Minimum Makespan Scheduling with Low Rank Processing Times*

Aditya Bhaskara † Ravishankar Krishnaswamy ‡ Kunal Talwar § Udi Wieder ¶

Abstract

We investigate approximation algorithms for the classical minimum makespan scheduling problem, focusing on instances where the rank of the matrix describing the processing times of the jobs is bounded. A bounded rank matrix arises naturally when the processing time of a job on machine depends upon a bounded set of resources. A bounded rank matrix also shows up when jobs have varying degrees of parallelizability, and the machines have multiple cores.

We are interested in studying the tractability of the problem as a function of the (positive) rank of the processing-time matrix. At one extreme is the case of unit rank, also known as *related machines*, which admits a PTAS [7], and at the other extreme is the full rank case (*unrelated machines*), which is NP-hard to approximate within a factor better than 3/2 [8].

Our main technical contribution is in showing that the approximability of the problem is *not* smooth with the rank of the matrix. From the inapproximability side, we show that the problem becomes APX-hard, *even for rank four matrices*. For rank seven matrices, we prove that it is hard to approximate to a factor 3/2, matching the inapproximability result for general unrelated machines. From the algorithmic side, we obtain a quasi-polynomial approximation scheme (i.e., a $(1+\varepsilon)$ approximation in time $n^{poly(1/\varepsilon,\log n)}$) for the rank two case. This implies that the problem is not APX-hard in this case, unless NP has quasi-polynomial algorithms. Our algorithm is a subtle dynamic program which runs in polynomial time in some interesting special cases. The classification of the three dimensional problem remains open.

1 Introduction

The makespan minimization problem is a classic and well-studied problem. In this problem, we are given a collection of n jobs and m machines, and the processing time incurred by scheduling job j on machine i is given by $p_{i,j} \geq 0$. The goal is to find an assignment of jobs to machines, that minimizes the total processing time assigned to any machine (i.e., the *makespan*). This is also the time to complete processing all jobs. Makespan minimization is one of the problems originally shown to be NP-Hard by Garey and Johnson [5]. The problem is NP-Hard even on two identical machines (through a reduction from the subset-sum problem). Since then most work focused on obtaining efficient *approximation algorithms*, and our work also concerns with precisely this line of study.

A well understood special case for this problem is the one of *identical machines* where the processing time of job j is p_j on each of the m machines. As mentioned earlier, the hardness reduction in [5] shows that an FPTAS is not possible (unless P=NP). In a seminal result Hochbaum and Shmoys [6] show an $(1+\epsilon)$-approximation algorithm which runs in polynomial time for constant ϵ. The algorithm was later extended to the case of *related machines* [7] where each machine may have a different speed, and the processing time of job j on machine i is $p_{i,j} = p_i \cdot s_i^{-1}$. On the other hand, in the problem of *unrelated machines* $p_{i,j}$ are arbitrary non negative numbers. Lenstra, Shmoys and Tardos [8] showed that this problem is NP-hard to approximate within a factor better than $3/2 - \epsilon$ for every constant $\epsilon > 0$. They also show a $2-$approximation algorithm, and this bound was subsequently improved to $2-1/m$ by Shchepin and Vakhania [10]. Closing the gap between the upper and lower bound is a long-standing open problem.

There have been several works trying to study the approximability of various special cases of this problem. This work looks at a new way of parametrizing the complexity of the problem, which we believe arises naturally and is practically relevant. We focus on the rank of matrix indicating the processing times. In this view, the input to the problem is captured by the matrix $P = \{p_{i,j}\}_{1\leq i\leq m, 1\leq j\leq n}$. If P is a non-negative *unit rank matrix*, then every row is a scaling of the other, which means that we can express the processing time p_{ij} as p_j/s_i for some scaling factor s_i. But this is just the instance of related machines, and admits a PTAS [7]. At the other extreme, the case of full rank matrices P

*Research done when the first two authors were visiting Microsoft Research

†EPFL
‡Princeton University
§Microsoft Research
¶Microsoft Research

corresponds to the problem of scheduling on unrelated machines, for which we know a $2 - 1/m$-approximation and a factor $3/2 - \epsilon$ hardness result.

The question we are interested in studying is the case of P having low non-negative rank[1], say 2. Does the problem admit a PTAS for constant rank matrices? Does the complexity of the problem grow exponentially with the rank of the matrix, to become APX-hard for super-constant ranks?

1.1 Motivation A low rank matrix arises naturally in several cases:

Bounded Resources. In this setting each job requires a small number of resources (say CPU operations, memory accesses, network utilization, etc.), and each machine has a different configuration with respect to these parameters. In this case job j may be characterized by a D-dimensional *size vector* $\mathbf{u}_j = (u_{j1}, ..., u_{jD})$, expressing its demand across the D resources. Similarly, each machine is characterized by a D-dimensional *speed vector* $\mathbf{v}_i = (v_{i1}, ..., v_{iD})$ measuring the scarcity (or supply) of the resources on the machine. The processing time is the inner product of these two vectors, $p_{i,j} = \sum_{d \leq D} u_{dj} v_{di}$. Clearly this leads to a matrix P with non-negative rank bounded by D. We remark that both our positive and negative results extend to the case when the sum is replaced by a max.

Multicore Scheduling. The positive rank two case can also model the following scenario: machine i has c_i cores and each job j has a sequential part of length σ_j and is a parallelizable part π_j. So the running time $p_{i,j} = \sigma_j + \pi_j / c_i$. It is easy to see that this problem can be cast as a rank two problem, where the machines have the vector $(1, 1/c_i)$ and the jobs have a vector (σ_j, π_j). More generally, the setting where the jobs can be partitioned into D types of parallelizibility can be captured by rank D instance.

Geometrically Restricted Scheduling. An important special case of the minimum makespan scheduling is the restricted assignment problem, where the running time of job j on any machine i is $\in \{p_j, \infty\}$. A positive rank D matrix can model the following geometric special case of the restricted assignment problem: each job j is associated with a point q_j in $\mathbb{R}^{D-1}$ and each machine is associated a point r_i also in $\mathbb{R}^D$. A job j has processing time p_j on machine i if $q_j \prec r_i$ (here $\prec$ denotes coordinate-wise dominance of q_j by r_i), and has processing time ∞ otherwise. From a technical standpoint, understanding the *geometrically restricted assignment problem* is a crucial stepping stone for the general LRS problem.

1.2 Problem Definition We call the problem of makespan minimization with positive rank D the LRS(D) problem. In this problem, we are given a $n \times D$ matrix J. Column j is the D-dimensional *size vector* $\mathbf{u}_j \succeq 0$ of job j. Similarly we are given a $D \times m$ matrix M where each row i is the D-dimensional *speed vector* $\mathbf{v}_i \succeq 0$ of the i'th machine. The running/processing time p_{ij} of job j on machine i is then $\mathbf{u}_j \cdot \mathbf{v}_i = \sum_{1 \leq d \leq D} u_{j,d} v_{i,d}$, so $P = JM$. The objective is to find an assignment of jobs to machines to minimize the makespan, i.e., the maximum time by which all jobs have been scheduled. We remark that if we are given P it is possible to recover J and M in polynomial time, as long as D is constant [1, 9].

1.3 Related Work Some special cases of the unrelated machine problem were shown to have an approximation ration better than 2. Ebenlendr et.al. [3] showed a 1.75 approximation for the case where each job can be processed on at most two machines. Svensson [11] showed a ≈ 1.94 approximation for the makespan in the *restricted assignment* case where $p_{ij} \in \{p_j, \infty\}$. Intriguingly Svensson's algorithms does not output a schedule that achieves this makespan. The results of Epstein and Tassa [4] imply a PTAS for the problem under the assumption that all the entries in J and M are bounded. A related but different problem called vector scheduling was considered by Chekuri and Khanna [2].

1.4 Our Results From the algorithmic side, we show the following results: We obtain PTAS algorithms for any fixed D, assuming the inputs are *well-conditioned*, i.e, the numbers involved are bounded by some factor Δ for *either* the machines or the jobs. This improves over the results of Eppstein and Tassa [4] who need all the numbers of jobs and machines to be bounded by constants (and we get a better dependence on Δ), although they solve a more general problem. For the special case of $D = 2$ however, we show a dynamic programming based approximation algorithm which computes a $1 + \epsilon$ approximation in quasi-polynomial time for all instances (without the well-conditioned assumption). So the positive rank 2 case is not APX-hard unless NP has quasi-polynomial time algorithms. Moreover, our algorithm for the 2-dimensional case runs in polynomial time for some interesting cases such as the sequential/parallelizable case.

Counterbalancing the above results, we show that in general, LRS(D) is in fact APX-hard for $D \geq 4$[2],

[1] We thank Anna Karlin for suggesting this problem

[2] Not surprisingly, the instances we use will not be well-

thus showing that the approximability is *not* smooth with the rank of the matrix. For $D \geq 7$ we recover the $3/2 - \epsilon$ hardness (which is the best bound known for unrelated machines), and for $4 \leq D \leq 6\}$ we get a smaller constant. We do not resolve the classification of the three dimensional case, and it remains an intriguing open problem.

2 Hardness Results

2.1 Factor 3/2 Hardness for Rank 7 scheduling

In this section, we show that the low rank scheduling problem is hard to approximate to a factor of $3/2 - \epsilon$, when the rank of the matrix is 7. This rules out a smooth trade-off between the approximability and the dimensionality of the problem. As mentioned before, this matches the best known hardness factor for the general makespan minimization problem.

THEOREM 2.1. *For every $\epsilon > 0$, it is NP-hard to approximate* $\mathsf{LRS}(7)$ *to within a factor of $3/2 - \epsilon$.*

Proof. The proof is by reduction from the 3-dimensional matching problem. In this problem we are given a ground set of vertices $U \uplus V \uplus W$, and a set of hyperedges $E \subseteq U \times V \times W$. Furthermore, suppose $|U| = |V| = |W| = n$, and $|E| = m$. A subset $F \subseteq E$ is said to form a perfect matching if each vertex of $U \uplus V \uplus W$ appears in exactly one hyperedge in F. The problem of deciding whether E contains a perfect matching F is known to be NP-complete [5].

We assume the nodes in U are numbered $1, 2, \ldots, n$ and slightly abuse notation by letting $u \in [n]$ indicate both a vertex in U and its index. Similarly v and w denote both vertices in V and W and their indices. Also, let t_u (t_v and t_w resply.) be the number of hyperedges which pass through u (v and w resply.), i.e., $t_u = |\{e \in E : e = (u, *, *)\}|$. We are now ready to define our hard scheduling instance. We set $N := n/\epsilon$.

Machines. For every edge $e = (u, v, w) \in E$, we have a corresponding machine $m(e)$, which has the following speed vector $\mathbf{v}_{m(e)}$ s.t.

$$\mathbf{v}_{m(e)} \equiv (N^{-u}, N^{u}, N^{-v}, N^{v}, N^{-w}, N^{w}, 1) \ .$$

Real Jobs. For each vertex $v \in V$, we have a job $j(v)$, which has the following size vector $\mathbf{u}_{j(v)}$ s.t

$$\mathbf{u}_{j(v)} \equiv (0, 0, \epsilon N^{v}, \epsilon N^{-v}, 0, 0, 1) \ .$$

Likewise, for each $w \in W$, we have a job $j(w)$, which has size vector $\mathbf{u}_{j(w)}$ s.t

$$\mathbf{u}_{j(w)} \equiv (0, 0, 0, 0, \epsilon N^{w}, \epsilon N^{-w}, 1) \ .$$

conditioned.

Dummy Jobs. Additionally, for each vertex $u \in U$, there are $t_u - 1$ dummy jobs of type u which all have size vectors $(\epsilon N^{u}, \epsilon N^{-u}, 0, 0, 0, 0, 2)$.

Processing Times. The processing time of a job (real or dummy) on a machine is the inner product of the corresponding vectors $\mathbf{v}_{m(\cdot)}$ and $\mathbf{u}_{j(\cdot)}$. It is easy to see that the resulting matrix of running times has rank 7.

Notice that real jobs correspond to vertices of V and W, and dummy jobs correspond to those of U. Furthermore, there are $2n$ real jobs and $\sum_u (t_u - 1) = m - n$ dummy jobs in total.

OBSERVATION 1. *In the above instance, the processing time of real job $j(v)$ (or $j(w)$) on machine $m(e)$ is $1+2\epsilon$ if e passes through v (or w resply). If e does not pass through v, then the processing time on $m(e)$ is at least $\Omega(N)$. Similarly, the processing time of dummy jobs of type u are $2 + 2\epsilon$ on machines $m(e)$ where e passes through u. It is at least $\Omega(N)$ on other machines.*

Completeness. We now show that if there is a perfect matching, then there is a feasible solution to the scheduling problem with makespan at most $2 + 4\epsilon$. Indeed, suppose there is a perfect matching $F \subseteq E$, and suppose the hyperedge $e = (u, v, w) \in F$. In this case, jobs $j(v)$ and $j(w)$ go to machine $m(e)$, and the $t_u - 1$ dummy jobs of type u, all go to the $t_u - 1$ machines corresponding to the unmatched edges (involving u). Clearly each machine $e \in F$ gets two jobs assigned to it, and each machine $e \notin F$ gets one dummy job assigned to it. From the above observation, the makespan on any machine is at most $2 + 4\epsilon$.

Soundness. We argue that given a schedule with makespan at most $3 + 5\epsilon$, it is possible to find in polynomial time a perfect matching. Since in this case the optimal schedule has a makespan of at most $2 + 4\epsilon$, we obtain an approximation hardness of $3/2 - \epsilon$.

To see this we first assume we are given a schedule of makespan at most $2 + 4\epsilon$. Now, each machine either has a dummy job, or has at most two real jobs (all other job assignments will exceed the bound of $2 + 4\epsilon$). For any vertex $u \in U$, the dummy jobs of type u have all been scheduled on machines $m(e)$ s.t e passes through u (by Observation 1). Since there are $t_u - 1$ such jobs and each occupies a unique machine, this leaves only one machine $m(e_u)$ for some edge e_u which has no dummy jobs scheduled on it. Indeed, these machines corresponding to edges $F = \{e_u : u \in U\}$ are the n machines on which the real jobs are scheduled. Furthermore, since there are $2n$ real jobs and $|F| = n$, each machine $m(e_u)$ has a load of exactly 2 real jobs. Now because job $j(v)$ can only be scheduled on a machine e that passes through v, we

have that each vertex $v \in V$ (and similarly for $w \in W$) passes through a unique edge in F. Therefore, the edges of F form a perfect matching. The argument is complete by the observation that the next lowest makespan value possible in our instance is $3 + 6\epsilon$, so we can recover a perfect matching out of any schedule with cost at most $3 + 5\epsilon$, giving us the desired $3/2 - \epsilon$ hardness of approximation.

2.2 APX-hardness for 4-Dimensions Building on the above ideas, we now show that the problem $\mathsf{LRS}(4)$ is APX-hard, i.e., hard to approximate to within a constant factor (smaller than 3/2).

At a high level, the idea is to use only one dimension for each partition of the $3D$ matching. Recall that in the previous reduction, we used 2 coordinates to specify that a job v could only run on a machine $e = (*, v, *)$. We did this by using one coordinate to ensure that v can only run on machines $(*, \leq v, *)$, and another coordinate which ensures v can run only on machines $(*, \geq v, *)$ leaving the edges (machines) passing through v as the only ones in the intersection. In the coming reduction, we will save one coordinate for each partition U, V, and W by only enforcing one (the $\leq$ constraint) of these constraints. But in such a situation, a job v meant to be scheduled on a machine $(*, v, *)$ can be scheduled on machines with lower coordinate value $v' \leq v$. We resolve this issue by creating sufficiently many jobs of earlier coordinate values, that such a situation does not arise. This comes at the cost of lowering the approximation constant. We now present the formal construction.

Recall that t_u = number of hyper-edges which pass through $u \in U$, and likewise, t_v and t_w are the number of hyper-edges which pass through $v \in V$ and $w \in W$ respectively.

Machines. For every edge $e = (u, v, w) \in E$, we have a machine $m(e)$ with speed vector

$$\mathbf{v}_{m(e)} = (N^u, N^v, N^w, 1)\,.$$

Real Jobs. For each vertex $u \in U$, we have a job $j(u)$ with size vector

$$\mathbf{u}_{j(u)} = (\epsilon N^{-u}, 0, 0, 1)\,.$$

Similarly, for each vertex $v \in V$, we have a job $j(v)$ with size vector $\mathbf{u}_{j(v)} = (0, \epsilon N^{-v}, 0, 1)$ and for each vertex $w \in W$, we have a job $j(w)$ with size vector $\mathbf{u}_{j(w)} = (0, 0, \epsilon N^{-w}, 1)$.

Dummy Jobs. Like before, there are $t_u - 1$ dummy jobs of type u which all have size vector $(\epsilon N^{-u}, 0, 0, 0.8)$. In addition, there are $t_v - 1$ dummy jobs of type v which have size vector $(0, \epsilon N^{-v}, 0, 0.9)$ and there are $t_w - 1$ dummy jobs of type w which have size vector $(0, 0, \epsilon N^{-w}, 1.3)$. Notice that now there are $m - n$ dummy jobs corresponding to vertices in U, V and W each, resulting in a total of $3(m - n)$ dummy jobs.

Completeness. We now show that if there is a perfect matching, then there is a feasible solution to the scheduling problem with makespan at most $3 + 3\epsilon$. Indeed, suppose there is a perfect matching $F \subseteq E$, and suppose the hyperedge $e = (u, v, w) \in F$. In this case, jobs $j(u)$, $j(v)$ and $j(w)$ go to machine $m(e)$, and the dummy jobs of type u, all go to the $t_u - 1$ machines corresponding to the unmatched edges (involving u). Likewise, the dummy jobs of type v go to the $t_v - 1$ unmatched edges for v, and likewise for $w \in W$. Clearly each machine $e \in F$ gets three jobs assigned to it, and each machine $e \notin F$ gets three dummy job assigned to it. Furthermore, the makespan on any machine is at most $3 + 3\epsilon$. To see this observe that the makespan of the matched edges is $3(1 + \epsilon)$ and the makespan of an unmatched edge is $1.3 + 0.9 + 0.8 + 3\epsilon$.

Soundness. Now, suppose we have a solution with makespan of at most $3.09 + 3\epsilon$.

LEMMA 2.1. *If the makespan is at most $3.09 + 3\epsilon$, then each machine has a load of 3 jobs, furthermore, a machine e either has three real jobs of types u, v, and w respectively, or has three dummy jobs of types u, v and w respectively.*

Proof. The total number of machines is m and the total number of jobs is $3m$. Observe that a machine with four jobs must have a makespan of at least $3.2 + 4\epsilon$, so it must be the case that each machine has exactly three jobs assigned to it. For the second part of the lemma we focus on the last dimension in the size vector of each job. The total contribution of the last dimension across all jobs is $3m$ so on average the makespan of each machine individually has to have an additive constant of 3 which could be be obtained either as $1 + 1 + 1$ or $0.8 + 0.9 + 1.3$. Deviating from this assignment implies that at least one machine has a makespan with an additive constant strictly larger than 3. The smallest possible configuration larger than 3 is $0.8+1+1.3 = 3.1$ which would result in a makespan of at least $3\epsilon + 3.1$. We conclude that it must be the case that each machine has 3 jobs of three distinct types.

LEMMA 2.2. *For all $1 \leq i \leq n$, the $t_{u_1} - 1$ dummy jobs (and the real job) corresponding to vertex u_i are all assigned to edges of the form $(u_i, *, *)$. Furthermore, no two jobs corresponding to the same vertex are scheduled on the same machine.*

Proof. The proof is by induction on u_i. For the base case, consider jobs of type u_1. Firstly, notice that both real jobs and dummy jobs of type u_1 cannot be scheduled on machines corresponding to edges of the form $(u_{>1}, *, *)$ (by the way we have defined the size vectors, such assignments have makespan $\Omega(N)$). Furthermore, by Lemma 2.1, no two of these jobs can be scheduled on the same machine. This establishes the base case. For inductive hypothesis, suppose the statement is true up to u_{i-1}. Now consider jobs corresponding to u_i. Again, notice that these jobs cannot be scheduled on machines corresponding to edges of the form $(u_{>i}, *, *)$ (by the way we have defined the size vectors, such assignments have makespan $\Omega(N)$). Furthermore, by the inductive hypothesis, all $\sum_{p=1}^{i-1} t_p$ edges of the form $(u_{<i}, *, *)$ are occupied by jobs associated with vertices $u_1, u_2, \ldots, u_{i-1}$.Therefore, by Lemma 2.1, jobs of type u_i can't be scheduled on machines $(u_{<i}, *, *)$. Therefore, these jobs have to be assigned to the t_{u_i} distinct machines/edges of the form $(u_i, *, *)$. This completes the proof.

To recover the matching, we look at the n edges on which the real jobs have been assigned. We have shown that it is NP-hard to distinguish between a makespan of $3 + \epsilon$ and $3.09 + \epsilon$ implying a hardness of approximation within a factor of 1.03. We made no attempt at optimizing the exact constant.

3 Algorithmic Results

We give the two main algorithmic results. The first is for the *well-conditioned* case (in which we assume that either the job-lengths or the machine speeds have a 'bounded aspect ratio'). In this case, we show that for any fixed dimension d, there exists a PTAS for the $\mathsf{LRS}(d)$ problem. Other than being of independent interest, the well conditioned case also serves as an ingredient in our second result concerning the rank-two (or two-dimensional) problem. In this case we give a QPTAS without any restriction on the aspect ratio. (Note that for $d \geq 4$, we do not expect such an algorithm in view of our APX-hardness results).

3.1 PTAS in case of Bounded Aspect Ratio

We now consider the special case of $\mathsf{LRS}(D)$ where either the jobs or the machines have bounded *aspect ratio*, which we define below.

Definition 1. (Aspect ratio) *A set M of machines in an instance of $\mathsf{LRS}(D)$ is said to have an **aspect ratio** of Δ, if for every $1 \leq d \leq D$, there exists an L_d s.t. all the machines have a speed in the interval $[L_d, \Delta \cdot L_d]$ in the dth dimension. That is, if $\mathbf{v}_i$ is the speed vector for some machine i, then $L_d \leq v_{i,d} \leq \Delta \cdot L_d$. An analogous definition holds for the case of jobs having bounded aspect ratio.*

We are now ready to formally state our theorem.

Theorem 3.1. *Consider an instance of $\mathsf{LRS}(D)$, with the additional constraint that the set of machines have an aspect ratio of at most Δ. Then there exists an algorithm which gives a $(1+\varepsilon)$-approximation to the minimum makespan, with running time $n^{(\frac{1}{\varepsilon}\log(\Delta D/\varepsilon))^{O(D)}}$. A similar result holds if the jobs are well-conditioned.*

Note that Theorem 3.1 essentially *justifies* having to use a large aspect ratio in our NP-hardness reductions: the approximation scheme is quasipolynomial for any polynomial aspect ratio. We start by making the following assumptions which are without loss of generality, and help simplify the presentation.

1. ***Machines are well-conditioned.*** Jobs being well-conditioned can be handled symmetrically.
2. ***Unit Makespan.*** To get this, we can appropriately scale either the job sizes or the machine speeds, without changing the optimal schedule (ensuring that it has a makespan of 1).
3. ***Machines speeds in*** $[1, \Delta]$. By re-scaling the job lengths along each dimension, we can assume that in each coordinate, the machine speeds lie in the interval $[1, \Delta]$. Indeed, if along some coordinate i the machine speeds are originally between $[L_i, \Delta \cdot L_i]$. Then, we scale up the job sizes (along this coordinate) by a factor of L_i and scale down the machine speeds also by the same factor of L_i, to get the desired property. Note that the optimal makespan has not changed, and still remains at 1.

Observation 2. *For any job j, all coordinates in $\mathbf{u}_j$ are bounded by Δ. That is, $\|\mathbf{u}_j\|_\infty \leq \Delta$.*

This is easy to see, and follows directly from Assumption 3 that the machines have coordinates bounded by Δ and that the optimal makespan is 1. Before we begin with our algorithm description, we recall the theorem of Lenstra, Shmoys and Tardos [8] which will serve as a crucial sub-routine.

Lemma 3.1. (Lenstra, Shmoys, Tardos) *Suppose we are given a collection of m machines and n jobs, and a matrix $\{p_{ij}\}_{1 \leq i \leq m, 1 \leq j \leq n}$ of processing times. Suppose there exists a scheduling s.t. the schedule on machine i ends at time T_i, for some (given) numbers T_i. Then we can find in polynomial time, a schedule s.t. for all i, the schedule on machine i ends at time $T_i + \max_j p_{ij}$.*

Algorithm Idea. Lemma 3.1 can provide a good approximation when all the p_{ij} are small compared to T_i. The high-level idea is to guess the optimal assignment of the "large jobs" so that the remaining jobs are small for every machine (i.e., with processing times at most ε), then we can use the above Lemma to compute a schedule with makespan $1 + O(\varepsilon)$. But how do we *guess* the assignment of large jobs, especially since there can be many of them in a diverse collection?

To this end, our main observation is that if a job is appropriately tiny (chosen to be $\varepsilon^3/\Delta D^2$ with hindsight) in some coordinates (but not all), we can essentially make those coordinates 0. Indeed, if a job has at least one large coordinate (say larger than ε^2/D), then there can't be too many of such jobs assigned to any machine (at most D/ε^2), and the total contribution from tiny coordinates of all such jobs is at most ε. Using this, we can discretize the jobs that have a large coordinate into $O(\log \Delta D/\varepsilon)^D$ many job types, and guess their optimal assignment. Then it is easy to see that we are left only with jobs which are small in all the coordinates, for which we use Lemma 3.1. We now proceed with the complete details.

Pre-processing the jobs. In the following, a job coordinate is called *small* if it is $< \varepsilon^2/D$. It is called *tiny* if it is $< \varepsilon^3/\Delta D^2$. Recall that all machine coordinates are bounded in $[1, \Delta]$.

A job is called *all-small* if all its coordinates are small. All other jobs are called *large*.

Step 1. Zeroing tiny coordinates. If a job is large, then we set *all its tiny coordinates* to 0. Call the resulting instance $\mathcal{I}'$ and the original instance $\mathcal{I}$.

LEMMA 3.2. *The optimal makespan of $\mathcal{I}'$ is at most 1, assuming the optimal makespan of $\mathcal{I}$ is 1. Likewise, if we have a schedule of makespan 1 in $\mathcal{I}'$, then the same schedule has makespan at most $1 + \varepsilon$ in $\mathcal{I}$.*

Proof. The first part of the lemma is trivial, since we only make some job coordinates smaller. For the second part, consider the same schedule as the one with makespan 1 for $\mathcal{I}'$. The main idea is that if a machine i processes jobs which are large, there can be at most $\Delta D/\varepsilon^2$ such jobs on machine i (since it has makespan 1), and thus the sum total of *all* the tiny coordinates of all such jobs can contribute a total of at most $(\Delta D/\varepsilon^2) \times D \times (\varepsilon^3/\Delta D^2) < \varepsilon$ to the makespan on that machine. Here, we have used the fact that the machine coordinates are well-conditioned with aspect ratio of Δ.

Step 2. Discretizing job coordinates. Henceforth, we focus on instance $\mathcal{I}'$. Now, we can divide the large jobs into constantly many bins, by rounding up the non-zero coordinates to the nearest power of $(1+\varepsilon)^{-1}$. This gives $k \approx (1/\varepsilon)\log(\Delta^2 D^2/\varepsilon^3)$ *classes* for each non-zero coordinate, and thus the large jobs can be put into $R := (k+1)^D$ bins, depending on the class of each coordinate (the '+1' is for zero coordinate).

Step 3. Discretizing machine coordinates. let us bucket the machines also based on speeds in each coordinate. Here there will be $S := ((1/\varepsilon)\log \Delta)^D$ classes, because of the well-conditioned assumption. Let the machines of each class s be denoted by M_s.

It is easy to see that the rounding causes the makespan to increase by at most a multiplicative factor of $(1 + O(\varepsilon))$. Let $\mathcal{I}''$ denote the final instance after all these pre-processing steps. The following lemma is immediate, from Lemma 3.2 and the fact that we only rounded-up to the next higher power-of-$(1+\epsilon)$.

LEMMA 3.3. *The optimal makespan of $\mathcal{I}''$ is at most $1+\varepsilon$, assuming the optimal makespan of $\mathcal{I}$ is 1. Likewise, if we have a schedule of makespan 1 in $\mathcal{I}'$, then the same schedule has makespan at most $1 + \varepsilon$ in $\mathcal{I}$.*

We now look to solving the instance $\mathcal{I}''$. In the following, let $R = (k+1)^D$ denote the number of job types for large jobs, $S = ((1/\varepsilon)\log \Delta)^D$ denote the number of machine types, and let the number of jobs in each of these classes be $n_1, n_2, \ldots, n_R$.

1. Pre-process and divide the large jobs and machines into classes as above.
2. "Guess" the assignment of large jobs into the machine classes. That is, for all $1 \le r \le R$, guess the *right split* (i.e., the way OPT splits these jobs) for each n_r into S *parts*.
3. For each $1 \le s \le S$, schedule the guessed job set L_s in the machines M_s by running the PTAS due to Hochbaum and Shmoys [7] for scheduling on identical machines.
4. Grant an additional *extra time* of ε in each machine (we can afford an ε addition to makespan), and schedule the all-small jobs on the machines, using Lemma 3.1.

THEOREM 3.2. *If there exists a scheduling of the jobs for $\mathcal{I}''$ with makespan 1, the algorithm finds one of makespan at most $1 + O(\varepsilon)$.*

Proof. Suppose OPT denotes the optimal schedule for the instance $\mathcal{I}''$. For each machine class $1 \le s \le S$, let L_s, and AS_s denote the set of large jobs and all-small jobs that OPT runs in machine class s. Now assume we have guessed the correct set of jobs of each type n_r (for $1 \le r \le R$) which are scheduled on the different

machine classes $1 \leq s \leq S$. That is, we have guessed the sets L_s, $1 \leq s \leq S$ in Step 2 above.

Now, notice that within each machine class, we have a set of identical machines. So consider a fixed machine class s: Since we made the right guess for L_s, the algorithm of Hochbaum and Shmoys (which we use in Step 3 above) will find a schedule of these large jobs L_s with makespan at most $1+\varepsilon$. Next, we claim that it is feasible to schedule *all* the all-small jobs AS_s that OPT schedules in this machine class, by using an additional space of ε on each machine. Indeed, suppose the all-small jobs AS_s don't fit into these machines, even when each machine is granted an additional makespan of ε. Then, the total volume of jobs in $L_s \cup AS_s$ exceeds the total capacity available in all the machines of M_s, which contradicts the fact that we correctly guessed L_s.

Therefore, after placing all the jobs L_s on the machines in Step 4, we are guaranteed a feasible solution for fitting in the small jobs into these machines, when provided with an additional ε time. We now directly appeal to Lemma 3.1 to complete our proof.

Note that the run time of the algorithm is dominated by the number of ways of dividing the R n_j's into S parts, which is $O(n^{RS})$. Plugging in the values of R and S, we get the claimed running time. Note that this is polynomial in the case whenever D and Δ are constants. This finishes the proof of Theorem 3.1.

4 The General LRS(2) Problem

Next we present our main algorithmic result, which is a QPTAS for the general 2-dimensional case, with no restrictions on the aspect ratio.

THEOREM 4.1. *There exists an algorithm for* LRS(2) *which gives a* $(1+\varepsilon)$ *approximation to the minimum makespan, with running time* $n^{poly(1/\varepsilon, \log n)}$.

Notation. Recall that we have n jobs to be scheduled on m machines. In this section, we denote the job vectors by (p_j, q_j) and the machine vectors by (s_i, t_i), and associate them with the corresponding points on the plane. The processing time p_{ij} is now taken to be $p_j/s_i + q_j/t_i$ (notice that this is the inner product if we invert the entries of the speed vector). This transformation makes it easier to interpret the problem geometrically. The goal as before is to find an assignment which minimizes the makespan. We begin by making a couple of assumptions about the problem instance (all of which are without loss of generality). We call such instances *canonical instances*.

Canonical Instances.

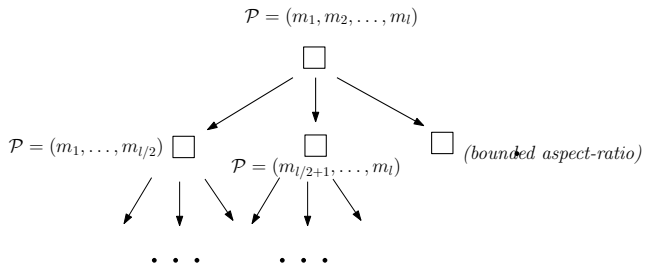
1. The optimal makespan is 1. This can be ensured by guessing the optimal makespan, and then scaling the machine coordinates by this value[3]
2. All the non-zero s_i, t_i, p_j, q_j are powers of $(1+\varepsilon)$. It is easy to see that rounding the non-zero coordinates to the nearest power-of-$(1+\epsilon)$will change the makespan by only a factor of $(1+O(\varepsilon))$.
3. Most importantly, we assume that for any job j and machine i, $\max\{p_j/s_i, q_j/t_i\} > \varepsilon/2n$. Indeed, this is w.l.o.g. because even if we schedule all such jobs j on the machine i, the makespan will only increase by $2n \cdot (\varepsilon/2n) < \varepsilon$, and so we can ignore all such jobs j.
4. Similarly, for any two machine i and i', it is not the case that both $\frac{s_i}{s_{i'}} \leq \varepsilon/mn$ and $\frac{t_i}{t_{i'}} \leq \varepsilon/mn$. Any such i can be removed and all its jobs in OPT can be moved to i' while increasing the makespan by at most an additive ε.

Outline of the algorithm. The algorithm proceeds by breaking up an instance into *sub-instances*, that are solved recursively. At each step the current set of machines M, is split into three sets M_1, M_2, M_3, each of size at most $|M|/2$, thus assuring that the depth of the recursion is at most logarithmic. Moreover, we try a quasipolynomial number of ways to partition the set J of jobs to three subsets J_1^k, J_2^k, J_3^k and recursively solve the instances $(M_1, J_1^k), (M_2, J_2^k), (M_3, J_3^k)$ for all $k \leq n^{poly(\log n)}$. The leaves in this partition serve as the base case of the induction. The number of these is bounded by $(n^{poly(\log n)})^{O(\log n)}$ which is quasipolynomial. The base cases will be of two kinds: either of *bounded aspect ratio*, or having a *unique Pareto optimal machine*. For these cases, we use algorithms from Sections 3.1 and 4.2 respectively.

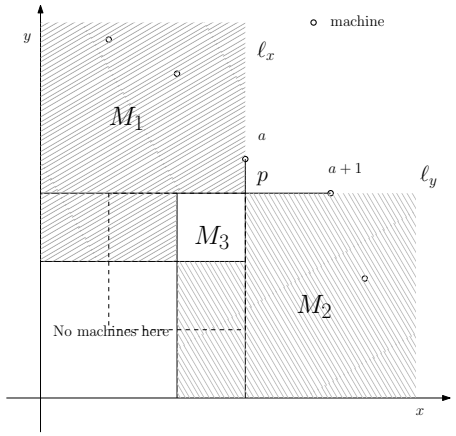
A natural way to divide the instance is by using the geometry of the problem. For example, we can draw a vertical line and split the instance into machines to the left of the line, and those to the right. But how do we split the jobs? Clearly, by the unit-makespan assumption, jobs to the right of the line can't be scheduled on machines to the left (as their processing times would exceed 1). However, jobs to the left can potentially be scheduled on machines to the right, and it is not easy to succinctly characterize how these jobs split.

While such an approach does not work, we show that we can split the instance, by drawing a vertical line *and* a horizontal line, and getting *three sub-instances*. We now present the details.

[3] Technically, we use binary search to ensure that the makespan is in $[1-\varepsilon, 1]$, rather than exactly 1. This does not affect the approximation ratio.



(a) Recursive Partitioning Tree



(b) One Level of Recursive Partitioning

4.1 Recursive Partitioning: Defining the Sub-instances We begin by re-defining the input to the instance in a form more suitable for our algorithm. To this end, let an instance $\mathcal{I}$ of the scheduling problem be defined by the set of machines M and a set of jobs J: for convenience, we characterize the jobs by a set of *job types* T, and a vector $y \in Z^T$ where y_t denotes the number of jobs of type t in the instance; here the type of a job is defined by its size (p_j, q_j).

Given that we have associated machines with points in $\Re^2$, we have a natural partial order $\succeq$ on machines where $(s,t) \succeq (s',t')$ if $s \geq s'$ and $t \geq t'$. Let $P = \{m_1, \ldots, m_l\}$ denote the set of maximal elements in this partial order: these machines form the *pareto curve* and have the property that for each $i \in M$, there is a $p \in P$ such that $p \succeq i$. Let the machines in P be sorted in increasing order of their first coordinate and let $a = m_{l/2}$ be the median machine in this ordering.

Let $L := \log mn/\varepsilon$. Intuitively, since our goal is to run in time roughly n^L, L would bound the size of instances we can compute directly. Recall that the size vectors are rounded to powers of $1+\epsilon$. We call all numbers that were rounded to the same power of $1+\epsilon$ a *cell*. It would be convenient to do arithmetic using cells, effectively working on a logarithmic scale. Thus when we say machine i is L cells to the left of machine i'. we mean that ratio of the rounded $s_{i'}$ to the rounded s_i is $(1+\epsilon)^L$.

Recursively Partitioning Machines.

If the set M of machines has aspect ratio at most $(1+\epsilon)^L$ (we call this the well-conditioned case), then we stop the recursion, and can solve this instance using the algorithm in Section 3.1. If the set M has a single machine which dominates all the rest, we stop the recursion and solve the instance using the algorithm in Section 4.2. Otherwise, we split the set of machines M into three groups, as shown in Figure 1(b) (where $a = m_{l/2}$ as above). Formally, let us denote by ℓ_x the vertical line $x = s_a$, and ℓ_y the horizontal line $y = t_{a+1}$, and let p be the point of intersection. We define the three sets of machines in each sub-instance.

1. **Class $\mathbf{M_1}$** Consists of machines above ℓ_y, and those below ℓ_y but at least L cells to the left of ℓ_x.
2. **Class $\mathbf{M_2}$** consists of machines right of ℓ_x, and those to the left of ℓ_x but at least L cells below ℓ_y.
3. **Class $\mathbf{M_3}$** consists of all machines which are in the $L \times L$ cells forming a rectangle to the left and below p (in other words, the $L \times L$ rectangle corresponds to a rectangle formed by L powers-of-$(1+\epsilon)$to the left of p and L powers-of-$(1+\epsilon)$below p).

We recall, by the canonical assumption (point 4) that the white bottom left region in the figure has no machines. The crucial point to note here is that, by the definition of the Pareto curve, we have that the Pareto curve for M_1 is $\{m_1, \ldots, m_{l/2}\}$, and the Pareto curve of M_2 is $\{m_{l/2+1}, \ldots, m_l\}$. This will help us bound the depth of our recursion by $O(\log n)$.

LEMMA 4.1. *The depth of the above recursive decomposition process is $O(\log n)$. Moreover, the sub-instances corresponding to the leaves are either well-conditioned, or have a unique Pareto machine.*

Proof. Each step of the recursion the machines in M_3 have an aspect ratio of at most L and therefore form a base case for the recursion. Furthermore, the number of machines in the Pareto curve in each of the instances M_1, M_2 is reduced by a factor of two. Once the number of machines in the Pareto curve drops to 1 the recursion stops. The Lemma follows.

Recursively Partitioning Jobs. Given the partition of machines into M_1, M_2, M_3 we need to 'guess' how to partition the set of jobs into corresponding J_1, J_2, J_3. Recall that $a := m_{l/2}$. Since we cannot assign any job (p,q) with $p > s_a$ on any machine to the left of ℓ_x, we place all such jobs in J_2. Likewise, we place all jobs (p,q)

with $q > t_{a+1}$ in J_1. Thus we only need to consider jobs which are below and to the left of p. We divide these jobs into three classes:

1. Jobs that are at most $2L$ cells to the left or below the point $p = (s_a, t_{a+1})$. Call this region $\mathcal{B}'$ (dotted in figure).
2. Jobs on the L vertical lines left or equal to line ℓ_x, and below the box $\mathcal{B}'$.
3. Jobs on the L horizontal lines below or equal to line ℓ_y, to the left of the box $\mathcal{B}'$.

These jobs need to be split between M_1, M_2 and M_3 and we will guess this split. To ensure that we can bound the number of instances in the recursion, we will encode this split using a small number of integers. First note that there are only $4L^2$ job types in the box $\mathcal{B}'$ so that a sub-instance corresponding to M_i only needs to know how many of each type in this box it needs to schedule. There may however be an unbounded number of job types on the vertical and horizontal lines. The main observation here is that jobs of type 2 above are Y-small for *all* the machines in M_1 and M_3. Thus if M_1 (or M_3) has to handle k jobs of type 2 on a line ℓ, we can assume without loss of generality that they are the ones with the largest Y-coordinate. To see this observe that swapping jobs only decreases the load of machines outside M_1, and increases the load on an M_1 machine by at most ε. Similarly, jobs of type 3 above are X-small for all the machines in M_2 and M_3, and we only need to remember the number on each line. Thus, for each of M_1, M_2, M_3, the sub instances that we need to solve are fully characterized by $4L^2 + 2L$ integers, the number of jobs of type 1 (each 'cell'), and the number of jobs of type 2 and 3.

Thus, checking that (M, J) is feasible is straightforward: for every possible 3-way partitioning of the jobs in the box $\mathcal{B}'$, and every way of partitioning the jobs of type 2 and 3 above we get three instances corresponding to M_1, M_2 and M_3. If for any of the partitioning, the three resulting instances are feasible, we can construct a makespan $(1+\varepsilon)$ schedule for (M, J) and return feasible. If for every possible partitioning of the jobs, at least one of the three instances is infeasible, we declare infeasibility. Note that the number of partitioning is only $n^{O(L^2)}$ so that given the solutions to sub instances, the solution to (M, J) can be computed in time $n^{O(L^2)}$.

Since the depth of the recursion is only $O(\log n)$, the set of jobs in any sub instance in the whole recursion tree is specified by $O(L^2 \log n)$ integers in $[n]$. The set of machines is easily seen to be describable by a length $O(\log n)$ ternary string. Thus we conclude that

LEMMA 4.2. *The total number of sub-instances in the entire program is bounded by* $\sum_{j=1}^{O(\log n)} 3^j n^{O(L^2 \log n)}$.

We now show how to handle each of the two types of base cases. First, recall that the Algorithm in Section 3.1 handles precisely the well-conditioned instances. Therefore, we would be done if we could handle the instances where there is a unique Pareto machine.

4.2 Unique Pareto optimal machine

Main idea: Suppose the Pareto curve of the machines consists of precisely one machine, say i. This means that every other machine i' satisfies $i \succeq i'$. By the canonical assumption, the coordinates of i' cannot *both* be *much smaller* than those of i (so also for the jobs), and thus we have that both the jobs and the machines are on roughly $L := \frac{1}{\varepsilon}\log(n/\varepsilon)$ different horizontal and vertical lines (corresponding to $(1+\varepsilon)^{-a}s_i$ and $(1+\varepsilon)^{-a}t_i$, for $1 \le a \le L$). The idea is to split the problem into two subinstances after a small amount of guessing. The two subinstances handle the machines on the L vertical lines, and the machines not on them (and thus on the L horizontal lines) respectively. This reduces the instance to a problem where all machines lie on L axis-parallel lines, and we show a simple "sweep" algorithm for this case. We now present the details.

Let the dominating machine have speed vector (s, t). By the canonical assumption, the instance in this case satisfies the property that all the machines and jobs lie on the L horizontal lines below $y = t$, and the L vertical lines to the left of $x = s$. (see Figure 1). We now show how to obtain a QPTAS in this case by *separating* the machines on horizontal and vertical lines.

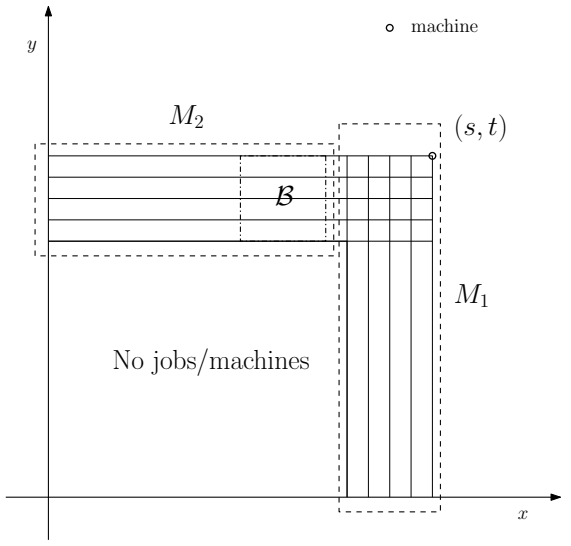


Figure 1: Unique pareto optimal machine

Let M_1 be the set of machines on the L vertical lines, as shown in the figure, and let M_2 be the machines on the horizontal lines, excluding the ones that are already in M_1. Next we partition the jobs. Note that all the jobs on the vertical lines *have* to be scheduled on the machines in M_1, or else the makespan would be more than 1. Thus we only have to decide the assignment of the jobs on the horizontal lines which are not on the vertical lines.

For these, we will guess two sets of integers, first of size L^2, which gives precisely the number of jobs of each type in the box $\mathcal{B}$ which will be scheduled on M_1, and another set of L integers, which is the number of jobs on the horizontal lines, but to the *left* of box $\mathcal{B}$ which we will schedule on M_1. Once we guess the *number* of such jobs, we choose to schedule the right-most of such jobs on the machines in M_1, because *all* these jobs are X-small for machines in M_1. (This is very similar to the argument for the recursive algorithm in the general case).

Thus, having guessed $L^2 + L$ integers, we have reduced the problem to that in which all the machines are on L axis-aligned lines, all vertical, or all horizontal. We will see how to handle these in the next section.

To check feasibility of the instance, it is sufficient to try all possible ways of splitting the jobs in the box $\mathcal{B}$ and the number of jobs outside of $\mathcal{B}$ handled by M_1 for each horizontal line, and check if both the resulting sub instances are feasible. If so, we get a makespan $(1+\varepsilon)$ schedule. If for every splitting, one of the resulting instances is infeasible, we can declare infeasibility.

4.3 Scheduling on r axis-aligned lines We now develop a QPTAS for a special case of $\mathsf{LRS}(2)$, in which all the jobs as well as all the machines lie on a set of r axis-aligned lines, which are either *all horizontal* or *all vertical*. As before let $L := \log mn/\varepsilon$. The algorithm will then run in time $n^{\mathrm{poly}(r,L)}$. As before, let the optimal makespan be 1.[4]

Without loss of generality, let us assume the lines are vertical, and let them be denoted $\ell_1, \ell_2, \ldots, \ell_r$. The case of horizontal lines can be handled in a symmetric fashion. The algorithm once again is based on dynamic programming. The idea is to do a *sweep* from top to bottom: we sort the machines in the order of decreasing y-coordinate (breaking ties arbitrarily), and 'process' them in this order (i.e., we schedule jobs on them). We begin with the following easy observation about OPT.

Observation 3. *Since the OPT makespan is* 1*, all jobs located above a machine* (s,t) *(i.e., job* (p_j, q_j) *such that* $q_j > t$*) have to be scheduled on machines above* (s,t) *(i.e.,* (s_i, t_i) *such that* $t_i \geq t$*).*

This is because the assignment cost of scheduling a job geometrically located above a machine (s,t) is strictly greater than 1, violating our assumption that the OPT makespan is 1. We thus ensure in our DP that when we "arrive" at a machine (in the sweep algorithm), all jobs above it (w.r.t the y-coordinate) have been scheduled . Therefore, to completely characterize the sub instance, it suffices to define the set of jobs which are below the machine, that are yet to be scheduled. We now argue how to do this using a vector of $rL + r$ integers in $[1, n]$.

State Information. At each step, we will maintain the following information: (a) the index of the machine we are scheduling jobs on, which defines the current 'height' t, i.e., the y coordinate of that machine; (b) a set of $r \times L$ integers, which is the number of jobs left to be done in each of $\ell_1, \ldots, \ell_r$, in the L horizontal levels below t; (c) the number of jobs on each of the r lines that are more than L horizontal levels below t. For type (c), it suffices to maintain just the *number* of such jobs, because we may in fact assume that the jobs remaining are precisely the ones with the *least* y-coordinates. This is because each of these jobs is Y-small for any of the machines we have processed so far, thus we could have scheduled those with the largest y-coordinates.

This is the dynamic program we will consider. Whenever we are at a machine i, the state information described above gives precisely the jobs left to be scheduled. We now guess $r \times L + r$ integers, which will let us decide which jobs we schedule on machine i, as well as the set of jobs we need to schedule on machines $i+1, \ldots, m$. For each guess, we check if the set of jobs thus determined for machine i can indeed be scheduled with makespan $(1+\varepsilon)$, and that the remaining set of jobs can be scheduled on the remaining machines. If for any guess, both sub instances are feasible, we have our makespan $(1+\varepsilon)$ schedule. If for all guesses, at least one of the sub instances is infeasible, we can declare infeasibility.

Running time. The size of the *table* in the dynamic program is $m \times n^{rL+r} = n^{O(rL)}$. Further updating an entry in the table depends on looking up n^{rL+r} entries of the table with a higher index i, and thus the dynamic programming algorithm runs in time $n^{O(rL)}$.

This completes the remaining base case of our dynamic programming algorithm. Thus in the recursive partitioning, every leaf can be evaluated in time $n^{O(L^2)}$, and every internal node can be evaluated, given its children, in time $n^{O(L^2)}$. Since we already bounded the

[4] Even after scaling, all the machines and jobs lie on at most $2r$ different lines.

total number of sub instance in our recursive partitioning by $n^{O(L^2 \log n)}$, we have established theorem 4.1.

4.4 Special Case: Sequential or Parallelizable Jobs Notice that we can cast the problem where each job is either sequential or parallelizable, as $\mathsf{LRS}(2)$, where the machines all line on *a vertical line*. Indeed, suppose a machine i is expressed as vector $(1, s_i)$ if it has s_i cores; A sequential job is expressed as a vector $(\sigma_j, 0)$ and a parallelizable job is expressed as $(0, \pi_j)$. For this special case, we can in fact modify and simplify the above QPTAS and obtain a true PTAS. To this end, we borrow inspiration from the PTAS for related machines [7], and perform a *bottom-up sweep* instead of a top-down sweep. By doing this, a job which is Y-small (coordinate less than $\varepsilon^2 s_i$) for a machine $(1, s_i)$ will remain Y-small for all subsequent machines $i' \geq i$.

So when we are at machine i, we will keep track of the exact number of jobs to be done for $O(1/\varepsilon)$ levels (w.r.t powers-of-$(1+\epsilon)$) below s_i. We will also keep track of the *total volume* of Y-small jobs which have already been scheduled on earlier machines. These will be for the parallelizable jobs. Again for sequential jobs, we will keep track of the exact number of jobs to be done for $O(1/\varepsilon)$ levels (w.r.t powers-of-$(1+\epsilon)$) to the left of machine i, and a total volume of X-small jobs which have already been scheduled.

5 Open Problems

We leave open several intriguing questions. The first one of course is the classification of the rank three case of the problem. Our algorithm for the rank two case is polynomial for many interesting cases, such as when the aspect ratio between machines is polynomially bounded. It seems natural to attempt to get a real PTAS for the rank two case. Breaking the factor two barrier for the general problem is long open, and several special cases have been studied for this problem. Designing better than two approximation algorithms for constant rank is a worthwhile special case to consider. Finally the low rank assumption seems like a natural one, and it would be interesting to study other scheduling problems with this restriction.

Acknowledgments We thank Anna Karlin for suggesting to us the problem of scheduling with low rank processing times. We express our gratitude to Anupam Gupta for useful discussions in the early stages of this work.

References

[1] Sanjeev Arora, Rong Ge, Ravi Kannan, and Ankur Moitra. Computing a nonnegative matrix factorization - provably. In *STOC*, 2012.

[2] Chandra Chekuri and Sanjeev Khanna. On multi-dimensional packing problems. *SIAM J. Comput.*, 33(4):837–851, April 2004.

[3] Tomáš Ebenlendr, Marek Křćal, and Jiří Sgall. Graph balancing: a special case of scheduling unrelated parallel machines. In *Proceedings of the nineteenth annual ACM-SIAM symposium on Discrete algorithms*, SODA '08, pages 483–490, Philadelphia, PA, USA, 2008. Society for Industrial and Applied Mathematics.

[4] Leah Epstein and Tamir Tassa. Vector assignment schemes for asymmetric settings. *Acta Inf.*, 42(6-7):501–514, 2006.

[5] M. R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[6] Dorit S. Hochbaum and David B. Shmoys. Using dual approximation algorithms for scheduling problems theoretical and practical results. *J. ACM*, 34:144–162, January 1987.

[7] 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.

[8] Jan Karel Lenstra, David B. Shmoys, and Éva Tardos. Approximation algorithms for scheduling unrelated parallel machines. In *FOCS*, pages 217–224, 1987.

[9] Ankur Moitra. A singly exponential time algorithm for computing nonnegative rank. Manuscript, 2012.

[10] Evgeny V. Shchepin and Nodari Vakhania. An optimal rounding gives a better approximation for scheduling unrelated machines. *Oper. Res. Lett.*, 33(2):127–133, 2005.

[11] Ola Svensson. Santa claus schedules jobs on unrelated machines. In *STOC*, pages 617–626, 2011.