one and two that is conjectured to achieve the best possible competitive ratio.

Keywords: scheduling, online algorithms, preemption, multiprocessor jobs

Contents

Introduction

In this thesis we approach the problem of online Preemptive Scheduling of Multiprocessor Jobs (PSMJ) where the goal is to minimize the makespan. That means we have a system of m processors and there are jobs that arrive over time. The jobs have a size (sometimes called width) that is equal to the number of processors that are required to run the job, a processing time (how long we need to run the job to finish it) and a release time. An algorithm for this problem needs to assign pending jobs (i.e., the jobs that are not yet done but were already released) to the processors such that each job is assigned either to zero processors or to the number of processors corresponding to its size, and every processor has at most one job assigned to it. Preemption is allowed, i.e., job processing can be interrupted and resumed later, possibly on a different set of processors. But in total the algorithm needs to schedule each job for as long as is its processing time. The goal of the algorithm is to produce a schedule with the smallest possible makespan where makespan is the time of completion of the last job.

It is an example of an online problem, a problem where the algorithm does not know the whole instance in advance and needs to do decisions continuously, only based on the history and on the part of the instance already revealed. Often, there is no optimal online algorithm, as a decision might be good or bad depending on the future input. In that situation, the goal is to come up with an algorithm with a guarantee not as strong as optimality. We say that an algorithm is c-competitive if its output is at most c-times worse (so in the case of a minimization problem, as in PSMJ, at most c-times bigger) than is the optimum constructed from the whole instance (i.e., constructed by all-powerful algorithm that can guess the future). For this competitive ratio we can provide lower bounds (by showing instances on which no algorithm can achieve better guarantee) and upper bounds (by finding algorithms that achieve this guarantee).

Many variants of this problem were studied before, as we will see in more detail in Chapter 2. For us, the most relevant works are Sgall and Woeginger [2015] and Johannes [2006] as they contain results related to PSMJ specifically. The former gives a classification for existence of 1-competitive algorithm based on the set of allowed sizes of jobs (see Theorem 2.7 later). The later gives a lower bound 6/5 on the problem (see Claim 2.12). We show that this lower bound is actually not computed correctly (see Corollary 4.5). Johannes [2006] also gives a simple $(2 - (1/m))$-competitive algorithm which can be viewed as a starting point that this thesis was supposed to surpass.

As we said, the work of Sgall and Woeginger [2015] answers the question for what sets of allowed sizes of jobs there is a 1-competitive algorithm but it does not say anything about the competitive ratio for the other cases. And that is exactly what we approach in this thesis. Specifically, we focused on the smallest such example, that is when we have 4 processors and the jobs have allowed sizes only 1 or 2. The small cases usually give insights to the general problem and it is naive trying to solve a general problem if we are not able to solve small cases first.

Unfortunately, we were unable to find a matching lower bound and algorithm but we have some partial results. Specifically, we proved a lower bound of 9/8 (see

Theorem 4.4) and an algorithm with competitive ratio 5/4 (see Corollary 5.13). We think that the lower bound could be tight and we support this by providing an algorithm we conjecture could match the lower bound (see Section 5.4). Many of the results also generalize for any fixed number of processors (see for example Theorems 4.7, 4.9, Corollary 5.12) or even to sizes of jobs bounded by any constant (see Corollary 5.15).

The structure of the thesis is as follows. In Chapter 1 we go over the definitions and notation we will use throughout the thesis. In Chapter 2 we summarize previous work in this area, focusing on the results directly connected to our methods. In Chapter 3 we reformulate some of the previously known theorems and introduce some new theorems and observations, regarding optimal schedules, that we believe are important for `PSMJ`. In Chapter 4 we discuss the lower bounds for `PSMJ`, most notably, providing a correct lower bound for `PSMJ` that replaces the incorrect lower bound claimed by Johannes [2006]. Finally, Chapter 5 contains algorithms that achieve better competitive ratio than the ones known previously. We start with $m/(m-1)$-competitive algorithm (see Corollary 5.7) that we enhance into a $(m+1)/m$-competitive algorithm for m even (see Corollary 5.12) and generalize to $m/(m-k+1)$-competitive algorithm for jobs of sizes at most k (see Corollary 5.15). In the last section of the chapter, Section 5.4, we introduce an algorithm we conjecture matches the lower bound 9/8 for $m = 4$.

1. Basic definitions

1.1 Approximation and online algorithms

Before we will formally define variants of PSMJ approached in this thesis and their corresponding notation, we will define the notions of approximation and online algorithms.

Definition 1.1 (Combinatorial optimization problem)**.** *A combinatorial optimization problem is a quadruple $(\mathcal{I}, \mathcal{F}, o, g)$ where $\mathcal{I}$ is the set of instances, for instance $I \in \mathcal{I}, \mathcal{F}(I)$ is the set of feasible solutions, function o is the objective function and for $I \in \mathcal{I}, S \in \mathcal{F}, o(I, S)$ is the objective value of the solution. At last, g is the goal function and is either* min *or* max*.*

The goal is to find for some instance $I \in \mathcal{I}$ an optimal solution S^* such that

$$o(I, S^*) = g\{o(I, S) \mid S \in \mathcal{F}(I)\}.$$

We call $o(I, S^*)$ the optimal value of the problem and denote it by $\texttt{OPT}(I)$ or OPT *if the instance is clear from the context.*

Definition 1.2 (Approximation algorithm)**.** *Let us have a combinatorial optimization problem $(\mathcal{I}, \mathcal{F}, o, g)$. We say that algorithm* ALG *is an α-approximation algorithm for the problem if for any instance $I \in \mathcal{I}$ the algorithm returns a feasible solution $S \in \mathcal{F}(I)$ in polynomial time and its objective value $\texttt{ALG}(I) = o(I, S)$ satisfies the inequalities*

$$\frac{1}{\alpha} \leq \frac{\texttt{ALG}(I)}{\texttt{OPT}(I)} \leq \alpha.$$

We see that based on the goal function g only one of the inequalities is meaningful, the second one follows from the fact that the solution is feasible.

Next we define an *online algorithm*. In online problems, instance is given by parts and the algorithm needs to construct the solution on the go, without the knowledge of the future parts. The definition that follows is given in the *online-time* model where the algorithm decides what to do at every time point $t \in \mathbb{R}_0^+$. In different definitions the instance is given by a finite sequence of instance parts and the online algorithm is only invoked once for each of them. As we will see later, the distinction is mostly formal for our problem (see Theorem 1.11).

Definition 1.3 (Online algorithm, competitive ratio)**.** *Let us have a combinatorial minimization problem $(\mathcal{I}, \mathcal{F}, o, \min)$ such that $I \in \mathcal{I}$ is a function $I(t), t \in \mathbb{R}_0^+$. Online algorithm for such problem receives $I(t)$ and then generates $S(t)$ at every time point t without knowing $I(t')$ for $t' > t$.*

This way, algorithm ALG *generates a solution $S = S(t)$ and if it is feasible, its objective value is $\texttt{ALG}(I) = o(I, S)$. We say that* ALG *is c-competitive (or that it achieves a competitive ratio c) if for any instance I the inequality*

$$\texttt{ALG}(I) \leq c \cdot \texttt{OPT}(I) + \alpha$$

is true, where α does not depend on I. We say it is strictly c-competitive if $\alpha = 0$.

A (strict) competitive ratio of a problem is the minimum c such that there exists a c-competitive algorithm for that problem.

For maximization problem the definition would be analogous.

In this thesis, we will focus on strict competitiveness as is often the case with scheduling. There is an informal observation that supports this. The scheduling instance can be scaled up arbitrarily and one would expect reasonable algorithm to do the same, return a solution that is just scaled up. But then, if $\texttt{ALG}(I) = c \cdot \texttt{OPT}(I) + \epsilon$ for some $\epsilon > 0$, we can scale I up to I' by $s = (\alpha + 1)/\epsilon$ and get $\texttt{ALG}(I') = s \cdot \texttt{ALG}(I) = s \cdot (c \cdot \texttt{OPT}(I)) + s \cdot \epsilon = c \cdot \texttt{OPT}(I') + (\alpha + 1)$. So ALG is not c-competitive. This shows the additive factor does not help, at least with the forementioned assumption about algorithms.

Further on, we will usually omit the word *strict* for strict competitiveness and always use *non-strict* for the other case.

1.2 Scheduling

In this section we will provide all necessary definitions related to PSMJ that we will use throughout the thesis. Notice that the following definitions and notation are focused on PSMJ specifically which means they do not necessarily capture all other variants of scheduling.

Definition 1.4 (Job, Instance)**.** *Job j is determined by three numbers, s_j is its size, p_j its processing time and r_j its release time. We say that j is a copy of $[s, p, r]$, or $j \leftarrow [s, p, r]$, if $s_j = s, p_j = p$ and $r_j = r$.*

The volume of a job j, denoted by $\mathrm{Vol}(j)$ is $s_j \cdot p_j$. The volume of a set of jobs J is $\mathrm{Vol}(J) = \sum_{j \in J} \mathrm{Vol}(j)$.

An instance I for an (online) scheduling problem is a finite set of jobs and the instance part released at time t is a set $I(t) = \{j \in I \mid r_j = t\}$.

By $\mathrm{Longest}(J, k)$ we denote the k-th longest processing time in a set of jobs J or zero if $|J| < k$.

For any set of jobs J, by J^k we denote the set $\{j \mid j \in J, s_j = k\}$ of jobs in J of size k.

We will also say that one job is longer than another job if its processing time is bigger.

Definition 1.5 (Schedule, Makespan)**.** *A schedule for m processor system and instance I is a function $S = S(t)$ that assigns a set of jobs to any time point $t \in \mathbb{R}_0^+$. The occurrence of a job j in the schedule is a set $S^j = \{t \mid j \in S(t)\}$ and we require that S^j is a union of a finite number of intervals $[t, t')$ (we allow only finitely many preemptions). By $|S^j|$ we denote the sum of lengths of these intervals.*

The schedule is feasible if for every $t, \sum_{j \in S(t)} s_j \leq m$ (jobs fit on the processors), $S(t) \subseteq \bigcup_{t' \leq t} I(t')$ (jobs were already released) and $|S^j| \leq p_j$ (job was not running after it was finished).

The schedule is complete if it is feasible and for all jobs $j \in I$ it is true that $|S^j| = p_j$. Otherwise the schedule is incomplete.

The completion time of a job j in a complete schedule is $C_j(S) = \sup\{t \mid j \in S(t)\}$.

The makespan (or a length) of a complete schedule is $C_{max}(S) = \max\{C_j(S) \mid j \in I\}$.

Definition 1.6 (Restricted schedule, Remaining instance)**.** *By restricting the schedule S to a time interval $[t, t')$ we get a schedule $S_{[t,t')}$ such that $S_{[t,t')}(t'') = S(t'' + t)$ if $t'' < t' - t$ and $S_{[t,t')}(t'') = \emptyset$ otherwise. Moreover, we denote by $S_{<t} = S_{[0,t)}$ the schedule up to a time t.*

The remaining processing time of a job j at time t (with respect to some schedule S) is $p_j(S,t) = p_j - |S^j_{<t}|$.

The remaining instance at time t is the set of jobs

$$R(S,t) = \{j(S,t) \leftarrow [s_j, p_j(S,t), r_j] \mid j \in \bigcup_{t' \leq t} I(t'), p_j(S,t) > 0\}.$$

We call the jobs in $R(S,t)$ pending.

A feasible schedule is complete if there is a time t such that the remaining processing time $p_j(S,t)$ is zero for every job j in the instance. The makespan of the schedule is the minimum such t.

In literature on scheduling we often meet a three field notation $\alpha \mid \beta \mid \gamma$ for the various scheduling problems. The problem PSMJ can be written in the three field notation as $P \mid size_j, r_j, pmtn \mid C_{max}$. Here $\alpha = P$ means that the processors have the same speeds. Other possibilities, such as $\alpha = Q, \alpha = R$ indicate problems were the processors have different speeds, possibly in a different way for every job. We will also use $\alpha = Pm$ to indicate that the number of processors m is a constant, rather than given to the algorithm at the beginning. In β we have several components, $size_j$ stands for the fact that the jobs are multiprocessor, i.e., their sizes can be bigger than 1, r_j indicates that jobs have release times and $pmtn$ means that preemption is allowed. In different problems, β can include other indicators such as d_j to denote jobs have deadlines. At last, γ is the optimization part and $\gamma = C_{max}$ means we want to minimize the makespan of the schedule. Other possible objective functions are for example maximum latency or average completion time.

Let us now formally define the problems we approach in this thesis.

Problem 1.7 (PSMJ, $P \mid size_j, r_j, pmtn \mid C_{max}$)**.** *The problem is a minimization problem defined by a quadruple $(\mathcal{I}, \mathcal{F}, o, \min)$ as seen by the Definition 1.1. Here $\mathcal{I}$ is the set of all instances, where instance contains also the number of processors m given to the algorithm at the beginning, $\mathcal{F}$ is the set of complete schedules for m processor system and o is the function that returns C_{max} for such schedule. See Definitions 1.4, 1.5 for the definitions of instance, complete schedule and C_{max}.*

We consider this problem both in the offline and online models. The online model we consider is as defined in the Definition 1.3 and corresponds to the online scheduling model *jobs arriving over time*. Other online scheduling models are not included in this thesis, see for example Sgall [1996] for their definitions.

Often our results apply only to some special cases of PSMJ, i.e., with some restrictions on the allowed instances. We will denote such special cases with PSMJ(*restrictions*). E.g., PSMJ($m = 5, p_j = 1, size_j \in \{1,2\}, r_j = 0$) corresponds to the problem $P5 \mid size_j \in \{1,2\}, p_j = 1, pmtn \mid C_{max}$.

Our main focus in this thesis will be on the following two special cases of PSMJ.

Problem 1.8 (PSMJ(m *const.*, $size_j \in \{1,2\}$))**.** *The same as Problem 1.7 but the number of processors m is a constant and sizes of jobs are restricted to one and two. In the the three field notation this is $Pm \mid size_j \in \{1,2\}, r_j, pmtn \mid C_{max}$.*

Problem 1.9 (PSMJ($m = 4, size_j \in \{1, 2\}$)). *The same as Problem 1.8 for $m = 4$. In the three field notation it is $P4 \mid size_j \in \{1, 2\}, r_j, pmtn \mid C_{max}$.*

Notice that any theorem that can be stated for Problem 1.8, i.e., universally for every m, also applies to the Problem 1.9 with $m = 4$. In these cases we will only state the result for the former, more general, problem.

Our goal is to find online algorithms for these problems with competitive ratios as good as possible. To avoid some technicalities and make the goal of finding an algorithm a bit easier, we will relax our requirements on the algorithm and we will focus on *nearly online* algorithms.

Definition 1.10 (Nearly online algorithm). *For scheduling problems with jobs arriving over time, including problem* PSMJ*, a nearly online algorithm is an online algorithm that is, at any time, also given the next release time of future jobs in the instance.*

In the context of nearly online algorithms, online algorithms from the Definition 1.3 are sometimes called *fully online*. Of course, giving an algorithm more knowledge gives it more power. But to see that the advantage is insignificant, refer to the following theorem mentioned in Sgall [1996].

Theorem 1.11. *For any non-strictly c-competitive nearly online algorithm* ALG *for* PSMJ *there is a non-strictly c-competitive fully online algorithm* ALG' *for* PSMJ*.*

For any strictly c-competitive nearly online algorithm ALG *for* PSMJ *there is a strictly $(c + \epsilon)$-competitive fully online algorithm* ALG' *for* PSMJ*.*

Proof. We will choose some $\delta > 0$ and adjust the instance by adding a job with zero processing time at every integer multiple of δ. Furthermore we will delay the release time of every job in the original instance by δ. We can easily make both of these adjustments to the instance online and at any time point, we know what is the next release time. So we can run the nearly online algorithm in the background. The optimum makespan of the adjusted instance is longer by exactly δ and so our algorithm returns a schedule of length at most $c \cdot (\mathtt{OPT} + \delta) + \alpha = c \cdot \mathtt{OPT} + (c \cdot \delta + \alpha)$. We allow an additive factor independent of the instance so our algorithm is again non-strictly c-competitive. This proves the first part of the statement.

In the second case we can wait until the first job j is released and then set $\delta = (\epsilon/c) \cdot p_j$ and continue by the same algorithm as above. The makespan of the received schedule will be at most

$$c \cdot (\mathtt{OPT} + \delta) = c \cdot \mathtt{OPT} + c \cdot \delta = c \cdot \mathtt{OPT} + \epsilon \cdot p_j \leq c \cdot \mathtt{OPT} + \epsilon \cdot \mathtt{OPT} = (c + \epsilon) \cdot \mathtt{OPT}.$$

and the second part of the statement is also proved. □

Fully online and nearly online algorithms would also become equivalent if we allowed infinite number of preemptions or some other form of time-sharing (more jobs using the same processor at reduced speed).

In the following text, when we speak about algorithms to online problems, we will mean nearly online algorithms, unless specified otherwise. Notice that by allowing nearly online algorithms, we could change the Definition 1.3 of online algorithms to contain only finitely many interactions. Then interaction would

consist of giving the algorithm newly released jobs and the next release time (if there is any) and the algorithm would return a schedule from now until the next release time.

The following definition is an example of a *resource augmentation*. In the case when solving the original problem is too hard, for example when there is no constant competitive algorithm, we may provide our algorithm additional power over the offline optimum that is used for comparison with our algorithm when computing the competitive ratio. This can mean providing our algorithm with additional processors, or, as in our case, providing faster processors.

Definition 1.12 (Speedup model)**.** *Let $v \geq 1$. We call this number a speedup.*

Let I be an instance of PSMJ*. By $I_{\downarrow v}$ we denote a downscaled instance by a factor of v, i.e., the same instance but with processing times of jobs multiplied by $1/v$.*

We say that a schedule is complete for I in the v-speedup model if it is a complete schedule for $I_{\downarrow v}$. This corresponds to the fact that on v-times faster processors all jobs are finished.

An algorithm for PSMJ *in the v-speedup model is c-competitive if it always returns a complete schedule in the v-speedup model with makespan at most c-times the optimal offline makespan.*

We also define the remaining processing time in the v-speedup model of a job j at time t with respect to schedule S as $p_j^{\uparrow v}(S,t) = p_j - v \cdot |S^j_{<t}|$ and the remaining instance as

$$R^{\uparrow v}(S,t) = \{j^{\uparrow v}(S,t) \leftarrow [s_j, p_j^{\uparrow v}(S,t), r_j] \mid j \in \bigcup_{t' \leq t} I(t'), p_j^{\uparrow v}(S,t) > 0\}.$$

We see that a remaining processing time of a job in the v-speedup model reflects the fact that the processors are v-times faster. Complete schedules again correspond to feasible schedules for which there is some t such that the remaining processing times at time t are zero for all jobs in the instance.

Speedup model is an interesting resource augmentation model on its own but for us it will be mostly a tool for finding c-competitive algorithms, as enabled by the following theorem.

Theorem 1.13. *For any algorithm* ALG *for* PSMJ *that is 1-competitive in the v-speedup model, there is a v-competitive algorithm* ALG' *for* PSMJ*.*

Proof. We will describe ALG' directly. We will run ALG in memory at slower rate and return the schedule of this algorithm. More specifically, at time t we will return $S'(t) = S(t/v)$, where S is the schedule produced by ALG and S' is the schedule produced by ALG'. We can do this because at time t we know the input $I(t/v)$, as $v \geq 1$. From 1-competitiveness of ALG we know that the makespan of S is at most OPT and so $S_{<\mathtt{OPT}} = S$. That means that for every job j we have

$$|S'^j_{<(v \cdot \mathtt{OPT})}| = v \cdot |S^j_{<\mathtt{OPT}}| = v \cdot |S^j| = v \cdot \frac{p_j}{v} = p_j.$$

The first equality follows from the fact that $S'^j_{<(v \cdot t)} = \{v \cdot t' \mid t' \in S^j_{<t}\}$ and one before last equality follows from the fact that S is a complete schedule in the v-speedup model.

This means that the makespan of ALG' is at most $v \cdot$ OPT and so ALG' is v-competitive. □

2. Previous work

In this chapter we discuss previous work related to the problem PSMJ. As there are so many variants of scheduling, there is a lot of results in this area and there is not enough space in this thesis to capture the state of the art for all of it. So we will focus only on the results that are strongly connected to PSMJ. Either because they speak about this problem directly or because we use them, or the techniques from their proofs, in our results later in the thesis. With this in mind, we will only include the proofs if they are simple enough and relevant and we will edit the statements and the proofs, compared to what can be found in the original work, so that it matches our needs as much as possible.

Drozdowski [1996] gives a nice overview of multiprocessor scheduling. Later, several books were written to capture the state of the art of scheduling. For example the *Handbook of scheduling* by Leung [2004] and most notably its chapter on *Online scheduling* by Pruhs et al. [2004], a chapter on *On-line scheduling* by Sgall [1996] in the book *Online algorithms, state of the art* and *Scheduling for parallel processes* by Drozdowski [2009].

As we said, we will focus only on problems highly relevant to PSMJ. Specifically, we will include only the results about problems that have the same objective, i.e., minimizing the makespan and where preemption is allowed.

We only focus on preemptive scheduling as the non-preemptive version behaves very differently. For example consider the problem of scheduling sequential jobs (jobs with size 1) without release times, i.e., $P \mid (pmtn) \mid C_{max}$. Whereas the preemptive version can be solved optimally even online (with release times, see Sahni and Cho [1979]), its non-preemptive counterpart is strongly NP-hard (by a reduction from 3-partition).

The following results are split into two sections, offline and online scheduling.

2.1 Offline scheduling

The study of offline algorithms is very important for the online version of the problem, as we will see later on. It should not be too surprising as we compare the online algorithms to offline optimums and so offline algorithms can give us insights for the analysis of our algorithms.

The following result from McNaughton [1959] is one of the first in the area and it has been used in many later results. It will be, along with its proof, of a considerable importance in this thesis as well.

Theorem 2.1 (McNaughton)**.** *Let us have a set of sequential jobs J with release times 0, i.e., an instance of* PSMJ$(size_j = 1, r_j = 0)$*, or equivalently $P \mid p_j, pmtn \mid C_{max}$. Then there exists a schedule of length l if and only if $p_j \leq l$ for every $j \in J$ and $\mathrm{Vol}(J) \leq m \cdot l$.*

Proof. It is clear that both conditions are necessary. For the other direction, let us construct the schedule by adding the jobs to it one by one. By induction, we will prove that at any point the schedule is feasible and some of the processors are entirely filled up, some are entirely empty and at most one processor is partially filled and it is filled in the time interval $[0, t)$ for some t. Let us denote the jobs

$j_1, j_2, \ldots, j_n$. Before adding the first job, the schedule is empty and therefore the induction hypothesis is true. Let us now suppose it is true after adding jobs with indices $1, \ldots, i-1$ and now we want to add a job j_i. Because $\mathrm{Vol}(J) \leq m \cdot l$ we know there is enough space in the schedule for the job, so specifically, at least one processor is not entirely filled up. Let P be the processor that is partially filled, or if it does not exist, one of the empty processors. Let t be the number such that P is filled on the time interval $[0, t)$.

If $p_{j_i} \leq l - t$ then we can schedule job j_i on P on the interval $[t, t + p_{j_i})$.

In the other case, if $p_{j_i} > l - t$ we choose $t' = p_{j_i} - (l - t)$ and we schedule j_i on P on the interval $[t, l)$ and on Q on the interval $[0, t')$. The schedule is feasible because $t' = p_{j_i} - (l - t) \leq l - (l - t) = t$.

In both cases job j_i is now in the schedule, the schedule is still feasible and at most one processor (P in the first case and Q in the second case) is only partially filled on some interval $[0, t')$. This concludes the induction and also the proof. □

Let us highlight the scheduling algorithm from the proof above, as it will be used later in this thesis. It is called *McNaughton's rule* algorithm and it schedules the jobs one by one, putting them on the processors from top to bottom, from left to right and wrapping them around the edges when necessary.

The following result moves us closer to PSMJ, as it solves the offline version of the problem, at least for a constant number of processors. Błażewicz et al. [1986] provide a linear program for the problem $Pm \mid size_j, pmtn \mid C_{max}$. Its complexity is $O(poly(n^m))$ as it contains a variable for any set of jobs that can be scheduled on m processors at the same time. Although originally the LP formulation does not account for release time, it can be easily adjusted to contain them (this is mentioned in Drozdowski [2009]).

Later, Jansen and Porkolab [2000] came up with a more subtle technique, again based on linear programming, that achieves a better performance of $O(n + poly(m))$ for the same problem. For us, the important part is just the following theorem.

Theorem 2.2. *Problem* PSMJ(m *const.*)*, i.e.,* $Pm \mid size_j, r_j, pmtn \mid C_{max}$ *can be solved offline in polynomial time.*

Notice that if m is part of the input, rather than a constant, the problem becomes NP-hard. Actually, the problem remains NP-hard even in the case of sequential jobs and in the absence of release times, as shows the following theorem by Drozdowski [1995].

Theorem 2.3. *Problem* PSMJ($size_j = 1, r_j = 0$)*, i.e.,* $P \mid size_j, pmtn, p_j = 1 \mid C_{max}$ *is* NP*-hard.*

This suggests the problem becomes considerably harder when m becomes part of the input. Therefore, in this thesis we focus only on the case when m is a constant.

There are some results even if we restrict the problem PSMJ further, in the direction towards Problems 1.8 and 1.9, with the condition $size_j \in \{1, 2\}$. Even though the problem is solved from the computational complexity view by the Theorem 2.2 even without this restriction, the following results are of importance

for us. The problem with the linear programming solution in the Theorem 2.2 is, as is often the case with linear programming, opaque and does not shed any light on how to solve the online version of the problem. On the contrary, the following results provide us with combinatorial insight into the problem, which we will use later in this thesis.

Specifically offline version of $\texttt{PSMJ}(size_j \in \{1,2\}, r_j = 0)$, i.e., the problem of preemptive scheduling of multiprocessor jobs without release times with jobs of sizes in $\{1,2\}$, is solved in Błażewicz et al. [1984]. That was later generalized to sizes $\{1,k\}$ in Błażewicz et al. [1986]. These results were further generalized in Błażewicz et al. [1990], resp. Błażewicz et al. [1994], to uniform duo-processor, resp. k-processor, systems, in which processors are in groups of two, resp. k, processors with the same speeds within the group but with different speeds across the groups.

The following results are all from Błażewicz et al. [1986].

Theorem 2.4 (A-schedule)**.** *Let I be an instance of $\texttt{PSMJ}(size_j \in \{1,k\}, r_j = 0)$, i.e., $P \mid size_j \in \{1,k\}, pmtn \mid C_{max}$. An A-schedule is a (preemptive) schedule of some length l that we get when we apply McNaughton's rule first on the I^k jobs on the time interval $[0,l)$ and then on the I^1 jobs (for every job in I^1 we start processing it at the first time t with the least number of used processors and when moving from left to right or wrapping around the edge, we always schedule the current job on the topmost empty processor).*

For any feasible schedule there exists an A-schedule of the same length.

Theorem 2.5. *Let I be an instance of $\texttt{PSMJ}(size_j \in \{1,k\}, r_j = 0)$, or equivalently $P \mid size_j \in \{1,k\}, pmtn \mid C_{max}$. Also, let*

$$\begin{aligned}
Y &= \frac{\mathrm{Vol}\left(I^k\right)}{k}, \\
C &= \max\left\{\frac{\mathrm{Vol}\left(I\right)}{m}, \frac{Y}{\left\lfloor \frac{m}{k} \right\rfloor}, \mathrm{Longest}\left(I,1\right)\right\}, \\
f &= \left\lfloor \frac{Y}{C} \right\rfloor, \\
m_1 &= m - (f+1)\cdot k, \\
C_i &= \frac{(i-m_1)Y + \sum_{i'=1}^{j} \mathrm{Longest}\left(I^1, i'\right)}{(i-m_1)f + i}.
\end{aligned}$$

If $f = \lfloor m/k \rfloor$ then the length of the optimal preemptive schedule is $C^*_{max} = C$. Otherwise it is

$$C^*_{max} = \max\{C, C_{m_1+1}, C_{m_1+2}, \ldots, C_{m_1+k-1}\}.$$

Combining these two theorems we get the following corollary that there is a fast and simple algorithm for the problem $\texttt{PSMJ}(size_j \in \{1,k\}, r_j = 0)$. First compute the value C^*_{max} from Theorem 2.5 and then construct an A-schedule of this length according to Theorem 2.4.

Corollary 2.6. *There is an $O(n)$ algorithm for $\texttt{PSMJ}(size_j \in \{1,k\}, r_j = 0)$, i.e., $P \mid size_j \in \{1,k\}, pmtn \mid C_{max}$.*

2.2 Online scheduling

As in the section on offline scheduling we start with a result on sequential jobs. This already brings us close to the PSMJ problem as omitting release times would bring us back to the offline section. Already in Sahni and Cho [1979] we can find a nearly online 1-competitive algorithm for the problem $Q \mid r_j, pmtn \mid C_{max}$, that is on a uniform multiprocessor system – a system of processors that can have different speeds. Later, for the case when processors are identical, i.e., for $P \mid r_j, pmtn \mid C_{max}$, this was improved to fully online 1-competitive algorithm by Hong and Leung [1992]. Both of these algorithms know the optimal offline makespan but they only use it for testing of infeasibility.

The result of Hong and Leung [1992] is now only a special case of the following theorem by Sgall and Woeginger [2015]. This theorem characterizes, for every m, for what sets of allowed sizes there exists a 1-competitive nearly and fully online algorithm.

Theorem 2.7. *For a number m of machines and a size s, we define the rank of s relative to m as $R(s,m) = \lceil m/s \rceil$. In other words, $R(s,m)$ denotes the maximum number of jobs of size s that can be processed simultaneously on m machines.*

A size s is called fat for m machines if $s > \frac{m}{2}$ (so that s has rank 1), and it is called skinny if $s \leq \frac{m}{2}$ (so that s has rank at least 2). For a set $S \subseteq \{1, 2, \ldots, m\}$, we denote by $S^- \subseteq S$ its skinny elements and by $S^+ \subseteq S$ its fat elements. Jobs are called fat respectively skinny if their size is fat respectively skinny.

Let $m \geq 1$ be the number of machines and let $S \subseteq \{1, 2, \ldots, m\}$ be the set of possible job sizes. There exists a 1-competitive nearly online scheduling algorithm on m identical parallel machines with job sizes in S, if and only if the following two conditions are both fulfilled:

(c1) All $a, b \in S^-$ satisfy $R(a,m) = R(b,m)$; in other words, all skinny sizes in S have the same rank relative to m.

(c2) All $a, b \in S^-$ and all $c \in S^+$ satisfy $R(a, m-c) = R(b, m-c)$; in other words, whenever a fat job blocks some of the machines, then all the skinny sizes in S have the same rank relative to the number of remaining machines.

Furthermore there exists a 1-competitive fully online scheduling algorithm on m identical parallel machines with job sizes in S, if and only if conditions (c1) and (c2) together with the following condition (c3) are fulfilled:

(c3) All $c \in S^+$ and all $a \in S^-$ satisfy $R(a, m-c) = 0$ or $R(a,m) = 2$.

The result gives us the following corollary, that also serves as a justification for us to focus on the problem PSMJ$(m = 4, size_j \in \{1, 2\})$ in this thesis.

Corollary 2.8. *There is a 1-competitive nearly online algorithm for the problem* PSMJ$(m \leq 3)$.

There is no 1-competitive algorithm for PSMJ$(m = 4, size_j \in \{1, 2\})$.

The Theorem 2.7 does not say anything more specific about the competitive ratios when there exists no 1-competitive algorithm. Some competitive ratios are known nevertheless. Specifically, Johannes [2006] shows that List Scheduling

is $(2 - 1/m)$-competitive for PSMJ. The List Scheduling algorithm dates all the way back to Graham [1966] and a competitive ratio of $(2 - 1/m)$ was proven in that paper (with a different terminology, though, as the definitions of online algorithms and competitiveness were not used at that time) but only for the case of sequential jobs and non-preemptive schedules.

The List Scheduling algorithm takes the jobs in an arbitrary order (adding newly released jobs at the end of the list) and whenever there is some event (a job finishes or a new job arrives), first m jobs in the list are scheduled.

Theorem 2.9. *The algorithm s_j-list-scheduling is a List Scheduling algorithm that always sorts the list of available jobs in a non-increasing order by their sizes.*

For an instance of PSMJ*, the length of the schedule constructed by s_j-list-scheduling algorithm is at most $2 - \frac{1}{m}$ times the optimal makespan.*

This can be applied directly to problems PSMJ(m *const.*, $size_j \in \{1, 2\}$) and PSMJ($m = 4, size_j \in \{1, 2\}$).

Corollary 2.10. *The competitive ratio of* PSMJ(m *const.*, $size_j \in \{1, 2\}$) *is at most* $2-1/m$*. The competitive ratio of* PSMJ($m = 4, size_j \in \{1, 2\}$) *is at most* $7/4$*.*

The competitive ratio of the s_j-list-scheduling algorithm is exactly $2-1/m$ as can be seen by the construction provided already by Graham [1966]. Let us have $m(m-1)$ sequential jobs with processing time one and one job with processing time m. The optimal schedule is of length m but as the long job is the last one, the schedule produced by the s_j-list-scheduling algorithm will have a makespan of length $2m - 1$.

Johannes [2006] also provides an example that proves that no non-preemptive List Scheduling algorithm (no matter how the list is sorted) can have a competitive ratio better than $2 - 2/m$.

Last but not least, Johannes [2006] also gives a lower bound using the following example.

Example 2.11. *There are four processors. Consider the instances*

$$\begin{aligned} I^- &= \{A \leftarrow [1, 2, 0], B_1 \leftarrow [2, 1, 0], B_2 \leftarrow [2, 1, 0]\}, \\ I^+ &= I^- \cup \{C \leftarrow [3, 2, 1]\}. \end{aligned}$$

Recall that $[s, p, r]$ stands for a job with size s, processing time p and release time r.

Claim 2.12. *Any online algorithm achieves a competitive ratio no less than* $6/5$ *on at least one of the instances I^-, I^+ from Example 2.11.*

As we will see later in Corollary 4.5, this claim is actually false.

3. Optimal schedules

In this section we present several results regarding offline and online optimums. The purpose is to make the analysis of both lower bounds and algorithms simpler. We also introduce some kind of "normal form" for schedules, called P-schedules. It is a set of rules the schedule needs to comply with, chosen such that any schedule can be transformed into a P-schedule without making it any longer and such that it is as simple as possible.

We start with a section on optimum for jobs without release times, for the case $m = 4$. We interpret previously known results in a more usable way. Then there are two sections, on offline and online schedules, that examine the restrictions we can put on optimal schedules, leading to the already mentioned definition of P-schedules.

3.1 Optimum for jobs without release times

One reason why the problem $P \mid pmtn \mid C_{max}$ is so simple is because the makespan of the optimal schedule can be described as $\max(\text{Longest}(I, 1), \text{Vol}(I)/m)$, an easy and descriptive formula. When we move to jobs with arbitrary sizes there is probably no such nice description of the makespan. Even if we restrict the sizes of jobs to two values 1 and k, the description we get, see Theorem 2.5, is not as nice. But if the sizes are 1 and 2 and $m = 4$, the situation gets better.

Definition 3.1 (Lower bound values)**.** *Let I be an instance for the problem* PSMJ$(m = 4, size_j \in \{1, 2\}, r_j = 0)$*. We call the following values lower bound values*

$$B_l(I) = \text{Longest}\left(I^1, 1\right) \tag{3.1}$$

$$B_L(I) = \text{Longest}\left(I^2, 1\right) \tag{3.2}$$

$$B_V(I) = \frac{\text{Vol}(I)}{4} \tag{3.3}$$

$$B_{V^2+l}(I) = \frac{\text{Vol}(I^2) + 2 \cdot \text{Longest}(I^1, 1)}{4} \tag{3.4}$$

$$B_{V^2+3l}(I) = \frac{\text{Vol}(I^2) + 2 \cdot (\text{Longest}(I^1, 1) + \text{Longest}(I^1, 2) + \text{Longest}(I^1, 3))}{6} \tag{3.5}$$

When the instance is clear from the context, we omit it and write just B_l instead of $B_l(I)$ and the same for the other lower bound values.

We call $B_l, B_L, B_V, B_{V^2+l}, B_{V^2+3l}$ lower bound values because, as the next theorem states, they provide a lower bound on the makespan of a schedule for instance I. We already know this is true for B_l, B_L and B_V, the length of the schedule needs to be at least as long as the longest job and also the volume of the jobs needs to fit in the space on the processors. We will interpret B_{V^2+l} and B_{V^2+3l} as well, it will be just a little bit more complicated.

The value B_{V^2+l} corresponds to the case when we have one job of size 1 much longer than the other jobs of size 1. E.g., if there is just one job of size 1. For

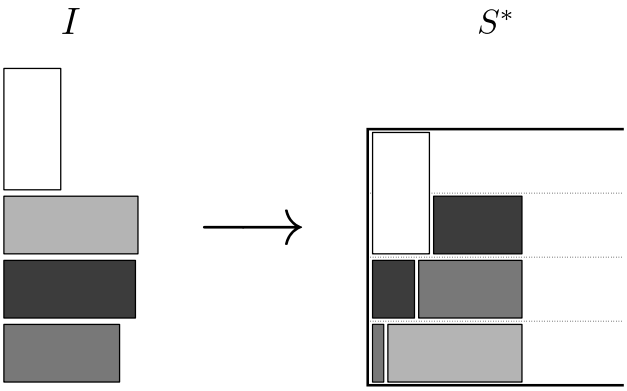

Figure 3.1: On the left we can see an instance I and on the right a schedule S^* with optimal makespan for this instance. Notice that B_l, B_L, B_V and B_{V^2+l} are all smaller than the makespan of this schedule but the schedule cannot be made any shorter. It is an example of an instance where the optimal makespan is defined by the value B_{V^2+3l}.

the sake of computing a lower bound, we now forget the remaining jobs of size 1. Then because m is even and all the remaining jobs have size 2, whenever we run this job, there is at least one empty processor. So it is the same as if we had this job twice in the instance. If we apply bound B_V on this adjusted instance we get exactly B_{V^2+l}. We can compute that the idle time on instance I is at least $B_{V^2+l} - B_V = \text{Longest}(I^1, 1) - \sum_{k=2}^{\infty} \text{Longest}(I^1, k)$ (this is the amount of time we need to run the longest job and no other job of size 1).

For the interpretation of the value B_{V^2+3l} consider the following instance. The jobs I^2 have volume $\text{Vol}(I^2) < 2 \cdot C^*_{max}$ and they are on the first two processors on the interval $[0, t)$ for $t = \text{Vol}(I^2)/2$ (we can assume that they are only on the first two processors because otherwise we can swap parts of the schedule to achieve this) and we have three small jobs j_1, j_2, j_3 such that $p_{j_1} = p_{j_2} = p_{j_3} \geq \frac{2}{3}\text{Vol}(I^2)$. Notice that in this instance $B_l, B_L, B_{V^2+l} < B_V$. So if there is some idle time in this instance it is because of a different reason than the reason behind B_l, B_L, B_{V^2+l}. And there is some idle time in the instance because in the time interval $[t, C^*_{max})$ there are four processors but only three jobs that can run on them. See Figure 3.1 for an example of instance where B_{V^2+3l} defines the optimal makespan.

To understand the exact value of B_{V^2+3l} we denote $X = \text{Vol}(I^2)$ (so $t = X/2$) and $Y = \text{Longest}(I^1, 1) + \text{Longest}(I^1, 2) + \text{Longest}(I^1, 3)$. We see that $Y/3$ is the average of the three longest jobs and that is certainly a lower bound on C^*_{max}. Now we have to take X into account. Notice that when jobs of size 2 are running, only two instead of three processors can be used to run the three jobs of size 1. So to process the whole volume Y the schedule needs to be long at least $t + (Y - 2 \cdot t)/3 = X/6 + Y/3 = B_{V^2+3l}$.

We informally showed that the lower bound values are really lower bounds for the makespan for an instance of $\texttt{PSMJ}(m = 4, size_j \in \{1, 2\}, r_j = 0)$. In fact, the makespan is determined by these values which is stated in the next theorem. This result is a direct consequence of Theorem 2.5 by Błażewicz et al. [1986].

Theorem 3.2 (Offline optimum with lower bound values)**.** *Let I be an instance of* PSMJ$(m = 4, size_j \in \{1, 2\}, r_j = 0)$*. Then*

$$C^*_{max} = \max(B_l, B_L, B_V, B_{V^2+l}, B_{V^2+3l}).$$

Proof. Let $C' = \max(B_l, B_L, B_V, B_{V^2+l}, B_{V^2+3l})$.

We apply Theorem 2.5. In the notation of this theorem we have

$$Y = \frac{\mathrm{Vol}\,(I^2)}{2}, \tag{3.6}$$

$$C = \max(B_l, B_L, B_V), \tag{3.7}$$

$$f = \left\lfloor \frac{Y}{C} \right\rfloor, \tag{3.8}$$

$$m_1 = m - (f + 1) \cdot 2, \tag{3.9}$$

$$C_{m_1+1} = \frac{Y + \sum_{i=1}^{m_1+1} \mathrm{Longest}\,(I^1, i)}{f + m_1 + 1}, \tag{3.10}$$

where the term $Y/(\lfloor m/k \rfloor)$ is missing from the maximum when computing C because the term is at most B_V. Obviously f is either 0, 1 or 2. If $f = 2$ then $\mathrm{Vol}\,(I^1) = 0$ and by Theorem 2.5 $C^*_{max} = C$. On the other hand, in this case $B_{V^2+l} \leq B_V$ and $B_{V^2+3l} \leq B_V$ so $C = C'$ and we have $C^*_{max} = C'$ as we wanted.

If $f < 2$ then $C^*_{max} = \max(C, C_{m_1+1})$. First let $f = 1$. Then $m_1 = 0$ and $C_{m_1+1} = C_1 = (Y + \mathrm{Longest}\,(I^1, 1))/2 = B_{V^2+l}$. So to have $C^*_{max} = C'$ we just need to show $B_{V^2+3l} \leq C^*_{max}$. From $f = 1$ we have $Y \geq C$ or also $\mathrm{Vol}\,(I^2) \geq 2C \geq 2B_V = \mathrm{Vol}\,(I^2)/2 + \mathrm{Vol}\,(I^1)/2$ so $\mathrm{Vol}\,(I^2) \geq \mathrm{Vol}\,(I^1)$. Using this we get

$$B_{V^2+3l} \leq \frac{\mathrm{Vol}\,(I^2) + 2 \cdot \mathrm{Vol}\,(I^1)}{6} \leq \frac{\frac{3}{2}\mathrm{Vol}\,(I^2) + \frac{3}{2}\mathrm{Vol}\,(I^1)}{6} = B_V \leq C \leq C^*_{max}.$$

Now let $f = 0$. Then $m_1 = 2$ and $C_{m_1+1} = C_3 = B_{V^2+3l}$. So we just need to show $B_{V^2+l} \leq C^*_{max}$. But $f = 0$ means $Y < C \leq B_V$ or also $\mathrm{Vol}\,(I^2)/2 < (\mathrm{Vol}\,(I^2) + \mathrm{Vol}\,(I^1))/4$, so $\mathrm{Vol}\,(I^2) < \mathrm{Vol}\,(I^1)$. If $\mathrm{Vol}\,(I^1) < 2 \cdot \mathrm{Longest}\,(I^1, 1)$ then $B_{V^2+l} \leq (\mathrm{Vol}\,(I^1) + 2 \cdot \mathrm{Longest}\,(I^1, 1))/4 < 4 \cdot \mathrm{Longest}\,(I^1, 1)/4 = B_l \leq C^*_{max}$ and otherwise $B_{V^2+l} \leq (\mathrm{Vol}\,(I^2) + \mathrm{Vol}\,(I^1))/4 = B_V \leq C^*_{max}$. □

Now we know how to compute the optimal makespan in a way that provides some structural insight. We are still missing understandable offline algorithm for the problem PSMJ$(m = 4, size_j \in \{1, 2\})$, though, as the previous theorem does not account for release times. We might not be able to construct such an algorithm but in the following section we at least make some assumptions on how the optimal schedule will look. That in turn can help us when we design an online algorithm with similar properties because in the analysis of the algorithm we can compare to this offline optimum and the common properties will hopefully make the analysis of the algorithm simpler.

3.2 Offline schedules

When we are deciding what jobs to run, it makes sense to run the longest jobs, as long jobs can have big completion times and that in turn leads to big makespan. This idea is supported by the following lemma.

Lemma 3.3 (Convexity)**.** *Let $I = J \cup \{j_1 \leftarrow [s, p_1, 0], j_2 \leftarrow [s, p_2, 0]\}$ and $I' = J \cup \{j'_1 \leftarrow [s, p'_1, 0], j'_2 \leftarrow [s, p'_2, 0]\}$ be two instances of* PSMJ*, such that $p_1 > p'_1 \geq p'_2 > p_2$ and $p_1 + p_2 = p'_1 + p'_2$. For every complete schedule S for I there is a complete schedule S' for I' with the same makespan.*

Proof. Let $K = S^{j_1} \setminus S^{j_2}$ be the parts of the schedule where j_1 is scheduled and j_2 not and $L = S^{j_1} \cap S^{j_2}$ the parts of the schedule with both of the jobs scheduled. We know $|K| = |S^{j_1} \setminus S^{j_2}| = |S^{j_1}| - |S^{j_1} \cap S^{j_2}| = p_1 - |L| > p'_1 - |L|$. Note that $p'_1 - |L| \geq p'_1 - p_2 > 0$, so there exists a time t such that $|K \cap [0, t)| = p'_1 - |L|$. We create S' as a copy of S (jobs j'_1 and j'_2 are scheduled at the same places as j_1 and j_2 in S) and only change the processing of job j'_1 to j'_2 for every $t \in K \cap [t, \infty)$. We get $|S'^{j'_1}| = |S'^{j'_1} \setminus S'^{j'_2}| + |S'^{j'_1} \cap S'^{j'_2}| = |K \cap [0, t)| + |L| = p'_1$ and because the overall amount of processing of jobs j'_1 and j'_2 remained the same as the overall amount of processing of jobs j_1 and j_2 and because $p'_1 + p'_2 = p_1 + p_2$, we know also $|S'^{j'_2}| = p'_2$. The schedule is feasible as we only changed the processing of the job j'_1 to j'_2 in the time intervals not yet containing j'_2. □

This can be also interpreted as follows.

Corollary 3.4. *Let I and I' be as in Lemma 3.3. Than $C^*_{max}(I') \leq C^*_{max}(I)$.*

This shows that among the jobs of the same size, it is always better to run a longer job than a shorter one. This can be generalized to more than two jobs, we can use it to say what are the best jobs to run if we know how many processors should be reserved for every size of jobs. Before stating and proving this result formally, we will need some definitions.

Intuitively, we try to imagine what is the best option for the offline optimum so that the remaining instance has the smallest possible makespan. The convexity tells us that we do not want long jobs and small jobs but rather jobs of equal length. So the "best thing to do" is to run the longest job until it is equal to the second longest job, than run both of these until they have the same length as the third longest job etc. Of course if we are creating a part of the schedule of length Δ, we cannot run any job longer than that and we cannot process more volume than $m \cdot \Delta$. With this in mind we can define a *requested instance* for a set of jobs, representing the best possible choice of jobs to run.

Definition 3.5 (Level, requested instance)**.** *Let J be a set of jobs, $m \geq 1$ be an integer and $\Delta > 0$ be a real number.*

We define a level L^J_Δ to be the smallest non-negative real number such that

$$V(L) \coloneqq \sum_{j \in J} \max(0, s_j \cdot \min(\Delta, p_j - L)) = m \cdot \Delta, \tag{3.11}$$

or $L^J_\Delta = 0$ if (3.11) does not have a solution for $L \geq 0$.

For every job $j \in J$ we call $q^j_\Delta = \max(0, \min(\Delta, p_j - L^J_\Delta))$ its requested processing time.

Let $V^J_\Delta = V(L^J_\Delta)$ denote the requested processing volume.

Let $\mathcal{I}^J_\Delta = \{i^j_\Delta \leftarrow [s_j, q^j_\Delta, r_j] \mid j \in J\}$ denote the requested instance.

Theorem 3.6. *We can compute the value L^J_Δ in polynomial time.*

Proof. For every $j \in J$ we try to set L to the two numbers p_j and $p_j - \Delta$. In both cases we check the value $V(L)$ and compare it to $m \cdot \Delta$. The function $V(L)$ is non-increasing so whenever it is bigger than $m \cdot \Delta$, we know L is too small and otherwise it is too big. This in turn indicates if $q^j_\Delta = 0$, $q^j_\Delta = s_j \cdot \Delta$ or $q^j_\Delta = s_j \cdot (p_j - L)$ so (3.11) becomes just a linear equation in L or L gets cancelled. In the second case if we get false equation, it means (3.11) does not have a solution and we return $L^J_\Delta = 0$. Otherwise we know $t \leq L \leq t'$ (where t and t' are some of the values we tried to substitute for L, t' can also be ∞) and we return $L^J_\Delta = t$. □

Notice that the previous algorithm can be implemented in $O(|J| \log |J|)$ time if we sort the set of numbers $\{p_j, p_j - \Delta \mid j \in J\}$ and then binary search the neighbouring values in this set between which L lies.

Theorem 3.7 (Requested instance properties)**.** *Let $J, \Delta, L^J_\Delta, q_\Delta, V^J_\Delta$ and $\mathcal{I}^J_\Delta$ be as in the Definition 3.5. Then*

$$\text{Longest}\left(\mathcal{I}^J_\Delta, 1\right) \leq \Delta \tag{3.12}$$

$$\text{Vol}\left(\mathcal{I}^J_\Delta\right) = V^J_\Delta. \tag{3.13}$$

$$\text{Vol}\left(\mathcal{I}^J_\Delta\right) \leq m \cdot \Delta. \tag{3.14}$$

$$p_j - q^j_\Delta < L^J_\Delta \implies q^j_\Delta = 0, \qquad \forall j \in J \tag{3.15}$$

$$p_j - q^j_\Delta > L^J_\Delta \implies q^j_\Delta = \Delta. \qquad \forall j \in J \tag{3.16}$$

Proof. For (3.12) we notice that

$$q^j_\Delta = \min(\max(0, \Delta), \max(0, p_j - L^J_\Delta)) = \min(\Delta, \max(0, p_j - L^J_\Delta)),$$

so $q^j_\Delta \leq \Delta$ for every $j \in J$. Thus $\text{Longest}\left(\mathcal{I}^J_\Delta, 1\right) = \max\{q^j_\Delta \mid j \in J\} \leq \Delta$.

The equation (3.13) follows straight from the definitions.

Now we will prove (3.14). If $L^J_\Delta > 0$ the inequality holds (with equality) directly from the Definition 3.5. So we can assume $L^J_\Delta = 0$. Suppose for contradiction that (3.14) does not hold, i.e., $V(0) = V(L^J_\Delta) > m \cdot \Delta$. From the Definition 3.5 this means $V(L) = m \cdot \Delta$ does not have any solution for $L \in R^+_0$. Obviously if $L > p_j$ for every $j \in J$ we have $V(L) = 0$. And the function V is continuous so by intermediate value theorem there is L' such that $V(L') = m \cdot \Delta$ and that is a contradiction.

For (3.15) we know $q^j_\Delta = \min(\Delta, \max(0, p_j - L^J_\Delta)) \leq \max(0, p_j - L^J_\Delta)$ and from the assumption $q^j_\Delta > p_j - L^J_\Delta$, so $q^j_\Delta \leq 0$ and so it is zero.

We are left with (3.16). We write $q^j_\Delta = \min(\Delta, \max(0, p_j - L^J_\Delta))$ and from the assumption $q^j_\Delta < p_j - L^J_\Delta$ we know $q^j_\Delta < \max(0, p_j - L^J_\Delta)$, so $q^j_\Delta = \Delta$. □

We will define the notion of *ideal schedule* to be a schedule that can process all the jobs from the requested instance with a makespan of at most Δ. The definition also accounts for a v-speedup model which will be useful later in Chapter 5.

Definition 3.8 (Ideal schedule)**.** *Let J be a set of jobs, $\Delta > 0$ and $v \geq 1$. Furthermore let $\mathcal{I}^J_\Delta$ be as in the Definition 3.5.*

We call a schedule S ideal for J and Δ (in the v-speedup model) if it is a complete schedule for $\mathcal{I}^J_\Delta$ (in the v-speedup model) and its makespan is at most Δ.

Creating an ideal schedule for a set of jobs of different sizes is not always possible. E.g., imagine $m = 2$, there are two jobs $A \leftarrow [1, 2, 0], B \leftarrow [2, 1, 0]$ and $\Delta = 1$. Then $q^A_\Delta = 1$ and $q^B_\Delta = 1/2$, so it is not possible to create an ideal schedule for this instance (without speedup) because we cannot run jobs A and B at the same time. On the other hand if all jobs have the same size and m is a multiple of this size, we can always create an ideal schedule.

Theorem 3.9. *Let J be a set of jobs that all have some size k. Let $m = k \cdot l$. Then for any $\Delta > 0$ we can create an ideal schedule for J and Δ.*

Proof. As all jobs have the same size k and the number of processors m is divisible by k, we can simply use Theorem 2.1, i.e., McNaughton's rule (it is stated for sequential jobs but we can imagine the jobs are sequential and we have only $l = m/k$ processors). The assumptions of the theorem are satisfied because of (3.12) and (3.14) from the Theorem 3.7. □

So in general it is not possible to create an ideal schedule but it is possible for jobs of the same size. If we knew how many processors to assign to each size of jobs, we would be able to create a schedule that consists of ideal schedules for each of theses sizes and that in turn would lead to the best possible remaining instance, in terms of convexity. We will state this formally in Theorem 3.11 but first, we need one more definition.

Definition 3.10 (Size signature)**.** *Let J be a set of jobs. Then τ_J, called the size signature of J, is defined as a function $\tau_J : k \mapsto |\{j \mid s_j = k\}|$, so for an integer k it returns the number of jobs in J with size k.*

Theorem 3.11. *Let I be an instance of* `PSMJ` *and S some complete schedule for I. Also let $0 = t_0 < t_1 < t_2 < \cdots < t_x = C_{max}(S)$ be time points such that every job in I is released at some t_i and for every $i < x$ and $t_i < t' < t_{i+1}$ we have $\tau_{S(t')} = \tau_{S(t_i)}$. Then there exists a schedule $\hat{S}$ for I with makespan $C_{max}(\hat{S}) = C_{max}(S)$ such that $\tau_{\hat{S}(t)} = \tau_{S(t)}$ for every $t \geq 0$ and for every t_i, the schedule $\hat{S}_{[t_i,t_{i+1})}$ can be split into groups of processors, where for $k \geq 1$ the group consists of $k \cdot \tau_{S(t_i)}(k)$ processors and they contain an ideal schedule for $R(\hat{S}, t_i)^k$ and $\Delta = t_{i+1} - t_i$.*

Proof. We will proceed by induction on x. If $x = 0$ the schedule is empty and then empty schedule meets the requirements for $\hat{S}$.

Now $x > 0$. Let $\Delta = t_1 - t_0$, $J = \{j_{[t_0,t_1)} \leftarrow [s_j, |S^j_{[t_0,t_1)}|, r_j] \mid j \in I\}$ and let $\tau^k = \tau_{S(t_0)}(k)$. For every $k \geq 1$ we create an ideal schedule on $m_k = k \cdot \tau^k$ processors for I^k and Δ as part of $\hat{S}_{[t_0,t_1)}$ using Theorem 3.9. First we will prove that for every k these processors are full. For that we only need to verify that $\mathrm{Vol}\left(\mathcal{I}^{I^k}_\Delta\right) = V^{I^k}_\Delta = m_k \cdot \Delta$. This is equivalent to showing that $V(0) \geq m_k \cdot \Delta$ for V from (3.11). Notice that the schedule S has the same signature τ^k everywhere on $[t_0, t_1)$ and so it uses exactly m_k processors for jobs of sizes k at any point in this time interval, i.e., $\mathrm{Vol}\left(J^k\right) = m_k \cdot \Delta$. As no job $j \in J^k$ is longer than Δ we know $V(0) \geq \mathrm{Vol}\left(J^k\right) = m_k \cdot \Delta$. This means $\mathrm{Vol}\left(\mathcal{I}^{I^k}_\Delta\right) = \mathrm{Vol}\left(J^k\right)$ and so $\mathrm{Vol}\left(R(S, t_1)^k\right) = \mathrm{Vol}\left(R(\hat{S}, t_1)^k\right)$.

Now we will proceed by a second induction, on the number of jobs $j \in I$ such that $p_j(\hat{S}, t_1) \neq p_j(S, t_1)$.

If this number of such jobs is zero, the remaining instances $R(S, t_1)$ and $R(\hat{S}, t_1)$ at time t_1 are the same and we can use the induction hypothesis for $x-1$ on $S_{[t_1,\infty)}$ to get $\hat{S}_{[t_1,\infty)}$ (we need to shift the instance to the left, i.e., change t_i to $t_i - t_1$ and also subtract t_1 from all the release times in $R(S, t_1)$).

Now let the number of such jobs be nonzero and let j be the job in I such that $p_j(\hat{S}, t_1) \neq p_j(S, t_1)$ and such that $p_j(S, t_1)$ is the largest among these jobs. Let $k = s_j$. We will prove that $p_j(S, t_1) > p_j(\hat{S}, t_1)$. It makes sense because $\hat{S}_{[t_0,t_1)}$ was constructed such that the longest jobs are shortest possible.

Suppose for contradiction that $p_j(S, t_1) < p_j(\hat{S}, t_1)$. Then there is $j' \in I^k$ such that $p_{j'}(S, t_1) > p_{j'}(\hat{S}, t_1)$ because $\mathrm{Vol}\left(R(\hat{S}, t_1)^k\right) = \mathrm{Vol}\left(R(S, t_1)^k\right)$. That means $p_{j'}(\hat{S}, t_1) < p_{j'}(S, t_1) \leq p_{j'}$, i.e., $q_\Delta^{j'} > 0$. From the property (3.15) from Theorem 3.7 we know $p_{j'}(\hat{S}, t_1) \geq L_\Delta^{I^k}$ and from the maximality of j we have $p_j(\hat{S}, t_1) > p_j(S, t_1) \geq p_{j'}(S, t_1) > p_{j'}(\hat{S}, t_1) \geq L_\Delta^{I^k}$. But from (3.16) $p_j(\hat{S}, t_1) > L_\Delta^{I^k}$ means $q_\Delta^j = \Delta$ and so $p_j(\hat{S}, t_1) = p_j - \Delta \leq p_j - p_{j_{[t_0,t_1)}} = p_j(S, t_1)$ which is a contradiction.

Now we know $p_j(S, t_1) > p_j(\hat{S}, t_1)$. Then there is $j' \in I^k$ such that $p_{j'}(S, t_1) < p_{j'}(\hat{S}, t_1)$ because $\mathrm{Vol}\left(R(\hat{S}, t_1)^k\right) = \mathrm{Vol}\left(R(S, t_1)^k\right)$. From (3.16), if $p_{j'}(\hat{S}, t_1) > L_\Delta^{I^k}$ then $q_\Delta^{j'} = \Delta$ and so $p_{j'}(\hat{S}, t_1) = p_{j'} - \Delta \leq p_{j'} - p_{j'_{[t_0,t_1)}} = p_{j'}(S, t_1)$ and that is a contradiction. So $p_{j'}(\hat{S}, t_1) \leq L_\Delta^{I^k}$. Now $p_j(\hat{S}, t_1) < p_j(S, t_1) \leq p_j$, so $q_\Delta^j > 0$ and from (3.15) $L_\Delta^{I^k} \leq p_j(\hat{S}, t_1)$. So $p_j(\hat{S}, t_1) \geq p_{j'}(\hat{S}, t_1)$. Together we have

$$p_j(S, t_1) > p_j(\hat{S}, t_1) \geq p_{j'}(\hat{S}, t_1) > p_{j'}(S, t_1). \tag{3.17}$$

Let

$$\begin{aligned} \delta &= \min(p_j(S, t_1) - p_j(\hat{S}, t_1), p_{j'}(\hat{S}, t_1) - p_{j'}(S, t_1)) \\ &= \min(q_\Delta^j - p_{j_{[t_0,t_1)}}, p_{j'_{[t_0,t_1)}} - q_\Delta^{j'}) > 0 \end{aligned} \tag{3.18}$$

Now let $f_1 = j_{[t_0,t_1)}$, $f_2 = j'_{[t_0,t_1)}$, $f_1' \leftarrow [s_{f_1}, p_{f_1} + \delta, r_{f_1}]$ and $f_2' \leftarrow [s_{f_2}, p_{f_2} - \delta, r_{f_2}]$. Notice that $p_{f_1'} = p_{f_1} + \delta = p_{j_{[t_0,t_1)}} + \delta \leq q_\Delta^j \leq \Delta$ and $p_{f_2'} = p_{f_2} - \delta \leq p_{j'_{[t_0,t_1)}} \leq \Delta$.

Furthermore, let $J' = J \setminus \{f_1, f_2\} \cup \{f_1', f_2'\}$. We see that J' satisfies the conditions

$$\begin{aligned} \mathrm{Longest}\,(J', 1) &\leq \max(\mathrm{Longest}\,(J, 1), p_{f_1'}, p_{f_2'}) \leq \Delta, \\ \mathrm{Vol}\,(J') &= \mathrm{Vol}\,(J) - p_{f_1} - p_{f_2} + p_{f_1'} + p_{f_2'} = \mathrm{Vol}\,(J) \leq m_k \cdot \Delta, \end{aligned}$$

so we can use Theorem 2.1 to create a complete schedule $S'_{[t_0,t_1)}$ of jobs J'.

We want to use Lemma 3.3 (convexity) to create $S'_{[t_1,\infty)}$ out of $S_{[t_1,\infty)}$. We will prove the assumptions of this lemma are satisfied for the remaining instance at $R(S, t_1)$ and $R(S', t_1)$ and jobs $j_1 = j(S, t_1), j_2 = j'(S, t_1), j_1' = j(S', t_1)$ and $j_2' = j(S', t_2)$. Let these jobs have processing times p_1, p_2, p_1', p_2' respectively. First notice that $R(S', t_1) = R(S, t_1) \setminus \{j_1, j_2\} \cup \{j_1', j_2'\}$ so we only need to prove $p_1 > p_1' \geq p_2' > p_2$ and $p_1' + p_2' = p_1 + p_2$ (the lemma is stated only for instances with release times zero but we as all the jobs j_1, j_2, j_1', j_2' were released before time t_1, we can assume that). We see that $p_1 = p_{j_1} = p_j(S, t_1), p_2 = p_{j_2} = p_{j'}(S, t_1)$

and $p'_1 = p_j(S', t_1) = p_j(S, t_1) - \delta, p'_2 = p_{j'}(S', t_1) = p_{j'}(S, t_1) + \delta$. So obviously $p'_1 + p'_2 = p_1 + p_2$.

By combining (3.17) and (3.18) we get

$$p_j(S, t_1) > p_j(S, t_1) - \delta \geq p_j(\hat{S}, t_1) \geq p_{j'}(\hat{S}, t_1) \geq p_{j'}(S, t_1) + \delta > p_{j'}(S, t_1),$$

i.e., $p_1 > p'_1 \geq p_j(\hat{S}, t_1) \geq p_{j'}(\hat{S}, t_1) \geq p'_2 > p_2$.

We see that the assumptions of Lemma 3.3 are satisfied and so we get $S'_{[t_1,\infty)}$ of the same makespan as $S_{[t_1,\infty)}$. Now because of the choice of δ we will have either $p_j(S', t_1) = p_j(\hat{S}, t_1)$ or $p_{j'}(S', t_1) = p_{j'}(\hat{S}, t_1)$. In both cases the number of jobs with different processing times at time t_1 is smaller by one and we can use induction hypothesis on S' to get $\hat{S}$. □

The previous theorem shows that if somebody gave us the size signature of an optimal schedule, we could construct a (possibly different) schedule with the optimal makespan. So the search for an optimal algorithm narrows down to deciding a correct size signature. For the problem $\texttt{PSMJ}(m = 4, size_j \in \{1, 2\})$ we can narrow it down even more.

Theorem 3.12. *Let I be an instance for the problem $\texttt{PSMJ}(m = 4, size_j \in \{1, 2\})$ and let S be a complete schedule for I. There exists a schedule $\hat{S}$ with a makespan of at most $C_{max}(S)$ that for every t satisfies*

$$\mathrm{Vol}\left(R(\hat{S}, t)^2\right) > 0 \implies \hat{S}(t) \cap I^2 \neq \emptyset.$$

Proof. We split the schedule S into maximal intervals $[t, t')$ such that $S(t'') = S(t)$ for every $t'' \in [t, t')$ and release time of every job is not in the middle of any interval. Let $B(S) \subseteq I^2$ be the set of jobs of sizes two such that for $j \in B(S)$ there is a nonempty set of intervals $W(j)$ on which j is scheduled and such that for each $w \in W(j)$ there is a nonempty set of intervals $V(w)$ such that every $[t, t') \in V(w)$ is between r_j and w and only small jobs are in $S(t)$. Let I be fixed and let us have any ordering on I^2. We will proceed by induction on $(|B(S)|, |W(j)|, |V(w)|)$ where we compare the triplets lexicographically, j is the first job in $B(S)$ (in the chosen ordering) and w is the first interval in $W(j)$.

If $|B(S)| = 0$ then S satisfies the conditions for $\hat{S}$.

Otherwise let $j \in B(S)$ be the first job in b, $w = [t_2, t_3)$ the first interval in $W(j)$ and $v = [t_0, t_1)$ the first interval in $V(w)$. Let $\delta = \min(t_3 - t_2, t_1 - t_0)$ and $w' = [t_3 - \delta, t_3)$ and $v' = [t_0, t_0 + \delta)$. We denote $t'_2 = t_3 - \delta$. Notice that by the choice of δ either $w' = w$ or $v' = v$. Let $S(t_0) = \{a_1, a_2, a_3, a_4\}$ be the four jobs of size one (if there are less than four jobs running, we will create imaginary jobs representing the idle time). As $j \in S(t'_2)$ and $j \notin S(t_0)$ we know $|S(t_0) \cap S(t'_2)| \leq 2$. So at least two of the jobs a_1, a_2, a_3, a_4 are not in $S(t'_2)$ and we can swap their occurrence in v' with j in w' to get a schedule S'.

If $w = w'$ this means that $|W(j)|$ is smaller by one (or if it is zero, $|B(S')|$ is smaller by one). Otherwise $w \neq w'$ and $v = v'$ means that $|V(j)|$ is smaller by one (or if it is zero, $|W(j)|$ or potentially $|B(S)|$ is smaller by one). In all the cases we can use induction hypothesis on S' to get the required schedule $\hat{S}$. □

This means that whenever there is at least one job of size 2 available, the schedule will contain at least one such job. So to provide an optimal algorithm

for the offline version of $\texttt{PSMJ}(m = 4, size_j \in \{1, 2\})$ we only need to decide, at any time point, whether to run two jobs of size 2 or one job of size two with two jobs of size one.

Notice that by applying Theorem 3.11 to some schedule S we obtain a schedule with the same size signature at ever time point. This means we can combine the Theorems 3.11 and 3.12 to construct even more specific schedule.

Definition 3.13 (P-schedule)**.** *Let I be an instance of the problem* $\texttt{PSMJ}(m = 4, size_j \in \{1, 2\})$*. A schedule S is called a P-schedule if it is feasible and if for every two consecutive release times t and t' (and for the pair $(t, t' = C_{max}(S))$ where t is the last release time) there are numbers $t \leq t_1 \leq t_2 \leq t'$ satisfying*

(i) If $t_1 > t$, the schedule in $[t, t_1)$ contains at any time point two jobs of size two and $S_{[t,t_1)}$ is an ideal schedule for $R(S, t)$.

(ii) If $t_2 > t_1$, the schedule in $[t_1, t_2)$ contains at any time point one job of size two and $S_{[t_1,t_2)}$ is an ideal schedule for $R(S, t_1)$.

(iii) If $t' > t_2$, there are no jobs of size two at time t_2, i.e., $R(S, t_2)^2 = \emptyset$ and the schedule $S_{[t_2,t')}$ is an ideal schedule for $R(S, t_2)$.

We call the schedule a P-schedule because if we imagine the jobs of size two in the schedule between any two consecutive release times, they have the shape of the letter P.

Theorem 3.14. *Let I be an instance of* $\texttt{PSMJ}(m = 4, size_j \in \{1, 2\})$*. There is a P-schedule S^* with optimal makespan.*

Proof. Let S be any schedule for I with the optimal makespan. First we apply Theorem 3.12 to obtain a schedule S' from S. Let t and t' be any two consecutive release times (or last release time and the makespan of S'). From the properties of S' there is a time t_2 such that at least one big jobs runs at all times in the time interval $[t, t_2)$ and if $t_2 < t'$ we have $R(S', t_2)^2 = \emptyset$. Now we can easily swap subsets of $[t, t_2)$ to achieve that in the first part there are always two jobs of size two running and in the second part only one job of size two is running. By doing this for every such interval $[t, t')$ we obtain a schedule S''. This schedule satisfies all conditions apart from the conditions that the individual intervals $[t, t_1), [t_1, t_2)$ and $[t_2, t')$ are ideal schedules for the remaining instance at the corresponding start of the interval.

Now we can use Theorem 3.11 on S'' with all the time points t, t_1, t_2, t' to obtain S^*. This does not change the size signature at any time point and the requested parts of the schedule are ideal schedules for the corresponding remaining instances. □

We did not find an optimal offline algorithm for $\texttt{PSMJ}(m = 4, size_j \in \{1, 2\})$ (better then the linear programming formulation from Theorem 2.2) but we have proved that we can make many assumptions on the shape of the optimal schedule. That can be useful when analyzing a performance of an online algorithm.

3.3 Online schedules

In this section we show that the result from Theorem 3.14 describing the offline optimal schedule can also be obtained in the online setting.

Theorem 3.15. *For any c-competitive (nearly) online algorithm for the problem* PSMJ$(m = 4, size_j \in \{1, 2\})$ *there is a c-competitive nearly online algorithm that always produces a P-schedule.*

Proof. Let I be the instance on the input and S the schedule produced by the algorithm we are given. As we are constructing a nearly online algorithm, we can, at any point, compute the schedule S up to the next release time. We will construct a P-schedule $\hat{S}$.

We will change the schedule S into $S_\sim$ in the same way as described in the proof of Theorem 3.12. Of course we do not know how the original schedule looks in the future so we need to make some adjustments. We remember some order of jobs of size two and whenever new jobs are released, we add them at the end in some arbitrary order. If $S(t)$ contains at least one job of size two, nothing changes and we return $S_\sim(t) = S(t)$. Now suppose it contains only jobs of size one. If there are no pending jobs of size two, we return $S_\sim(t) = S(t)$. Otherwise we already know what is the job j of size two that we will swap to this t, it is the smallest job in our ordering that was already released and that is not yet completed. So we can replace it for two jobs of size one from $S(t)$. We do not know which two so we make a guess, later we will show it is not important. We will remember this swap. Now at the first time[1] t' such that $j \in S(t')$ we need to finish the swap. So we will remove j from the schedule and include two jobs from $S(t)$. We will ignore the guess we made and swap j with the two jobs such that $S_\sim$ is feasible – we need to change also $S_\sim(t)$ to achieve feasibility of $S_\sim$. This change of history is only for our future decisions (we want $S_\sim$ to remain feasible), of course we cannot change what the algorithm already returned.

By the above procedure we generate some schedule S' online which is equal to $S_\sim$ with the exception that some jobs of sizes one are different in the schedules. The schedule S' may be even unfeasible. But it is important that the size signature at any time point is the same in both schedules.

Let t be the current release time and t' the next release time. We will change S' to S'' by rearranging subsets of $[t, t')$ so that in $[t, t_1)$ there are always two jobs of size two scheduled, in $[t_1, t_2)$ one job of size two and in $[t_2, t')$ no jobs of size two. Actually, thanks to the way we generated S', we only need to rearrange $[t, t_2)$. We can do this online because at time t we can access $S'(t'')$ for any $t'' \in [t, t)$ as no new jobs are released in this interval.

Finally, we change S'' to $\hat{S}$ using Theorem 3.11, always for t, t_1, t_2, t'. This procedure is again almost online, only using $S''(t'')$ for $t'' < t'$ at time t, which is fine.

Now we need to prove that $\hat{S}$ is a P-schedule with at most the same makespan as S. For this we define $S''_\sim$ from $S_\sim$ in the same way as we defined S'' from S' and

[1] Technically, in Theorem 3.12 we are not swapping with the first time that contains job j but with the last time so that we can still perform all swaps. But this is still possible to do online. At t' we remember the remaining processing time of job j that also counts with the swaps. We start swapping j back when this processing time is zero.

$\hat{S}_\sim$ form $S''_\sim$ in the same way as $\hat{S}$ was defined from S''. We already mentioned that S' and $S'_\sim$ have the same size signature at any time point. But when we generate S'' and $S''_\sim$ we only decide based on the size signatures, so the same is true for these two schedules as well. Finally, when generating $\hat{S}$ and $\hat{S}_\sim$, we only use the size signatures from the input schedules. So $\hat{S} = \hat{S}_\sim$. But we know $\hat{S}_\sim$ is a P-schedule with makespan at most $C_{max}(S)$ because it was generated from S in the same way as $\hat{S}$ in Theorem 3.14.

□

The power of this theorem is not in the fact that we can transform algorithms so that they produce some specific schedule, it is more of an existential significance. We know that whatever is the competitive ratio of the problem $\texttt{PSMJ}(m = 4, size_j \in \{1, 2\})$, there is an algorithm having this competitive ratio and following the above rules. This makes our task for finding an optimal algorithm easier, as it is enough to only consider algorithms of some specific type. Moreover, the prescribed rules for the algorithm are simple, thus guiding us toward a simpler, rather than more complex, optimal algorithm. In Chapter 5.4 we introduce an algorithm following these rules that we conjecture might have the best possible competitive ratio.

4. Lower bounds

In this chapter we discuss the lower bounds for problem PSMJ, specifically for its variants PSMJ(m *const.*, $size_j \in \{1, 2\}$) and PSMJ($m = 4, size_j \in \{1, 2\}$). Lower bounds are an important part of finding a competitive ratio of a problem. Not only it is one of the two necessary parts, as we know that no 1-competitive algorithm exists, but they also help us design the algorithms. If one has a weak lower bound he might be trying to come up with an algorithm that cannot exist. Also if we believe that a lower bound is tight and we want to design an algorithm matching this bound, it puts a restriction on the algorithm – we know how it needs to behave on the example that gave us the lower bound.

We start with lower bounds on competitiveness and then we move onto lower bounds for the speedup model, always starting with the case $m = 4$ and then generalizing the example for larger m. Theorem 1.13 states that for a 1-competitive algorithm in a v-speedup model there is v-competitive algorithm. But the reverse does not need to hold and it would be quite surprising if it did. Especially if we view the speedup model through the following equivalent definition.

Definition 4.1 (Delay model)**.** *An algorithm in a d-delay model is an algorithm for* PSMJ *that at time $t \cdot d$ receives the instance part $I(t)$ (i.e., the jobs are delayed by a factor of d). It is c-competitive in this model if it creates a complete schedule with a makespan of length at most $c \cdot d \cdot C^*_{max}$, where C^*_{max} is the optimal offline makespan.*

Theorem 4.2 (Equivalence of speedup and delay models)**.** *There exists an algorithm for* PSMJ *that is c-competitive in the d-delay model if and only if there exists an algorithm that is c-competitive in the d-speedup model.*

Proof. We can change the algorithm for the d-delay model to the d-speedup model simply by internally dividing the current time, all release times and processing times of jobs by d.

The other direction is analogous. □

With this in mind, it would be surprising if a c-competitive algorithm could be easily adjustable into a 1-competitive algorithm in the c-delay model, as the only difference in these models is that c-delay model receives jobs with a delay of factor c. So the same example that is used to show a lower bound for competitiveness could provide us with a better bound for the speedup model.

In both of the following sections, lower bounds on the competitive ratio and lower bounds for the speedup model, we will always start with the case $m = 4$ and then generalize the result for $m > 4$.

4.1 Lower bounds on the competitive ratio

4.1.1 Lower bound $9/8$ for $m = 4$

We start with a lower bound for PSMJ($m = 4, size_j \in \{1, 2\}$). Our starting point will be Example 2.11 by Johannes [2006]. We cannot use it directly as in the instance I^+ there is a job of size 3. We will use the following adjusted example.

Example 4.3. *There are four processors. Consider the instances*

$$\begin{aligned} I_4^- &= \{A \leftarrow [1,2,0], B_1 \leftarrow [2,1,0], B_2 \leftarrow [2,1,0]\}, \\ I_4^+ &= I_4^- \cup \{C \leftarrow [2,2,1], D \leftarrow [1,2,1]\}. \end{aligned}$$

Theorem 4.4. *Any online algorithm for* $\texttt{PSMJ}(m = 4, size_j \in \{1,2\})$ *achieves a competitive ratio no less than* $9/8$ *on at least one of the instances* I_4^-, I_4^+ *from Example 4.3.*

Proof. In the time interval $[0,1)$ the algorithm only sees the instance $I_4^- = I_4^+(0)$ so it will create the same schedule $S_{<1}$ in this time interval for both instances. Now we distinguish two cases.

Case 1, $|S^A_{<1}| \leq x = \frac{3}{4}$: Let S be the schedule for instance I_4^-. We know $C_{max}(S) \geq C_A(S) \geq 1 + p_A(S,1) \geq 1 + 2 - x = 3 - x$. On the other hand $C^*_{max} = 2$ as can be seen in Figure 4.1. So in this case, algorithm achieves a competitive ratio of at least $C_{max}(S)/C^*_{max} \geq (3-x)/2 = 9/8$.

Case 2, $|S^A_{<1}| > x = \frac{3}{4}$: Let S be the schedule for instance I_4^+. Let $R_1 = R(S,1)$ be the set of remaining jobs at time 1. We know that along the job A at most one job of size 2 can be running in $S_{<1}$ and so $\mathrm{Vol}\,(R_1^2) \geq \mathrm{Vol}\left((I_4^+)^2\right) - 2x - 4(1-x) = 4 + 2x$. From Theorem 3.2 we know that

$$\begin{aligned} C_{max}(S) &\geq 1 + B_{V^2+l}(R_1) = 1 + \frac{\mathrm{Vol}\,(R_1^2) + 2 \cdot \mathrm{Longest}\,(R_1^1, 1)}{4} \\ &\geq 1 + \frac{(4+2x) + 2 \cdot 2}{4} = 3 + \frac{x}{2}. \end{aligned}$$

From Figure 4.1 we see that $C^*_{max} = 3$ and so the algorithm achieves a competitive ratio of at least $C_{max}(S)/C^*_{max} \geq (3 + x/2)/3 = 9/8$. □

From the proof and also from the Figure 4.1 we can see that Example 4.3 cannot provide us with a better bound. On the Figure 4.1 we can also see that if we merged the jobs C and D into a job of size 3, obtaining Example 2.11 by Johannes [2006], we could change the schedules easily without making them any longer. That leads us to the following corollary.

Corollary 4.5. *The Claim 2.12 by Johannes [2006] does not hold.*

Proof. We will provide one such contradicting algorithm `ALG`. Notice that it is enough to define its behaviour so that it constructs a feasible schedule for the instance I^- and I^+. Suppose it is given one of these instances.

It will schedule the jobs according to the bottom part of the Figure 4.1, with C and D merged into one job. It is important that the schedule for I^- and I^+ is the same in the time interval $[0,1)$ so these schedules can really be produced by the same algorithm. We see, that for $x = 3/4$ the schedule lengths for both instances are only $9/8$ apart from the optimal makespans. □

The mistake that was in the proof of Johannes [2006] was in the *Case 2* of the proof of Theorem 4.4 above (with the slight difference that in their proof there was $x = 3/5$). They stated that $C_{max}(S) \geq 3 + x$ instead of the correct bound $C_{max}(S) \geq 3 + x/2$.

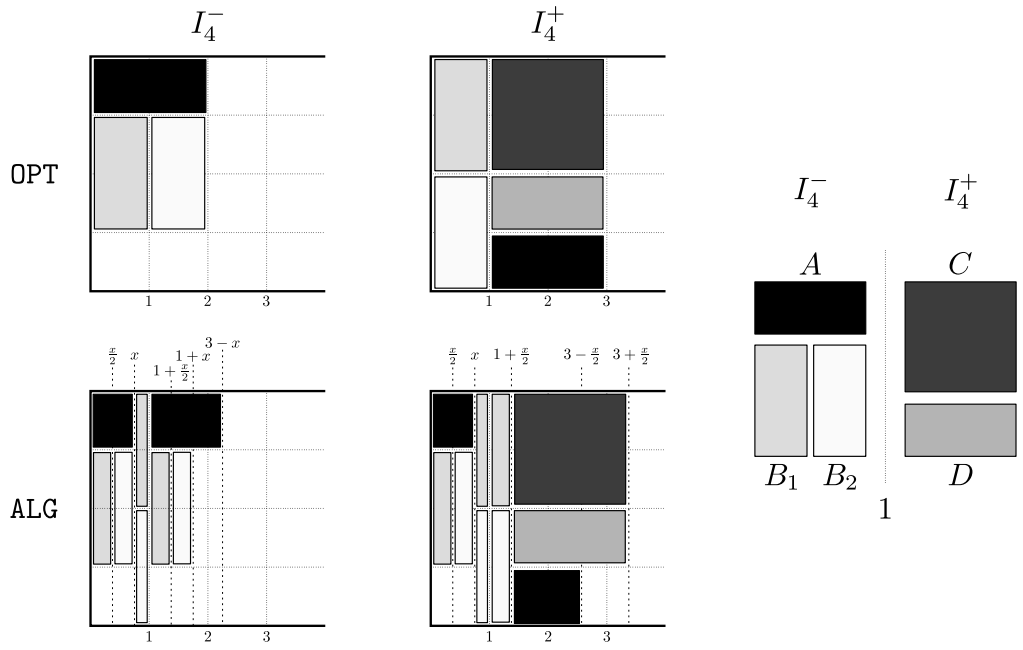

Figure 4.1: In the top part of the figure there are optimal schedules for scenarios I_4^-, I_4^+. In the bottom part we can see the schedules that can be produced by some algorithm ALG and that have a makespan only 9/8-times bigger than the optimum. The value of x is 3/4.

4.1.2 Lower bound $1 + 2/(3m + 4)$ for m even

We will generalize Example 4.3.

Example 4.6. *There are $m = 2n$ processors. Consider the instances*

$$\begin{aligned} I_{2n}^- &= \{A \leftarrow [1, 2, 0], B_i \leftarrow [2, 1, 0] \mid i = 1, \ldots, n\}, \\ I_{2n}^+ &= I_{2n}^- \cup \{C_i \leftarrow [2, 2, 1], D \leftarrow [1, 2, 1] \mid i = 1, \ldots, n-1\}. \end{aligned}$$

Theorem 4.7. *Any online algorithm for* PSMJ*(m const., $size_j \in \{1, 2\}$) with $m = 2n \geq 4$ achieves a competitive ratio no less than $1 + 2/(3m + 4)$ on at least one of the instances I_{2n}^-, I_{2n}^+ from Example 4.6.*

Proof. In the time interval $[0, 1)$ the algorithm only sees the instance $I_{2n}^- = I_{2n}^+(0)$ so it will create the same schedule $S_{<1}$ in this time interval for both instances. Now we distinguish two cases.

Case 1, $|S_{<1}^A| \leq x = 3m/(3m + 4)$: Let S be the schedule for instance I_{2n}^-. We know $C_{max}(S) \geq C_A(S) \geq 1 + p_A(S, 1) \geq 1 + 2 - x = 3 - x$. On the other hand $C_{max}^* = 2$, so in this case algorithm achieves a competitive ratio of at least $C_{max}(S)/C_{max}^* \geq (3 - x)/2 = 1 + 2/(3m + 4)$.

Case 2, $|S_{<1}^A| > x = 3m/(3m + 4)$: Let S be the schedule for instance I_{2n}^+. At time 1 we have exactly two jobs of size 1, one of length $p_A(S, 1) \leq 2 - x$ and one of length $p_D(S, 1) = 2$. As the total number of processors is even, when job D is running and job A is not, at least one processor is idle, so there will be at least x units of idle time after time 1. There is also at least x units of idle time in the time interval $[0, 1)$ when job A is running. That gives us the following lower

bound on the makespan of the schedule

$$C_{max}(S) \geq \frac{\text{Vol}\left(I^+_{2n}\right) + 2x}{m} = \frac{3m + 2x}{m} = 3 + 2\frac{x}{m}.$$

So the algorithm achieves a competitive ratio of at least

$$\frac{C_{max}(S)}{C^*_{max}} \geq \frac{3 + 2\frac{x}{m}}{3} = 1 + \frac{\frac{6}{3m+4}}{3} = 1 + \frac{2}{3m+4}.$$

□

4.1.3 Lower bound $1 + 2/(5m + 8)$ for m odd

For m odd the generalization of Example 4.3 is not that straightforward. The problem is that in the proofs above we used the fact that when m is even and we have only one job of size one then whenver it is running there is at least one empty processor. For m odd this does not hold.

On the other hand, for m odd we know that at least one job of size one will always be running if there is such a pending job. If we added a job of size equal to the optimal makespan it would fill the extra processor and even number of processors would be remaining for the rest of the schedule. At first glance this works and we get the same lower bound as for $m - 1$ but unfortunately it is not correct. By this we would give the algorithm some extra information (the length of the optimal makespan) so the algorithm could, for example, distinguish between I^- and I^+ at time $t = 0$.

Instead, we will have an algorithm choose to run either two long jobs of size one and have one processor empty or to run only one long job of size one and have no idle time.

Example 4.8. *There are $m = 2n + 1$ processors. Consider the instances*

$$I^-_{2n+1} = \{A_1 \leftarrow [1, 2, 0], A_2 \leftarrow [1, 2, 0], B_i \leftarrow [2, 1, 0] \mid i = 1, \ldots, n\},$$
$$I^+_{2n+1} = I^-_{2n+1} \cup \{C_i \leftarrow [2, 3/2, 1], D \leftarrow [1, 3/2, 1] \mid i = 1, \ldots, n - 1\}.$$

Theorem 4.9. *Any online algorithm for* $\texttt{PSMJ}(m \text{ const.}, size_j \in \{1, 2\})$ *with* $m = 2n + 1 \geq 5$ *achieves a competitive ratio no less than* $1 + 2/(5m + 8)$ *on at least one of the instances* I^-_{2n+1}, I^+_{2n+1} *from Example 4.8.*

Proof. First we will compute the optimal schedules for the two instances.

We know $C^*_{max}(I^-_{2n+1}) \geq \text{Longest}\left(I^-_{2n+1}, 1\right) = 2$. Now consider the following schedule. In the time interval $[0, 1)$ schedule jobs A_1, A_2 and $B_1, B_2, \ldots, B_{n-1}$ and in the time interval $[1, 2)$ schedule A_1, A_2, B_n. All jobs are done at $t = 2$ and $C^*_{max}(I^-_{2n+1}) = 2$.

We know $C^*_{max}(I^+_{2n+1}) \geq \text{Vol}\left(I^+_{2n+1}\right)/m = 5/2$. Now consider the following schedule. In the time interval $[0, 1/2)$ schedule jobs A_1 and $B_1, B_2, \ldots, B_n$, in the time interval $[1/2, 1)$ jobs $A_2, B_1, B_2, \ldots, B_n$ and in the time interval $[1, 5/2)$ schedule $A_1, A_2, D, C_1, C_2, \ldots, C_{n-1}$. All jobs are done at $t = 5/2$ and $C^*_{max}(I^+_{2n+1}) = 5/2$.

Now let S be the schedule created on one of the instances I^-_{2n+1} or I^+_{2n+1}. In both cases $S_{<1}$ will be the same and let $p = |S^{A_1}_{<1} \cap S^{A_2}_{<1}|$ be the amount of processing time with jobs A_1 and A_2 running at the same time.

Case 1, $p \leq x = 5m/(5m+8)$: Let S be the schedule for instance I^-_{2n+1}. We know

$$\begin{aligned} C_{max}(S) &\geq \max(C_{A_1}(S), C_{A_2}(S)) \geq \max(1 + p_{A_1}(S,1), 1 + p_{A_2}(S,1)) \\ &\geq 1 + \frac{p_{A_1}(S,1) + p_{A_2}(S,1)}{2} \\ &= 1 + \frac{p_{A_1} + p_{A_2} - 2|S^{A_1}_{<1} \cap S^{A_2}_{<1}| - |S^{A_1}_{<1} \setminus S^{A_2}_{<1}| - |S^{A_2}_{<1} \setminus S^{A_1}_{<1}|}{2} \\ &\geq 1 + \frac{4 - 2p - (1-p)}{2} = 1 + \frac{3-p}{2} \geq \frac{5-x}{2}. \end{aligned}$$

We know $C^*_{max} = 2$, so in this case algorithm achieves a competitive ratio of at least $C_{max}(S)/C^*_{max} \geq (5-x)/4 = 1 + (1-x)/4 = 1 + 2/(5m+8)$.

Case 2, $p > x = 5m/(5m+8)$: Let S be the schedule for instance I^+_{2n+1}. We know that there is at least p units of idle time in the schedule because when both A_1 and A_2 are running, there are $m - 2 = 2n - 1$ processors left for jobs of size 2. So $C_{max}(S) \geq (\mathrm{Vol}\left(I^+_{2n+1}\right) + p)/m = 5/2 + p/m \geq 5/2 + x/m$.

In this case algorithm achieves a competitive ratio of at least $C_{max}(S)/C^*_{max} \geq (5/2 + x/m)/(5/2) = 1 + 2x/(5m) = 1 + 2/(5m+8)$.

□

The lower bound we achieved for m odd is not as good as for m even. The difference compared to m even is that in the *Case 2* we can create a schedule such that $S_{[1,C_{max}(S))}$ has no idle time.

4.2 Lower bounds for the speedup model

In this section we use the same examples as in the section above but the bounds we obtain are stronger. The fact that we get stronger lower bounds for this model is not the only reason to compute them. As we will see later, in Chapter 5, we use the speedup model to design competitive algorithms. And so it is useful to know the limits of this model.

4.2.1 Lower bound $8/7$ for $m = 4$

Theorem 4.10. *If there exists an algorithm for* $\texttt{PSMJ}(m = 4, size_j \in \{1, 2\})$ *that is 1-competitive in the v-speedup model, then* $v \geq 8/7$.

Proof. We will use the instances I^-_4, I^+_4 from Example 4.3.

Let S be the schedule constructed by this algorithm on one of these instances, we will specify later on which one. But in both cases the partial schedule $S_{<1}$ is going to be the same. Again, we distinguish between two cases.

Case 1, $|S^A_{<1}| \leq x = 3/4$: Let S be the schedule for instance I^-_4. Because the algorithm in question is 1-competitive, we know $2 = C^*_{max} \geq C_{max}(S) \geq C_A(S) \geq 1 + p^{\uparrow v}_A(S,1)/v \geq 1 + (2 - vx)/v = 1 + 2/v - x$ and so $v \geq 2/(1+x) = 8/7$.

Case 2, $|S^A_{<1}| > x = 3/4$: Let S be the schedule for instance I^+_4. As in the proof of Theorem 4.7, we know that there will be at least $2x$ units of idle time in the schedule. Including this idle time, processors need to run, altogether, for $\mathrm{Vol}\left(I^+_4\right)/v + 2x = 12/v + 2x$ units of time and so

$$3 = C^*_{max} \geq C_{max}(S) \geq \frac{\frac{12}{v} + 2x}{4} = \frac{3}{v} + \frac{x}{2},$$

which leads to $v \geq 3/(3 - x/2) = 8/7$. □

4.2.2 Lower bound $1 + 2/(3m + 2)$ for m even

Again, we will use the same example as for the lower bound on competitive factor for m even, that is Example 4.6.

Theorem 4.11. *Let $m = 2n \geq 4$ and $v \geq 1$. If there exists an algorithm for* PSMJ*(m const., $size_j \in \{1, 2\}$) that is 1-competitive in the v-speedup model, then $v \geq 1 + 2/(3m + 2)$.*

Proof. Let I^-_{2n}, I^+_{2n} be the instances from Example 4.6.

The proof to this theorem is analogous to the proof of the Theorem 4.10. In here we choose $x = 3m/(3m + 4)$.

In *Case 1* we know $2 = C^*_{max} \geq C_{max}(S) \geq C_A(S) \geq 1 + 2/v - x$. That gives us $v \geq 2/(1 + x) = 1 + 2/(3m + 2)$.

In *Case 2* we know

$$3 = C^*_{max} \geq C_{max}(S) \geq \frac{\frac{\mathrm{Vol}\left(I^+_{2n}\right)}{v} + 2x}{m} = \frac{3}{v} + \frac{2x}{m}.$$

That gives us

$$v \geq \frac{3}{3 - \frac{2x}{m}} = 1 + \frac{2}{3m + 2}.$$

□

4.2.3 Lower bound $1 + 2/(5m + 2)$ for m odd

We will use the same example as for the lower bound on competitive factor for m odd, that is Example 4.8.

Theorem 4.12. *Let $m = 2n + 1 \geq 5$ and $v \geq 1$. If there exists an algorithm for* PSMJ*(m const., $size_j \in \{1, 2\}$) that is 1-competitive in the v-speedup model, then $v \geq 1 + 2/(5m + 2)$.*

Proof. Let I^-_{2n+1}, I^+_{2n+1} be the instances from Example 4.8.

The proof to this theorem follows the same structure as the proof of the Theorem 4.10. In here we choose $x = 5m/(5m + 4)$. Let $p = |S^{A_1}_{<1} \cap S^{A_2}_{<1}|$.

In *Case 1*, $p \leq x$, we know

$$\begin{aligned} 2 = C^*_{max} \geq C_{max}(S) &\geq \max(C_{A_1}(S), C_{A_2}(S)) \geq 1 + \frac{p^{\uparrow v}_{A_1}(S, 1) + p^{\uparrow v}_{A_2}(S, 1)}{2} \\ &\geq 1 + \frac{4 - 2pv - (1 - p)v}{2} = 3 - \frac{v(p + 1)}{2} \geq 3 - \frac{v(x + 1)}{2}. \end{aligned}$$

That gives us $v \geq 2/(1+x) = 1 + (1-x)/(1+x) = 1 + 2/(5m+2)$.

In *Case 2*, $p > x$, we know there is at least p idle time and so

$$\frac{5}{2} = C^*_{max} \geq C_{max}(S) \geq \frac{\frac{\mathrm{Vol}\left(I^+_{2n+1}\right)}{v} + x}{m} = \frac{5}{2v} + \frac{x}{m}.$$

That gives us

$$v \geq \frac{\frac{5}{2}}{\frac{5}{2} - \frac{x}{m}} = 1 + \frac{2}{5m+2}.$$

□

5. Algorithms

In the following three sections we provide 1-competitive algorithms for the problem PSMJ(m *const.*, $size_j \in \{1, 2\}$) in the $m/(m-1)$-speedup model, for the same problem but only for m even in the $(m+1)/m$-speedup model and for the problem PSMJ(m *const.*, $size_j \leq k$) in the $m/(m-k+1)$-speedup model. They are called LPTIS, LPTIS+ and LPTISK, where LPTIS stands for *Longest Processing Times Ignoring Sizes*. The idea behind these algorithms is that jobs with long processing times are potential problems for constructing short schedules. If at all times we were somehow able to have all jobs shorter than the optimum, we would surely create a schedule at most as long as the offline optimum. Of course this is not possible as we do not know what the optimum, we are comparing to, will do. But if we relax this requirement just a bit, now requiring that our longest job is at most as long as the longest job of the optimum, the same being true for the sum of processing times of the two longest jobs, three, four, etc. (the invariant is going to be a bit different as we need to handle the job sizes correctly), we are still able to construct a schedule at most as long as the optimum. And this is something we can achieve with the right amount of speedup.

To achieve this we start this chapter with some definitions and theorems that will be used in all of these three algorithms.

As said above, there is an invariant that the LPTIS algorithms will keep throughout their computation that somehow expresses that the remaining instance of the algorithm has shorter jobs then the remaining instance of any optimal schedule. The invariant is following. We sort the jobs in the non-increasing order by their processing times and create a sequence of these processing times, repeating every processing time as many times as is the size of the job. This explains the part *ignoring sizes* in the name of the algorithms, as in this sequence we forget what were the original sizes of jobs. The invariant states that the sequence created from the remaining instance of the algorithm is, in some sense, smaller than the sequence created from the remaining instance of the offline optimum we are comparing to.

Definition 5.1 (Sequence ignoring sizes)**.** *Let J be a set of jobs and let the jobs in J be $\sigma(1), \ldots, \sigma(n)$, ordered by their non-increasing processing times, ties broken arbitrarily. We define a sequence ignoring sizes for J to be an infinite sequence $a = a(J)$ such that*

$$a_j = p_{\sigma(i)} \qquad \text{for} \qquad 1 \leq i \leq n \quad \text{and} \quad 1 + \sum_{k=1}^{i-1} s_{\sigma(k)} \leq j \leq \sum_{k=1}^{i} s_{\sigma(k)},$$

$$a_j = 0 \qquad \text{for} \qquad j > \sum_{k=1}^{n} s_{\sigma(k)}.$$

For two infinite sequences a, b we write $a \preccurlyeq b$ and say a is smaller than b if

$$\sum_{i=1}^{k} a_i \leq \sum_{i=1}^{k} b_i \qquad \forall k \geq 1.$$

Notice that the definition of $a(I)$ is sound because the sequence is going to be the same no matter how we broke the ties when ordering the jobs.

As we mentioned above, LPTIS algorithms will behave in such a way, that the sequence ignoring sizes created from its remaining instance is going to be smaller than the sequence ignoring sizes of some offline optimum. For this invariant to make sense, we need it to hold after new jobs are released. This is formally stated in the Theorem 5.3 below. But first we need the following lemma.

Lemma 5.2. *Let $a, b : \mathbb{N} \to \mathbb{R}$ be two infinite sequences of real numbers and let $\pi : \mathbb{N} \to \mathbb{N}$ be a permutation such that $b = a \circ \pi$ and b is non-increasing. Then $a \preccurlyeq b$.*

Proof. Let $k \geq 1$. We want to prove $\sum_{i=1}^{k} a_i \leq \sum_{i=1}^{k} b_i$. Let $[k] = \{1, 2, \ldots, k\}$. We denote $M = \pi^{-1}([k])$ and we construct a bijective function $r : M \to [k]$ that returns a rank of i in M when M is sorted in the ascending order. Now for every $i \in M$ we have $r(i) \leq i$, so $b_i \leq b_{r(i)}$ and therefore

$$\sum_{i \in [k]} a_i = \sum_{i \in [k]} b_{\pi^{-1}(i)} = \sum_{i \in M} b_i \leq \sum_{i \in M} b_{r(i)} = \sum_{i \in [k]} b_i.$$

□

Theorem 5.3. *Let I, I' and J be pairwise disjoint sets of jobs such that $a(I) \preccurlyeq a(I')$. Then also $a(I \cup J) \preccurlyeq a(I' \cup J)$.*

Proof. We will prove the theorem by induction on the size of the set J. If J is empty the statement is trivial.

Now let $J = \{j\}, a = a(I), b = a(I'), a^j = a(I \cup J)$ and $b^j = a(I' \cup J)$. If $p_j = 0$ the statement is trivial. Otherwise, let k be the index such that $a_k \geq p_j$ and $a_{k+1} < p_j$. Then

$$a_i^j = \begin{cases} a_i & i \leq k, \\ p_j & k+1 \leq i \leq k + s_j, \\ a_{i-s_j} & i > k + s_j. \end{cases}$$

Similarly to this, we create a sequence b':

$$b'_i = \begin{cases} b_i & i \leq k, \\ p_j & k+1 \leq i \leq k + s_j, \\ b_{i-s_j} & i > k + s_j. \end{cases}$$

From $a \preccurlyeq b$ we clearly have also $a^j \preccurlyeq b'$. But b' is a permutation of b^j and b^j is a non-increasing sequence so by Lemma 5.2 we also know $b' \preccurlyeq b^j$. Together $a^j \preccurlyeq b' \preccurlyeq b^j$ and the base case of induction is done.

Finally suppose $|J| = k > 1$ and let $J = J' \cup \{j\}$. By the induction hypothesis $a(I \cup J') \preccurlyeq a(I' \cup J')$ and once again by the induction hypothesis

$$a(I \cup J) = a((I \cup J') \cup \{j\}) \preccurlyeq a((I' \cup J') \cup \{j\}) = a(I' \cup J).$$

□

In order for LPTIS algorithms to preserve the invariant we mentioned, we need to run longer jobs with higher priority. We try to imagine what is intuitively the best option for the offline optimum, in order to make its sequence ignoring sizes as small as possible. In the ordering of the infinite sequences we defined, the first terms of the sequence have the most impact on the ordering. This can be observed by the fact that if we take infinite sequences a and a' that differ only on positions i and j and we know $a_i < a'_i$ and $a_i + a_j = a_i + a'_j$, then $a \preccurlyeq a'$. So we want to primarily make a_1 as small as possible, secondarily make a_2 as small as possible etc. Of course, running the job corresponding to a_i such that a_i becomes smaller then a_{i+1} does not help us because the sequence ignoring sizes is always sorted. So when processing times of two jobs are equalized, we want to run them at the same speed, possibly sharing the time between more jobs on the available processors. Equalizing jobs in this way is exactly what the Definition 3.5 does. In the Chapter 3 we only used it for jobs of the same size and we said that it is not always possible to create an ideal schedule otherwise. But when we consider enough speedup, it is possible. Using the Definition 3.5 on the whole remaining instance exactly corresponds to running the longest jobs, ignoring sizes.

If we are able to construct an ideal schedule from the Definition 3.8 and if we prove that no algorithm (without speedup) can achieve a smaller sequence ignoring sizes than by running the jobs for the time equal to their requested processing times, we can easily create an algorithm that maintains our invariant.

The following theorem, although it is quite technical, is the core of the LPTIS algorithms. It states that by creating an ideal schedule, the invariant remains true. This can be interpreted as an informal statement that running jobs according to the requested instance leads to the smallest possible new sequence ignoring sizes than any, i.e., also the optimal, algorithm can achieve without speedup.

Theorem 5.4. *Let I be an instance of the problem* PSMJ *and let $v \geq 1$. Let S and S' be two feasible schedules for I and $0 \leq t < t'$ be two numbers such that $I(t'') = \emptyset$ for every $t < t'' < t'$. Denote $\Delta = t' - t$.*

Suppose that $a(R^{\uparrow v}(S,t)) \preccurlyeq a(R(S',t))$ and that the schedule $S_{[t,t')}$ is ideal for $R^{\uparrow v}(S,t)$ and Δ in the v-speedup model. Then $a(R^{\uparrow v}(S,t')) \preccurlyeq a(R(S',t'))$.

Proof. By Theorem 5.3 we know that adding new jobs does not change the inequality. So we can assume $I(t') = \emptyset$ as well.

Let $J = R^{\uparrow v}(S,t)$, $a = a(R^{\uparrow v}(S,t)), b = a(R(S',t)), a' = a(R^{\uparrow v}(S,t'))$ and $b' = a(R(S',t'))$. Also let L^J_Δ and q_Δ be defined as in the Definition 3.5. Because $S_{[t,t')}$ is an ideal schedule for J and Δ in the v-speedup model we know that $p_j^{\uparrow v}(S,t') = p_j^{\uparrow v}(S,t) - q^j_\Delta$ for every $j \in J$.

First we need to show that whenever $p_j^{\uparrow v}(S,t) \geq p_{j'}^{\uparrow v}(S,t)$ then also $p_j^{\uparrow v}(S,t') \geq p_{j'}^{\uparrow v}(S,t')$. We will split the proof into three cases.

Case I: $p_j^{\uparrow v}(S,t') < L^J_\Delta$. Then by (3.15) from the Theorem 3.7 we know $q^j_\Delta = 0$ and so $p_{j'}^{\uparrow v}(S,t') \leq p_{j'}^{\uparrow v}(S,t) \leq p_j^{\uparrow v}(S,t) = p_j^{\uparrow v}(S,t')$.

Case II: $p_j^{\uparrow v}(S,t') \geq L^J_\Delta$ and $p_{j'}^{\uparrow v}(S,t') \leq L^J_\Delta$. Then trivially $p_j^{\uparrow v}(S,t') \geq L^J_\Delta \geq p_{j'}^{\uparrow v}(S,t')$.

Case III: $p_j^{\uparrow v}(S,t') \geq L^J_\Delta$ and $p_{j'}^{\uparrow v}(S,t') > L^J_\Delta$. Then by (3.16) we have $q^{j'}_\Delta = \Delta$ and also $p_j^{\uparrow v}(S,t) \geq p_{j'}^{\uparrow v}(S,t) = p_{j'}^{\uparrow v}(S,t') + \Delta > L^J_\Delta + \Delta$ and so $q^j_\Delta = \Delta$ as well.

We see that both of the numbers are decreased by the same amount, so their order is preserved.

Because of this lemma, when we construct the sequence $a' = a(R^{\uparrow v}(S,t'))$, the jobs can have the same order as when we constructed the sequence $a = a(R^{\uparrow v}(S,t))$. So

$$a'_i = a_i - \max(0, \min(\Delta, a_i - L^J_\Delta)).$$

Let $\pi : \mathbb{N} \to \mathbb{N}$ be a permutation such that $b'_{\pi(i)}$ corresponds to the processing time of the same job as b_i. From Lemma 5.2 we know that for every $j \geq 1$

$$\sum_{i=1}^{j} b'_{\pi(i)} \leq \sum_{i=1}^{j} b'_i. \tag{5.1}$$

Now we can prove the theorem itself. We start with the case $L^J_\Delta = 0$. Then $a'_i = a_i - \Delta$ for $a_i > \Delta$ and $a'_i = 0$ otherwise. Let k be the largest index such that $a'_i > 0$. Then it is enough to show that $\sum_{i=1}^{j} a'_i \leq \sum_{i=1}^{j} b'_i$ for $j \leq k$, for larger j it follows from the case $j = k$ as we only add zeros to the left hand side. From (5.1), because no job can run for longer than Δ and from the assumption $a \preccurlyeq b$ we have the required inequality

$$\sum_{i=1}^{j} b'_i \geq \sum_{i=1}^{j} b'_{\pi(i)} \geq \sum_{i=1}^{j} (b_i - \Delta) \geq \sum_{i=1}^{j} (a_i - \Delta) = \sum_{i=1}^{j} a'_i.$$

Now $L^J_\Delta > 0$. From the Definition 3.5 we know that (3.11) holds and using (3.13) we get

$$\sum_{i=1}^{\infty} a_i - \sum_{i=1}^{\infty} a'_i = \mathrm{Vol}\left(\mathcal{I}_\Delta^{R^{\uparrow v}(S,t)}\right) = V_\Delta^{R^{\uparrow v}(S,t)} = m \cdot \Delta \geq \sum_{i=1}^{\infty} b_i - \sum_{i=1}^{\infty} b'_i, \tag{5.2}$$

because no schedule on m processors of length m can process more volume than $m \cdot \Delta$.

Suppose for contradiction that

$$\sum_{i=1}^{j} a'_i > \sum_{i=1}^{j} b'_i \tag{5.3}$$

and j is the smallest such index. From the minimality of j we have $b'_j < a'_j$.

If $b'_j \geq L^J_\Delta$ then $a'_j > L^J_\Delta$ and so $a'_i = a_i - \Delta$ for $i \in \{1, \ldots, j\}$ by (3.16). That gives us

$$\sum_{i=1}^{j} b'_i \geq \sum_{i=1}^{j} b'_{\pi(i)} \geq \sum_{i=1}^{j} (b_i - \Delta) \geq \sum_{i=1}^{j} (a_i - \Delta) = \sum_{i=1}^{j} a'_i$$

and that is a contradiction with (5.3).

This means $b'_j < L^J_\Delta$. Let k be the largest index such that $a'_k < a_k$. If $k \leq j$, then using the assumption $a \preccurlyeq b$ and (5.2) we get

$$\sum_{i=1}^{j} a'_i = \sum_{i=1}^{j} a_i - \sum_{i=1}^{j} (a_i - a'_i) = \sum_{i=1}^{j} a_i - \sum_{i=1}^{\infty} (a_i - a'_i) \leq \sum_{i=1}^{j} b_i - \sum_{i=1}^{\infty} (b_i - b'_i) \leq \sum_{i=1}^{j} b'_i,$$

Algorithm 5.1 Schema for `LPTIS` algorithms. It depends on the speedup $v \geq 1$ and a procedure `Schedule`.

```
1: t ← 0
2: S ← empty schedule
3: I ← instance available at time 0
4: while not all jobs released or I ≠ ∅ do
5:    if not all jobs released then
6:       t' ← get next release time
7:    else
8:       t' ← t + max(Longest(I, 1), Vol(I)/m)
9:    end if
10:   S_[t,t') ← Schedule(I, t' − t)
11:   I ← change I according to schedule S_[t,t') and speedup v
12:   if not all jobs released then
13:      I ← add jobs to I received at time t'
14:   end if
15: end while
16: return S
```

which is again a contradiction with (5.3). So $k > j$. Then by the contrapositive of (3.15) we have $a'_1 \geq \cdots \geq a'_k \geq L^J_\Delta > b'_j \geq b'_{j+1} \geq \cdots \geq b'_k$ and so $a'_i > b'_i$ for $j+1 \leq i \leq k$. Using this, the assumption $a \preccurlyeq b$ and (5.3) we have

$$\begin{aligned}
\sum_{i=1}^{\infty} a_i - \sum_{i=1}^{\infty} a'_i &= \sum_{i=1}^{k} (a_i - a'_i) \\
&= \sum_{i=1}^{k} a_i - \sum_{i=1}^{j} a'_i - \sum_{i=j+1}^{k} a'_i \\
&< \sum_{i=1}^{k} b_i - \sum_{i=1}^{j} b'_i - \sum_{i=j+1}^{k} b'_i \\
&= \sum_{i=1}^{k} (b_i - b'_i) \\
&\leq \sum_{i=1}^{\infty} b_i - \sum_{i=1}^{\infty} b'_i,
\end{aligned}$$

which is a contradiction with (5.2). □

Now we are prepared to present the schema that all the `LPTIS` algorithms follow. It is recorded using pseudocode in Algorithm 5.1. The differences between the algorithms are in the problem they are trying to solve, in the way how they construct the ideal schedule and in the speedup v they use. The following theorem states that creating an ideal schedule is the only missing part to obtain a 1-competitive algorithm in the v-speedup model for the problem `PSMJ`.

Theorem 5.5 (`LPTIS` schema)**.** *Let us have an algorithm following the schema of Algorithm 5.1 with some $v \geq 1$. Furthermore, suppose that the* `Schedule` *procedure called with parameters I and Δ returns an ideal schedule for these parameters in the v-speedup model. Then the algorithm is 1-competitive for* `PSMJ` *in the v-speedup model.*

Proof. Let I_0 be the instance on the input. Let S^* be some feasible schedule with optimal makespan for I_0. Let T be the set containing 0 and all the release times of jobs in I_0. By induction on T we will prove that for any $t \in T$ the schedule $S_{<t}$ is feasible and $a(R^{\uparrow v}(S,t)) \preccurlyeq a(R(S^*,t))$. It is true for $t = 0$. If it is true for t, by Theorem 5.4 it is also true for the following release time and the claim is true.

Now let t be the last release time. We know that $a(R^{\uparrow v}(S,t)) \preccurlyeq a(R(S^*,t))$. Let I and t' have the values as in the Algorithm 5.1 on line 8, so $I = R^{\uparrow v}(S,t)$ and $t' = \max\left(t + \text{Longest}\,(I,1), t + \frac{\text{Vol}(I)}{m}\right)$. Also, let $I^* = R(S^*,t)$ be the set of jobs remaining in the schedule S^* at time t. We know that

$$t + \text{Longest}\,(I,1) = t + a(I)_1 \leq t + a(I^*)_1 = t + \text{Longest}\,(I^*,1),$$

and

$$t + \frac{\text{Vol}\,(I)}{m} = t + \frac{\sum_{i=1}^{\infty} a(I)_i}{m} \leq t + \frac{\sum_{i=1}^{\infty} a(I^*)_i}{m} = t + \frac{\text{Vol}\,(I^*)}{m}.$$

Both values on the right side are lower bounds on the length of the schedule $C^*_{max} = C_{max}(S^*)$ so for their maximum t'' it is true that $t'' \leq C^*_{max}$. The maximum of the left sides is t' so $t' \leq t'' \leq C^*_{max}$.

Finally we will prove that $C_{max}(S) \leq t'$. As the schedule $S_{[t,t')}$ is ideal for I and $\Delta = t' - t$ in the v-speedup model, let $\mathcal{I}^I_\Delta$ be the corresponding ideal instance. We will prove that I and $\mathcal{I}^I_\Delta$ are equivalent instances (their jobs have the same parameters). The theorem then follows because $S_{[t,t')}$ is complete for $\mathcal{I}^I_\Delta$ in the v-speedup model and so if $\mathcal{I}^I_\Delta$ and I are equivalent, $S_{[t,t')}$ is also complete for I in the v-speedup model and so t' is an upper bound on the makespan of S.

We know $L^I_\Delta = 0$ because by putting $L = 0$ into the left hand side of (3.11) we get $V(0) \leq \text{Vol}\,(I) \leq m \cdot \Delta$ because of $\Delta = t' - t \geq \frac{\text{Vol}(I)}{m}$. As V is non-increasing, the smallest value L for which $V(L) = m \cdot \Delta$ either does not exist or is $L = 0$.

This means $q^j_\Delta = \min(\Delta, p_j)$ for $j \in I$. But from $\Delta = t' - t \geq \text{Longest}\,(I,1)$ we also know $\Delta \geq p_j$. This proves $\mathcal{I}^I_\Delta$ is equivalent to I.

We proved $C_{max}(S) \leq t' \leq C^*_{max}$ so the considered algorithm is 1-competitive. □

Now it suffices to create an ideal schedule. That is what are the following three sections about. Note that we did state all the previous theorems for the problem `PSMJ` even though our primary motive was solving the problem `PSMJ`(m *const.*, $size_j \in \{1,2\}$). We present two algorithms for this problem and then we exploit the generality of the previous theorems and provide one algorithm for the problem `PSMJ`(m *const.*, $size_j \leq k$).

5.1 `LPTIS`: $m/(m-1)$-speedup

In this section we describe the algorithm `LPTIS` that is 1-competitive for the problem `PSMJ`(m *const.*, $size_j \in \{1,2\}$) in the $m/(m-1)$-speedup model. As mentioned above, we only need to design the procedure `Schedule` with this speedup. For that we will closely follow the McNaughton's rule algorithm.

Algorithm 5.2 Procedure `Schedule` for `LPTIS` algorithm.

1: read I and Δ
2: $J \leftarrow (\mathcal{I}^I_\Delta)_{\downarrow v}$
3: $S \leftarrow$ empty schedule
4: $x \leftarrow \lceil (\mathrm{Vol}\,(J^2)/2)/\Delta \rceil$
5: add jobs J^2 to S using McNaughton's rule on processors $1, \ldots, 2x$
6: add jobs J^1 to S using McNaughton's rule on processors $2x, \ldots, m$
7: **return** S

Theorem 5.6. *Let I be an instance of* `PSMJ`$(m$ *const.*$, size_j \in \{1, 2\})$*, also let $v = m/(m-1)$ and $\Delta > 0$. We can construct an ideal schedule for I and Δ in the v-speedup model.*

Proof. Let $J = (\mathcal{I}^I_\Delta)_{\downarrow v}$. We need to construct a complete schedule for J of makespan at most Δ.

We create the schedule using McNaughton's rule algorithm. First we schedule the big jobs J^2 and then the small jobs J^1.

We can schedule the big jobs J^2 using the McNaughton's rule because if we scale their sizes to 1 (denote the new instance as J') and the number of processors to $m' = \lfloor m/2 \rfloor$ we can use Theorem 2.1 directly. We just need to verify the two conditions. By (3.12) from Theorem 3.7

$$\mathrm{Longest}\,(J', 1) \leq \mathrm{Longest}\,(J, 1) = \frac{\mathrm{Longest}\left(\mathcal{I}^I_\Delta, 1\right)}{v} \leq \frac{\Delta}{v} < \Delta, \tag{5.4}$$

and by (3.14)

$$\mathrm{Vol}\,(J') = \frac{\mathrm{Vol}\,(J^2)}{2} \leq \frac{\mathrm{Vol}\,(J)}{2} = \frac{\mathrm{Vol}\left(\mathcal{I}^I_\Delta\right)}{2v} \leq \frac{m \cdot \Delta}{2v} = \frac{m-1}{2} \cdot \Delta \leq m' \cdot \Delta.$$

Now we want to add the small jobs J^1 to the schedule. We would like to continue using McNaughton's rule but in the proof of Theorem 2.1 the invariant, that ensured that the schedule can be finished, was that there is at most one partially filled processor. Now there are at most two such processors. But we can ignore the remaining space on one of them (or fill it with an imaginary job). Then there is only one partially filled processor. Moreover, exactly as in (5.4) we know $\mathrm{Longest}\,(J^1, 1) < \Delta$ so we just need to check that the jobs fit into the remaining space. As we threw away at most Δ units of space in the schedule, the jobs will fit in the remaining space if $\mathrm{Vol}\,(J) \leq (m-1) \cdot \Delta$. This again follows from (3.14) in Theorem 3.7

$$\mathrm{Vol}\,(J) = \frac{\mathrm{Vol}\left(\mathcal{I}^I_\Delta\right)}{v} \leq \frac{m \cdot \Delta}{v} = (m-1) \cdot \Delta.$$

□

The algorithm for creating an ideal schedule occurring in the proof of the previous theorem is described in Algorithm 5.2.

Theorems 5.5 and 5.6 give us directly the following corollary.

Corollary 5.7. *The algorithm defined by Algorithms 5.1, 5.2 is 1-competitive for* PSMJ$(m$ *const.*$, size_j \in \{1,2\})$ *in the* $m/(m-1)$*-speedup model.*

Using Theorem 1.13 this gives us also the following corollary.

Corollary 5.8. *The competitive ratio of* PSMJ$(m$ *const.*$, size_j \in \{1,2\})$ *is at most* $m/(m-1)$.

5.2 LPTIS+: $(m+1)/m$-speedup for m even

In the proof of Theorem 5.6 there is one place with strict inequality, even when non-strict equality is needed. Specifically, we needed Longest $(J_{\downarrow v}, 1) \leq \Delta$ but in (5.4) we showed even Longest $(J_{\downarrow v}, 1) \leq \frac{\Delta}{v}$. The reason is simple, we need to run the longest jobs for at least as long as the optimum. That means for at least Δ. But because of the speedup, it only needs to occupy $\frac{\Delta}{v}$ units in the schedule.

So the question is if we can do better. We can, at least for m even. The reason why we need the speedup at all is because we cannot run the longest jobs and also ensure that we use all processors. So we guarantee the use of only $m-1$ processors and the speedup ensures we process enough volume. To make the speedup smaller, we will create part of the schedule by running jobs that fill all m processors and rest of the schedule by running the longest jobs as previously. And because of the non-tightness of (5.4) we will still process enough units of the long jobs.

The bound from (5.4) also guides us how long should be these parts of the schedule. For bound (5.4) to still hold (with non-strict inequality) we need the second part of the schedule to be at least Δ/v long. Then the first part will be long $(1-1/v)\cdot\Delta$.

A lemma follows that we will use to construct the first part of the schedule.

Lemma 5.9. *Let* $v = (m+1)/m$, $t_v = 1 - 1/v = 1/(m+1)$ *and let* $m \geq 4$ *be even. Also let* J *be a set of jobs and* $\Delta > 0$ *such that* Longest $(J, 1) \leq \Delta$ *and*

$$\mathrm{Vol}\,(J) \geq v \cdot (m-1) \cdot \Delta. \tag{5.5}$$

There is a schedule S *of length* $t = t_v \cdot \Delta$ *such that* $p_j(S,t) \leq p_j/v$ *for every* $j \in J$ *and there is no idle time in the schedule.*

Proof. Let the jobs in J be $\sigma(1), \sigma(2), \ldots, \sigma(n)$ ordered by p_j from the largest to the smallest value. Let $a = a(J)$ and let $c(i)$ be the index of the job which corresponds to a_i. We distinguish between two cases.

First suppose $a_m \geq t_v \cdot \Delta$. Then the jobs $\sigma(1), \ldots, \sigma(c(m))$ all satisfy $p_j \geq t_v\Delta$. If $c(m) \neq c(m+1)$ the sum of the sizes of the jobs $\sigma(1), \ldots, \sigma(c(m))$ is precisely m. So we can schedule these jobs, each on its corresponding number of processors in the time interval $[0, t_v \cdot \Delta)$. Otherwise $c(m) = c(m+1)$. The sum of sizes of jobs $\sigma(1), \ldots, \sigma(c(m))$ is then $m+1$ which is an odd number and so one of these jobs is of size 1. The remaining jobs have sum of sizes equal to m and we can create the schedule out of them, scheduling each on the time interval $[0, t_v \cdot \Delta)$.

Now suppose $a_m < t_v \cdot \Delta$. Let

$$J' = \{\sigma'(i) \leftarrow [s_{\sigma(i)}, \min(p_{\sigma(i)}, t_v\Delta), r_{\sigma(i)}] \mid \sigma(i) \in J]\}.$$

Notice that

$$\begin{aligned}\operatorname{Vol}(J') &\geq \operatorname{Vol}(J) - (m-1)\cdot\Delta\cdot(1-t_v) \geq v\cdot(m-1)\cdot\Delta - \frac{(m-1)\Delta}{v} \\ &= \frac{(2m+1)(m-1)}{m(m+1)}\cdot\Delta \geq \frac{m+2}{m+1}\cdot\Delta = (m+2)\cdot t_v\cdot\Delta, \end{aligned} \tag{5.6}$$

where the first inequality is true because the jobs $\sigma'(1),\ldots,\sigma'(c(m)-1)$ of sum of sizes at most $m-1$ lose at most $p_j - \min(p_j, t_v\Delta) \leq (1-t_v)\Delta$ (as $p_j \leq \Delta$) of their processing time each and the remaining jobs do not lose any processing time, the second inequality is from the assumption (5.5) and the last inequality is equivalent to $m^2-3m-1\geq 0$ which is true for $m\geq 4$.

We will create a schedule of length $t_v\cdot\Delta$ using McNaughton's rule. We can do that because for every job $\sigma'(i)\in J'$, $p_{\sigma'(i)}\leq t_v\cdot\Delta$. We start by scheduling all the jobs of size 2. If we do not fill the whole schedule, at most two processors will be partially filled. We throw away what is scheduled on them and schedule jobs of sizes 1 on these and the following processors. Now we will certainly fill all processors because we threw away at most $2\cdot t_v\cdot\Delta$ volume and $\operatorname{Vol}(J')-2\cdot t_v\cdot\Delta \geq m\cdot(t_v\cdot\Delta)$ by (5.6). □

We will generalize the key part of the proof of Theorem 5.6 into the following lemma.

Lemma 5.10. *Let I be an instance, $\Delta>0$ and $v\geq 1$. Suppose that $\operatorname{Vol}(I)\leq v\cdot(m-1)\cdot\Delta$ and $\operatorname{Longest}(I,1)\leq v\cdot\Delta$. Then there is a complete schedule for I and Δ in the v-speedup model of makespan at most Δ.*

Proof. The proof goes along the same lines as the proof of Theorem 5.6. We put jobs $I^2_{\downarrow v}$ to the schedule using McNaughton's rule which we can do because

$$\operatorname{Longest}\left(I^2_{\downarrow v},1\right)\leq\frac{\operatorname{Longest}(I,1)}{v}\leq\Delta \quad\text{and}\quad \frac{\operatorname{Vol}\left(I^2_{\downarrow v}\right)}{2}\leq\frac{\operatorname{Vol}(I)}{2v}\leq\left\lfloor\frac{m}{2}\right\rfloor\cdot\Delta.$$

At this point at most two processors are partially filled. We will fill one of them with imaginary job (that creates an idle time of at most Δ units) and then use McNaughton's rule on the jobs $I^1_{\downarrow v}$. It will fit in the schedule because

$$\operatorname{Longest}\left(I^1_{\downarrow v},1\right)\leq\Delta \quad\text{and}\quad \operatorname{Vol}(I_{\downarrow v})+\Delta\leq\frac{\operatorname{Vol}(I)}{v}+\Delta\leq m\cdot\Delta.$$

□

Theorem 5.11. *Let I be an instance of* PSMJ(m *const.*, $size_j\in\{1,2\}$)*, also let $v=(m+1)/m$, $\Delta>0$ and $m\geq 4$ be even. We can construct an ideal schedule for I and Δ in the v-speedup model.*

Proof. Let $J=\mathcal{I}^I_\Delta$. We need to construct a complete schedule for J in the v-speedup model of makespan at most Δ.

First suppose that (5.5) holds. Let $t_v=1-1/v=1/(m+1)$ and create a schedule $S_{<(t_v\cdot\Delta)}$ using Lemma 5.9. Let $J'=R^{\uparrow v}(S,t_v\cdot\Delta)$. By (3.12) from Theorem 3.7 we know that

$$\operatorname{Longest}(J',1)\leq\operatorname{Longest}(J,1)\leq\Delta=v\cdot\frac{\Delta}{v},$$

Algorithm 5.3 Procedure `Schedule` for `LPTIS+` algorithm.

read I and Δ
$J \leftarrow (\mathcal{I}^I_\Delta)_{\downarrow v}$
$S \leftarrow$ empty schedule
if $\mathrm{Vol}\,(J) \geq v \cdot (m-1) \cdot \Delta$ **then**
  $t_v \leftarrow 1 - 1/v$
  if $a(J)_m \geq t_v \Delta$ **then**
    $x \leftarrow$ minimum number such that the sum of sizes of the first x longest jobs is at least m
    $X \leftarrow$ set of x longest jobs
    if $\sum_{j \in X} s_j > m$ **then**
      delete from X a job of size 1
    end if
    $S_{[0,t_v\Delta)} \leftarrow$ schedule jobs in X on the whole interval $[0, t_v\Delta)$
  else
    $J' = \{j' \leftarrow [s_j, \min(p_j, t_v\Delta), r_j] \mid j \in J]\}$
    $x \leftarrow \lfloor (\mathrm{Vol}\,(J'^2)/2)/(t_v\Delta) \rfloor$
    add jobs J'^2 to $S_{[0,t_v\Delta)}$ using McNaughton's rule on processors $1, \ldots, 2x$ until they are full
    add jobs J'^1 to $S_{[0,t_v\Delta)}$ using McNaughton's rule on processors $(2x+1), \ldots, m$ until they are full
  end if
  create schedule $S_{[t_v\Delta,\Delta)}$ using Algorithm 5.2 on $R^{\uparrow v}(S, t_v\Delta)$
else
  create schedule $S_{[0,\Delta)}$ using Algorithm 5.2 on I
end if
return S

and because $S_{<(t_v \cdot \Delta)}$ is filled completely, by (3.14) from Theorem 3.7 and because $v \cdot m \cdot t_v = 1$ we also know that

$$\mathrm{Vol}\,(J') = \mathrm{Vol}\,(J) - v \cdot m \cdot (t_v \cdot \Delta) \leq m \cdot \Delta - (v \cdot m \cdot t_v) \cdot \Delta = v \cdot (m-1) \cdot \frac{\Delta}{v}.$$

This means that by Lemma 5.10 for J' and Δ/v we can create a complete schedule $S_{[t_v \cdot \Delta, t_v \cdot \Delta + \Delta/v)} = S_{[t_v \cdot \Delta, \Delta)}$ for J' in the v-speedup model. So S is a schedule of makespan Δ that is complete for J in the v-speedup model.

On the other hand, if (5.5) does not hold, we can use Lemma 5.10 directly for J and Δ and obtain the whole schedule $S_{[0,\Delta)}$. □

We capture the `Schedule` algorithm from the proof of previous theorem in Algorithm 5.3.

Theorems 5.5 and 5.11 give us the following corollary.

Corollary 5.12. *For $m \geq 4$ even the algorithm defined by Algorithms 5.1, 5.3 is 1-competitive for* $\texttt{PSMJ}(m\ const., size_j \in \{1,2\})$ *in the $(m+1)/m$-speedup model.*

Using Theorem 1.13 this gives us also the following corollary.

Corollary 5.13. *The competitive ratio of* $\texttt{PSMJ}(m\ const., size_j \in \{1,2\})$ *for even number of processors $m \geq 4$ is at most $(m+1)/m$.*

Algorithm 5.4 Procedure `Schedule` for `LPTISK` algorithm.

```
1: read I and Δ
2: J ← (I^I_Δ)_↓v
3: S ← empty schedule
4: p ← 0
5: for l = k to 1 do
6:    add jobs J^l to S using McNaughton's rule starting on processor p
7:    if l ≠ 1 then
8:       p ← first processor that is not full
9:       if p is partially full then
10:         mark the empty space on processor p as thrown away
11:         p ← p + 1
12:      end if
13:   end if
14: end for
15: return S
```

The natural question is whether we can do better with this approach. For m odd the problem is in Lemma 5.9. If we had $(m+1)/2$ jobs of size 2 and no job of size 1 we could not create a schedule without idle time. And as we do not know anything about the job sizes of the optimal solution we are comparing to, in general it could have a job of size 1 and create a schedule without idle time. For m even Lemma 5.9 works fine and we could even create a longer schedule without idle time, the bounds used in the proof are not all tight. Though imagine the algorithm has an instance with the longest job of size 1 and then $m/2$ jobs of size 2. The only possibility for the first part of the schedule to be without idle time is to not run the longest job and if we made this part of the schedule any longer than in `LPTIS+`, the optimum could, on principle, process more units of this longest job. So the invariant would not hold.

5.3 `LPTISK`: $m/(m-k+1)$-speedup for $size_j \leq k$

In this section we generalize the algorithm `LPTIS` from Section 5.1 to the problem `PSMJ`(m *const.*, $size_j \leq k$). In `LPTIS` we are running the longest jobs and we can guarantee to always run jobs on at least $m-1$ machines (if there are enough jobs remaining). The $m/(m-1)$-speedup than guarantees that we process enough volume. So if we allow sizes of jobs to be at most k, hopefully we can guarantee that at least $m-k+1$ machines are used at all times and so with $m/(m-k+1)$-speedup we again process enough volume.

The following theorem uses the same ideas as the corresponding Theorem 5.6 for `LPTIS`.

Theorem 5.14. *Let I be an instance of* `PSMJ`(m *const.*, $size_j \leq k$)*, also let $v = m/(m-k+1)$ and $\Delta > 0$. We can construct an ideal schedule for I and Δ in the v-speedup model.*

Proof. Let $J = (\mathcal{I}^I_\Delta)_{\downarrow v}$. We need to construct a complete schedule for J of makespan at most Δ. We will do that using Algorithm 5.4.

By induction we will prove that jobs $J^k, J^{k-1}, \ldots, J^l$ fit into the schedule, after adding them, if $l > 1$, there is at most $(k-l+1)\Delta$ idle time in the schedule that is marked as thrown away. Counting this idle time as being occupied by jobs, there are either zero or $l-1$ partially full processors.

The base case is $l = k+1$. There are no jobs (they fit into the schedule), there is no idle time marked as thrown away and there are no partially full processors.

Now suppose jobs $J^k, \ldots, J^{l+1}$ are already in the schedule, there is at most $(k-l)\Delta$ idle time marked as thrown away and apart from that either zero or l partially full processors. We will schedule the jobs J^l by McNaughton's rule. As they are either zero or l partially full processors, we can use it. The longest job is at most Δ so we just need to verify that the volume of the jobs is at most the empty space in the schedule.

When scheduling the jobs by McNaughton's rule we continue scheduling from left to right, wrapping around the edges, at the place we finished with jobs of size $l+1$. There will be no idle time but at most $l-1$ last processors may be unusable as they do not fit a job of size l. So we just need to check that the space that is already thrown away, the volume of jobs of sizes $k, k-1, \ldots, l$ and $(l-1)\Delta$ for the unusable processors is at most the total space in the schedule $m\Delta$. We can see it is true

$$\begin{aligned}(k-l)\Delta + \operatorname{Vol}\left(\bigcup_{i=l}^{k} J^i\right) + (l-1)\Delta &\leq \operatorname{Vol}(J) + (k-1)\Delta \\ &= \frac{\operatorname{Vol}\left(\mathcal{I}_\Delta^l\right)}{v} + (k-1)\Delta \\ &\leq \frac{m\Delta}{v} + (k-1)\Delta \\ &= m\Delta.\end{aligned}$$

This means that jobs J^l fit into the schedule. If $l = 1$ the whole theorem is proven. Otherwise we need to prove the rest of the induction hypothesis. If there are no partially full processors, the induction step is proved. Otherwise there are exactly l partially filled processors. We mark the remaining space on the first of them as thrown away and we see that the induction hypothesis holds again. □

Theorems 5.5 and 5.14 give us the following corollary.

Corollary 5.15. *The algorithm defined by Algorithms 5.1, 5.4 is 1-competitive for* $\texttt{PSMJ}(m \text{ const.}, size_j \leq k)$ *in the* $m/(m-k+1)$*-speedup model.*

Using Theorem 1.13 this gives us also the following corollary.

Corollary 5.16. *The competitive ratio of* $\texttt{PSMJ}(m \text{ const.}, size_j \leq k)$ *is at most* $m/(m-k+1)$.

5.4 Optimal algorithm conjecture

In this section we propose an algorithm for $\texttt{PSMJ}(m = 4, size_j \in \{1, 2\})$ that we conjecture to be 9/8-competitive, thus matching the lower bound 9/8 from the Theorem 4.4. We will call jobs of size one *small* and jobs of size two *big*.

Algorithm 5.5 Algorithm LPTDS. It has a parameter α.

$I, R \leftarrow$ empty instance
$S \leftarrow$ empty schedule
$t \leftarrow 0$
$g \leftarrow 0$
while schedule is not complete **do**
  if new jobs released at time t **then**
    add new jobs to I and R
    $l \leftarrow$ compute minimum makespan for I using Theorem 2.2
    $g \leftarrow l \cdot \alpha$
  end if
  signature $\leftarrow$ BIG-BIG
  if Longest $(R^2, 1) = 0$ **then**
    signature $\leftarrow$ BIG-SMALL
  else if Longest $(R^2, 2) = 0$ **then**
    signature $\leftarrow$ SMALL-SMALL
  end if
  if *signature* = BIG-BIG and $(t + B_l(R) = g$ or $t + B_{V^2+3l}(R) = g)$ **then**
    signature $\leftarrow$ BIG-SMALL
  end if
  $t' \leftarrow \infty$
  if not all jobs received **then**
    $t' \leftarrow$ get next release time
  end if
  if *signature* = SMALL-SMALL **then**
    $m_1, m_2 \leftarrow 4, 0$
  else if *signature* = BIG-SMALL **then**
    $m_1, m_2 \leftarrow 2, 2$
    $t' \leftarrow \min(t', \text{Vol}\,(R^2)/2)$
  else if *signature* = BIG-BIG **then**
    $m_1, m_2 \leftarrow 0, 4$
    $t' \leftarrow \min(t', \text{Vol}\,(R^2)/4)$
    $t' \leftarrow \min(t', (\text{Vol}\,(R^2) - 2 \cdot \text{Longest}\,(R^2, 1))/2)$
    $t' \leftarrow \min(t', g - \text{Longest}\,(R^1, 1))$
    $t' \leftarrow \min(t', 3g - 3B_{V^2+3l}(R) - 2t)$
  end if
  $\Delta \leftarrow t' - t$
  $S_{[t,t')} \leftarrow$ create ideal schedule for R^1 and Δ on m_1 processors
  $S_{[t,t')} \leftarrow$ merge with ideal schedule for R^2 and Δ on m_2 processors
  $R \leftarrow$ edit the remaining instance
  $t \leftarrow t'$
end while
return S

We call the algorithm LPTDS(α) which stands for *Longest processing time, delay small* and it is described in pseudocode in Algorithm 5.5. It has a parameter α that is the competitive ratio it is trying to achieve. Using Theorem 2.2 it computes the current offline optimal makespan and sets a goal makespan by

multiplying this value by α. It is constructing a P-schedule (see Definition 3.13), so it only needs to decide when to stop running two big jobs and run one big job with two small jobs instead. At any time t, we say that a *signature* at t is BIG-BIG (or that algorithm at t runs according to BIG-BIG) if it runs two big jobs, it is BIG-SMALL if it runs one big job with at most two small jobs and SMALL-SMALL if it runs only small jobs.

The decision when to change the signature from BIG-BIG to BIG-SMALL is based on the goal it has set and on the lower bound values B_l and B_{V^2+3l} from the Definition 3.1. Specifically, it runs according to BIG-BIG while it has two big jobs in the remaining instance, until the next release time or until time t' such that $t' + B_l$ or $t' + B_{V^2+3l}$ is equal to the current goal. Then it runs according to BIG-SMALL while it has big jobs and until the next release time. If there are no big jobs in the remaining schedule, it runs according to SMALL-SMALL.

To compute the time t' when $t' + B_l$ or $t' + B_{V^2+3l}$ is equal to the current goal g is simple if we suppose that the algorithm will run according to BIG-BIG until then. For the first case we just compute $t' = g - B_l$ because B_l will not decrease. For the second case we know B_{V^2+3l} will decrease by $2/3$ per a unit of time as we are running two big jobs. So if the current time is t then we want t' such that $t' + (B_{V^2+3l} - (t' - t) \cdot 2/3) = g$. This is a linear equation that can be solved easily.

Now we try to reason why we think this algorithm could match the lower bound $9/8$ (for $\alpha = 9/8$). Of course we should check what does the algorithm do on the $9/8$ lower bound example. The following claim can be easily verified by running the algorithm on the instances I_4^- and I_4^+.

Claim 5.17. `LPTDS`$(9/8)$ *achieves a competitive ratio* $9/8$ *on* I_4^- *and* I_4^+ *from Example 4.3.*

We will try to intuitively explain why we think the algorithm is good. It constructs a P-schedule and by Theorem 3.15 if α is the competitive ratio of `PSMJ`$(m = 4, size_j \in \{1, 2\})$ then there is an α-competitive algorithm that constructs a P-schedule. So the question is if the way how it decides when to change the signature from BIG-BIG to BIG-SMALL is good. We see that for the instances I_4^- and I_4^+ it chooses this time correctly. Notice that there is only one possible time at which the switch from BIG-BIG to BIG-SMALL can happen to achieve a ratio of $9/8$ on both instances I_4^- and I_4^+.

We see that `LPTDS` makes the switch as late as possible. It cannot do it later because it would not finish by the computed goal and hence would not achieve the required competitive ratio. The disadvantage of running according to BIG-SMALL is that if there is only one small job, we have idle time in the schedule. Perhaps in the future a new small job arrives and so if we did not run according to BIG-SMALL now, we could run the old and new job of size one at the same time and have no idle time in the schedule. Also, if we have the decision to run either two small jobs or one big job, it makes sense to get rid of the big job because two small jobs can easily run at the same time imitating a big job but they can also run independently. So by having more small jobs rather than big jobs we have more possibilities in the future.

Now we consider yet another angle. Using the Theorem 3.2 we can compute the minimum length of the schedule according to the current remaining instance.

We can look on the speed by which the lower bound values decrease depending on the signature. If it is BIG-BIG, the values B_L, B_V and B_{V^2+l} all decrease as much as they can, B_V and B_{V^2+l} with derivative 1 (the numbers decrease by one per unit of time) and B_L can decrease slower (if at least the three longest big jobs have equal processing times) but still faster than when the signature is BIG-SMALL. So the only two numbers that do not decrease at their maximum speeds are B_l and B_{V^2+3l}. If we lose on volume, compared to the optimum, a lot of jobs can arrive, the value B_V gets to the maximum and our deficit shows up. On the other hand, deficit for B_l does not really cause problems. If at some point a new job arrives that changes B_l, it also changes the value B_l of the optimum to the same value. When no such job arrives, we know that B_l is going to be zero by the time g, thanks to line 17 in Algorithm 5.5. The value B_{V^2+3l} is more complicated. If new big jobs arrive, then the value changes by the same amount for the `LPTDS` and for the optimum. If small jobs arrive, they are either so short that they do not influence the value of B_{V^2+3l} or they are long and then they are also quite big for the optimum – we are trying to run big jobs primarily and so our small jobs should not be much shorter than the small jobs of the optimum.

Of course these observations are only informal and do not prove anything. We present one formal result as well, a theorem stating that `LPTDS` algorithm finishes in time if there are no new jobs arriving.

Theorem 5.18. *Let $R(t)$ be the remaining instance of* `LPTDS` *at some point t and g be the goal of* `LPTDS` *such that $t + C^*_{max}(R(t)) \leq g$. Then the algorithm finishes the jobs by the time g. Also if t' is a next release time but there are no new jobs arriving (resp. jobs with processing time zero), then at time t' for the remaining instance $R(t')$ we know $t' + C^*_{max}(R(t')) \leq g$.*

Proof. We can prove the second part of the statement only, as the first part follows from it by choosing $t' = g$. We will prove that for each of the lower bound values, if it is currently equal to the maximum of all the lower bound values, it is decreasing with derivative one. Then the maximum is also always decreasing with derivative one and we are done.

Let R be the current remaining instance. First suppose $B_l = C_{max}$. Then the condition on line 17 in Algorithm 5.5 is true and the signature is not BIG-BIG. First suppose that $\mathrm{Vol}\,(R^2) > 0$. Then the signature is BIG-SMALL. For contradiction, suppose that $\mathrm{Longest}\,(R, 1) = \mathrm{Longest}\,(R, 2) = \mathrm{Longest}\,(R, 3) = l$. Then $B_{V^2+3l} = (\mathrm{Vol}\,(R^2) + 2(l + l + l))/6 > l = B_l$ and that is a contradiction. So at most the two longest jobs have equal processing times and so we can run them at the two available processors with speed 1. Now suppose $\mathrm{Vol}\,(R^2) = 0$. Then the signature is SMALL-SMALL. From $B_l \geq B_V$ we know that at most four longest jobs have equal processing times and we have four processors available so we can run all the longest jobs at speed 1.

Now let $B_L = C_{max}$, then the signature is BIG-BIG or BIG-SMALL. From $B_L \geq B_V$ we know that at most two longest big jobs have equal processing times. So if the signature is BIG-BIG, we run the longest big jobs with speed one. Now let the signature be BIG-SMALL. If the two longest big jobs do not have equal processing times we can run the longest job at speed one. If they have equal processing times we know that $\mathrm{Vol}\,(R^1) > 0$ (otherwise the signature type could not be BIG-SMALL) and so $B_V = (\mathrm{Vol}\,(R^2) + \mathrm{Vol}\,(R^1))/4 > (2 \cdot \mathrm{Longest}\,(R^2, 1) + 2 \cdot \mathrm{Longest}\,(R^2, 2))/4 = B_L$ and that is a contradiction.

Now let $B_V = C_{max}$. If the signature is BIG-BIG we have at least two big jobs and the volume decreases with derivative one. If the signature is BIG-SMALL, then from $B_V \geq B_{V^2+l}$ we know that $\mathrm{Vol}\,(R^1) \geq 2 \cdot \mathrm{Longest}\,(R^1, 1)$ and so there are at least two small jobs and at least one big job (otherwise the signature would be SMALL-SMALL) so the volume decreases with derivative one. If the signature is SMALL-SMALL, form $B_V \geq B_l$ we know there are at least four small jobs and again the volume decreases with derivative one.

Let $B_{V^2+l} = C_{max}$. If the signature is BIG-BIG then there are at least two big jobs and the value decreases with derivative one. If the signature is BIG-SMALL then we have at least one big job and at least one small job (otherwise $B_{V^2+l} = B_L/2 < B_L$). The three longest small jobs cannot have the same length because then $B_V \geq (\mathrm{Vol}\,(R^2) + 3 \cdot \mathrm{Longest}\,(R^1, 1))/4 > B_{V^2+l}$. So we run some big job and at most two longest small jobs and the value decreases with derivative one. The signature cannot be SMALL-SMALL because that would mean $B_{V^2+l} = B_l/2 < B_l$.

Finally, let $B_{V^2+3l} = C_{max}$. Then the condition on line 17 in Algorithm 5.5 is true and the signature is not BIG-BIG. First let the signature be BIG-SMALL. If the four longest small jobs have the same length l then

$$\begin{aligned} B_V &\geq \frac{\mathrm{Vol}\,(R^2) + 4l}{4} = \frac{\mathrm{Vol}\,(R^2)}{4} + l > \frac{\mathrm{Vol}\,(R^2)}{6} + l \\ &= \frac{\mathrm{Vol}\,(R^2) + 2(\mathrm{Longest}\,(R^1, 1) + \mathrm{Longest}\,(R^1, 2) + \mathrm{Longest}\,(R^1, 3))}{6} \\ &= B_{V^2+3l}, \end{aligned}$$

and that is a contradiction. So at most three longest small jobs have equal processing times and we run them with speed 2/3 (resp. we switch between pairs of them). We also run one big job and so the value decreases with derivative $(2 + 2(2/3 + 2/3 + 2/3))/6 = 1$. If the signature is SMALL-SMALL we can run the three longest jobs all at speed one unless we need to share between five small jobs. But if the five longest small jobs have processing times equal to l, then $B_V \geq 5l/4 > l = B_{V^2+3l}$, a contradiction. □

This means that we just need to prove that when adding a job, the makespan of our remaining instance increases by at most 9/8 of the increase of the makespan of the remaining instance of optimum. To prove this we would probably need some sort of invariant on the lower bound values (something stronger than that they are all smaller than the goal time minus current time) that holds throughout the run of the algorithm. Unfortunately, we were not able to find such an invariant.

Notice that the Theorem 5.18 above works also with $\alpha = 1$, so it is an optimal algorithm for $\texttt{PSMJ}(m = 4, size_j \in \{1, 2\}, r_j = 0)$. It still uses the LP program for computing the optimal makespan but in the case without release times this could be replaced by computing the maximum of the five lower bound values, i.e., by using Theorem 3.2.

Conclusion

In this thesis we study the online version of PSMJ, i.e., preemptive scheduling of multiprocessor jobs. We focus primarily on its special cases when jobs have sizes only one or two and the number of processors is either a general constant m or $m = 4$.

After we define the notion of online algorithms and scheduling, we explore previous results in the area, with main focus on results directly relevant to this thesis.

We devote the next chapter to better understanding of optimal schedules. Focusing on the special cases of PSMJ, we derive a more comprehensible way for computing offline optimums without release times than are the previously known results. We also prove the existence of optimal schedules following some specific rules, the so called P-schedules. This helps with the goal of finding a combinatorial algorithm for computing offline optimum. Also we get an optimum of a very special form and if we decide to compare an online algorithm to this optimum we can use many of its properties. In this chapter we also generalize the result to online algorithms, showing that we can focus only on the algorithms creating P-schedules, without decreasing the optimal competitive ratio.

Then there is a chapter on lower bounds for PSMJ(m *const.*, $size_j \in \{1, 2\}$). First we show that for $m = 4$ we get a lower bound of $9/8$. Johannes [2006] claimed a lower bound of $6/5$ for this case but there was a mistake in the proof that we detect. We use practically the same example and derive the correct bound. We also generalize the lower bound for other m, getting a lower bound of $1 + 2/(3m + 4)$ for m even and $1 + 2/(5m + 8)$ for m odd. Then we use the same examples to derive new bounds, on the minimum speedup of 1-competitive algorithms in the v-speedup model. For $m = 4$, m even and m odd we receive bounds $8/7$, $1 + 2/(3m + 2)$ and $1 + 2/(5m + 2)$, respectively. We see that the lower bounds for m even and m odd differ significantly. We believe that at least when sizes of jobs are only one or two, these problems are significantly different. The following observations also point in this direction. For m even having one job of size one means that we will have idle time in the schedule, which is not true for m odd. On the other hand, for m odd we know that at least one job of size one will be scheduled at all times if it exists.

In the last chapter we provide several online algorithms, focusing especially on the problem PSMJ(m *const.*, $size_j \in \{1, 2\}$). We start with a 1-competitive algorithm in the $m/(m - 1)$-speedup model called LPTIS. This algorithm is based on the idea that at every release time we want our algorithm to have, in some sense, shorter jobs than any optimum. To achieve this, it uses the speedup to run the longest jobs, not caring about the sizes. This way, it can guarantee to fill only $m - 1$ processors out of m but because of the speedup, the amount of processed volume is as if it filled all m processors without speedup.

The following section describes an algorithm LPTIS+, an enhanced version of LPTIS for m even. It is a 1-competitive algorithm in the $(m + 1)/m$-speedup model. The key observation is that in the schedule of LPTIS we have unnecessarily enough space for long jobs. So we split the schedule into two parts. In the first part we schedule the jobs in such a way that all m processors are filled. In the

second part we schedule the longest jobs again. This way we can afford smaller speedup and still guarantee that enough volume is processed. Notice that for $m = 4$ this gives us a 5/4-competitive algorithm and so it puts the competitive ratio of PSMJ($m = 4, size_j \in \{1, 2\}$) somewhere in the interval $[9/8, 5/4]$.

The next presented algorithm is LPTISK and it is a generalization of LPTIS for problem PSMJ(m *const.*, $size_j \leq k$). It achieves the 1-competitiveness with $m/(m - k + 1)$-speedup. This means that for jobs with sizes bounded by a constant we approach a 1-competitive algorithm for $m \to \infty$.

Last presented algorithm, LPTDS, is an algorithm we conjecture matches the lower bound 9/8 for PSMJ($m = 4, size_j \in \{1, 2\}$). We provide an intuition why we think it is good and prove that if it is in a feasible state (the goal it is trying to reach can be met with the remaining instance of the algorithm) it will remain in a feasible state throughout the run of the algorithm, until the goal is met or until new jobs are released. So to prove this algorithm is 1-competitive in the 9/8-speedup, we only need to prove that it does not change its state to unfeasible when new jobs arrive.

For future work there are several directions to follow. First would be to determine whether LPTDS is really 9/8-competitive and then hopefully generalize it to other special cases of PSMJ. If we could generalize it to 9/8-competitive algorithm for $m \leq 7$ then we would have 9/8-competitive algorithm for PSMJ($size_j \in \{1, 2\}$) as for $m = 8$ we could use LPTIS+ and for $m > 8$ LPTIS to guarantee this ratio (these algorithms are stated for m constant but their time complexity is only polynomial in m, so we could really use them). Another possible direction would be to use the LPTIS schema in Algorithm 5.1 together with Theorem 5.5 to design algorithms for further special cases of PSMJ, for example when sizes of jobs are restricted to powers of two.

Bibliography

Jacek Błażewicz, Jan Weglarz, and Mieczyslaw Drabowski. Scheduling independent 2-processor tasks to minimize schedule length. *Information Processing Letters*, 18(5):267–273, 1984.

Jacek Błażewicz, Mieczyslaw Drabowski, and Jan Weglarz. Scheduling multiprocessor tasks to minimize schedule length. *IEEE Transactions on Computers*, 35(5):389–393, 1986.

Jacek Błażewicz, Maciej Drozdowski, Günter Schmidt, and Dominique De Werra. Scheduling independent two processor tasks on a uniform duo-processor system. *Discrete Applied Mathematics*, 28(1):11–20, 1990.

Jacek Błażewicz, Maciej Drozdowski, Günter Schmidt, and Dominique de Werra. Scheduling independent multiprocessor tasks on a uniform k-processor system. *Parallel Computing*, 20(1):15–28, 1994.

Maciej Drozdowski. On the complexity of multiprocessor task scheduling. *Bulletin of the Polish Academy of Sciences Technical Sciences*, 43(3):381–392, 1995.

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

Maciej Drozdowski. *Scheduling for parallel processing*. Springer, 2009.

Ronald L. Graham. Bounds for certain multiprocessing anomalies. *Bell System Technical Journal*, 45(9):1563–1581, 1966.

Kwang Soo Hong and Joseph Y.-T. Leung. On-line scheduling of real-time tasks. *IEEE transactions on Computers*, 41(10):1326–1331, 1992.

Klaus Jansen and Lorant Porkolab. Preemptive parallel task scheduling in o(n)+poly(m) time. In D. T. Lee and Shang-Hua Teng, editors, *Algorithms and Computation, 11th International Conference, ISAAC 2000, Proceedings*, volume 1969 of *Lecture Notes in Computer Science*, pages 398–409. Springer, 2000.

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

Joseph Y.-T. Leung. *Handbook of scheduling: algorithms, models, and performance analysis*. CRC Press, 2004.

Robert McNaughton. Scheduling with deadlines and loss functions. *Management Science*, 6(1):1–12, 1959.

Kirk Pruhs, Jiří Sgall, and Eric Torng. Online scheduling. In Joseph Y.-T. Leung, editor, *Handbook of scheduling: algorithms, models, and performance analysis*, chapter 15, pages 320–362. CRC Press, 2004.

Sartaj Sahni and Yookun Cho. Nearly on line scheduling of a uniform processor system with release times. *SIAM Journal on Computing*, 8(2):275–285, 1979.

Jiří Sgall. On-line scheduling. In Amos Fiat and Gerhard J. Woeginger, editors, *Online Algorithms, The State of the Art*, volume 1442 of *Lecture Notes in Computer Science*, pages 196–231. Springer, 1996.

Jiří Sgall and Gerhard J. Woeginger. Multiprocessor jobs, preemptive schedules, and one-competitive online algorithms. In Evripidis Bampis and Ola Svensson, editors, *Approximation and Online Algorithms – 12th International Workshop, WAOA 2014, Revised Selected Papers*, volume 8952 of *Lecture Notes in Computer Science*, pages 236–247. Springer, 2015.