Preemptive scheduling on selfish machines

Leah Epstein[1] and Rob van Stee[2]

[1] Department of Mathematics, University of Haifa, 31905 Haifa, Israel. lea@math.haifa.ac.il.
[2] Department of Computer Science, University of Karlsruhe, D-76128 Karlsruhe, Germany. vanstee@ira.uka.de.

Abstract. We consider the problem of scheduling on parallel uniformly related machines, where preemptions are allowed and the machines are controlled by selfish agents. Our goal is to minimize the makespan, whereas the goal of the agents is to maximize their profit. We show that a known algorithm is monotone and can therefore be used to create a truthful mechanism for this problem which achieves the optimal makespan. We extend this result for additional common goal functions.

1 Introduction

Internet users and service providers act selfishly and spontaneously, without an authority that monitors and regulates network operation in order to achieve some social optimum such as minimum total delay. Selfish behavior may affect the performance, and it is interesting to identify the problems in which this happens, and to find how much performance can be lost as a result of lack of coordination. Many algorithmic problems, in which we investigate the cost of the lack of coordination arise. The study of lack of coordination can be compared to the lack of information (that is assumed in online algorithms) or the lack of unbounded computational resources (assumed in polynomial time approximation algorithms).

There has been a large amount of previous research into approximation and online algorithms for a wide variety of computational problems, but most of this research has focused on developing good algorithms for problems under the implicit assumption that the algorithm can make definitive decisions which are always carried out. On the internet, this assumption is no longer valid, since there is no central controlling agency.

To solve problems which occur, e.g., to utilize bandwidth efficiently (according to some measure), we now not only need to deal with an allocation problem which might be hard enough to solve in itself, but also with the fact that the entities that we are dealing with (e.g. agents that wish to move traffic from one point to the other) do not necessarily follow our orders but instead are much more likely to act selfishly in an attempt to optimize their private return (e.g. minimize their latency).

Mechanism design is a classical area of research with many results. Typically, the fundamental idea of mechanism design is to design a game in such a way that truth telling is a dominant strategy for the agents: it maximizes the profit for each agent individually. That is, each agent has some private data that we have no way of finding out, but by designing our game properly we can induce them to tell us what that is (out of well-understood self-interest), thus allowing us to optimize some objective while relying on the truthfulness of the data that we have. This is done by introducing *side payments* for the agents. In a way, we reward them (at some cost to us) for telling us the truth. The role of the mechanism is to collect the claimed private data (bids), and based on these bids to provide a solution that optimizes the desired objective, and hand out payments to the agents. The agents know the mechanism and are computationally unbounded in maximizing their utility.

A seminal paper by Archer and Tardos[4] considered the general problem of one-parameter agents. The class of one-parameter agents contain problems where any agent i has a private value t_i and his valuation function has the form $w_i \cdot t_i$, where w_i is the work assigned to agent i. Each agent i makes a bid b_i depending on its private value and the mechanism, and each agent wants to maximize its own profit.

The paper [4] shows that in order to achieve a truthful mechanism for such problems, it is necessary and sufficient to design a *monotone* algorithm, and use a payment function of the form

$$P_i(b_{-i}, b_i) = h_i(b_{-i}) + b_i w_i(b_{-i}, b_i) - \int_0^{b_i} w_i(b_{-i}, u)\, du. \quad (1)$$

Here (b_{-i}, x) is the bid vector b where the element b_i has been replaced by x, the h_i are arbitrary functions, and $w_i(b_{-i}, x)$ is the work assigned to agent i if the bid vector is (b_{-i}, x).

An algorithm is monotone if for every agent, the amount of work assigned to it does not increase if its bid increases. More formally, an algorithm is monotone if given two vectors of length m, b, b' which represent a set of m bids, which differ only in one component i, i.e., $b_i > b'_i$, and for $j \neq i$, $b_j = b'_j$, then the total size of the jobs (the work) that machine i gets from the algorithm if the bid vector is b is never higher than if the bid vector is b'.

Using this result, monotone (and therefore truthful) approximation algorithms were designed for several classical problems, like scheduling on related machines to minimize the makespan, where the bid of a machine is the inverse of its speed [4, 2, 6, 1, 13], shortest path [5, 9], set cover and facility location games [7], and combinatorial auctions [14, 16, 3].

Problem definition In this paper, we consider the problem of scheduling jobs in a multiprocessor setting where jobs may be preempted, and where the performance measure is the makespan. The makespan of a given schedule is the time at which the last task finishes.

Preemption means that a job may be split into parts, which can be possibly assigned to distinct machines. A part of job of size p must be assigned to a time slot on one of the machines. The length of the time slot should be $\frac{p}{s}$ for a machine of speed s. The time slots assigned to the parts of one job on the different machines must all be disjoint.

We denote the number of processors by m and the number of jobs by n. We consider the version of this problem where the machines are related: each machine has a speed at which it runs, which does not depend on the job being run.

Denote the size of job j by p_j ($j = 1, \ldots, n$). Denote the speed of machine i by s_i ($i = 1, \ldots, m$). In our model, each machine belongs to a selfish user. The private value (t_i) of user i is equal to $1/s_i$, that is, the cost of doing one unit of work. The load on machine i, L_i, is the total size of the jobs assigned to machine i divided by s_i. The total work assigned to a machine i, denoted by W_i, is the total size of jobs assigned to it, i.e., $W_i = s_i \cdot L_i$. The profit of user i is $P_i - L_i$, where P_i is the payment to user i by the payment scheme defined by (1).

Our goal is to minimize the makespan. The classical version of this problem can be solved in polynomial time [12, 11, 17, 8]. As is generally the case in algorithmic mechanism design, we are not interested in maximizing the total profit of the users.

As mentioned above, in order to imply a truthful mechanism, we need to show an algorithm for which an increase in a speed of a machine does not reduce the amount of work it receives.

Our results We show that the algorithm given by Epstein and Tassa[10] is in fact monotone and can therefore also be used in this setting. We describe the algorithm which computes *the load* of every machine. The algorithm which creates the actual assignment is omitted, since we are only interested in the loads of machines and not in the exact assignment. This algorithm can be found in [10] as well.

Note that even though in principle idle time is allowed, the algorithm does not create idle time on any machine. The algorithm actually creates a *strongly optimal* schedule, in the sense that not only the maximum

load is minimized, but also every subsequent load is minimized after the larger loads have been fixed. We thus show that it is possible to achieve an optimal makespan even with selfish agents.

We extend this result to preemptive scheduling with the goal of minimizing the ℓ_p norm of the loads vector, for $1 \leq p < \infty$. This is again done by using the algorithm of [10] for the cases $1 < p < \infty$, and a simple algorithm for $p = 1$.

Throughout the paper, we assume that the jobs are sorted in order of non-increasing size ($p_1 \geq p_2 \geq \ldots \geq p_n$), and the machines are sorted in a fixed order of non-decreasing bids (i.e. non-increasing speeds, assuming the machine agents are truthful, $s_1 \geq s_2 \geq \ldots \geq s_m$). In case of ties, i.e., machines of identical speeds, each machine also carries an identifier (a number in $\{1, \ldots, m\}$), and a set of machines with the same speed are ordered in a increasing order of identifiers. We call the order implied by the identifiers a *lexicographical ordering* of the machines.

2 Makespan minimization (ℓ_∞)

2.1 Algorithm

The algorithm is based on the following m lower bounds on the optimal makespan, given already by Liu and Yang[15] for a special case.

- For $k = 1, \ldots, m-1$,
$$\sum_{i=1}^{k} p_i / \sum_{i=1}^{k} s_i \ ,$$
That is, the total size of the k largest jobs, divided by the sum of k largest speeds.
- The last lower bound is
$$\sum_{i=1}^{n} p_i / \sum_{i=1}^{m} s_i \ ,$$
That is, the total size of all the jobs, divided by the sum of the speeds.

It is known that the maximum of all these bounds equals the optimal makespan [12, 11, 17, 8]. The algorithm below is presented in [10]. The algorithm repeatedly executes the following steps until all jobs are assigned. Based on the bounds above, it determines a value k (an index of a machine) which determines the smallest maximum load that can be achieved for the remaining machines. If $k = m$, it assigns the jobs to the machines so that the load on all machines is equal and halts. Otherwise, it assigns the k largest jobs to k fastest machines (this set of machines will be called a *group*).

We use the following notations.

$$P_k = \begin{cases} \sum_{j=1}^{k} p_j & 1 \leq k \leq m-1 \\ \\ \sum_{j=1}^{n} p_j & k = m \end{cases} \ ,$$

and

$$S[a:b] = \sum_{i=a}^{b} s_i .$$

Algorithm 1

1. *Set $t = 0$ and $k_t = 0$ (at each stage k_t equals the number of values W_j that were already determined).*
2. *For every $k_t + 1 \leq k \leq m$, compute*

$$q_k = \frac{P_k - P_{k_t}}{S[k_t + 1 : k]} ,$$

   and set k_{t+1} to be the (minimal) value of k for which q_k is maximal. The set of machines $\{k_t + 1, \ldots, k_{t+1}\}$ is defined to be the $t + 1$-th group.
3. *For all $k_t + 1 \leq j \leq k_{t+1}$, set*

$$W_j = s_j \cdot \frac{P_{k_{t+1}} - P_{k_t}}{S[k_t + 1 : k_{t+1}]} .$$

4. *If $k_{t+1} < m$ set $t = t + 1$ and go to Step 2.*

This algorithm is optimal. In our analysis, we will use the following property.

Lemma 1. *[10] The loads are monotonically non-increasing as a function of the machine indices. Within a group, loads are identical.*

The following corollary holds since machines are sorted by speed.

Corollary 1. *The work assigned to machines is a monotonically non-increasing function of the machine indices.*

Proof. Consider machines i and $i + 1$ in the sorted list, for some $1 \leq i \leq m - 1$. By Lemma 1, $L_i \geq L_{i+1}$. We have $s_i \geq s_{i+1}$ and so

$$W_i = s_i \cdot L_i \geq s_{i+1} L_{i+1} = W_{i+1}.$$

□

2.2 Monotonicity

In this section, we prove the following theorem.

Theorem 2. *Algorithm 1 is monotone.*

We number the groups in the order of creation by the algorithm. We use additional notations. Let $s_a(b)$ be the speed of the a-th machine in group b, we use (a, b) to denote this machine. Let $S_a(b)$ be the sum of the a largest speeds among machines in groups $b, b + 1, \ldots$. If group b consists of at least a machines, then this is actually the sum of a largest speeds of machines in this group. Let $p_a(b)$ be the size of the a-th largest job remaining after the first $b - 1$ steps of the algorithm, and let $P_a(b)$ be the total size of the a largest such jobs.

We consider the situation where one machine (the j-th machine of group g) becomes faster, that is, decreases its bid. We denote the new speed of this machine by $s'_j(g)$ and we let

$$\varepsilon = s'_j(g) - s_j(g) > 0.$$

We use $S'_a(b)$ to denote the sum of the a largest speeds in groups $b, b + 1, \ldots$, after the speed change. All other bids remain unchanged. We call the instance with the original speed the *original instance*, and the instance with the changed speed the *new instance* or the *modified instance*.

The following lemmas reduces the number of cases to be considered.

Lemma 2. *Consider machine (j, g) of the original instance that changes its speed. If the new location of this machine is later than the machines of groups $1, \ldots, h$ of the original instance (for some $h \leq g-1$), the groups $1, \ldots, h$ created by the algorithm for the new instance consist of the same machines as created for the original instance.*

Proof. Assume by contradiction that the algorithm does not act in the same way on the first h groups, and let $1 \leq c \leq h$ be the first group that is different for the new instance compared to the original instance. Let k be the number of machines in group c for the original instance and let k' be the number of machines for the new instance, where $k \neq k'$. Since the algorithm chose a group with k machines for the original instance, we have

$$\frac{P_k(c)}{S_k(c)} > \frac{P_{k'}(c)}{S_{k'}(c)}$$

(since the algorithm chooses a group of minimal number of machines in case of ties). If $k' < k$, by the assumption above, the machines of groups $1, \ldots, c$ of the original instance are earlier in the ordering than the machines which changes its speed. Therefore, the k machines of group c of the original instance remain in the same location in the ordering and there is no change in the speeds of any machines of group c for both instances, and so $S_{k'}(c) = S'_{k'}(c)$. This derives an immediate contradiction since we get

$$\frac{P_k(c)}{S'_k(c)} = \frac{P_k(c)}{S_k(c)} > \frac{P_{k'}(c)}{S_{k'}(c)} \geq \frac{P_{k'}(c)}{S'_{k'}(c)},$$

which would imply our algorithm makes a group of size k instead of k'. Otherwise, if $k' > k$, we have $S_k(c) = S'_k(c)$ and $S_{k'}(c) \leq S'_{k'}(c)$, and the contradiction is derived similarly. □

Below we consider the cases where a machine that increases its speed either does not change its location in the sorted list of machines, or changes places with its predecessor in the sorted list. The following lemma shows that these cases are sufficient to prove that the algorithm is monotone.

Lemma 3. *If there exists an instance for which a machine increases its speed and is allocated less work by the algorithm as a result, then there exists such an instance where as a result of the speed change the machine does not change its location in the sorted list, or appears just one place earlier.*

Proof. Assume that there exists an instance which disproves monotonicity. We may assume that the machine which changes its speed moves to a location which is at least two places earlier in the sorted list, as a result.

We split the process of change in speed into several phases. Let z be the index of the machine which changes its speed in the ordering for the original instance and $z' < z$ its location for the new instance. A single phase consists of an increase of speed for a machine until it changes places with the machine before it. There are $z - z'$ such phases, and thus we consider $z - z' + 1$ instances, starting with the original instance, and considering also every instance that results from each additional phase.

Since the machine that changed its speed gets a smaller amount of work, there must exist at least one phase in which its work decreases. The instances which is defined just before this phase, with the speed change that results in the instance just after this phase prove the claim. □

We therefore need to consider three cases. In the first two cases we assume that (j, g) is located in the ordering of the new instance later than all machines of groups $1, \ldots, g-1$ in the original instance.

Case 1.1 Machine (j, g) remains in the same group. By Lemma 2 this means that groups $1, \ldots, g-1$ remain unchanged as a result of the change in speed. We consider the case that group g contains machine j in both instances.

Let k denote the number of machines in group g for the original instance, and let k' be the number of machines in this group after the increase of speed. Note that all three cases $k < k'$, $k = k'$ and $k > k'$ are possible in principle.

The original work assigned to machine (j, g) is

$$s_j(g)\frac{P_k(g)}{S_k(g)} .$$

The work assigned to this machine after the speed change is

$$s'_j(g)\frac{P_{k'}(g)}{S'_{k'}(g)} .$$

We show that

$$\frac{P_{k'}(g)}{S'_{k'}(g)} \geq \frac{P_k(g)}{S'_k(g)}$$

holds. This statement is trivial in case $k = k'$. For $k \neq k'$ it follows from the choice of the algorithm to create a group of k' machines and not of k machines. Therefore,

$$s'_j(g)\frac{P_{k'}(g)}{S'_{k'}(g)} \geq s'_j(g)\frac{P_k(g)}{S'_k(g)} .$$

We have that

$$s'_j(g)\frac{P_k(g)}{S'_k(g)} \geq s_j(g)\frac{P_k(g)}{S_k(g)},$$

since

$$\frac{s_j(g)+\varepsilon}{(S_k(g)+\varepsilon} \geq \frac{s_j(g)}{S_k(g)} \text{ for } s_j(g) \leq S_k(g),$$

which obviously holds because $S_k(g) = \sum_{i=1}^{k} s_i(g)$ and j belongs to the g-th group, i.e., $1 \leq j \leq k$.

Case 1.2 We now consider the case where machine (j, g) does not remain in the same group. By Lemma 2 groups $1, \ldots, g-1$ still remain unchanged as a result of the change in speed, but group g changes in a way that it does not contain machine (j, g) of the original instance. This machine becomes a part of a later group $c > g$.

This situation is possible. The increase of a speed of a machine in the group has the following effect. The lower bounds for the maximum load decrease starting from this machine. The bound that determined the last machine (k) of the group g may no longer be a maximum. In this case, one or more new groups are formed before the one that contains j. Since we assume that the machine which changed its speed is located in the ordering of the new instance later than all machines of groups $1, \ldots, g-1$ in the original instance, this situation can only mean that the length of the g-th group has decreased, since in the new ordering, machine (j, g) cannot appear later than in the original ordering (by the definition of the sorting, where ties are broken in a consistent way).

For an example, see Figures 1, 2 and 3. The set of jobs is $\{J_1, \ldots, J_6\}$, and their sizes are 110, 70, 55, 18, 9 and 9, respectively. The output of the algorithm on the original set of speeds (20, 10, 10, 6 and 3) is

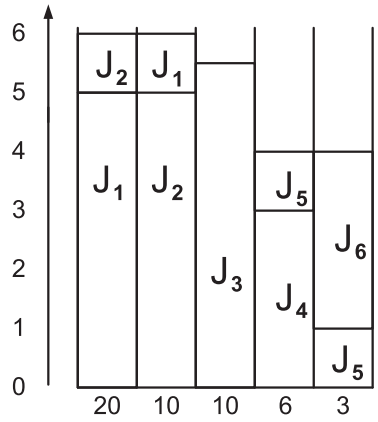

Fig. 1. The output for the original input

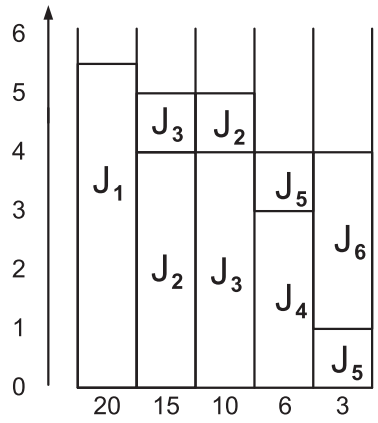

Fig. 2. The output after the second machine changes its speed

given in Figure 1. Figure 2 shows the change in the output after the second machine increases its speed to 15. Figure 3 shows the change in the output after an additional change of speed, in which the fifth machine increases its speed to 3.

Note that we may assume that group g is the first group. Since for both instances the algorithm creates the same groups $1, \ldots, g-1$, running the same jobs, we can omit these machines and jobs from the instance. Thus we assume that machine 1 is the first machine of group g, and number the machines starting from the first machine of group g before the speed change, and the jobs excluding the jobs that are scheduled on the machines of groups $1, \ldots, g-1$.

Let k' be the last machine of the last group before (j, g) (in its new location) after the speed change. Let k'' be the last machine of the group which contains (j, g) after the speed change.

Case 1.2.a: Machine (j, g) remains in the same location in the ordering.

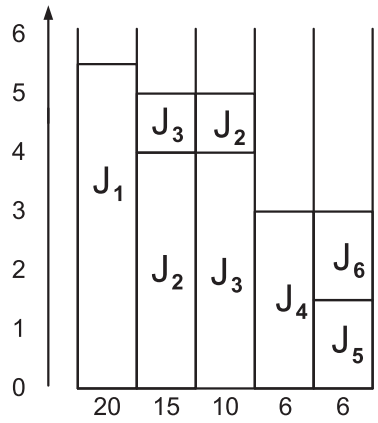

Fig. 3. The output after the fifth machine changes its speed as well

Given the original instance (without the jobs we omitted as described above), denote the total size of the k' largest jobs by L, and the remaining total size of jobs assigned to machines $1 \ldots, k$ by M (that is the difference between the size of all jobs assigned to these machines, and L). Denote the total size assigned to machines $k'+1, \ldots, k''$ after the speed change by N.

We define similarly the total speeds of these three groups of machines as S, T, and U, where S is the total speed of machines $1, \ldots, k'$, T is the total speed of machines $k'+1, \ldots, k$, and N is the total speed of machines $k'+1, \ldots, k''$. We denote the speed of the machine which changes its speed by s and the new speed by $s' = s + \varepsilon$.

We clearly have $T \geq s$, since the machine of this speed is in the set $\{k'+1, \ldots, k\}$. We also let $T' = T + \varepsilon$ denote the total speed of machines in $\{k'+1, \ldots, k\}$ after the speed change, and by $U' = U + \varepsilon$ the total speed of machines in $\{k'+1, \ldots, k''\}$.

Since the algorithm chose a group g with machines $1, \ldots, k$ for the original instance, and not $1, \ldots, k'$, we have

$$\frac{L}{S} \leq \frac{L+M}{S+T},$$

implying

$$LT \leq MS. \tag{2}$$

After the speed change, when the algorithm examines the set of machines $k'+1, \ldots, m$, it chooses the index k'' rather than k (although it may be that $k'' = k$), so we have

$$\frac{N}{U+\varepsilon} \geq \frac{M}{T+\varepsilon}. \tag{3}$$

The work assigned to the machine which changes its speed, before the change, is $\frac{s(L+M)}{S+T}$, and after the change, it is $\frac{s'N}{U'}$, thus we would like to show

$$\frac{s(L+M)}{S+T} \leq \frac{s'N}{U'}.$$

Using (3), it is enough to show that

$$\frac{(s+\varepsilon)M}{T+\varepsilon} \geq \frac{s(L+M)}{S+T}.$$

This holds when

$$\begin{aligned} Ms(S+T) + M\varepsilon(S+T) &\geq s(L+M)(T+\varepsilon) \\ &= (L+M)sT + (L+M)s\varepsilon, \end{aligned}$$

or

$$s(MS-LT) + \varepsilon(M(S+T) - (L+M)s) \geq 0$$

which holds because $MS \geq LT$ and

$$M(S+T) - (L+M)s \geq M(S+T) - (L+M)T,$$

so

$$M(S+T) - (L+M)s \geq 0$$

holds when

$$M(S+T) \geq T(L+M),$$

or

$$MS \geq LT,$$

which is true by (2).

Case 1.2.b: The location of the machine that changed it speed (for the new instance) is one place before its location for the original instance. In this case we again assume that the speed changes gradually, and split the change in speed into three parts, the increase before the change of location, the swap, and an additional increase. We only need to consider the swap, which happens when the pair of machines have the same speed, or just after that. By Corollary 1 we have that the swap can only increase the work of the machine that becomes faster.

Case 2 Machine j is now a part of the previous group. By Lemma 2, this can only happen if the speed of machine j becomes larger or equal to the speed of the slowest machine in the group $g-1$ of the original instance, and j changes its location in the sorted list of machines. By Lemma 3 we need to consider the case where it moves to a location that is just before its previous location.

We consider a process in which the speed of the machine increases gradually, and partition the speed increase into before the swap, the swap, and after the swap. Denote the two locations that we consider by $i, i+1$.

The proofs of cases 1.1 and 1.2 covers the speed changes before and after the swap, thus we only need to consider the swap, which takes place when the two machines have the same speed, or just after that. This means that the machines changed roles, and thus the machine that used to be in location $i+1$ now gets the work $W_i \geq W_{i+1}$, by Corollary 1.

3 Other norms

We start with the simple case of the ℓ_1 norm. In this case it is noted in [10] that the jobs are assigned to the set of fastest machines, that is, let b be a maximal index such that $s_1 = \ldots = s_b$, then all the jobs are assigned to the first b machines. We use a specific variant of this algorithm that assigns all jobs to machine 1. It is straightforward to see the following. We call this algorithm *Algorithm 2*.

Proposition 1. *Algorithm 2 is monotone.*

We next consider the other cases $(1 < p < \infty)$. We use the terminology of [10] and define

$$S_p[a:b] = \sum_{i=a}^{b} s_i^{\frac{p}{p-1}}.$$

The algorithm works as follows.

Algorithm 3

1. *Set $t = 0$ and $k_t = 0$ (at each stage k_t equals the number of values W_j that were already determined).*
2. *For every $k_t + 1 \leq k \leq m$, compute*

$$q_k = \frac{P_k - P_{k_t}}{S_p[k_t + 1 : k]},$$

   and set k_{t+1} to be the (minimal) value of k for which q_k is maximal. The set of machines $\{k_t + 1, \ldots, k_{t+1}\}$ is defined to be the $t+1$-th group.
3. *For all $k_t + 1 \leq j \leq k_{t+1}$, set*

$$W_j = s_j^{p/(p-1)} \cdot \frac{P_{k_{t+1}} - P_{k_t}}{S_p[k_t + 1 : k_{t+1}]}.$$

4. *If $k_{t+1} < m$ set $t = t + 1$ and go to Step 2.*

It can be seen that the algorithm acts in the same way that Algorithm 1 would work if the set of speeds were $s_1^{\frac{p}{p-1}}, \ldots, s_m^{\frac{p}{p-1}}$. Since we are not interested in the exact schedule but only in the work that each machine receives, in order to reduce to the proof in the previous section, we only need the following fact, that follows from $p > 1$.

Fact 1 *For a pair of speeds $s, s' > 0$, and $1 < p < \infty$ we always have $s' > s$ if and only if $s'^{\frac{p}{p-1}} > s^{\frac{p}{p-1}}$.*

We can thus prove the following.

Theorem 4. *Algorithm 3 is monotone.*

Proof. As mentioned above, Algorithm 3 simply runs algorithm 1 with adapted speeds. Thus we need to show that an increase of a speed s occurs if and only if an increase in "speed" $s^{\frac{p}{p-1}}$ occurs. This follows from Fact 1. □

4 Conclusion

We have shown that for a class of preemptive scheduling problems, it is possible to obtain truthful mechanisms simply by applying previously known algorithms. This is usually not the case for non-preemptive problems, which are typically NP-hard, whereas polynomial time approximation schemes lack the structure of optimal solutions. Note also that for makespan minimization in non-preemptive scheduling, if running time is not limited, it is possible to find an optimal algorithm which is monotone as follows. The machines are ordered by their lexicographical ordering. Given this ordering, the algorithm chooses an optimal schedule which has a smallest load vector (lexicographically) [4].

References

1. Nir Andelman, Yossi Azar, and Motti Sorani. Truthful approximation mechanisms for scheduling selfish related machines. In *Proc. of 22nd International Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 69–82, 2005.
2. Aaron Archer. *Mechanisms for Discrete Optimization with Rational Agents*. PhD thesis, Cornell University, 2004.
3. Aaron Archer, Christos Papadimitriou, Kunal Talwar, and Éva Tardos. An approximate truthful mechanism for combinatorial auctions with single parameter agents. In *Proc. of 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 205–214, 2003.
4. Aaron Archer and Éva Tardos. Truthful mechanisms for one-parameter agents. In *Proc. 42nd Annual Symposium on Foundations of Computer Science*, pages 482–491, 2001.
5. Aaron Archer and Éva Tardos. Frugal path mechanisms. In *Proc. of 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 991–999, 2002.
6. Vincenzo Auletta, Roberto De Prisco, Paolo Penna, and Giuseppe Persiano. Deterministic truthful approximation mechanisms for scheduling related machines. In *Proc. of 21st International Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 608–619, 2004.
7. Nikhil R. Devanur, Milena Mihail, and Vijay V. Vazirani. Strategyproof cost-sharing mechanisms for set cover and facility location games. In *ACM Conference on E-commerce*, pages 108–114, 2003.
8. Tomás Ebenlendr and Jiri Sgall. Optimal and online preemptive scheduling on uniformly related machines. In *Proc. of the 21st Annual Symposium on Theoretical Aspects of Computer Science (STACS2004)*, pages 199–210, 2004.
9. Edith Elkind, Amit Sahai, and Ken Steiglitz. Frugality in path auctions. In *Proc. of 15th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 701–709, 2004.
10. Leah Epstein and Tamir Tassa. Optimal preemptive scheduling for general target functions. *Journal of Computer and System Sciences*, 72(1):132–162, 2006.
11. Teofilo Gonzalez and Sartaj Sahni. Preemptive scheduling of uniform processor systems. *Journal of the ACM*, 25(1):92–101, 1978.
12. Edward C. Horvath, Shui Lam, and Ravi Sethi. A level algorithm for preemptive scheduling. *Journal of the ACM*, 24(1):32–43, 1977.
13. Annamária Kovács. Fast monotone 3-approximation algorithm for scheduling related machines. In *Proc. of 13th Annual European Symposium on Algorithms (ESA)*, pages 616–627, 2005.
14. Daniel J. Lehmann, Liadan O'Callaghan, and Yoav Shoham. Truth revelation in rapid, approximately efficient combinatorial auctions. In *ACM Conference on Electronic Commerce*, pages 96–102, 1999.
15. Jane W. S. Liu and Ai-Tsung Yang. Optimal scheduling of independent tasks on heterogeneous computing systems. In *Proceedings of the ACM National Conference*, volume 1, page 3845, 1974.
16. Ahuva Mu'alem and Noam Nisan. Truthful approximation mechanisms for restricted combinatorial auctions. In *Proc. of the 18th National Conference on Artificial Intelligence and 14th Conference on Innovative Applications of Artificial Intelligence (AAAI/IAAI)*, pages 379–384, 2002.
17. Hadas Shachnai, Tami Tamir, and Gerhard J. Woeginger. Minimizing makespan and preemption costs on a system of uniform machines. *Algorithmica*, 42(3-4):309–334, 2005.