A deterministic truthful PTAS for scheduling related machines

George Christodoulou[*] Annamária Kovács[†]

Abstract

Scheduling on related machines ($Q||C_{\max}$) is one of the most important problems in the field of Algorithmic Mechanism Design. Each machine is controlled by a selfish agent and her valuation function can be expressed via a single parameter, her *speed*. Archer and Tardos [3] showed that, in contrast to other similar problems, a (non-polynomial) allocation that minimizes the makespan can be truthfully implemented. On the other hand, if we leave out the game-theoretic issues, the complexity of the problem has been completely settled — the problem is strongly NP-hard, while there exists a PTAS [9, 8].

This problem is the most well-studied in single-parameter Algorithmic Mechanism Design. It gives an excellent ground to explore the boundary between truthfulness and efficient computation. Since the work of Archer and Tardos, quite a lot of deterministic and randomized mechanisms have been suggested. Recently, a breakthrough result [7] showed that a randomized, truthful-in-expectation PTAS exists. On the other hand, for the deterministic case, the best known approximation factor is 2.8 [10, 11].

It has been a major open question whether there exists a deterministic truthful PTAS, or whether truthfulness has an essential, negative impact on the computational complexity of the problem. In this paper we give a definitive answer to this important question by providing a truthful *deterministic* PTAS.

1 Introduction

Algorithmic Mechanism Design (AMD) is an area originated in the seminal paper by Nisan and Ronen [14, 15] and it has flourished during the last decade. It studies combinatorial optimization problems, where part of the input is controlled by selfish agents that are either unmotivated to report them correctly, or strongly motivated to report them erroneously, if a false report is profitable. In classical mechanism design more emphasis has been put on incentives issues, and less to computational aspects of the optimization problem at hand. On the other hand, traditional algorithm design disregards the fact that in some settings the agents might have incentive to lie. Therefore, we end up with algorithms that are fragile against selfish behavior. AMD carries challenges from both disciplines, aiming at the design of algorithms that optimize some global objective that, at the same time, make selfish users interested in reporting truthfully, and so are also immune to strategic behavior.

A fundamental optimization problem that has been suggested in [15] as a ground to explore the design of truthful mechanisms, is the *scheduling problem*, where a set of n tasks need to be processed by a set of m machines. There are two important variants with respect to the processing capabilities of the machines, that have been studied within the AMD framework. The machines

[*]University of Liverpool, Liverpool, U.K.. Email: gchristo@liv.ac.uk This author has been partially supported by DFG grant Kr 2332/1-3 within Emmy Noether Program and by EPSRC grant EP/F069502/1.

[†]Department of Informatics, Goethe University, Frankfurt M., Germany. Email: panni@cs.uni-frankfurt.de Research supported by the German Research Foundation (DFG) grant KO 4083/1-1.

can be *unrelated*, i.e., each machine i needs t_{ij} units of time to process task j; or *related*, where machine i comes with a speed s_i, while task j has processing requirement p_j, that is, $t_{ij} = p_j/s_i$ (we will use the settled notation $Q||C_{\max}$ to refer to the latter problem). The objective is to allocate the jobs to the machines so that the maximum finish time of the machines, i.e. the *makespan* is minimized.

In the game-theoretic setting, each machine i is owned by a rational agent who controls[1] the *private* values of row t_i. In order to motivate the machines to cooperate, we pay them to execute the tasks. It is assumed that each agent wants to maximize his utility, and without a proper incentive he will lie (report false t_i), if this can trick the algorithm to assign less work or higher payment to him. A *mechanism* consists of two parts: an *allocation algorithm* that assigns the tasks to the machines, and a *payment scheme* that compensates the machines in monetary terms. We are interested in devising *truthful* mechanisms in *dominant* strategies, where each agent maximizes his utility by telling the truth, regardless of the reports of the other agents.

The scheduling problem provides an excellent framework to study the computational aspects of truthfulness. It is a well-studied problem from the algorithmic perspective with a lot of algorithmic techniques that have been developed. Moreover, it is conceptually close to (additive) combinatorial auctions, so that solutions and insights can be transferred from the one problem to the other.[2] Indeed, the scheduling problem comes with a variety of objectives to be optimized, that are different than the objectives used in classical mechanism design.

The mechanism design version of scheduling on related machines was first studied by Archer and Tardos [3]. It is the most central and well-studied among single-parameter problems, where each player controls a single real value and his objective is proportional to this value (see also Ch. 9 and 12 of [13]). In particular, in the scheduling setting the *cost* of player i is $t_i \cdot W_i$, where $t_i = 1/s_i$ is his private value, and W_i is the total processing requirement (sum of the p_j) allocated to machine i. The *utility* (profit) that the agent tries to maximize is the payment he receives minus his cost.

Myerson [12] gave a characterization of truthful algorithms for one-parameter problems, in terms of a monotonicity condition. Archer and Tardos [3] found a similar monotonicity characterization, and they showed that a certain type of *optimal* allocation is monotone and consequently truthful (albeit exponential-time). A monotone allocation rule for related scheduling is defined as follows:

Definition 1. *An allocation rule for related machine scheduling is* monotone*, if increasing the input speed s_i of any single machine i, while leaving the other speeds unchanged, monotonically increases the workload W_i allocated to this machine, i.e., $s_i < s'_i \Rightarrow W_i \leq W'_i$.*

The fact that truthfulness does not exclude optimality, in contrast to the multi-parameter variant of scheduling (the unrelated case)[3], makes the problem an appropriate example to explore the interplay between truthfulness and computational complexity. It has been a major open problem

[1]Technically, this is not the case for the related machines setting, as the sizes of the jobs are beyond agents' control. In that case a machine can only control her own speed.

[2]Take for example the machines scheduling on unrelated machines, and a combinatorial auction setting with additive valuations. The main difference is that in the former case an agent is a *cost minimizer*, while in the latter case an agent is a *utility maximizer*. Similarly, in the former case we *pay* the agent to compensate her for processing the jobs, while in the latter case we *charge* the agent a price.

[3]With the scheduling on unrelated machines, we are more in the dark *(see [6] for a recent overview of results).* There are impossibility results that show that there does not exist any truthful mechanism with approximation ratio better than a constant *even in exponential time.* Therefore, more primitive questions need to be answered before we settle the complexity of the problem. The only known algorithm for the problem is the VCG that has approximation ratio equal to the number of machines. For the special, but natural case of anonymous mechanisms, Ashlagi, Dobzinski and Lavi [4] showed that the best possible mechanism is VCG.

whether or not a *deterministic* monotone PTAS exists for $Q||C_{\max}$[4]. In this work, we give a definitive positive answer to that central question and conclude the problem.

1.1 Related Work

Auletta et al. [5] gave the first deterministic polynomial-time monotone algorithm for any fixed number of machines, with approximation ratio 4. This result was improved to an FPTAS by Andelman, Azar, and Sorani [1]. For an arbitrary number of machines, the authors of [1] gave a 5-approximation deterministic truthful mechanism, and Kovács improved the approximation ratio to 3 [10] and to 2.8 [11], which was the previous record for the problem.

Randomization has been successfully applied. There are two major concepts of randomization for truthful mechanisms, *universal truthfulness*, and *truthfulness-in-expectation*. The first notion is strongest, and consists of randomized mechanisms that are probability distributions over deterministic truthful mechanisms. In the latter notion, by telling the truth a player maximizes his expected utility. Only the second notion of randomized truthfulness has been applied to the problem. Archer and Tardos [3] gave a truthful-in-expectation mechanism with approximation ratio 3, that was later improved to 2 [2]. Recently, Dhangwatnotai et al. [7], settled the status for the randomized version of the problem by giving a randomized PTAS that is truthful-in-expectation. Both mechanisms apply (among other methods) a randomized rounding procedure. Interestingly, randomization is useful only to guarantee truthfulness and has no implication on the approximation ratio. Indeed, both algorithms can be easily derandomized to provide deterministic mechanisms that preserve the approximation ratio, but violate the monotonicity condition.

1.2 Our results and techniques

We provide a deterministic monotone PTAS for $Q||C_{\max}$. The corresponding payment scheme [3] is polynomially computable, and with these payments our algorithm induces a $(1 + 3\epsilon)$-approximate deterministic truthful mechanism, settling the status of the problem. As opposed to [10], where a variant of the LPT heuristic was shown to be monotone, our algorithm is the first deterministic monotone adaptation of the known PTAS's [9, 8]. Next we describe the main ideas that led to this result. We assume that input speeds are indexed so that $s_1 \leq s_2 \leq \ldots \leq s_m$ holds. For any set of jobs $P = \{p_1, p_2, \ldots, p_j\}$, the *weight* or *workload* of the set is $|P| = \sum_{r=1}^{j} p_r$. We will view an allocation of the jobs to the machines as an *(ordered) partition* $(P_1, P_2, \ldots, P_m)$ of the jobs into m *sets*. We search for an output where the workloads $|P_i|$ are in increasing order.[5]

The PTAS in [8] – which is a simplified and polished version of the very first PTAS [9] – defines a directed network on $m + 1$ layers depending on the input job set, where each arc leading between the layers $i - 1$ and i represents a possible realization of the set P_i, and directed paths leading over the $m + 1$ layers correspond to the possible job partitions. An optimal solution is then found using a shortest path computation in this network, for minimizing the maximum edge-weight – i.e. finish time – over the path.

The difficulty in applying any known PTAS to construct a deterministic monotone algorithm for $Q||C_{\max}$ is twofold. First, in all of the known PTAS's, sets of input jobs of approximately the same size form groups, s.t. in the optimization process a common (rounded or smoothed) size is assumed for all members of the same group. Second, jobs that are tiny compared to the total workload of a

[4]We say that a mechanism runs in polynomial time when both the allocation algorithm and the payment algorithm run in polynomial time.

[5]We use the terms increasing (decreasing) in the sense of non-decreasing (non-increasing).

machine do not turn up individually in the calculations, but just as part of an arbitrarily divisible (e.g., in form of small blocks) total volume.

Note that it must be relatively easy to find an allocation procedure that is in a way 'approximately monotone'. However, (exact) monotonicity intuitively requires exact determination and knowledge of the allocated workloads. To justify this, we just point out that in every monotone (in expectation) algorithm for $Q||C_{\max}$ provided so far, the (expected) workloads either occur in increasing order wrt. increasing machine speeds, or constitute a lexicographically minimal optimal solution wrt. a fixed solution set over labeled machines.

Thus, both of the mentioned simplifications of the input set – which, to some extent, seem necessary to admit polynomial time optimization – appear to be condemned to destroy any attempt to make a deterministic adaptation monotone. (The authors of [7] used randomization at both points to obtain the monotone in expectation PTAS.) Our ideas to eliminate the above two sources of inaccuracy of the output are the following, respectively:

1. As for rounding the job sizes, note that grouping is necessary only to narrow the set of schedules considered. We can achieve basically the same restriction if for any group of jobs of similar size we fix the order of jobs in which they appear in the allocation (e.g., in increasing order), and calculate with the *exact* job sizes along the optimization process. Now, if reducing a machine speed increases the makespan of the (previously optimal) solution, that means that this machine became a bottleneck, so a new upper bound on the optimum makespan over the considered set of outputs is induced *exactly* by the (previous) workload of the changed machine (the same argument as used in [3, 1, 7]).

2. Concerning tiny jobs, we observe that with these we can fill up some of the fastest machines nearly to the makespan level. Further, it is easy to show [2] that pre-rounding machine speeds to powers of some predefined $(1+\epsilon)$ does not spoil monotonicity and increases the approximation bound only by a factor of $(1+\epsilon)$. Assuming now that the coarsity of tiny blocks is *much* finer than the coarsity of machine speeds, we can be sure that (full) machines of higher speed receive more work than slower machines. Moreover, having reduced the speed of such a machine, tiny jobs in its workload 'flow' to other machines to enable a new optimum makespan 'much' smaller than implied by the previous workload of this machine.

It is quite a technical challenge to combine these two ideas so smoothly that in the end yields a correct monotonicity proof. We optimize over a special form of schedules (see Figure 1): Each machine i has a set L_i of large (non-tiny) jobs, so that the $L_1, L_2, \dots, L_m$ have increasing and exactly known weights, and with a fix order of jobs within each size-category (see point 1 above). Also, each machine has a set S_i of small jobs (some of these are uniform tiny blocks, while some are known exactly). We note that the same job size may be large for a slower machine, while it is small on a faster machine. The total set of small jobs is flexible in the sense that we can always move a small job to a higher index machine, and obtain a valid schedule. We set the objectives so that in an optimum solution the small jobs are moved towards the higher index machines as much as possible, and fill them up to the makespan.

Our monotonicity proof becomes subtle in case of the first (and so, not necessarily full) machine containing small jobs. It is especially so when only one machine (say, m) has small jobs, not leaving space for manipulating the small jobs in the output as needed. In order to circumvent this problem we restrict the search to allocations where at least two machines do have some tiny blocks (unless too few tiny jobs exist). Moreover, it seems crucial that every machine has the possibility to get rid of all its tiny blocks (i.e., those inducing imprecise workload) if this is provoked by a reduction of its speed. Combining these two requirements we treat the last *three* machines as a single entity. A carefully optimized assignment of an 'obligatory' set of tiny blocks, and later of the actual tiny jobs to these machines then implies monotonicity.

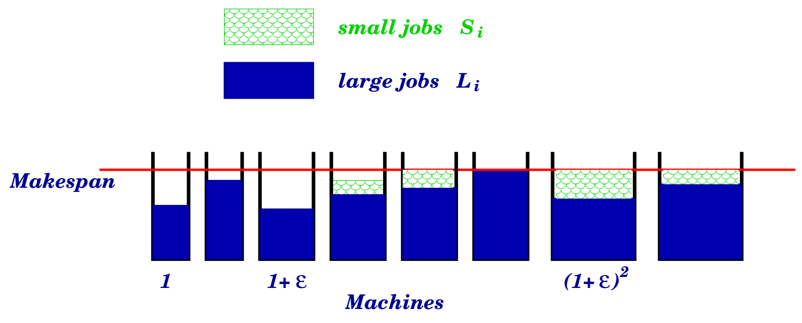

Figure 1: The type of schedule we look for

1.3 Roadmap

Following the Preliminaries, in Section 2 we introduce the special form of allocations (*canonical allocations*) that we consider, as delineated in the previous paragraphs. We show (Theorem 1) that for arbitrary input, a canonical allocation with near-optimal makespan exists. We proceed in Section 3 with describing a succinct (imprecise) representation of allocations of tasks to the machines called *configuration*. In Section 4 we build a digraph of configurations with m layers, where each path leading through all layers corresponds to a feasible allocation. We show (Theorem 2) that an optimal path of this graph represents a near-optimal schedule. In Section 5 we describe the algorithm PTAS, that essentially consists of two procedures: one finding an optimal path in the digraph, and another, distributing the jobs consistently with the path presentation. Finally, in Section 6 we show that our algorithm is monotone (Theorem 4), and therefore it can become truthful by using the appropriate payment scheme, that we discuss in Section 6.1.

1.4 Preliminaries

The input is given by a set P_I of n input jobs, and a vector s (or σ) of input speeds $s_1 \leq \ldots \leq s_m$. For a job $p \in P_I$ we use p both to denote the job itself, and the *size* of this job in a given formula.

For a desired approximation bound $1 + 3\epsilon$, we choose a $\delta \ll \epsilon$, that will be the rounding precision of the job sizes. For ease of exposition, we will assume that $(1+\delta)^t = 2$ for some $t \in \mathbb{N}$.[6] Furthermore, we define ρ as the unique integer power of 2 in $[\delta/6, \delta/3]$.

Definition 2 (job classes)**.** *If p denotes (the size of) a job, then $\overline{p}$ denotes this job rounded up to the nearest integral power of $(1+\delta)$. A job p is in the* job class C_l, *iff $\overline{p} = (1+\delta)^l$.*

Let $C_l = \{p_{l1}, p_{l2}, \ldots, p_{ln_l^{\max}}\}$ be the jobs of C_l in some fixed increasing order of size. We use the notation $C_l[a] = \{p_{l1}, \ldots, p_{la}\}$ for $0 \leq a \leq n_l^{\max}$, and $C_l(a,b] = C_l[b] \backslash C_l[a]$ for $0 \leq a \leq b \leq n_l^{\max}$.

If $P = \{p_1, p_2, \ldots, p_j\}$ is a job set, then $\overline{P} = \{\overline{p}_1, \overline{p}_2, \ldots, \overline{p}_j\}$ denotes the corresponding set of rounded jobs. The *weight* or *workload* of P is $|P| = \sum_{r=1}^{j} p_r$; the *rounded weight* is $|\overline{P}| = \sum_{r=1}^{j} \overline{p}_r$. Assuming that the jobs are in *decreasing* order of size, we denote the subset of the r largest jobs in P by $P^r = \{p_1, p_2, p_3, \ldots, p_r\}$.

We use the interval notation (e.g., $[1, m]$) for a set of consecutive machine indices.

All logarithms are base 2 unless otherwise specified.

[6]This simplifying assumption is unrealistic for computations, but it is not necessary for the result to hold. We could equally well use the rounding function of [8] or [7]. Also, since our result is of purely theoretical interest, we do not try to optimize the ratio δ/ϵ; it will be clear that, e.g., $30\delta < \epsilon$ suffices in the proofs.

2 Canonical allocations

This section characterizes the type of allocations – we call them *canonical* – that we will consider. Definitions 3 and 4 describe the necessary restrictions on the output job partition.

Definition 3 (δ-division). *We say that a set of jobs P is* δ-divided into the pair of sets (L, S) *if*

(D0) $P = L \cup S$ *and* $L \cap S = \emptyset$,

(D1) $\overline{p} > \frac{\delta \cdot |L|}{(1+\delta)^2}$ *for every* $p \in L$, *and*

(D2) $\overline{q} \leq \delta |L|$ *for every* $q \in S$.

The subsets L and S will be the *large* resp. the *small* jobs on a single machine. Note that we cannot set a sharp border between their sizes. For instance, having uniform jobs, $1/\delta$ of them will form L, while the rest (arbitrarily many) belong to S.

Definition 4 (canonical allocation). *For a given input, an allocation $P_1, P_2, \ldots, P_m$ is called* canonical*, if for every $i \in [1, m]$, the set P_i can be δ-divided into (L_i, S_i), so that the following hold:*

(A1) *if $i < i'$, then* $|L_i| \leq |L_{i'}|$;

(A2) *for jobs p and q of the same job class $p \leq q$ holds if and only if*

- *(a)* $p \in L_i$ *and* $q \in S_{i'}$ *for some* $i, i' \in [1, m]$, *or*
- *(b)* $p \in L_i$ *and* $q \in L_{i'}$ *and* $i \leq i'$, *or*
- *(c)* $p \in S_i$ *and* $q \in S_{i'}$ *and* $i \leq i'$.

In words, taking the jobs of any job-class in increasing order, they will get filled first into the large sets L_i by increasing machine speeds, followed by the small sets S_i, again by increasing machine speeds (wherever jobs of this class are intended in the solution).

As the main result of the section, Theorem 1 states that for any input, and any $\delta > 0$, a canonical allocation exists that provides $(1 + 3\delta)$-approximation to the optimum makespan. Before presenting the details, we sketch the proof here. We will modify an optimal partition of the *rounded input* jobs $\overline{P_I}$ to get the canonical allocation. First we take the *core* set of each set in the partition as defined by Definition 5. In the sense of Proposition 1, this yields a 'preliminary δ-division' of the sets. Next, we order the partition sets by increasing order *of core size.* It is easy to show that *small* jobs (those outside the cores) can be shifted towards fast machines, in order to 'fit below' the original makespan again. Finally, we apply Lemma 1 to make the cores (now with original job sizes) fulfill (A2) (b). After all, the cores and the small jobs induce a δ-division on each machine.

Definition 5 (core). *Given a set P of jobs, we define the* core $cr(P)$ *of P as follows. Consider the jobs $P = \{p_1, p_2, \ldots\}$ in a fixed decreasing order of size. Let j be minimum with the property that $\overline{p}_j \leq \frac{\delta}{1+\delta} |P^{j-1}|$, then $cr(P) \overset{\text{def}}{=} P^{j-1} = \{p_1, \ldots, p_{j-1}\}$. If no such j exists, then $cr(P) \overset{\text{def}}{=} P$.*

Proposition 1. *Let P be a set of jobs, then*

(d1) $\forall p \in cr(P) \quad \overline{p} > \frac{\delta}{(1+\delta)^2} |cr(P)|$;

(d2) $\forall q \in P \setminus cr(P) \quad \overline{q} \leq \frac{\delta}{1+\delta} |cr(P)|$.

Proof. Since job sizes are in decreasing order, (d2) holds by definition. Moreover, also by definition $\overline{p}_{j-1} > \frac{\delta}{(1+\delta)}|P^{j-2}|$. Therefore

$$(1+\delta)^2\overline{p}_{j-1} > (1+\delta)\overline{p}_{j-1} + \delta p_{j-1} > \delta(|P^{j-2}| + p_{j-1}) = \delta|P^{j-1}| = \delta|cr(P)|.$$

The same holds for all jobs of size at least p_{j-1}, which proves (d1). □

Lemma 1. *Let $(Q_1, \ldots, Q_m)$ be a partition of a subset Q of the input jobs such that $|\overline{Q}_1| \leq \ldots \leq |\overline{Q}_m|$. There exists a partition $(L_1, \ldots, L_m)$ of Q that satisfies*

(i) $|L_1| \leq \ldots \leq |L_m|$;

(ii) *for any job class C_l, if job p_{lj} belongs to L_i and job p_{lk} to $L_{i'}$ where $i < i'$, then $j < k$;*

(iii) *for all i*
$$\frac{1}{1+\delta}|\overline{Q}_i| < |L_i| \leq |\overline{Q}_i|.$$

Proof. Let $Q_i \subset R$. We say that we *maximize Q_i wrt. R*, if for every class l we replace the jobs in $Q_i \cap C_l$ by the largest possible jobs in $R \cap C_l$ (i.e., if there are r such jobs then with the r largest jobs of $R \cap C_l$). We will denote the maximized set by Q_i^R. Clearly, if $S \subset R$, then $|Q_i^S| \leq |Q_i^R|$.

Now we construct the new partition recursively. We define L_m as a set of maximum workload among $\{Q_1^Q, Q_2^Q, \ldots, Q_m^Q\}$ (notice that the latter contains subsets of Q but is not a partition of Q). Assuming that $L_m = Q_i^Q$, now L_{m-1} is defined to be a set of maximum workload among $\{Q_1^{Q\setminus L_m}, \ldots, Q_{i-1}^{Q\setminus L_m}, Q_{i+1}^{Q\setminus L_m} \ldots, Q_m^{Q\setminus L_m}\}$, etc. In every recursive step we selected a set that has larger weight than any other remaining set, even if those sets get the largest remaining jobs of the respective classes. This proves (i), whereas (ii) holds by construction.

Next we argue that (iii) holds as well. Observe that $\{\overline{Q}_1, \overline{Q}_2, \ldots, \overline{Q}_m\}$ and $\{\overline{L}_1, \overline{L}_2, \ldots, \overline{L}_m\}$ (as sets) are exactly the same. The proof of

$$\frac{1}{1+\delta}|\overline{Q}_i| < |L_i| \leq |\overline{Q}_i|$$

is simply the fact that there exist at least i jobsets among the $\overline{Q}_l$, so that $|\overline{Q}_l| < (1+\delta)|L_i|$, (namely, the sets of rounded jobs $\overline{L}_1, \overline{L}_2, \ldots, \overline{L}_i$), on the other hand there exist at least $m-i+1$ sets among the $\overline{Q}_l$, so that $|L_i| \leq |\overline{Q}_l|$ (namely, the sets $\overline{L}_i, \overline{L}_{i+1}, \ldots, \overline{L}_m$). □

Theorem 1. *For arbitrary increasing input speeds and input jobs, a canonical allocation inducing a schedule with makespan at most $(1+3\delta)OPT$ exists, where OPT is the optimum makespan.*

Proof. Let P_I be the set of all jobs and $s_1 \leq \ldots \leq s_m$ be the input speeds. We process this set of jobs in three steps to finally obtain the desired canonical allocation. In the first two steps we consider only the set of rounded jobs $\overline{P}_I$.

1. (core division) We start from an optimal schedule of $\overline{P}_I$. Let this be $(P_1, P_2, \ldots, P_m)$ and its makespan be $M \leq (1+\delta)OPT$. The inequality trivially holds, since any schedule of P_I induces a schedule of $\overline{P}_I$ of makespan increased by a factor of at most $(1+\delta)$.

Moreover, for every P_i let $S_i'' = P_i \setminus cr(P_i)$. In the rest of the proof we call jobs in $\bigcup_{i=1}^m cr(P_i)$ *large*, and jobs in $\bigcup_{i=1}^m S_i''$ *small.*

2. (core sorting) In this step we start from the schedule $(P_1, P_2, \ldots, P_m)$ with $P_i = cr(P_i)\dot\cup S_i''$ (disjoint union) and we result in a schedule $P_1', \ldots, P_m'$ with $P_i' = L_i'\dot\cup S_i'$, where $L_1', L_2', \ldots, L_m'$ is simply the set of cores $cr(P_i)$ sorted by weight.

We sort the *complete* P_i sets *by core weight* $|cr(P_i)|$. After that, we don't move large jobs, but take the small jobs in the same order as they are allocated now, starting from (small) jobs on machine $m, m-1, m-2, \ldots$ and fill them on the machines up to the time M, in decreasing order of machines. When a job reached or exceeded the finish time M, we continue on the next machine.

For the obtained (L_i', S_i') divisions, property (d1) of Proposition 1 trivially holds, since we did not change the large sets. Next we show that (d2) holds as well. We claim that every small job that was previously in S_i'', has been moved to a some core set L_h', with $|L_h'| \geq |cr(P_i)|$. This, in turn, implies (d2). Assume the contrary, that during redistribution, some small job is moved from a machine k to a lower index machine $h < k$. This, however, would mean that the total set of jobs on machines $[k, m]$ (after the sorting), does not fit on the fastest $m-k+1$ machines below finish time M, contradicting the fact that originally these were job sets on *some* $m-k+1$ machines, fitting below M. We claim that the resulting schedule has makespan at most $(1+\delta)M \leq (1+\delta)^2 OPT$. A folklore argument [8] shows that sorting the cores cannot increase the makespan *induced by the cores*, which is at most M. During redistribution, the last small job on each machine increased the makespan by a factor of at most $(1+\delta)$, due to (d2).

3. (permutation) Finally, we turn to the original jobsizes. Working from the fastest machine to the slowest, we replace each small rounded job by the largest unrounded job in its class that has not already been placed on a faster machine. Thus we obtain the sets S_i so that (A2) (c) holds. Then we replace the large rounded jobs in each L_i' by the remaining original jobs of the respective classes, and apply Lemma 1 on the resulting partition of large jobs, to obtain $L_1, \ldots, L_m$. Now (A1) and (A2) hold for the partition. Moreover, by Lemma 1 (iii), $\frac{1}{1+\delta}|L_i'| < |L_i| \leq |L_i'|$ holds for every i. This implies, on the one hand, that the makespan did not increase. On the other hand, we obtained δ-divisions (L_i, S_i) : to see (D1), note that the sets $\overline{L}_i$ are the permuted core sets L_i', so for any job p in L_i we have by (d1) that

$$\overline{p} > \frac{\delta}{(1+\delta)^2}|\overline{L}_i| \geq \frac{\delta}{(1+\delta)^2}|L_i|;$$

(D2) holds, because if $q \in S_i$ then $\overline{q} \in S_i'$, and by (d2)

$$\overline{q} \leq \frac{\delta}{(1+\delta)}|L_i'| < \delta|L_i|.$$

□

3 Configurations

Like in [8, 7], we introduce so called *configurations* $\alpha(w, \mu, \vec{n}^o, \vec{n}^1)$ in order to represent any possible job set P_i of the partition, up to δ accuracy. The μ and w determine which jobs are considered small and which are considered large (resp. considered at all) in the delta-division of P_i. The $\vec{n}^o$ describes a set of jobs in this scope, while $\vec{n}^1$ describes a superset of those jobs, P_i being roughly the difference of the two sets.

We use the configurations to define the vertices of a directed graph $\mathcal{H}_I$ over m layers. An optimal path over all layers will then determine our output schedule: if a path goes through a configuration at layer i, it means that machine i gets the job set of this configuration.[7]

The first component of any configuration is a *magnitude* w which is an integer power of 2. As we proceed from slow machines to fast machines in a schedule, the monotonically increasing magnitude

[7]Roughly speaking, our graph can be thought of as the *line graph* of the graph G defined in [8] (with simple modifications). That is, the vertices of $\mathcal{H}$ correspond to edges of G.

keeps track of the largest job size allocated so far, which must be some size in the interval $(w/2, w]$. Thus, the current magnitude indicates which (larger) job sizes are not yet relevant, and which (tiny) jobs need not be taken into account *individually* anymore in the configuration (note that here we exploit that the $|L_i|$ must be increasing, cf. [8]). This motivates the next definition.

Definition 6 (valid magnitude)**.** *The value* $w = 2^z$ $(z \in \mathbb{Z})$ *is a* valid magnitude *if an input job* $p \in P_I$ *exists so that* $w/2 < p \leq w$. *Let* $w_{\min}$ *and* $w_{\max}$ *denote the smallest and the largest valid magnitudes, respectively. We call a job* tiny for w *if it has size at most* ρw.

Recall that ρ is the integer power of 2 between $\delta/6$, and $\delta/3$. Having a magnitude w fixed, let $\lambda = \log_{(1+\delta)} \rho w = t \cdot \log(\rho w)$, and $\Lambda = \log_{(1+\delta)} w = t \cdot \log w$, where $(1+\delta)^t = 2$. Notice that both λ and Λ are integers, and by Definition 2, the jobs of size in $(\rho w, w]$ belong to the classes $C_{\lambda+1}, \ldots, C_\Lambda$. These will constitute the relevant job classes, if the largest jobsize on the current or slower machines is between $w/2$ and w. The integer parameter $\mu \in (\lambda, \Lambda]$ is the index of the job class on the border between large and small jobs, as we discuss later.

If the configuration α represents the set P_i in a job partition, then the so-called *size vector* $\vec{n}^o = (n^o_\lambda, n^o_{\lambda+1}, \ldots, n^o_\Lambda)$ describes the jobs in the cumulative job set $A_{i-1} \overset{\text{def}}{=} \bigcup_{h=1}^{i-1} P_h$ as follows. For $\lambda < l \leq \Lambda$, $(l \neq \mu, \mu+1)$, exactly the first (smallest) n^o_l jobs of the class C_l are in the set A_{i-1}. The meaning of n^o_λ is that in A_{i-1} the total weight of jobs from $\bigcup_{l \leq \lambda} C_l$ is in the interval $((n^o_\lambda - 1) \cdot \rho w, (n^o_\lambda + 1) \cdot \rho w)$. However, the particular subset of these small jobs inside A_{i-1}, is not determined by α. The vector $\vec{n}^1$ represents the set $A_i \overset{\text{def}}{=} \bigcup_{h=1}^{i} P_h$, analogously.

A major difference to the configurations of [8], is that our configurations should not only represent a job set P_i, but also its δ-division (L_i, S_i). In particular, we will distinguish four types of job sizes in a configuration. *Tiny* jobs have size at most ρw, and, as already seen, are represented by the first coordinates n_λ of the two size vectors with their total size rounded to an integer multiple of ρw. Correspondingly, we will talk about *blocks* of size ρw which are simply re-tailored tiny jobs so as to make our procedure efficient.

Definition 7 (block)**.** Blocks *are imaginary tiny jobs, each having size* ρw *for some valid magnitude* w. *We use* $T(n_\lambda, \rho w)$ *to denote a set of* n_λ *blocks of size* ρw.

Small jobs are those that (together with the tiny jobs), can only appear in the set S_i of the δ-division, whereas *large* jobs can only be in the set L_i. However, there exist job classes – we will call them *middle* size jobs –, which might occur in both L_i and S_i, since by Definition 3 there is a flexible boundary between the job sizes in L_i and in S_i. Therefore, exactly two job classes, μ and $\mu + 1$ will be represented by a *triple* of (increasing) non-negative integers instead of scalar values in both of the vectors $\vec{n}^o, \vec{n}^1$. In the case of $\underline{n}^o_\mu = (n^o_{\mu\ell}, n^o_{\mu m}, n^o_{\mu s})$, the meaning of the three numbers will be that in the set A_{i-1}, from the class C_μ exactly the jobs in $C_\mu(n^o_{\mu m}, n^o_{\mu s}]$ are allocated as small jobs, that is, to one of the sets $S_1, S_2, \ldots, S_{i-1}$, and exactly the jobs in $C_\mu[n^o_{\mu\ell}]$ as large jobs, i.e., in one of $L_1, \ldots, L_{i-1}$, and similarly in case of $\vec{n}^1_\mu$ for the set A_i. The meaning of the numbers for $\mu+1$ is analogous. We summarize the above description in the next, somewhat technical definitions.

Definition 8 (size vector)**.** *A* size vector $\vec{n} = (n_\lambda, \ldots, n_\Lambda)$ *with* middle size $\mu \in [\lambda+1, \Lambda]$, *is a vector of integers, except for the entries* $\underline{n}_\mu = (n_{\mu\ell}, n_{\mu m}, n_{\mu s})$ *and* $\underline{n}_{\mu+1} = (n_{(\mu+1)\ell}, n_{(\mu+1)m}, n_{(\mu+1)s})$ *both of which consist of three integers, so that* $n_{\mu\ell} \leq n_{\mu m} \leq n_{\mu s}$, *and* $n_{(\mu+1)\ell} \leq n_{(\mu+1)m} \leq n_{(\mu+1)s}$ *holds. All integer entries belong to* $[0, n]$.

Definition 9 (configuration)**.** *A* configuration $\alpha(w, \mu, \vec{n}^o, \vec{n}^1)$ *consists of four components: a valid magnitude* w, *and two size vectors* $\vec{n}^o = (n^o_\lambda, \ldots, n^o_\Lambda)$, *and* $\vec{n}^1 = (n^1_\lambda, \ldots, n^1_\Lambda)$ *with* middle size $\mu \in [\lambda+1, \Lambda]$, *such that*

(C1) *$n^o_l \le n^1_l \le n^{\max}_l$ for $\lambda < l \le \Lambda$, $l \notin \{\mu, \mu+1\}$;*

(C2) *if $w \ne w_{\min}$ then $n^1_l > 0$ for at least one $l \in (\Lambda - t, \Lambda]$;*

(C3) *$n^o_\lambda \le n^1_\lambda \le \max\{n^o_\lambda, \left\lfloor \frac{\sum_{l \leqslant \lambda} |C_l|}{\rho w} \right\rfloor - 1\}$;*

(C4) *$n^o_{\mu\ell} \le n^1_{\mu\ell} \le n^o_{\mu m} = n^1_{\mu m} \le n^o_{\mu s} \le n^1_{\mu s} \le n^{\max}_\mu$, and analogously for $\mu + 1$;*

$$T_\alpha \overset{\text{def}}{=} T(n^1_\lambda - n^o_\lambda, \rho w).$$

$$S_\alpha \overset{\text{def}}{=} T_\alpha \cup C_\mu(n^o_{\mu s}, n^1_{\mu s}] \cup C_{(\mu+1)}(n^o_{(\mu+1)s}, n^1_{(\mu+1)s}] \cup \bigcup_{l=\lambda+1}^{\mu-1} C_l(n^o_l, n^1_l], \text{ and}$$

$$L_\alpha \overset{\text{def}}{=} C_\mu(n^o_{\mu\ell}, n^1_{\mu\ell}] \cup C_{(\mu+1)}(n^o_{(\mu+1)\ell}, n^1_{(\mu+1)\ell}] \cup \bigcup_{l=\mu+2}^{\Lambda} C_l(n^o_l, n^1_l].$$

T_α, S_α, and L_α are the tiny, small, and large jobs (respectively) in configuration α.

(C5) *either $\vec{n}^o \ne \vec{n}^1$, and $(1+\delta)^{(\mu+1)} \le \delta \cdot |L_\alpha| < (1+\delta)^{(\mu+2)}$;*
or α is the empty configuration *$(w_{\min}, \lambda_{\min} + 1, \vec{0}, \vec{0})$ where $\lambda_{\min} = t \cdot \log(\rho w_{\min})$.*

Clearly, (C1), (C3), and (C4) reflect how $\vec{n}^o$ and $\vec{n}^1$ represent the cumulative job-sets; (C2) implies that w is always the smallest possible magnitude for representing these job-sets.

(C5) is different in flavor from the previous properties: it establishes the relation between the weight of L_α and of the job sizes at the boundary of L_α, and S_α. In particular, observe that the inequalities of (C5) impose (D1) and (D2) of Definition 3. Moreover, since $\lambda \le \mu - 1$, for the tiny blocks (of size $\rho w = (1+\delta)^\lambda$) (C5) implies the strict inequality

$$\rho w < \delta |L_\alpha|. \tag{1}$$

Notation. *We refer to the whole represented job set $L_\alpha \cup S_\alpha$ simply by α (abusing notation), and $|\alpha|$ stands for the total work of the set α. Note that α is a virtual jobset, since S_α also includes tiny blocks. By property (C5), for every non-empty configuration α, the sets (L_α, S_α) form a δ-division of the set α. We denote the set without tiny blocks by $\widetilde{\alpha} = \alpha \setminus T_\alpha$.*

4 The directed graph $\mathcal{H}_I$

In this section, for arbitrary input instance I, we define a directed, layered graph $\mathcal{H}_I$. All vertices of this graph are configurations, labeled and 'chained' to form the graph in a suitable way.

First, for an arbitrary configuration α, we define a set of configurations called $Scale(\alpha)$. These are the possible configurations of an end-vertex of any arc starting at a vertex labelled by α. In particular, if $\alpha = (w, \mu, \vec{n}^o, \vec{n}^1)$, $\beta = (w', \mu', \vec{n}'^o, \vec{n}'^1)$, and $\beta \in Scale(\alpha)$, then $\vec{n}'^o$ must represent the same job set A_i, as $\vec{n}^1$, from the point of view of a (possibly) increased magnitude w' and a (possibly) increased middle size μ'. Whenever $w' > w$, this involves collecting in A_i the jobs of classes that become tiny, and the tiny blocks, and forming the proper number of new bigger tiny blocks out of this job set. Since all magnitudes are powers of 2, the size ρw of tiny blocks either remains unchanged, or at least doubles as we proceed to faster machines. This prevents that rounding errors in the total (tiny) job size cumulate to an unpredictable error (see also [8]).

Furthermore, the middle size μ is allowed to increase if A_i already contains all large jobs in the class μ, that is, $n^1_{\mu\ell} = n^1_{\mu m}$. The exact definition of $Scale(\alpha)$ can be found in Appendix A.

The vertices of $\mathcal{H}_I$ are arranged in m *layers*, and in *levels* I and II, which are orthogonal to the layers. The configurations on level I must have an empty set of small jobs, i.e., $S_\alpha = \emptyset$, and here the layers $\{m-2, m-1, m\}$ are empty. Level II has $m-1$ 'real' layers, and we add a single dummy vertex v_m adjacent to every vertex on layer $m-1$, that alone forms the last layer m.[8] In general, the ith layer stands for the ith set P_i. Any directed path of m nodes leads to v_m over the m layers, and from level I (or II) to level II. Such a path we call an *m-path*. The m-paths represent partitions of the input P_I. For a given m-path, the very first vertex on level II is in some layer $k \leq m-2$; we will call it the *switch vertex*, and k the *switch machine*. (Note that k is thus the first machine possibly receiving small jobs.) We denote the vertex-sets of the two levels by V_I, and V_{II}, respectively.

Notation. *For any given directed path* $(v_1, v_2, \ldots v_r)$, *the corresponding configurations of the nodes will be denoted by* $(\alpha_1, \alpha_2, \ldots, \alpha_r)$.

In what follows, we consider the last three sets P_{m-2}, P_{m-1}, P_m of the partition. Note first that for any m-path the last configuration α_{m-1} alone represents $\bigcup_{h=1}^{m-1} P_h$. Thus, we can use α_{m-1} to uniquely determine the 'hidden configuration' α_m (not appearing explicitly in the path). We define α_m to have $w_m = w_{m-1}$, $\mu_m = \mu_{m-1}$, and all jobs not allocated in $\bigcup_{h=1}^{m-1} P_h$. In particular, note that α_m includes *all* jobs of class higher than Λ_{m-1}, and that this job set can be handled as a single huge chunk without violating the running time bounds. For technical reasons we overestimate the amount of tiny blocks on m and set $n^1_\lambda = \left\lceil \frac{\sum_{l \leqslant \lambda} |C_l|}{\rho w} \right\rceil + 3$. (We relax on (C3) for $m-1$ and m, and allow this bound on n^1_λ.) The definition of other size vector entries in α_m is obvious.

Furthermore, our monotonicity argument requires that even the last *three* workloads $\alpha_{m-2}, \alpha_{m-1}$, and α_m be dealt with as a single entity. Among other restrictions, we will demand that either all of them have the same magnitude, and therefore use $w'_{m-1} = w_{m-2}$ instead[9] of w_{m-1}, or that w_{m-2} is *much* smaller than w_{m-1}, so that all jobs on $m-2$ (if they exist), are tiny for machines $m-1$ and m. Correspondingly, α_{m-2} and α_{m-1} of any path must adhere to either type (X) or (Y) as specified next. Observe that in case (X) on the last three machines, resp. in case (Y) on the last two machines the size of tiny blocks is the same (i.e. well-defined):

(X) $w_{m-2} > \rho^2 \cdot w_{m-1}$; then we modify the last magnitude to be $w'_{m-1} := w_{m-2}$, and require

- (i) either *all* tiny jobs (at most 18 blocks) are on machines $m-1$ and m, or
- (ii) machines $\{m-2, m-1, m\}$ have at least 18 tiny blocks, and at least two of them have each at least 6 tiny blocks.

(Y) $w_{m-2} \leq \rho^2 \cdot w_{m-1}$; then

- (i) all machines but $\{m-1, m\}$ are empty, or
- (ii) $m-1$ and m together have at least 6 tiny blocks.

[8]More precisely, we will unite layers $m-2$ and $m-1$, and use *double vertices* in the united layer, but it simplifies the discussion to think of these as pairs of individual vertices.

[9]The exposition may look sloppy here; a rigorous definition of *double configurations* would have been *direct*, e.g., using common magnitude w_{m-2} and a size vector of triple length in case (X). The words "instead" and "modify" clearly reflect the development of our construction. We hope this will prove rather reader-friendly than confusing.

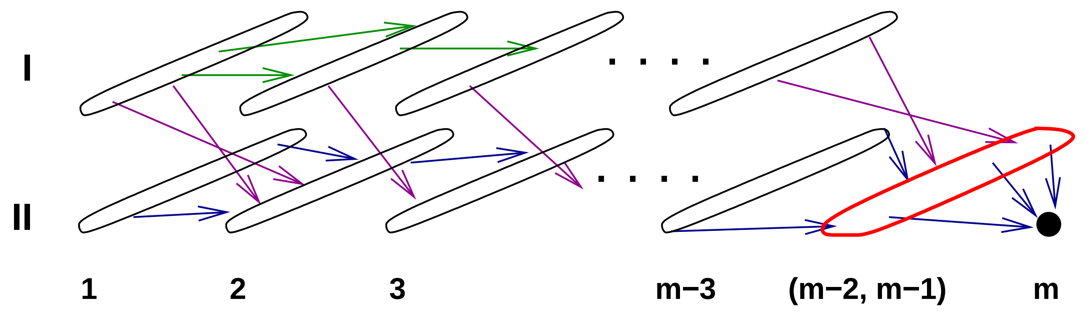

Figure 2: The graph $\mathcal{H}_I$. Level I consists of $m-3$ layers, level II has m layers. Arcs lead between consecutive layers within level I, within level II, and from level I to level II. Layers $m-2$ and $m-1$ are combined and contain the double vertices.

The requirements (X) and (Y) can be included in the graph definition, e.g., by using only special *double vertices* v^D_{m-2} with *double configurations* $(\alpha_{m-2}, \alpha_{m-1})$ in the layers $(m-2, m-1)$ on level II. Applying $w'_{m-1} := w_{m-2} > \rho^2 \cdot w_{m-1}$ can be done by using size vectors of triple length for the double vertices of type (X). Clearly, all restrictions can be represented by the double configurations, and we need polynomially many of them.

The following definition of graph $\mathcal{H}_I$ is independent of the speed vector s, and depends only on the job set P_I. We assume, w.l.o.g. that $m \geq 3$, otherwise we add a machine of speed (close to) 0.

Definition 10 (graph $\mathcal{H}_I$)**.** *$\mathcal{H}_I(V, E)$ is a directed graph, where every vertex $v \neq v_m$ is a triple $v = (d, i, \alpha)$, so that $d \in \{I, II\}$ is a level, i is a layer, and $\alpha = (w, \mu, \vec{n}^o, \vec{n}^1)$ is a configuration. Triples that fulfill $i \in [1, m-3]$, and (V1)-(V2), are vertices, and those fulfilling (V3) are double-vertices.*

(V1) *if $i = 1$, then for $l \neq \mu, \mu+1$ $n^o_l = 0$, while for $l \in \{\mu, \mu+1\}$ $n^o_{l\ell} = 0$ and $n^o_{lm} = n^o_{ls}$;*

(V2) *if $d = I$, then $S_\alpha = \emptyset$.*

There is an arc from $v = (d, i, \alpha)$ to $v' = (d', i+1, \beta)$ if an only if

(E1) *$\beta \in Scale(\alpha)$, and*

(E2) *$|L_\alpha| \leq |L_\beta|$, and*

(E3) *$d \leq d'$;*

(V3) *for $d = II$, and combined layers $i = (m-2, m-1)$, include* double vertices *v^D with* double configurations *$(\alpha_{m-2}, \alpha_{m-1})$ so that for $(\alpha_{m-2}, \alpha_{m-1}, \alpha_m)$ the requirements (E1) (E2) and either (X) or (Y) hold. Moreover, $|\widetilde{\alpha}_{m-2}| \leq |\widetilde{\alpha}_{m-1}| \leq |\widetilde{\alpha}_m|$, and $|\alpha_{m-2}| \leq |\alpha_{m-1}| \leq |\alpha_m|$.*

Next, to each vertex we assign a weight called *finish time*, and define the *makespan* of a path accordingly. These values do depend on the machine speeds s. Observe that machines having tiny (that is, non-exact) jobs use an upper-bound as finish time, except for the switch machine.

Definition 11 (finish time of a vertex)**.** *Let $v = (d, i, \alpha)$ be a vertex of $\mathcal{H}_I$, where $\alpha = (w, \mu, \vec{n}^o, \vec{n}^1)$. The* finish time *of v is then $f(v) = \frac{|\alpha|+\rho w}{s_i}$ if α does have tiny blocks ($n^o_\lambda < n^1_\lambda$), and $f(v) = \frac{|\alpha|}{s_i}$ otherwise.*

Definition 12 (makespan of a path)**.** *Let $\mathcal{Q} = (v_1, v_2, \ldots, v_r)$ be a directed path in $\mathcal{H}_I$. If $\mathcal{Q} \subset V_I$, or $\mathcal{Q} \subset V_{II}$, then the* makespan *of $\mathcal{Q}$ is $M(\mathcal{Q}) = \max_{h=1}^{r} f(v_h)$. If $\mathcal{Q}$ is an m-path with switch vertex $v_k = (II, k, \alpha_k)$, then $M(\mathcal{Q}) = \max\{\frac{|\alpha_k|}{s_k}, \max_{h \neq k} f(v_h)\}$.*

The following theorem, saying that an m-path having makespan close to the optimum makespan of the scheduling problem always exists, is a consequence of Theorem 1. The proof is straightforward, and requires a technical translation of real schedules to m-paths in $\mathcal{H}_I$. The reader can find the complete proof in Appendix B.

Theorem 2. *For every input $I = (P_I, s)$ of the scheduling problem, the optimal makespan over all m-paths in $\mathcal{H}_I$ is at most $OPT \cdot (1 + \mathcal{O}(\delta))$, where OPT denotes the optimum makespan of the scheduling problem.*

5 The deterministic algorithm

This section describes the deterministic monotone algorithm, in form of two procedures and the main algorithm Ptas. We will make use of an arbitrary fixed total order $\prec$ over the set of all configurations, such that configurations of smaller total workload are smaller according to $\prec$.

Among the m-paths of $\mathcal{H}_I$, the algorithm selects an m-path having *minimum* makespan, as the primary objective. Among paths of minimum makespan, we maximize the index of the switch machine k. A further order of preference, is to be of type (X), then (Y). This selection of the optimal path is done by Procedure OptPath (see Figure 3). This procedure is a common dynamic programming algorithm that finds an m-path of minimum makespan in $\mathcal{H}_I$. However, we do not simply proceed from left to right over the m graph layers, but select an optimal prefix path from the first layer to every node in V_I, and similarly, an optimal suffix path from layer m to each node in V_{II}. For all $v \in V_I$ and $v \in V_{II}$, the pointers *pred*() and *succ*() determine an incoming, respectively an outgoing path of minimum makespan. When the makespan of two prefix (or suffix) paths is the same, we break ties according to $\prec$. For every $v \in V_I$ and $v \in V_{II}$, the respective optimum values are recorded by *opt*(v).

Finally, we test each vertex in V_{II} to be a potential switch vertex (i.e., we find optimal paths leading to the switch vertex from both end-layers). The optimal path makespan, assuming some $v \in V_{II}$ as switch vertex, is recorded in $M(v)$. We choose a switch vertex v_k providing optimum makespan, and of maximum possible k.

The case $k = m - 2$ needs careful optimization. Among solutions of minimum makespan, we choose deterministically by some fixed order of the double configurations $(\alpha_{m-2}, \alpha_{m-1})$, except for the tiny blocks that are always distributed so as to minimize the (local) makespan of the last three machines. The flexibility provided by three machines with 'many' tiny blocks, facilitates a monotone allocation in this degenerate case as well.

Proposition 2. *If $\mathcal{Q}$ is the m-path output by* OptPath *with configurations $(\alpha_1, \ldots, \alpha_m)$, in the path vertices, and switch machine $k \neq m - 2$, then $|\alpha_i| \geq (1 - 4\delta) \cdot M(\mathcal{Q}) \cdot s_i$ for all $i > k$.*

Proof. Assume the contrary, that $|\alpha_i| < (1 - 4\delta) \cdot M \cdot s_i$, where $M = M(\mathcal{Q})$. By optimality of the switch machine k, $S_{\alpha_k} \neq \emptyset$. We claim that we can 'put' a small job $p \in S_{\alpha_k}$ from k to i by modifying the path $\mathcal{Q}$. Suppose that we put one job from class $l \geq \lambda$ from machine k to $k + 1$, where $\alpha(w, \mu, \vec{n}^o, \vec{n})$, and $\beta(w', \mu', \vec{n}', \vec{n}^1)$ with smallest job-classes λ and λ' are the configurations of v_k and v_{k+1}, respectively. In α we reduce n_l by 1. As for β, if $l > \lambda'$, then we reduce n'_l by 1 as well, and we are done. If $l \leq \lambda'$, then we scale the reduced $\vec{n}$ size vector according to (S5) in order to obtain the new $n'_{\lambda'}$. In this way, $n'_{\lambda'}$ either reduces by 1, or keeps its original value. In the

Procedure 1 OptPath

Input: The directed graph $\mathcal{H}_I$ and speeds $s_1 \leq s_2 \leq \ldots \leq s_m$.
Output: The optimal m-path $\mathcal{Q} = (v_1, \ldots, v_m)$ of $\mathcal{H}_I$.

```
1. for every double vertex $v^D_{m-2} = (\alpha_{m-2}, \alpha_{m-1}) \in V_{II}$ do
   $opt(v^D_{m-2}) := \max\{f(v_{m-2}), f(v_{m-1}), f(v_m)\}$   (where $\alpha_{m-1}$ determines $\alpha_m$, see Section 4)
   $M(v^D_{m-2}) := \max\{\frac{|\alpha_{m-2}|}{s_{m-2}}, \frac{|\alpha_{m-1}|}{s_{m-1}}, \frac{|\alpha_m|}{s_m}\}$;
   for $i = m-3$ downto 1 do
      for every $v = (d, i, \alpha) \in V_{II}$ do
    (i) $succ(v) := w$, if $opt(w) = \min\{opt(y) \mid (v, y) \in E\}$, and among such vertices of minimum
        $opt()$ the configuration $\alpha$ of vertex $w$ is $\prec$-minimal;
   (ii) $opt(v) := \max\{f(v), opt(succ(v))\}$;
        $M(v) := \max\{\frac{|\alpha|}{s_i}, opt(succ(v))\}$.

2. for every $v = (d, 1, \alpha) \in V_I$ do   $opt(v) := f(v)$;
   for $i = 2$ to $m-2$ do
      for every $v = (d, i, \alpha) \in V_I \cup V_{II}$ do
    (i) $pred(v) := w \in V_I$, if $opt(w) = \min\{opt(y) \mid y \in V_I, (y, v) \in E\}$, and among such
        vertices of minimum $opt()$ the configuration $\alpha$ of vertex $w$ is $\prec$-minimal;
   (ii) if $v \in V_I$ then $opt(v) := \max\{f(v), opt(pred(v))\}$;
        if $v \in V_{II}$ then $M(v) := \max\{M(v), opt(pred(v))\}$.

3. select an optimal switch vertex $v_k = (II, k, \alpha_k) \in V_{II}$, by the following objectives:
    (i) $M(v_k) = \min\{M(v) \mid v \in V_{II}\}$;
   (ii) the layer $k$ is maximum over all $v$ of minimum $M(v)$;
  (iii) if $k < m-2$ then the configuration $\alpha_k$ is minimal wrt. $\prec$ over all $v$ of minimum $M(v)$
        in layer $k$;
   (iv) if $k = m-2$, then among all double vertices $v^D_{m-2} = (\alpha_{m-2}, \alpha_{m-1})$ of minimum $M(v^D_{m-2})$
        select $v_k = v^D_{m-2}$ by the following objectives:
        (a) keep the order (X) (Y) and then (i) (ii) (cf. Section 4);
        (b) in case of (X), select an $(\widetilde{\alpha}_{m-2}, \widetilde{\alpha}_{m-1}, \widetilde{\alpha}_m, (T_{\alpha_{m-2}} \cup T_{\alpha_{m-1}} \cup T_{\alpha_m}))$ (i.e., with a common
        pool of tiny blocks) by some predefined ordering, then – by redistributing the tiny blocks
        – minimize also the second highest finish time of $\frac{|\alpha_{m-2}|}{s_{m-2}}$, $\frac{|\alpha_{m-1}|}{s_{m-1}}$, and $\frac{|\alpha_m|}{s_m}$ (as $M(v^D_{m-2})$
        had been minimized before);
        (c) in case of (Y), select an $(\alpha_{m-2}, \alpha_{m-1}, \alpha_m)$ by some predefined ordering for which
        $|\alpha_{m-1}| + |\alpha_m|$ is maximized;

4. for $i = k-1$ downto 1 do   $v_i := pred(v_{i+1})$;
   for $i = k+1$ to $m$ do   $v_i := succ(v_{i-1})$;
   $\mathcal{Q} := (v_1, \ldots, v_k, \ldots, v_m)$.
```

Figure 3: Procedure OptPath finds an m-path $\mathcal{Q}$ of minimum $M(\mathcal{Q})$ in the directed graph.

latter case we can say that T_β *swallowed* the job. In order to put a job from k to $i > k$, we can repeatedly apply the above changes to the configurations, until the job gets swallowed, or we arrive at machine i. These changes decrease $|\alpha_k|$, and increase $|\alpha_i|$ by (at most) one small job, that is, by at most $\delta|L_{\alpha_i}| \le \delta|\alpha_i|$. (If $i = m-2$ and so $|\alpha_{m-2}|$ increased, then we may have to increase $|\alpha_{m-1}|$, and $|\alpha_m|$, so as to keep them nondecreasing; that is why we need 4δ instead of 2δ in the claim.)

The workload of machines between k and i did not increase, because jobs in class C_l get shifted to the right. Altogether, we either found a smaller (by $\prec$) α_k, or a valid path with switch vertex $k+1$ if no small jobs remained on k, without having increased the makespan, so $\mathcal{Q}$ was not optimal. $\square$

Once an optimal m-path is found, we have to allocate the jobs of P_I to the machines. This is obvious for jobs that appear individually in some configuration of the path, but we need to define how the *tiny* jobs are distributed, given the block representation. Procedure Partition is depicted in Figure 4. Importantly, depending on whether the switch machine k is filled high (close to the makespan) or low, it gets filled with tiny jobs below, resp. above $|\alpha_k|$.

Distributing the tiny jobs for $k = m-2$ (see 3b. in Partition) is a slightly more subtle procedure, operating with the same principle (a *low* machine is filled over $|\alpha_i|$). Note also that here $|Q_{m-2}| \le |Q_{m-1}| \le |Q_m|$ can be achieved, because both the $\widetilde{\alpha}_i$ and the α_i are nondecreasing, and the total number of tiny blocks is overestimated by 3 blocks in α_m. Further, if the set Q_i is increased because of 3b (i) or (ii), then i still gets much less work than any higher index machines, so the $|Q_i|$ remain increasing.

The output of Partition is, indeed, a partition of P_I, and the induced allocation $(Q_1, \dots, Q_m)$ is canonical with the choice $L_i := L_{\alpha_i}$. This is clear, since the configurations represent δ-divisions, and (A1) and (A2) are imposed by the construction. Finally, the upper bound in (C3) on the number of tiny blocks in any configuration, and the rules of rounding in (S5) imply[10] that there are enough jobs in T from the classes $l \le \lambda$ to fulfill 3a (ii). In other words, the tiny jobs allocated to any machine i have size at most ρw_i, as intended.

The next technical proposition describes essential properties of the output job-partition.

Proposition 3. *If $\mathcal{Q} = (v_1, \dots, v_m)$ is the input path, then for the output $Q_1, \dots, Q_m$ of* Partition $\frac{|\alpha_i| - 8\rho w_i}{s_i} \le \frac{|Q_i|}{s_i} \le M(\mathcal{Q})$ *for every machine i.*

Proof. For $i < k$, α_i contains no tiny jobs, and $Q_i = \alpha_i$. So, in this case $\frac{|Q_i|}{s_i} = \frac{|\alpha_i|}{s_i} = f(v_i) \le M(\mathcal{Q})$.

In Partition, the variable Π_i stands for the total size of tiny blocks in α_i.

If $k < m-2$, then the machine $i = k$ receives an amount of tiny jobs of total size between $\Pi_k - \rho w_k$ and $\Pi_k + \rho w_k$, and not more than Π_k in case of *high-k*. The last machine having nonzero tiny blocks ($m-1$ or m) receives at most Π_i, and at least $\Pi_i - 6\rho w_i$ work (due to the estimate $n_\lambda^1 = \left\lceil \frac{\sum_{l \leqslant \lambda} |C_l|}{\rho w} \right\rceil + 3$ in the hidden configuration α_m, which is a bound on the total rounding error). The other machines $i > k$ get tiny jobs of total size in $[\Pi_i - \rho w_i, \Pi_i + \rho w_i]$. The $+\rho w_i$ overhead was calculated in these machines' finish times $f(v)$, and therefore in the makespan $M(\mathcal{Q})$.

Finally, in case $k = m-2$, only a low machine $i \in \{m-2, m-1, m\}$ may receive more work than $|\alpha_i|$; and each machine receives at least $\Pi_i - 8\rho w_i$ work of tiny jobs, even after the corrections done in 3b (i) or (ii). $\square$

The monotone Ptas is presented in Figure 5. A substantial property of the output is that on machines $i < k$, sets $Q_i = P_i$ are determined exactly, and have increasing workloads. On the other

[10]Note that Π denotes the total size of tiny *blocks* in $\bigcup_{h=1}^{i} \alpha_h$ in 3a (i). Let τ denote the total size of *jobs* tiny for w_i in $\bigcup_{h=1}^{i-1} \widetilde{\alpha}_h$. By (S5) $\Pi + \tau < n_\lambda^1 \cdot \rho w_i + \rho w_i$; by (C3) $n_\lambda^1 \cdot \rho w_i \le \sum_{l \leqslant \lambda} |C_l| - \rho w_i$. So, $\Pi + \tau < \sum_{l \leqslant \lambda} |C_l|$.

Procedure 2 Partition

Input: The job set P_I, and an m-path $\mathcal{Q} = (v_1, \ldots, v_m)$ with switch vertex v_k in the graph $\mathcal{H}_I$.
Output: A partition $Q_1, Q_2, \ldots, Q_m$ of the set P_I.

```
1.  for $i = 1$ to $m$ do
        $Q_i := \widetilde{\alpha}_i$;
2.  let $T = \{t_1, t_2, t_3, \ldots, t_\mu\} = P_I \setminus \bigcup_{i=1}^m Q_i$, so that the jobs $t_j$ are in increasing order;
3a. if $k < m - 2$ then
        Case *low-k* : $|\alpha_k|/s_k \le (1 - \epsilon/2) \cdot M(\mathcal{Q})$
        Case *high-k* : $|\alpha_k|/s_k > (1 - \epsilon/2) \cdot M(\mathcal{Q})$.
        let $\Pi = 0$ and $r = 0$;
        for $i = k$ to $m - 1$ do
            given $\alpha_i = (w, \mu, \vec{n}^o, \vec{n}^1)$ and $\lambda = \log \rho w$, let $\Pi_i := (n^1_\lambda - n^o_\lambda) \cdot \rho w$;
          (i)   $\Pi := \Pi + \Pi_i$;
          (ii)  if *high-k* then let $u$ be the maximum index in $T$ so that $\sum_{j=1}^u t_j \le \Pi$;
                if *low-k* then let $u$ be the minimum index in $T$ so that $\sum_{j=1}^u t_j \ge \Pi$;
          (iii) $Q_i := Q_i \cup \{t_{r+1}, t_{r+2}, \ldots, t_u\}$;
          (iv)  $r := u$.
        $Q_m := Q_m \cup \{t_{r+1}, t_{r+2}, \ldots, t_\mu\}$.
3b. if $k = m - 2$ then
        start with an allocation of tiny jobs to $\{m-2, m-1, m\}$ (in the given order) s. t. each machine
        $i$ gets at most $|T_{\alpha_i}|$ amount of tiny jobs, and the $|Q_{m-2}|, |Q_{m-1}|, |Q_m|$ are nondecreasing.
        let $M = \max\{\frac{|\alpha_{m-2}|}{s_{m-2}}, \frac{|\alpha_{m-1}|}{s_{m-1}}, \frac{|\alpha_m|}{s_m}\}$;
        Call $i \in \{m-2, m-1, m\}$ *low*, if $|\alpha_i|/s_i \le (1 - 2\epsilon/3) \cdot M$, and *high* if $|\alpha_i|/s_i \ge (1 - \epsilon/2) \cdot M$;
        correct the partition of tiny jobs (with keeping the job order) so that
        (i) if there is one low machine $i$, and two high machines, then $i$ receives at least $|\alpha_i|$ work;
        (ii) if there are two non-high machines, then both receive at least $6\rho w$ work of tiny jobs.
```

Figure 4: Procedure Partition allocates the jobs based on path $\mathcal{Q}$ output by OptPath.

Algorithm 3 Ptas

```
Input: machine speeds $\sigma_1 \leq \sigma_2 \leq \ldots \leq \sigma_m$, and job set $P_I = \{p_1, p_2, \ldots, p_n\}$, desired precision $\epsilon$.
Output: A partition $P_1, P_2, \ldots, P_m$ of $P_I$.

1. for each $i \in [1, m]$, round the speed $\sigma_i$ up to the nearest power of $(1 + \epsilon)$;
2. based on the jobs $P_I$, and an appropriate $\delta \ll \epsilon$, construct the graph $\mathcal{H}_I$;
3. with $\mathcal{H}_I$ and the rounded speeds $s_1 \leq s_2 \leq \ldots \leq s_m$, run Procedure OptPath in order to
   obtain the optimal $m$-path $\mathcal{Q} = (v_1, \ldots, v_m)$;
4. from $\mathcal{Q}$ compute the partition $Q_1, Q_2, \ldots, Q_m$ by Procedure Partition;
5. let $P_1, P_2, \ldots, P_m$ be the sets of $\{Q_1, Q_2, \ldots, Q_m\}$, sorted by increasing order of weight $|Q_i|$;
   output $P_1, P_2, \ldots, P_m$.
```

Figure 5: The deterministic monotone Ptas.

hand, the machines $i > k$ have finish time close to the makespan $M(\mathcal{Q})$. The characterization of the final output allocation $P_1, \ldots, P_m$ is summarized by the next lemma.

Lemma 2. *Let P_I denote the set of input jobs and s be the vector of rounded speeds. Let $\mathcal{Q} = (v_1, \ldots, v_m)$ be the path output by* OptPath *with configurations $(\alpha_1, \ldots, \alpha_m)$ in the vertices, and $v_k = (II, k, \alpha_k)$ be the switch vertex. In* Ptas *the sets Q_i get permuted only among machines of equal rounded speed. In particular, if $P_1, \ldots, P_m$ is the partition output by* Ptas*, then*

(a) for $i < k$, $P_i = Q_i = \alpha_i$.

Moreover, if $k < m - 2$, then

(b) for $i > k$, $(1 - 12\delta) \cdot M(\mathcal{Q}) \leq \frac{|P_i|}{s_i} \leq M(\mathcal{Q})$;

(c) for k either $P_k = Q_k$ or (b) holds.

Proof. Point (a) is obvious, since $S_{\alpha_i} = \emptyset$ for every vertex $v_i \in V_I$, and by step 1. of Partition, $Q_i = \alpha_i = L_{\alpha_i}$. Moreover, by (E2) L_{α_i} has the ith smallest weight, so $Q_i = P_i$.

For $i > k$ we have by Proposition 3 that $M(\mathcal{Q}) \cdot s_i \geq |Q_i| \geq |\alpha_i| - 8\rho w_i$; by inequality (1) that $\rho w_i < \delta|\alpha_i|$; and by Proposition 2 that $|\alpha_i| \geq (1 - 4\delta) \cdot M(\mathcal{Q}) \cdot s_i$. Therefore, $M(\mathcal{Q}) \cdot s_i \geq |Q_i| \geq |\alpha_i| - 8\rho w_i > |\alpha_i|(1 - 8\delta) > (1 - 12\delta) \cdot M(\mathcal{Q}) \cdot s_i$. This proves (b) for the $|Q_i|$, that is, before ordering the workloads. Furthermore, since $\delta \ll \epsilon$, this implies that the Q_i get permuted only among machines of equal s_i which, in turn, proves (b) also for the permuted workloads $|P_i|$.

If $|Q_i| \geq (1 - 12\delta) \cdot M(\mathcal{Q}) \cdot s_i$ holds for $i = k$, then (b) holds for $i = k$ as well by the previous argument; otherwise Q_k has the kth smallest weight, so $P_k = Q_k$. □

Since the Q_i get permuted only if $k < m - 2$ (and only among machines of the same speed), Proposition 3 and Lemma 2 imply $|P_i|/s_i \leq M(\mathcal{Q})$ in every case; i.e., the makespan $M(\mathcal{Q})$ of the output *path* is always an upper bound on the makespan of the output *schedule.*

We conclude the section with proving the approximation and the running-time bound.

Theorem 3. *For arbitrary input I, and any given $0 < \epsilon \leq 1$, the deterministic algorithm* Ptas *outputs a $(1+3\epsilon)$-approximate optimal allocation in time $Poly(n,m)$.*

Proof. By Theorem 2, for $\delta \ll \epsilon$ the optimal path $\mathcal{Q}$ in $\mathcal{H}_I$ has makespan $M(\mathcal{Q}) < (1+\epsilon)OPT$, and by Lemma 2 this remains an upper bound for the makespan of the output. Since Ptas rounds the input speeds to integral powers of $(1+\epsilon)$, we obtain an overall approximation factor of at most $(1+\epsilon)^2 \leq (1+3\epsilon)$.

In order to prove the running time bound, we show that for constant ϵ, step 2. of Ptas can be computed efficiently. The number of graph vertices $|V|$ is $\mathcal{O}(m \cdot C)$, where C denotes the number of different configurations. Every configuration (including the double configurations of triple length) is determined by $\mathcal{O}(\log_{(1+\delta)} 1/\rho)$ coordinates (where $\rho = \theta(\delta)$). Since close to zero $\log(1+\delta) = \theta(\delta)$ holds, this means $\mathcal{O}((1/\delta) \cdot \log 1/\delta)$ coordinates, each of which (including μ) can be assumed to take at most $n+1$ different values, so $|C| = n^{\mathcal{O}((1/\delta)\cdot \log 1/\delta)}$. Finally, for any $v, v' \in V$ deciding whether $(v, v') \in E$, takes time linear in the number of these coordinates. Thus for fixed δ, step 2. is *poly*(n,m), and all other steps obviously take polynomial time. □

6 Monotonicity and payments

A high-level formulation of the monotonicity argument is the following. Since the workloads $|P_i|$ of the output partition are increasing, it is easy to see that it suffices to prove monotonicity in the special case when a single rounded speed $s_i = (1+\epsilon)$ is reduced to $s'_i = 1$. Let the output path of OptPath be $\mathcal{Q}$ in the first, and $\mathcal{Q}'$ in the second case, moreover let f and M denote finish times and makespan for input (s_i, s_{-i}) and f' and M' for input (s'_i, s_{-i}), respectively.

The makespan of any (sub)path in $\mathcal{H}_I$ cannot decrease by reducing a machine speed. The objectives concerning the optimal path are defined so that $\mathcal{Q}' \neq \mathcal{Q}$ may occur only if machine i becomes a bottleneck machine, either concerning the whole path (meaning $M(\mathcal{Q}) < M'(\mathcal{Q}) = f'(v_i)$), or some relevant subpath (e.g., prefix path, or the subpath (v_{m-2}, v_{m-1}, v_m)). In any of these cases, $\mathcal{Q}$ is a possible solution of (local) makespan $f'(v_i)$, and $\mathcal{Q}'$ is preferred only in case the respective (sub)path of $\mathcal{Q}'$ has no higher (local) makespan than $f'(v_i)$, implying that i gets not more workload than $f'(v_i) = (1+\epsilon)f(v_i)$. This proves monotonicity if $(1+\epsilon)f(v_i)$ is the exact workload (e.g., if $i < k$), or at least a lower bound on what i received with the original speed.

On the other hand, if $(1+\epsilon)f(v_i)$ was just a δ-estimate, then the machine was close to full with input s_i, and, by reallocating 'many' or all tiny blocks, the new makespan (no matter how the output path looks like!) becomes 'much' smaller than $f(v_i)(1+\epsilon)$.

Next we present the theorem with complete proof.

Theorem 4. *Algorithm* Ptas *is monotone.*

Proof. Assume that machine i alone decreased its speed σ_i to σ'_i in the input. If the vector of rounded speeds $(s_1, s_2, \ldots, s_m)$ remains the same, then the deterministic Ptas outputs the same allocation, and i receives the same, or smaller workload, since the output workloads are in increasing order. Assuming that the rounded speed s_i decreased as well, it is enough to consider the special case when i is the first (smallest index) machine of rounded speed $s_i = (1+\epsilon)$ in input $I(P, s)$, and after reducing its speed, it becomes the last (highest index) machine of rounded speed 1 in input $I'(P, s')$. Since the workloads in the final allocation $P_1, \ldots, P_m$ are ordered, this implies monotonicity for every 'one-step' speed change (like $(1+\epsilon) \to 1$). Monotonicity in general can then be obtained by applying such steps repeatedly.

Note that for both inputs the algorithm constructs the same graph, independently of the speed vector. We assume that with inputs I, and I' OptPath outputs $\mathcal{Q} = (v_1, \ldots, v_m)$, and $\mathcal{Q}' =$

$(v'_1, \ldots, v'_m)$, where the switch vertices have index k and k', respectively. Finish time, makespan, etc. wrt. the *new* speed vector s' are denoted by $f'(), M'()$ etc. We prove that $|P_i| \geq |P'_i|$.

We start with a simple observation. Since we decreased a machine speed, it follows from the definition of makespan that for any path $\mathcal{R} = (v_1, v_2, \ldots, v_r)$ within level V_I, or within level V_{II}, and for any m-path, $M'(\mathcal{R}) \geq M(\mathcal{R})$. Similarly, for any vertex v, $opt'(v) \geq opt(v)$, and for any $v \in V_{II}$ $M'(v) \geq M(v)$ (cf. Procedure OptPath). Obviously, also the optimum makespan over all m-paths could not decrease.

CASE 1. $i < k$;

In this case, $\alpha_i = P_i$ by Lemma 2 (a), so the machine receives exactly $|\alpha_i|$ work with speed s_i.

If $M'(\mathcal{Q}) > M(\mathcal{Q})$, that means that machine i with the new speed $s'_i = 1$, becomes a bottleneck in path $\mathcal{Q}$, that is, $M'(\mathcal{Q}) = f'(v_i) = \frac{|\alpha_i|}{1}$. Thus, $\mathcal{Q}$ is a path with makespan $|\alpha_i|$, and by the optimality of $\mathcal{Q}'$ we obtain $|P'_i| = |P'_i|/s'_i \leq M'(\mathcal{Q}') \leq M'(\mathcal{Q}) = |\alpha_i| = |P_i|$.

If $M'(\mathcal{Q}) = M(\mathcal{Q})$, and $\mathcal{Q}' = \mathcal{Q}$, then $P'_i = \alpha'_i = \alpha_i = P_i$. Suppose $M'(\mathcal{Q}) = M(\mathcal{Q})$, but $\mathcal{Q} \neq \mathcal{Q}'$. Since the optimum makespan could not decrease by decreasing a speed, $\mathcal{Q}$ remains an optimal path for the new speeds as well, that is, $M'(\mathcal{Q}') = M'(\mathcal{Q})$. We claim that also $k' = k$. Assume for contradiction that $k' > k$, then $\mathcal{Q}'$ would have been preferred to $\mathcal{Q}$ for input s as well, because $M(\mathcal{Q}') \leq M'(\mathcal{Q}') = M'(\mathcal{Q}) = M(\mathcal{Q})$. In turn, also $v'_k = v_k$ (meaning double vertices if $k = m-2$) otherwise $\alpha' \prec \alpha$ would hold, and $\mathcal{Q}'$ would have been preferred for input s as well.

Now, since $\mathcal{Q} \neq \mathcal{Q}'$, a maximum $h < k$ exists so that $v_h \neq v'_h$. This means that $pred(v_{h+1}) \neq pred'(v_{h+1})$. Recall that $opt(v_h)$ is the optimum makespan over all paths leading to v_h from layer 1. If v'_h was preferred to v_h in $\mathcal{Q}'$ because $\alpha'_h \prec \alpha_h$, then $opt(v_h) < opt(v'_h) \leq opt'(v'_h) \leq opt'(v_h)$. The first inequality holds, otherwise $v'_h = pred(v_{h+1})$ would have been the choice for input s. The second holds for every vertex. The third holds, otherwise $v_h = pred'(v_{h+1})$ would have been the choice of OptPath for input s'. Similarly, if v'_h was preferred in $\mathcal{Q}'$ because $opt'(v'_h) < opt'(v_h)$, then $opt(v_h) \leq opt(v'_h) \leq opt'(v'_h) < opt'(v_h)$. In both cases we obtained $opt(v_h) < opt'(v_h)$. This value could strictly increase only if $i \leq h$, and i with workload $\alpha_i = P_i$ and speed $s'_i = 1$ became a bottleneck machine in $(v_1, v_2, \ldots, v_h)$. Therefore, $|P'_i| \leq opt'(v'_h) \leq opt'(v_h) \leq \frac{|\alpha_i|}{s'_i} = |P_i|$.

CASE 2. $k < m-2$ and $i > k$;

According to Lemma 2 (b), with speed $s_i = 1 + \epsilon$, machine i has a workload of at least $|P_i| \geq (1 - 12\delta) \cdot M(\mathcal{Q}) \cdot (1 + \epsilon)$. Note that $|P_i| \geq |L_{\alpha_i}|$ also holds, since $|Q_h| \geq |L_{\alpha_i}|$ for *all* $h \geq i$, and this remains true for the sorted sets P_h for $h \geq i$.

We modify the path $\mathcal{Q}$ and construct a new path $\mathcal{Q}''$ where, with input speed $s'_i = 1$, machine i is a bottleneck (just like in path $\mathcal{Q}$), and that has a makespan $M'(\mathcal{Q}'')$ of at most $|P_i|$. Therefore, the *optimal* path makespan $M'(\mathcal{Q}')$ is also at most $|P_i|$. Since the optimal path makespan is an upper bound on the makespan of the output schedule, this will prove $|P'_i| = |P'_i|/s'_i \leq M'(\mathcal{Q}') \leq |P_i|$.

We construct $\mathcal{Q}''$ as follows. Assuming $i < m$, we 'put' small jobs from S_{α_i} (tiny or non-tiny) onto machine $i+1$ until either the moved jobs have a total weight of at least $13\delta \cdot M(\mathcal{Q}) \cdot (1 + \epsilon)$, or S_{α_i} becomes empty. In the latter case the new finish time of i is at most $|L_{\alpha_i}|/s'_i = |L_{\alpha_i}| \leq |P_i|$; in the former case (using also (1) in Section 3) it is at most $M(\mathcal{Q}) \cdot (1 + \epsilon) \cdot (1 - 13\delta) + \rho w_i \leq M(\mathcal{Q}) \cdot (1 + \epsilon) \cdot (1 - 12\delta) \leq |P_i|$. If $i = m$, then we put only *tiny* blocks of the common magnitude w_{m-1} onto $m-1$, and use $|\widetilde{\alpha}_m|$ instead of $|L_{\alpha_i}|$ in the calculation. The job moving procedure is the same as described in Proposition 2.

It remains to show that the new finish time is, indeed, the makespan of $\mathcal{Q}''$, i.e., machine i is a bottleneck in $\mathcal{Q}''$. By Proposition 2 and Proposition 3, $|Q_i| = (1 - \mathcal{O}(\delta)) \cdot M(\mathcal{Q}) \cdot (1 + \epsilon)$, and from this workload we removed total jobsize of $\mathcal{O}(\delta) \cdot M(\mathcal{Q})$. Since $\epsilon \gg \delta$, the (work and) finish time of i is still at least $M(\mathcal{Q})(1 + 13\delta)$, which is an upper bound on all other machines' finish time.

CASE 3. $k < m-2$ and $i = k$;

By Lemma 2 (c), for k either $P_k = Q_k$ or $(1-12\delta)\cdot M(\mathcal{Q}) \leq \frac{|P_i|}{s_i}$ holds. In the latter case, we can include the case $i = k$ in CASE 2 above.

Assume that $i = k$ and $P_k = Q_k$. If *high-k* holds for $\mathcal{Q}$, meaning $|\alpha_k| > (1-\epsilon/2)\cdot M(\mathcal{Q})\cdot(1+\epsilon) > (1+\epsilon/3)\cdot M(\mathcal{Q})$, then the proof is analogous to CASE 2: from Partition it is clear that $|P_k| = |Q_k| \geq \max\{|\widetilde{\alpha}_k|, |\alpha_k| - \rho w_k\}$. By putting one tiny block onto the next machine (if there are any blocks), we still obtain a path $\mathcal{Q}''$, with makespan $M'(\mathcal{Q}'') = |\alpha''_k| = \max\{|\widetilde{\alpha}_k|, |\alpha_k| - \rho w_k\} > M(\mathcal{Q})$, and we are done.

If *low-k* is the case, then with input s_k the machine receives at least $|\alpha_k|$ work in Partition. If machine k still becomes a bottleneck, that is, $M'(\mathcal{Q}) > M(\mathcal{Q})$, then the new makespan $|\alpha_k|/s'_k = |\alpha_k|$ is an upper bound on the new optimum makespan and therewith on the new workload $|P'_k|$.

Finally, if the makespan of $\mathcal{Q}$ does not increase, then $\mathcal{Q}$ remains the output path: assuming the contrary that some $\mathcal{Q}' \neq \mathcal{Q}$ would be preferred by any of the optimization criteria, this would have been the case also with the original input speeds. It follows now from the Partition procedure that k cannot get more workload (only less if *low-k* changes to *high-k*).

CASE 4. $k = m-2$ and $i \geq k$

It follows from the definition (V3) of double vertices, and of Partition, that in this case the $|Q_i|$ are nondecreasing, so no permutation in step 5. of Ptas takes place. We show $|P'_i| \leq |Q_i|$.

In cases (X)(i) and (Y)(i) (concerning the path $\mathcal{Q}$), all tiny jobs – i.e., of size at most ρw_{m-2} or ρw_{m-1}, respectively – are on the last two machines. Importantly, these cases are trivially recognizable from the double configuration v^D_{m-2}. Moreover, the tiny jobs can be allocated to machines $\{m-1, m\}$ by any fixed rule (say, m gets all tiny jobs), and the makespan can be computed *exactly* during the path optimization. It can also be assumed that $|\alpha_{m-2}| \leq |\alpha_{m-1}| \leq |\alpha_m|$. The proof is therefore analogous to that of CASE 1.

Next, suppose that (X)(ii) holds for path $\mathcal{Q}$. Let $M = \max\{\frac{|\alpha_{m-2}|}{s_{m-2}}, \frac{|\alpha_{m-1}|}{s_{m-1}}, \frac{|\alpha_m|}{s_m}\}$. Assume first, that machine $i \geq m-2$ with speed s_i was a non-low machine in 3b of Partition, meaning $|\alpha_i| > (1-2\epsilon/3)\cdot M\cdot(1+\epsilon)$. Then, with speed $s'_i = 1$ machine i becomes a bottleneck in the subpath (v_{m-2}, v_{m-1}, v_m) (and possibly in the path $\mathcal{Q}$). Moreover, due to $\delta \ll \epsilon$, machine i is still a bottleneck in a modified (sub)path $\mathcal{Q}''$, where we put 8 tiny blocks, or *all* tiny blocks from i to the other two of the last three machines (obeying the sorted order and the restrictions (X)(ii)). That is, the makespan of $\mathcal{Q}''$ (resp. the local makespan on $(m-2, m-1, m)$) is $\max\{|\widetilde{\alpha}_i|, |\alpha_i| - 8\rho w_i\}$, which is an upper bound on $|P'_i|$, and a lower bound on $|Q_i|$, by Proposition 3.

Assume that i was a low machine in 3b of Partition.

If there was another non-high machine i' then only i and i' have tiny blocks in $\mathcal{Q}$ (cf. 3 (iv)(b) of OptPath). Having changed the speed to s'_i, we construct a path $\mathcal{Q}''$ by putting tiny blocks (when necessary) from i to i' so that their maximum finish time is minimized. If, with the optimized tiny blocks, the local makespan remains M then $\mathcal{Q}''$ is the new output solution. (Any path preferred to $\mathcal{Q}''$ would have been preferred with speed s_i as well.) Obviously, in Partition i receives a subset of the tiny jobs that it received with speed s_i. If in $\mathcal{Q}''$ the local makespan M increases, then i becomes a (local) bottleneck so it has at most 6 blocks in $\mathcal{Q}''$ (any further block would be put on i'). Thus, $|\widetilde{\alpha}_i| + 6\rho w_i$ is an upper bound on the new local makespan and also on $|P'_i|$; whereas, by Partition 3b (ii) it is a lower bound on $|P_i|$.

Suppose that i was a low machine with speed s_i, and the other two were both high machines. Then i received all but 6 tiny blocks (see 3 (iv) of OptPath), whereas $|Q_i| \geq |\alpha_i|$ (see 3b (i) of Partition). Consider now the same path $\mathcal{Q}$ with speed s'_i. If i becomes a (local) bottleneck, then $|\alpha_i|$ is an upper bound on the optimal (local) makespan of $\mathcal{Q}'$, so that $|P'_i| \leq |\alpha_i|$, and we are done. If i has the second highest finish time among $\{m-2, m-1, m\}$, then the output path might change only by (possibly) optimizing the second finish time by removing blocks from i; in this case i is not a low machine anymore, and $|P'_i| \leq |\alpha_i|$. If i has still the lowest finish time, then the output path is the same, and in PARTITION i gets the same set, or a subset of his previous jobs in case he becomes a non-low machine.

Finally, we turn to the case when for path $\mathcal{Q}$ (Y) (ii) holds. We claim that by the optimality of $\mathcal{Q}$, machines $m-1$ and m have finish time of at least $(1-6\delta)M(\mathcal{Q})$. Moreover, the allocation of PARTITION is essentially the same as in the case $k < m-2$ (with $m-2$ being a *low* or a *non-low* machine), and the monotonicity proof is analogous.

In the rest of the proof, we show that the claim holds. Assume that $m-1$ or m are not filled to at least $(1-6\delta)\cdot M(\mathcal{Q})$. In case (Y) (ii), OPTPATH maximizes $|\alpha_{m-1}| + |\alpha_m|$. If machine $m-2$ had at least one small job, then we could shift a small job from $m-2$ to the last two machines, increasing $|\alpha_{m-1}| + |\alpha_m|$, without increasing $M(\mathcal{Q})$, a contradiction. Assume that $m-2$ has no small jobs, that is, $\alpha_{m-2} = L_{m-2}$. We show that the *whole* job set L_{m-2} has size of at most ρw_{m-1}. By definition of case (Y), $\rho^2 w_{m-1} > w_{m-2}$. Since w_{m-2} is the maximum job size in L_{m-2}, and by property (D2) of δ-divisions, we have $w_{m-2} > \frac{\delta|L_{m-2}|}{(1+\delta)^2} > \rho|L_{m-2}|$. The latter inequality follows from $\rho < \delta/3$ for small δ. Putting it together, we obtain $\rho^2 w_{m-1} > \rho|L_{m-2}|$. We define a path $\mathcal{Q}''$ as follows. We 'put' the whole L_{m-2} to $m-1$ or to m in form of at most one tiny block of size $\rho \cdot w_{m-1}$ as imposed by (S5) (see Appendix A). Also, we shift every workload $\alpha_i = L_i$ to the next machine for machines $i < m-2$. The new 'partition' remains canonical, and we found a path better than the optimum $\mathcal{Q}$, a contradiction. □

6.1 Computing the payments

In order to get a truthful mechanism, we extend the monotone algorithm with the payment scheme, as defined in [12, 3]. For every machine the formula for the payment $\mathcal{P}_i$, as a function of the reported speeds and the allocation, is essentially uniquely determined[11]: it is composed of the *cost* ('reported' finish time) of i and of the integral of the *work curve* on the interval $[\frac{1}{s_i}, \infty)$. The work curve is the workload assigned to machine i by PTAS, as a function of his reported inverse speed, for fixed reported speeds of the other machines. This is a step-function, having breaks at integer powers of $(1+\epsilon)$, and at the reported inverse speeds of the other $m-1$ machines. Moreover, the curve vanishes when $\frac{1}{s_i} \geq \frac{n \cdot p_{\max}}{p_{\min} \cdot \hat{s}_m}$, where $\hat{s}_m$ stands for the maximum speed of the *other* machines – that is, i gets no work if he reports a very low speed; cf. also [3, 1, 7]. The calculation requires running the algorithm PTAS $\mathcal{O}(\log_{(1+\epsilon)} \frac{n \cdot p_{\max} \cdot s_i}{p_{\min} \cdot s_m})$ number of times which is polynomial in the input size.

References

[1] Nir Andelman, Yossi Azar, and Motti Sorani. Truthful approximation mechanisms for scheduling selfish related machines. *Theory of Computing Systems*, 40(4):423–436, 2007.

[2] Aaron Archer. *Mechanisms for Discrete Optimization with Rational Agents.* PhD thesis, Cornell University, January 2004.

[3] Aaron Archer and Éva Tardos. Truthful mechanisms for one-parameter agents. In *42nd Annual Symposium on Foundations of Computer Science (FOCS)*, pages 482–491, 2001.

[11] I.e., if we demand that all machines achieve nonnegative utility, and machines doing no work get zero payment.

[4] Itai Ashlagi, Shahar Dobzinski, and Ron Lavi. An optimal lower bound for anonymous scheduling mechanisms. In *Proceedings 10th ACM Conference on Electronic Commerce (EC-2009)*, pages 169–176, 2009.

[5] Vincenzo Auletta, Roberto De Prisco, Paolo Penna, and Giuseppe Persiano. Deterministic truthful approximation mechanisms for scheduling related machines. In Volker Diekert and Michel Habib, editors, *STACS*, volume 2996 of *Lecture Notes in Computer Science*, pages 608–619. Springer, 2004.

[6] George Christodoulou and Elias Koutsoupias. Mechanism design for scheduling. *Bulletin of EATCS*, 2009.

[7] Peerapong Dhangwatnotai, Shahar Dobzinski, Shaddin Dughmi, and Tim Roughgarden. Truthful approximation schemes for single-parameter agents. In *FOCS*, pages 15–24, 2008.

[8] Leah Epstein and Jiří Sgall. Approximation schemes for scheduling on uniformly related and identical parallel machines. *Algorithmica*, 39(1):43–57, 2004.

[9] Dorit S. Hochbaum and David B. Shmoys. A polynomial approximation scheme for scheduling on uniform processors: Using the dual approximation approach. *SIAM J. Comput.*, 17(3):539–551, 1988.

[10] Annamária Kovács. Fast monotone 3-approximation algorithm for scheduling related machines. In *Algorithms - ESA 2005, 13th Annual European Symposium*, pages 616–627, 2005.

[11] Annamária Kovács. Tighter approximation bounds for lpt scheduling in two special cases. *J. Discrete Algorithms*, 7(3):327–340, 2009.

[12] Roger B. Myerson. Optimal auction design. *Mathematics of Operations Research*, 6(1):58–73, 1981.

[13] N. Nisan, T. Roughgarden, E. Tardos, and V.V. Vazirani. *Algorithmic Game Theory*. Cambridge University Press, 2007.

[14] Noam Nisan and Amir Ronen. Algorithmic mechanism design (extended abstract). In *Proceedings of the Thirty-First Annual ACM Symposium on Theory of Computing (STOC)*, pages 129–140, 1999.

[15] Noam Nisan and Amir Ronen. Algorithmic mechanism design. *Games and Economic Behavior*, 35:166–196, 2001.

Appendix

A The definition of $Scale()$

We use the notation $\lambda = \log_{(1+\delta)}(\rho w),\ \Lambda = \log_{(1+\delta)} w,\ \lambda' = \log_{(1+\delta)}(\rho w'),$ and $\Lambda' = \log_{(1+\delta)} w'$.

Definition 13 ($Scale(\alpha)$)**.** *Let* $\alpha = (w, \mu, \vec{n}^o, \vec{n}^1)$, *and* $\beta = (w', \mu', \vec{n}'^o, \vec{n}'^1)$ *be two configurations, where* $\vec{n}^1 = \vec{n} = (n_\lambda, \dots, n_\Lambda)$, *resp.* $\vec{n}'^o = \vec{n}' = (n'_{\lambda'}, \dots, n'_{\Lambda'})$; *then* $\beta \in Scale(\alpha)$ *iff*

(S1) $w \le w'$, *and* $\mu \le \mu'$;

(S2) *if* $\mu' = \mu$ *then* $\underline{n}'_{\mu} = \underline{n}_{\mu}$ *and* $\underline{n}'_{\mu+1} = \underline{n}_{\mu+1}$; *if* $\mu' > \mu$, *then* $n_{\mu\ell} = n_{\mu m}$ *and* $n'_{(\mu'+1)m} = n'_{(\mu'+1)s}$;
if $\mu' = \mu + 1$ *then* $\underline{n}'_{\mu'} = \underline{n}_{(\mu+1)}$; *if* $\mu' > \mu + 1$, *then* $n_{(\mu+1)\ell} = n_{(\mu+1)m}$ *and* $n'_{\mu' m} = n'_{\mu' s}$;

For the sake of a concise presentation, in the next three requirements we assume that $n_{\mu} \overset{\text{def}}{=} n_{\mu s}$, *and* $n'_{(\mu'+1)} \overset{\text{def}}{=} n'_{(\mu'+1)\ell}$, *whenever* $\mu < \mu'$ *holds, furthermore* $n_{(\mu+1)} \overset{\text{def}}{=} n_{(\mu+1)s}$, *and* $n'_{\mu'} \overset{\text{def}}{=} n'_{\mu'\ell}$, *if additionally* $\mu + 1 < \mu'$ *holds.*

(S3) *if* $\Lambda < l \leq \Lambda'$, *then* $n'_l = 0$;

(S4) *if* $\lambda' < l \leq \Lambda$, *then* $n'_l = n_l$;

(S5) *Let* $\tau_\alpha = n_\lambda \rho w + \sum_{l=\lambda+1}^{\lambda'} |C_l[n_l]|$. *If* $\tau_\alpha = 0$, *then let* $n'_{\lambda'} = 0$. *Otherwise let* $n'_{\lambda'}$ *be the nearest positive integer to* $\tau_\alpha / \rho w'$ *(breaking ties to smaller).*

Disregarding (S2), the conditions above are the natural 'scaling requirements', as they also appear in [8]. By (S3) and (S4), in $\vec{n}'$ a 0 stands for job classes not appearing in $\vec{n}$, whereas those appearing explicitly in both $\vec{n}$ and $\vec{n}'$ must be represented by the same number in both size vectors. Less obvious is (S5). By the first condition we want to achieve that $n'_{\lambda'} = 0$ if and only if *no* jobs of size at most $\rho w'$ have been allocated in A_i. In general, it holds that

$$\mathbb{R}^+ \cap (\tau_\alpha - \rho w, \tau_\alpha + \rho w) \subset ((n'_{\lambda'} - 1) \cdot \rho w', (n'_{\lambda'} + 1) \cdot \rho w').$$

That is, up to a rounding error of $\pm \rho w$, we keep track of the total size of tiny jobs allocated in the cumulative job set. Here we exploit that the magnitudes, and ρ are exact powers of 2, and in particular, if the magnitude increases during scaling, then it at least doubles.

Finally, we turn to the meaning of (S2). If μ is smaller than the new middle size μ', that means that the jobs of $C_\mu[n_{\mu m}]$, that have to be allocated as large jobs, are already in A_i, that is, $n_{\mu\ell} = n_{\mu m}$ and so $C_\mu[n_{\mu m}] = C_\mu[n_{\mu\ell}] \subseteq A_i$. Moreover, $C_\mu[n_{\mu s}]$ are now *all* the jobs allocated from this class, so we can define (the scalar) $n_\mu \overset{\text{def}}{=} n_{\mu s}$. Similarly, if μ' is not a middle size in the old vector $\vec{n}$, then no jobs of class $C_{\mu'}$ have been allocated as small jobs in the set A_i, and this is expressed by $n'_{\mu' m} = n'_{\mu' s}$, and by the notation $n'_{\mu'} \overset{\text{def}}{=} n'_{\mu'\ell}$.

B The proof of Theorem 2

Theorem 2. *For every input* $I = (P_I, s)$ *of the scheduling problem, the optimal makespan over all* m*-paths in* $\mathcal{H}_I$ *is at most* $OPT \cdot (1 + \mathcal{O}(\delta))$, *where* OPT *denotes the optimum makespan of the scheduling problem.*

Proof. We define an appropriate m-path in $\mathcal{H}_I$ by the following steps.
1. We collect tiny jobs from P_I into a set T starting from the smallest job class, and proceeding in increasing order of job classes, but in decreasing order *within* each job class. We stop collecting, if either $|T| \geq 18 \rho w_{\max}$, or the next job has size larger than $\rho w_{\max}$. Let $P := P_I \setminus T$.
2. According to Theorem 1, a canonical allocation $P_1, \ldots, P_m$, $P_i = (L_i, S_i)$ of P exists with makespan of at most $OPT(1 + 3\delta)$. We start from this allocation.
3. We define w_i for each $i < m$ to be the magnitude of the largest job in $\bigcup_{h=1}^{i} P_i$. Since the $|L_i|$ are increasing, now we have $w_i/2 < |L_i|$; together with $\rho < \delta/3$ this implies

$$\frac{3}{2}\rho w_i < \delta |L_i|. \tag{2}$$

In turn, by (D2) jobs of size at most ρw_i can only be in S_i (and not in L_i). Let these be the set T_i of tiny jobs in P_i. (We note that the largest job in $\bigcup_{h=1}^{i} P_i$ might appear in some S_i, but it cannot belong to T_i.) For m let $w_m := w_{m-1}$.
4. We put the set T of tiny jobs to machine m, increasing its workload by a factor of at most $\mathcal{O}(\delta)$. Assume $w_{m-2} > \rho^2 w_{m-1}$. Now either T contains all jobs tiny for w_{m-2}, so that (X) (i) holds, or T has enough tiny jobs so that (X) (ii) holds. In the latter case we redistribute T over machines $m-1$ and m. We also set $w'_m = w'_{m-1} = w_{m-2}$.

If $w_{m-2} \le \rho^2 w_{m-1}$, then either all machines $i \le m-2$ are empty, and (Y) (i) holds, or $m-2$ is non-empty, meaning that T has at least $18\rho w_{\max} \ge 6\rho w_{m-1}$ amount of jobs of size at most w_{m-2}, which are jobs tiny for w_{m-1}, so after creating tiny *blocks* (Y) (ii) will hold.

We update the sets T_{m-1}, T_m, S_m etc., as implied by the changes above.
5. In order to facilitate a transparent definition of our m-path, we modify the current partition of P_I so that the sets contain tiny *blocks* instead of the tiny jobs. The jobs in T_i make $|T_i|/(\rho w_i)$ (fractional) blocks. We build integral blocks out of these for every $i < m$ by a simple procedure (cf. [7]) which packs (fractional) tiny jobs from a fractional block on some machine i into a fractional block on machine $h > i$, until one of them gets rounded to an integer number of blocks. Note that the (full size of) any repacked job remains tiny on its new machine. If there is just one machine $i < m$ left with a fractional block, we put this fractional block on machine m. Finally, we put one block of size ρw (if exists) for each valid magnitude $w < w_m$ onto machine m, from an arbitrary machine (with this we enforce (C3) on the configurations). The workload on every machine i increased by at most $2\rho w_i$.
6. We shift small jobs in $S_{m-2} \cup S_{m-1} \cup S_m$ to the right so that $|\widetilde{\alpha}_{m-2}| \le |\widetilde{\alpha}_{m-1}| \le |\widetilde{\alpha}_m|$ holds, increasing the makespan by at most $3\delta OPT$. Finally we redistribute tiny blocks when needed, so that $|\alpha_{m-2}| \le |\alpha_{m-1}| \le |\alpha_m|$ also holds.

The resulting job partition is denoted by $Q_1, Q_2, \ldots, Q_m \quad Q_i = (L_i, S'_i)$. It is easy to verify that during the whole process, we did not confuse the order of jobs within the classes (the ordering within tiny blocks is irrelevant).
7. Let $\lambda_i = \log_{(1+\delta)} \rho w_i$, and $\Lambda_i = \log_{(1+\delta)} w_i$. Given $|L_i|$, the property (C5) of configurations uniquely determines μ_i for $i < m$; moreover the μ_i are increasing since the same holds for the $|L_i|$. By (D1) and (C5), the jobs in L_i have *rounded* size of at least $(1+\delta)^{\mu_i}$ (we exploit the strict inequality in (D1)), and by definition, at most $w_i = (1+\delta)^{\Lambda_i}$. Thus, $\mu_i \le \Lambda_i$. By (D2) and (C5), the jobs in S'_i have *rounded* size of at most $(1+\delta)^{(\mu_i+1)}$ (we exploit the strict inequality in (C5)). For small δ, inequality (2) above, and (C5) imply $\lambda_i < \mu_i$. For m we define $\mu_m := \mu_{m-1}$.[12]
8. It is now straightforward to define a configuration α_i for each Q_i in a recursive manner. If $Q_i = \emptyset$, then let α_i be the empty configuration. Otherwise let $\vec{n}^o$ of α_i be the null vector for $i = 1$, and be the second size vector in α_{i-1} scaled to w_i for $i > 1$. Based on the calculations of step 7, we can construct the $\vec{n}^1$ of α_i so that $L_{\alpha_i} = L_i$, and $S_{\alpha_i} = S'_i$. In the end, we can define a double configuration $(\alpha_{m-2}, \alpha_{m-1})$ consistent with (V3), since the partition fulfills (X) or (Y).

Clearly, the vertices $v_i = (II, i, \alpha_i)$ exist in V_{II}, and form an m-path in level II, as easily follows from the graph definition. We increased every workload by $\mathcal{O}(\delta) \cdot OPT$, so the theorem follows. □

[12]As a technical consequence of $\mu_{m-1} = \mu_m$, we can only require $(1+\delta)^{(\mu+1)} \le \delta \cdot |L_{\alpha_m}|$ instead of property (C5) of configurations. That is, on the last machine the division (L_m, S_m) does not fulfill (D1). This relaxation for α_m does not affect the rest of the argument.