ETH-Tight FPT Algorithm for Makespan Minimization on Uniform Machines

Lars Rohwedder*

Abstract

Given n jobs with processing times $p_1, \dots, p_n \in \mathbb{N}$ and $m \leq n$ machines with speeds $s_1, \dots, s_m \in \mathbb{N}$ our goal is to allocate the jobs to machines minimizing the makespan. We present an algorithm that solves the problem in time $p_{\max}^{O(d)} n^{O(1)}$, where $p_{\max}$ is the maximum processing time and $d \leq p_{\max}$ is the number of distinct processing times. This is essentially the best possible due to a lower bound based on the exponential time hypothesis (ETH).

Our result improves over prior works that had a quadratic term in d in the exponent and answers an open question by Koutecký and Zink. The algorithm is based on integer programming techniques combined with novel ideas based on modular arithmetic. They can also be implemented efficiently for the more compact high-multiplicity instance encoding.

1 Introduction

We consider a classical scheduling problem, where we need to allocate n jobs with processing times $p_1, \dots, p_n$ to $m \leq n$ machines with speeds $s_1, \dots, s_m$. Job j takes time p_j / s_i if executed on machine i and only one job can be processed on a machine at a time. Our goal is to minimize the makespan. Formally, the problem is defined as follows.

Makespan Minimization on Uniform Machines
Input: $n \geq m \in \mathbb{N}$, $p_1, \dots, p_n \in \mathbb{N}$, $s_1, \dots, s_m \in \mathbb{N}$
Task: Find assignment $\sigma : \{1, \dots, n\} \to \{1, \dots, m\}$ that minimizes

$$\max_{i=1,\dots,m} \sum_{j:\sigma(j)=i} \frac{p_j}{s_i} \ .$$

The special case with $s_1 = \cdots = s_m = 1$ is called Makespan Minimization on *Identical* Machines. Either variant is strongly NP-hard and has been studied extensively towards approximation algorithms. On the positive side, both variants admit an EPTAS [10, 8], that is, a $(1+\epsilon)$-approximation algorithm in time $f(\epsilon) \cdot n^{O(1)}$ for any $\epsilon > 0$. Here, $f(\epsilon)$ is a function that may depend exponentially on ϵ.

More recently, the problem has also been studied regarding exact FPT algorithms, where the parameter is the maximum (integral) processing time $p_{\max} = \max_j p_j$ or the number of different processing times $d = |\{p_1, \dots, p_n\}|$, or a combination of both. Note that $d \leq p_{\max}$. An algorithm is fixed-parameter tractable (FPT) in a parameter k, if its running time is bounded by $f(k) \cdot \langle \text{enc} \rangle^{O(1)}$, that is, the running time can have an exponential (or worse) running time dependence on the parameter, but not on the overall instance encoding length $\langle \text{enc} \rangle$. The

*rohwedder@sdu.dk, University of Southern Denmark, Odense, Denmark. Supported by Dutch Research Council (NWO) project "The Twilight Zone of Efficiency: Optimality of Quasi-Polynomial Time Algorithms" [grant number OCEN.W.21.268]

study of FPT algorithms in the context of our problem was initiated by Mnich and Wiese [15], who showed, among other results, that for identical machines there is an FPT algorithm in $p_{\max}$. The running time was improved through the advent of new generic integer programming (ILP) tools. Specifically, a series of works led to fast FPT algorithms for highly structured integer programs called n-fold Integer Programming, see e.g. [5]. Makespan Minimization on Uniform Machines (and, in particular, the special case on identical machines) can be modeled in this structure and one can directly derive FPT results from the algorithms known for n-fold Integer Programs [13]. Namely, the state-of-the-art for n-fold ILPs [5] leads to a running time of

$$p_{\max}^{O(d^2)} n^{O(1)} \leq p_{\max}^{O(p_{\max}^2)} n^{O(1)} .$$

Koutecký and Zink [14] stated as an open question whether the exponent of $O(d^2)$ can be improved to $O(d)$. This is essentially the best one can hope for: even for identical machines Chen, Jansen, and Zhang [3] have shown that there is no algorithm that given an upper bound $U \in \mathbb{N}$ decides if the optimal makespan is at most U in time $2^{U^{0.99}} n^{O(1)}$, assuming the exponential time hypothesis (ETH). Since $d \leq p_{\max} \leq U$ there cannot be an algorithm for our problem with running time $2^{O(p_{\max}^{0.99})} n^{O(1)}$ or $p_{\max}^{O(d^{0.99})} n^{O(1)}$ either.

A similar gap of understanding exists for algorithms for integer programming in several variants, see [17] for an overview. Since no improvement over the direct application of n-fold Integer Programming is known, Makespan Minimization on Uniform Machines can be seen as a benchmark problem for integer programming techniques. For brevity we omit a definition of n-fold Integer Programming here and refer the reader to [5] for further details.

Jansen, Kahler, Pirotton, and Tutas [9] proved that for the case where the number of distinct machine speeds, that is, $|\{s_1, \ldots, s_m\}|$, is polynomial in $p_{\max}$, the running time of $p_{\max}^{O(d)} n^{O(1)}$ can be achieved. Note that this includes the identical machine case. Jansen et al. [9] credit a non-public manuscript by Govzmann, Mnich, and Omlo for discovering the identical machine case earlier and for some proofs used in their result.

Our contribution. We fully settle the open question by Koutecký and Zink [14].

Theorem 1. *Makespan Minimization on Uniform Machines can be solved in time* $p_{\max}^{O(d)} n^{O(1)}$.

We first prove this for the following intermediate problem, which is less technically involved and simplifies the presentation of our algorithm.

Multiway Partitioning
Input: $n \geq m \in \mathbb{N}$, $p_1, \ldots, p_n \in \mathbb{N}$, $T_1, \ldots, T_m \in \mathbb{N}$ with

$$p_1 + \cdots + p_n = T_1 + \cdots + T_m .$$

Task: Find partition $A_1 \dot\cup \cdots \dot\cup A_m = \{1, \ldots, n\}$ such that

$$\sum_{j \in A_i} p_j = T_i \quad \text{for all } i = 1, \ldots, m .$$

For consistency, we also use the job-machine terminology when talking about Multiway Partitioning. As a subroutine we solve following generic integer programming problem that might be of independent interest.

Multi-Choice Integer Programming
Input: $n, d, \Delta \in \mathbb{N}$, $A \in \mathbb{Z}^{d \times n}$ with $|A_{ij}| \leq \Delta$, $b \in \mathbb{Z}^d$, $c \in \mathbb{Z}^n$. Further, a partition P of $\{1, \dots, n\}$ and $t_S \in \mathbb{N}$ for each $S \in P$.
Task: Find $x \in \mathbb{Z}^n_{\geq 0}$ maximizing $c^\mathsf{T} x$, subject to $Ax = b$ and $\sum_{i \in S} x_i = t_S$ for all $S \in P$.

Theorem 2. *Multi-Choice Integer Programming can be solved in time*

$$(m\Delta|P|)^{O(m)}(n+t)^{O(1)} ,$$

where $t = \sum_{S \in P} t_S$.

Our algorithm is based on an approach that Eisenbrand and Weismantel [6]. introduced, where they use the Steinitz Lemma for reducing the search space in integer programming.

We note that Jansen et al. [9] also used a tailored integer programming algorithm to obtain their result. There are similarities to our ILP algorithm, which is partly inspired by it. The method in [11] also reduces the search space, but via a divide-and-conquer approach due to Jansen and Rohwedder [11] rather than the Steinitz Lemma. It is the author's impression that this method may also be able to produce a guarantee similar to Theorem 2, but since it is not stated in a generic way, we cannot easily verify this and use it as a black box. It seems that the approach in [9] suffers from significantly more technical complications than ours. Our proof is arguably more accessible and compact.

An important aspect in the line of work on FPT algorithms for Makespan Minimization is *high-multiplicity encoding*. Since the number of possible processing times is bounded, one can encode an instance efficiently by storing d processing times $p_1, \dots, p_d$ and a multiplicity $n_1, \dots, n_d$. Semantically, this means that there are n_i jobs with processing time p_i. The encoding can be much more compact than encoding n many processing times explicitly. In fact, the difference can be exponential and therefore obtaining a polynomial running time in the high-multiplicity encoding length can be much more challenging than in the natural encoding. Our algorithm can easily be implemented in time $p_{\max}^{O(d)} \langle \text{enc} \rangle^{O(1)}$, when given an input in high-multiplicity encoding of length $\langle \text{enc} \rangle$. Alternatively, a preprocessing based on a continuous relaxation and proximity results can be used to reduce n sufficiently and apply our algorithm as is, see [2]. For readability, we use the natural instance encoding throughout most of this document.

Other related work. The special case of Multiway Partitioning where $m = 2$ is exactly the classical Subset Sum problem. This problem has received considerable attention regarding the maximum item size as a parameter lately. Note that in contrast to the other mentioned problems, Subset Sum is only weakly NP-hard and admits algorithms pseudopolynomial in the number of items and the maximum size. Optimizing this polynomial (also in the more general Knapsack setting) has been subject of a series of recent works, see [16, 12, 4, 1].

It is natural to ask whether Makespan Minimization on Uniform Machines (or any of the previously mentioned variants) admits an FPT algorithm only in parameter d (and not $p_{\max}$). In the identical machine case, this depends on the encoding type. For high-multiplicity encoding there is a highly non-trivial XP algorithm due to Goemans and Rothvoss [7], that is, an algorithm with running time $\langle \text{enc} \rangle^{f(d)}$, and it is open whether an FPT algorithm exists. For natural encoding the result of Goemans and Rothvoss directly implies an FPT algorithm, see [14]. For uniform machines, the problem is $W[1]$-hard in both encodings, even under substantial additional restrictions, as shown by Koutecký and Zink [14].

Overview of techniques. Key to our result is showing and using the perhaps surprising fact that feasibility of a certain integer programming formulation is sufficient for feasibility of Multiway Partitioning. In essence, this model relaxes the load constraints for machines with large values of T_i, requiring only congruence modulo a for a particular choice of $a \in \mathbb{N}$. We refer to Section 2 for details. It is not trivial to see that the model can be solved in the given time. We achieve this via a new algorithm for Multi-Choice Integer Programming, see Section 3, that we then use in Section 4 to solve our model for Multiway Partition. The result transfers to Makespan Minimization on Uniform Machines by a straight-forward reduction. Finally, we sketch how to adapt the algorithm to high-multiplicity encoding in Section 5.

2 Modulo Integer Programming Formulation

Our model uses a pivot element $a \in \{p_1, \dots, p_n\}$. The selection of a is intricate as its definition is based on the unknown solution to the problem. We can avoid this issue by later attempting to solve the model for each of the d possible choices of a.

A machine $i \in \{1, \dots, m\}$ is called small if $T_i < p_{\max}^4$ and big otherwise. We denote the set of small machines by S and the big machines by $B = \{1, \dots, m\} \setminus S$. Define *mod-IP(a)* as the following mathematical system:

$$\sum_{j=1}^{n} p_j x_{ij} = T_i \qquad \text{for all } i \in S \tag{1}$$

$$\sum_{j=1}^{n} p_j x_{ij} \equiv T_i \mod a \qquad \text{for all } i \in B \tag{2}$$

$$\sum_{j: p_j = a} \sum_{i \in B} x_{ij} \geq p_{\max}^2 \cdot |B| \tag{3}$$

$$\sum_{i=1}^{m} x_{ij} = 1 \qquad \text{for all } j = 1, \dots, n \tag{4}$$

$$x_{ij} \in \{0, 1\} \qquad \text{for all } j = 1, \dots, n, \ i = 1, \dots, m$$

Here, (4) guarantees that the solution is an assignment of jobs to machines, encoded by binary variables x_{ij}. Constraint (1) forces the machine load of small machines to be correct. Instead of requiring this also for big machines, (2) only guarantees the correct load modulo a. Furthermore, we require that a sufficient number of jobs with processing time a are assigned to the big machines. There always exists a pivot element, for which this system is feasible. We defer the details to Section 4 and dedicate the rest of this section to proving that any feasible solution for mod-IP(a) can be transformed efficiently into a feasible solution to the original problem. In particular, feasibility of mod-IP(a) implies feasibility of the original problem, regardless of the choice of a.

2.1 Algorithm

Phase I. Starting with the solution for mod-IP(a), from each big machine we remove all jobs of processing time a. Furthermore, as long as there a processing time $b \neq a$ such that at least a many jobs of size b are assigned to the same big machine $i \in B$, we remove a many of these jobs from i. Note that both of these operations maintain Constraint (2).

However, Constraint (4) will be temporarily violated, namely some jobs are not assigned. After the operations have been performed exhaustively, there are at most

$(d-1)(a-1) \leq p_{\max}^2$ jobs on each big machine i and, using the definition of big machines, it follows that their total processing time is less than T_i.

Phase II. We now assign back the jobs that we previously removed. First, we take each bundle of a many jobs with the same processing time that we had removed together earlier. In a Greedy manner we assign the jobs of each bundle together to some big machine i, where they can be added without exceeding T_i. As we will show in the analysis, there always exists such a machine.

Phase III. Once all bundles are assigned, we continue with the jobs with processing time a. We individually assign them Greedily to big machines i, where they do not lead to exceeding T_i.

2.2 Analysis

Lemma 3. *Let z_{ij} be the current assignment at some point during Phase II. Then there is a big machine $i \in B$ with*

$$\sum_{j=1}^{n} p_j z_{ij} \leq T_i - p_{\max}^2 \ .$$

In particular, adding any bundle to i will not exceed T_i.

Proof. Let x_{ij} be the initial solution to mod-IP(a), from which we derived z_{ij}. Recall that by problem definition we have $p_1 + \cdots + p_n = T_1 + \cdots + T_m$. Further, it holds that

$$\sum_{i \in S} \sum_{j=1}^{n} p_j x_{ij} = \sum_{i \in S} T_i \ .$$

Together, these statements imply that

$$\sum_{i \in B} \sum_{j=1}^{n} p_j x_{ij} = \sum_{i \in B} T_i \ .$$

Consider the operations on x_{ij} that led to z_{ij} and focus on one particular job j. This jobs j was either removed from some big machine and added back to another big machine, which does not change the left-hand side of the equation above; or its assignment did not change, which also does not affect the left-hand side; or it was removed without being added back, which decreases the left-hand side by p_j. The latter is the case at least for the jobs with processing time equal to a. It follows that

$$\sum_{i \in B} \sum_{j=1}^{n} p_j z_{ij} \leq \sum_{i \in B} T_i - \sum_{i \in B} \sum_{j : p_j = a} p_j x_{ij} \leq \sum_{i \in B} T_i - p_{\max}^2 \cdot |B| \ .$$

The last inequality follows from Constraint (3). It follows that there is a big machine $i \in B$ with

$$\sum_{j=1}^{n} p_j z_{ij} \leq \sum_{i \in B} T_i - p_{\max}^2 \ .$$

Since each bundle consists of at most $a - 1 \leq p_{\max}$ jobs with processing time at most $p_{\max}$, adding this bundle to i will not exceed T_i. $\square$

Lemma 4. *Let z_{ij} be the current assignment at some point during Phase III, but before all jobs have been assigned back. Then there is a big machine $i \in B$ with*

$$\sum_{j=1}^{n} p_j z_{ij} \leq T_i - a \ .$$

In particular, a job with processing time a can be added to i without exceeding T_i.

Proof. With the same argument as in the proof of Lemma 3 it follows that

$$\sum_{i \in B} \sum_{j=1}^{n} p_j z_{ij} < \sum_{i \in B} T_i \ .$$

Here, we do not have the same gap as in Lemma 3, since some jobs of size a might already be assigned to big machines, but we still have strict inequality, since not all jobs are assigned. Thus, there is a big machine $i \in B$ with $\sum_{j=1}^{n} p_j z_{ij} < T_i$. Notice that $\sum_{j=1}^{n} p_j z_{ij} \equiv T_i \mod a$, since the initial solution x_{ij} for mod-IP(a), from which z_{ij} was derived, satisfies this and all operations we perform only add or subtract a multiple of a from the machine load. It follows that

$$\sum_{j=1}^{n} p_j z_{ij} \leq T_i - a \ . \qquad \square$$

Theorem 5. *Given a feasible solution to mod-IP(a), the procedure described in Section 2.1 constructs a feasible solution to Multiway Partitioning in time polynomial in n.*

Proof. By the previous Lemmas the algorithm succeeds in finding an assignment z_{ij} where each machine i has load $\sum_{j=1}^{n} p_j z_{ij} \leq T_i$. Since $p_1 + \cdots + p_n = T_1 + \cdots + T_m$ equality must hold for each machine. The polynomial running time is straightforward due to the Greedy nature of the algorithm. $\square$

3 Multi-Choice Integer Programming

This section is dedicated to proving Theorem 2. We refer to Section 1 for the definition Multi-Choice Integer Programming.

3.1 Algorithm

Let $t = \sum_{S \in P} t_S$. On a high level we start with $x = 0$ and then for iterations $k = 1, \dots, t$ increase a single variable by one. We keep track of the right-hand side of the partial solution at all times. We do not, however, want to explicitly keep track of the current progress $\sum_{i \in S} x_i$ for each $S \in P$. Instead, we fix in advance, which set we will make progress on in each iteration, ensuring that for each $S \in P$ there are exactly t_S iterations corresponding to it. Further, we want to make sure that all sets progress in a balanced way, which will later help bound the number of right-hand sides we have to keep track of. For intuition, we think of a continuous time $[0, 1]$. At 0 all variables are zero; at 1 all sets are finalized, that is, $\sum_{i \in S} x_i = t_S$ for each $S \in P$. For a set $S \in P$ we act at the breakpoints $1/t_S, 2/t_S, \dots, t_S/t_S$. This almost defines a sequence of increments, except that some sets may share the same breakpoints, in which case the order is not clear. We resolve this ambiguity in an arbitrary way. Let $S_1, \dots, S_t \in P$ be the resulting sequence and $d_1, \dots, d_t$ the corresponding breakpoints. Formally, we require that $d_1 \leq \cdots \leq d_t$ and for each

$S \in P$ and each $i = 1, \dots, t_S$ there is some $k \in \{1, \dots, t\}$ such that $S_k = S$ and $d_k = i/t_S$.

We now model Multi-Choice Integer Programming as a path problem in a layered graph. There are t sets of vertices $V_1, \dots, V_{t+1}$. The vertices V_k correspond to right-hand sides $b' \in \mathbb{Z}^d$, which stand for a potential right-hand side generated by the partial solution constructed in iterations $1, \dots, k-1$. Formally, V_k contains one vertex for every $b' \in \mathbb{Z}^d$ with $\|b' - d_k \cdot b\|_\infty \leq 4d\Delta|P|$. Let $v' \in V_k$ and $v'' \in V_{k+1}$ and let $b', b'' \in \mathbb{Z}^d$ be the corresponding right-hand sides. There is an edge from v' to v'' if there is some $i \in S_k$ with $A_i = b'' - b'$. Intuitively, choosing this edge corresponds to increasing x_i by one. The weight of the edge is c_i, or the maximum such value if there are several $i \in S_k$ with $A_i = b'' - b'$.

We solve the longest path problem in the graph above from the 0-vertex of V_0 to the b-vertex of V_{t+1}. Since the graph is a DAG, this can be done in polynomial time in the number of vertices of the graph, which is polynomial in $(8d\Delta|P|+1)^d \cdot (t+1)$. From this path we derive the solution x by incrementing the variable corresponding to each edge, as described above.

3.2 Analysis

It is straight-forward that given a path of weight C in the graph above, we obtain a feasible solution of value C for Multi-Choice Integer Programming. However, because we restrict the right-hand sides allowed in $V_1, \dots, V_{t+1}$ it is not obvious that the optimal solution corresponds to a valid path. In the remainder, we will prove this.

Lemma 6. *Given a solution x of value $c^\mathsf{T} x$ for Multi-Choice Integer Programming, there exists a path of the same weight $c^\mathsf{T} x$ in the graph described in Section 3.1.*

This crucially relies on the following result.

Proposition 7 (Steinitz Lemma [18], see also [6]). *Let $\|\cdot\|$ be an arbitrary norm. Let $d \in \mathbb{N}$ and $v_1, \dots, v_n \in \mathbb{R}^d$ with $v_1 + \dots + v_n = 0$ and $\|v_i\| \leq 1$ for all $i = 1, \dots, n$. Then there exists a permutation $\sigma \in \mathcal{S}_n$ such that for all $i = 1, \dots, n$ it holds that*

$$\|v_{\sigma(1)} + \dots + v_{\sigma(i)}\| \leq d \ .$$

Proof of Lemma 6. Consider first one set $S \in P$. Let

$$x_i^{(S)} = \begin{cases} x_i & \text{if } i \in S \\ 0 & \text{otherwise} \end{cases}$$

be the solution restricted to S. For each $i \in S$ we define a vector

$$v_i = \frac{A_i}{2\Delta} - \frac{Ax^{(S)}}{2\Delta t_S} \in \mathbb{R}^d \ .$$

Observe that $\|v_i\|_\infty \leq 1$ for all $i \in S$ and

$$\sum_{i \in S} x_i v_i = \frac{1}{2\Delta}(\sum_{i \in S} A_i x_i - \frac{Ax^{(S)}}{t_S} \sum_{i \in S} x_i) = \frac{1}{2\Delta}(Ax^{(S)} - Ax^{(S)}) = 0 \ .$$

Thus, by the Steinitz Lemma we can find a bijection $\sigma_S : \{1, \dots, t_S\} \to S$ such that

$$\|v_{\sigma_S(1)} + \dots + v_{\sigma_S(i)}\|_\infty \leq d \quad \text{for all } i = 1, \dots, t_S \ .$$

Using the definition of v_i we obtain that

$$\|A_{\sigma_S(1)} + \dots + A_{\sigma_S(i)} - Ax^{(S)} \frac{i}{t_S}\|_\infty \leq 2d\Delta \quad \text{for all } i = 1, \dots, t_S \ .$$

Next we define the following increments: for an iteration $k \in \{1, \ldots, t\}$ with $S := S_k$ and $d_k = i/t_S$ we increment the variable $\sigma_S(i)$. In other words, for each set $S \in P$ we follow exactly the order given by σ_S.

It remains to bound the right-hand side of each partial solution. Consider again an iteration $k \in \{1, \ldots, t\}$. Let $s_S \leq t_S$ be the number of increments to set S that have been performed during iterations $1, \ldots, k$. Then s_S must be such that $(s_S - 1)/t_S \leq d_k \leq s_S/t_S$. In other words,

$$s_S \in \{\lceil d_k t_S \rceil,\ \lfloor d_k t_S \rfloor + 1\} \ .$$

Let x' be the partial solution after iteration k. Then using $\sum_{S \in P} Ax^{(S)} = Ax = b$ and several triangle inequalities, we calculate.

$$\begin{aligned}
\|Ax' - d_k b\|_\infty &= \| \sum_{S \in P} \sum_{i=1}^{s_S} A_{\sigma_S(i)} - d_k b\|_\infty \\
&\leq \sum_{P \in P} \|A_{\sigma_S(\lfloor d_k t_S \rfloor + 1)}\|_\infty + \| \sum_{S \in P} \sum_{i=1}^{\lceil d_k t_S \rceil} A_{\sigma_S(i)} - d_k b\|_\infty \\
&\leq |P| \cdot \Delta + \| \sum_{S \in P} \sum_{i=1}^{\lceil d_k t_S \rceil} [A_{\sigma_S(i)} - d_k \cdot Ax^{(S)}]\|_\infty \\
&\leq |P| \cdot \Delta + \| \sum_{S \in P} (d_k - \frac{\lceil d_k t_S \rceil}{t_S}) \cdot Ax^{(S)}\|_\infty \\
&\quad + \| \sum_{S \in P} \sum_{i=1}^{\lceil d_k t_S \rceil} [A_{\sigma_S(i)} - \frac{\lceil d_k t_S \rceil}{t_S} \cdot Ax^{(S)}]\|_\infty \\
&\leq |P| \cdot \Delta + |P| \cdot \Delta + |P| \cdot 2d\Delta \leq |P| \cdot 4d\Delta \ .
\end{aligned}$$

It follows that solution x can be emulated as a path P in the given graph using the increment sequence defined above. Since the weights of the edges correspond to the values of c, this path has weight $c^\mathsf{T} x$. □

We conclude this section by showing that the previous result extends to the case of Multi-Choice Integer Programming with inequalities instead of equalities.

Corollary 8. *Let $A \in \mathbb{Z}^{d \times n}$ with $|A_{ij}| \leq \Delta$, $b \in \mathbb{Z}^d$, and $c \in \mathbb{Z}^n$. Let P be a partition of $\{1, \ldots, n\}$ and $t_S \in \mathbb{N}$ for each $S \in P$. In time*

$$(m\Delta|P|)^{O(m)}(n + t)^{O(1)} \ ,$$

where $t = \sum_{S \in P} t_S$, we can solve

$$\begin{aligned}
\max\ & c^\mathsf{T} x \\
Ax &\leq b \\
\sum_{i \in S} x_i &= t_S && \text{for all } S \in P \\
x &\in \mathbb{Z}^n_{\geq 0}
\end{aligned}$$

Proof. We reduce to Theorem 2 by adding slack variables. First, we remove all trivial constraints. If $b_j \geq \Delta t$ then the corresponding constraint cannot be violated. For each row j of A we add two variables $s_j, \bar{s}_j$, which form a new set in the partition P with required cardinality $s_j + \bar{s}_j = 2t\Delta$. We add s_j to the left-hand side of jth inequality and replace it by an inequality. Given a solution x to the ILP with inequalities, we can set $\bar{s}_j = (b - Ax)_j$ and $s_j = 2t\Delta - s_j \geq 0$ for $j = 1, \dots, m$ to obtain a feasible solution for the ILP with equalities. For a feasible solution for the ILP with equalities, the same settings of variables x is also feasible for the ILP with inequalities. Hence, both ILPs are equivalent. The transformation increases $|P|$ by d, n by $2d$, and t by $2t\Delta$. These changes do not increase the running time bound asymptotically. □

4 Main Result

We first need to verify that mod-IP(a) is indeed feasible for some choice of a.

Lemma 9. *Given a feasible instance of Multiway Partitioning, there exists a pivot $a \in \{p_1, \dots, p_n\}$ such that mod-IP(a) is feasible.*

Proof. Consider the assignment y_{ij} corresponding to the solution of Multiway Partitioning. By definition of the problem, this assignment satisfies Constraint (4) and for each $i = 1, \dots, m$ that

$$\sum_{j=1}^{n} p_j y_{ij} = T_i \ . \tag{5}$$

This implies that Constraints (1) and (2) are satisfied, for any choice of a. Recall that each big machine $i \in B$ has $T_i \geq p_{\max}^4$. In particular, (5) implies that

$$\sum_{j=1}^{n} y_{ij} \geq \frac{T_i}{p_{\max}} \geq p_{\max}^3 \quad \text{for all } i \in B \ . \tag{6}$$

Thus,

$$\sum_{j=1}^{n} \sum_{i \in B} y_{ij} \geq p_{\max}^3 \cdot |B| \ . \tag{7}$$

Thus, there exists some $a \in \{p_1, \dots, p_n\}$ with

$$\sum_{j : p_j = a} \sum_{i \in B} y_{ij} \geq \frac{p_{\max}^2 |B|}{d} \geq p_{\max}^2 \cdot |B| \ . \tag{8}$$

In other words, Constraint (3) holds for this choice of a, which concludes the proof. □

We will now model the problem of solving mod-IP(a) as an instance of Multi-Choice Integer Programming. The following is a relaxation of mod-IP(a):

$$\begin{aligned}
\sum_{j=1}^{n} p_j x_{ij} &= T_i && \text{for all } i \in S \\
\sum_{j=1}^{n} p_j x_{ij} &\equiv T_i \mod a && \text{for all } i \in B \\
\sum_{j : p_j = a} \sum_{i \in S} x_{ij} &\leq |\{j \mid p_j = a\}| - p_{\max}^2 \cdot |B| && \\
\sum_{i=1}^{m} x_{ij} &\leq 1 && \text{for all } j = 1, \dots, n \\
x_{ij} &\in \{0, 1\} && \text{for all } j = 1, \dots, n,\ i = 1, \dots, m
\end{aligned}$$

Here, we swap the constraint on jobs of size a to the small machines instead of the large ones and, more importantly, we do not require all jobs to be assigned. This model and mod-IP(a) are in fact equivalent, since all jobs that are unassigned must have a total processing time that is divisible by a (because of $p_1 + \cdots + p_n = T_1 + \cdots T_m$ and the constraints). Thus, one can derive a solution to mod-IP(a) by adding all unassigned jobs to an arbitrary big machine, assuming $B \neq \emptyset$. If, on the other hand, $B = \emptyset$ then the requirement that small machines have the correct load implies that all jobs are assigned, making the model exactly equivalent to mod-IP(a).

We can therefore focus on solving the model above, which is done with the help of the standard modeling technique of *configurations*.

Notice that in the model above small machines can have at most $p_{\max}^4$ jobs assigned to each. For big machines, we may also assume without loss of generality that at most $(a-1)d \leq p_{\max}^4$ jobs are assigned to each, since otherwise we can remove a many jobs of the same processing time without affecting feasibility.

Further, there are only a small number of machine *types*: for the small machines there are only $p_{\max}^4$ possible values of T_i and all machines with the same value of T_i behave in the same way; for big machines, all machines with the same value of $T_i \mod a$ behave the same and thus there are only a many types. For one of the $p_{\max}^4 + a$ many types τ, we say that a vector $C \in \mathbb{Z}_{\geq 0}^d$ is a configuration if the given multiplicities correspond to a potential job assignment, namely $\sum_{k=1}^d p_j C_j = T(\tau)$ if $i \in S$ and $\sum_{k=1}^d p_j C_j \equiv T(\tau) \mod a$ if $i \in B$. Here, $T(\tau)$ is the target (or remainder modulo a) corresponding to the type τ. We denote by $\mathcal{C}(\tau)$ the set of configurations for type τ and by $m(\tau)$ the number of machines of type τ. In the following model, we use variables $y_{\tau,C}$ to describe how many machines of type τ use configuration $C \in \mathcal{C}(\tau)$.

$$\sum_{C \in \mathcal{C}(\tau)} y_{\tau,C} = m(\tau) \qquad \text{for all types } \tau$$

$$\sum_{\tau \text{ small}} \sum_{C \in \mathcal{C}(\tau)} C_a \cdot y_{\tau,C} \leq |\{j \mid p_j = a\}| - p_{\max}^2 \cdot |B|$$

$$\sum_{\tau} \sum_{C \in \mathcal{C}(\tau)} C_b \cdot y_{\tau,C} \leq |\{j \mid p_j = b\}| \qquad \text{for all } b \in \{p_1, \ldots, p_n\}$$

$$y_{\tau,C} \in \mathbb{Z}_{\geq 0} \qquad \text{for all } \tau, C$$

It is straight-forward that this model is indeed equivalent to the previous one. The integer program has the structure of Multi-Choice Integer Programming (with inequalities) partitioned into the sets $\{y_{\tau,C} \mid C \in C(\tau)\}$ for each type τ. The maximum coefficient of the constraint matrix is $p_{\max}^4$, the number of rows of constraint matrix A is $d+1$, and the sum of cardinality requirements t is m. Applying Corollary 8 this leads to a running time of $p_{\max}^{O(d)} m^{O(1)}$, assuming the values $|\{j \mid p_j = b\}|$ have been precomputed.

Makespan Minimization on Uniform Machines. We use a binary search framework to the problem, where given $U \in \mathbb{R}_{\geq 0}$ our goal is to determine if there is a solution σ with machine loads $\sum_{j:\sigma(j)=i} p_j \leq T_i := \lfloor s_i U \rfloor$ for each machine i. Since the optimal value has the form L/s_i for some $i \in 1, \ldots, m$ and $L \in \{0, 1, \ldots, np_{max}\}$, a binary search all these values increases the running time by a factor of only $O(\log(m \cdot n \cdot p_{\max}))$, which is polynomial in the input length. Our techniques rely heavily on exact knowledge of machine loads. To emulate this, we add $T_1 + \cdots + T_m - p_1 - \cdots - p_n$ many dummy jobs with processing time 1. Clearly, this maintains feasibility and, more precisely, creates a feasible instance of Multiway Partitioning, assuming that U is a valid upper bound. Note that d may increase by one, which is negligible with respect to our running time. We can now solve the resulting instance using the algorithm for Multiway Partitioning.

5 High-Multiplicity Encoding

Recall that in the high-multiplicity setting we are given d processing times $p_1, \ldots, p_d$ with multiplicities $n_1, \ldots, n_d$ (next to the machine speeds $s_1, \ldots, s_d$). The encoding length is therefore

$$\langle \mathrm{enc} \rangle = \Theta(d \log(p_{\max}) + d \log(n) + m \log(s_{\max})) \ .$$

A solution is encoded by vectors $x_{ij} \in \mathbb{Z}_{\geq 0}$ that indicate how many jobs of processing time p_j are assigned to machine i, which is of size polynomial in $\langle \mathrm{enc} \rangle$. Through a careful implementation we can solve Makespan Minimization on Uniform Machines also in time $p_{\max}^{O(d)} \cdot \langle \mathrm{enc} \rangle^{O(1)}$. The binary search explained at the end of Section 4 adds only an overhead factor of $O(\log(nmp_{\max})) \leq \langle \mathrm{enc} \rangle^{O(1)}$ Notice that the ILP solver in Section 4 already runs in that time of $p_{\max}^{O(d)} \cdot m^{O(1)}$, which is sufficiently fast. This needs to be repeated d times for each guess of a. Afterwards, we need to implement the Greedy type of algorithm in Section 2. Instead of removing one bundle or job at a time, we iterate over all machines and processing times and remove as many bundles as possible in a single step. This requires only time $O(md)$. Similarly, we can add back bundles and jobs of size a in time $O(md)$ by always adding as many bundles as possible in one step.

References

[1] Karl Bringmann. Knapsack with small items in near-quadratic time. In *Proceedings of STOC*, pages 259–270, 2024.

[2] Hauke Brinkop, David Fischer, and Klaus Jansen. Structural results for high-multiplicity scheduling on uniform machines. *CoRR*, abs/2203.01741, 2024.

[3] Lin Chen, Klaus Jansen, and Guochuan Zhang. On the optimality of approximation schemes for the classical scheduling problem. In *Proceedings SODA*, pages 657–668, 2014.

[4] Lin Chen, Jiayi Lian, Yuchen Mao, and Guochuan Zhang. Faster algorithms for bounded knapsack and bounded subset sum via fine-grained proximity results. In *Proceedings SODA*, pages 4828–4848, 2024.

[5] Jana Cslovjecsek, Friedrich Eisenbrand, Christoph Hunkenschröder, Lars Rohwedder, and Robert Weismantel. Block-structured integer and linear programming in strongly polynomial and near linear time. In *Proceedings of SODA*, pages 1666–1681, 2021.

[6] Friedrich Eisenbrand and Robert Weismantel. Proximity results and faster algorithms for integer programming using the steinitz lemma. *ACM Transactions on Algorithms (TALG)*, 16(1):1–14, 2019.

[7] Michel X Goemans and Thomas Rothvoß. Polynomiality for bin packing with a constant number of item types. *Journal of the ACM (JACM)*, 67(6):1–21, 2020.

[8] Klaus Jansen. An eptas for scheduling jobs on uniform processors: using an milp relaxation with a constant number of integral variables. *SIAM Journal on Discrete Mathematics*, 24(2):457–485, 2010.

[9] Klaus Jansen, Kai Kahler, Lis Pirotton, and Malte Tutas. Improving the parameter dependency for high-multiplicity scheduling on uniform machines. *CoRR*, abs/2409.04212, 2024.

[10] Klaus Jansen, Kim-Manuel Klein, and José Verschae. Closing the gap for makespan scheduling via sparsification techniques. *Mathematics of Operations Research*, 45(4):1371–1392, 2020.

[11] Klaus Jansen and Lars Rohwedder. On integer programming, discrepancy, and convolution. *Mathematics of Operations Research*, 48(3):1481–1495, 2023.

[12] Ce Jin. 0-1 knapsack in nearly quadratic time. In *Proceedings STOC*, pages 271–282, 2024.

[13] Dusan Knop and Martin Koutecký. Scheduling meets n-fold integer programming. *Journal on Scheduling*, 21(5):493–503, 2018.

[14] Martin Koutecký and Johannes Zink. Complexity of scheduling few types of jobs on related and unrelated machines. In *Proceedings of ISAAC*, volume 181, pages 18:1–18:17, 2020.

[15] Matthias Mnich and Andreas Wiese. Scheduling and fixed-parameter tractability. *Mathematical Programming*, 154(1-2):533–562, 2015.

[16] Adam Polak, Lars Rohwedder, and Karol Węgrzycki. Knapsack and subset sum with small items. In *Proceeding of ICALP*, pages 1–19, 2021.

[17] Lars Rohwedder and Karol Wegrzycki. Fine-grained equivalence for problems related to integer linear programming. In *Proceedings of ITCS*, 2025.

[18] S Sevast'janov. Approximate solution of some problems of scheduling theory. *Metody Diskret. Analiz*, 32:66–75, 1978.