# Scheduling theory prevails against powerful selfish clients

(Author names removed for anonymity.)

January 4, 2017

**Abstract**

You run a job processing machine. Every day, many clients send you jobs to complete. You process the jobs in a uniformly random order. Everyone is happy.

One day, you notice that many size-1 jobs are waiting for your machine to solve a size-1000 job. You code in an optimization: process the smallest job first. You tuck yourself into bed secure in the knowledge that you've decreased your clients' total waiting time.

A week later, you notice your machine isn't receiving large jobs anymore. In fact, every day, your machine processes more and more jobs, and those jobs are smaller and smaller. What's happened? Maybe you can figure it out before reading on.

Oh no! Your clients are selfish! They don't like waiting, and they've noticed your clever optimization. They're breaking up all of their jobs into the smallest possible pieces to force themselves through your queue first. Your noble intentions have paved the road to hell.

If only there were a *truthful* job scheduling system: a way to process jobs faster than uniformly at random, that doesn't allow an adversary to shorten their waiting time by cleverly breaking up their job into small pieces.

In this paper, we construct truthful job scheduling algorithms, and prove upper and lower bounds on the problem's competitive ratio in both an offline and an online setting.

## 1 Introduction

In [4], Nisan and Ronen first connected scheduling theory to algorithmic game theory. They introduced the idea of a *truthful* algorithm:

**Definition.** A job scheduling algorithm is *truthful* if no client can improve their job's expected completion time by dishonestly manipulating their job's parameters.

How can an adversarial client dishonestly manipulate their job's parameters? A client can overreport their job's running time, e.g. submit a 1-minute job claiming that it takes 10 minutes. (However, we do not allow underreporting, since such lying is easily detectable; we can punish the lie by simply stopping the job after its reported time is up.) Also, in a model with release times, a client can wait before releasing their job.

Other authors have extensively studied these so-called *basic adversaries*. See [1, 2].

In this paper, we introduce a more powerful adversary called a *parallel adversary*. These clients, in addition to padding their job length and (when applicable) releasing later than necessary, can split up their job into many independent sub-jobs which can be executed in parallel. Specifically, a client with a job $J$ of length $t$ can instead submit many sub-jobs $j_1, j_2, \ldots, j_i$ of total length at least $t$. The job processing server does not know which jobs came from the same client. Each client selfishly wants to minimize their completion time, i.e., the maximum completion time of any of their sub-jobs.

Parallel adversaries pose a difficult challenge. A naive algorithm, such as shortest first, incentivizes clients to split their jobs into the smallest chunks possible.

We believe our parallel adversary model to be more realistic in most contexts than the basic adversary model. However, one issue with our model is that we have only formulated it in the case where the job

processing server is a single machine. When the job processing server has many machines, it is natural to *want* clients to break their jobs up into small pieces to be processed in parallel. So the truthfulness condition, that clients cannot improve their expected completion time by breaking their job up into many pieces, is unrealistic. It would be an interesting extension of this paper to define "truthful" for parallel adversaries in the multi-machine case.

In this paper, we analyze two settings for parallel adversaries:

- *Offline* – All jobs are known and available to process at the starting time $t = 0$.
- *Online* – Jobs are received at arbitrary times.

We prove the following results:

- In the offline setting, there is a 2-competitive truthful algorithm.
- In the online setting, there is a 6-competitive truthful algorithm.
- In both the offline and online settings, there is no 1.1292-competitive truthful algorithm.

## 2 Related Work

The following is a survey of related work in truthful scheduling theory. This section is not necessary for understanding our new results.

In [2], the authors consider basic adversaries in a multiple-machine model. Unlike us, they wish to minimize the makespan (the time at which all jobs have been completed) rather than the total waiting time. They provide randomized, truthful, $\frac{3}{2}$-competitive algorithms for the problem of minimizing the makespan. The rough idea of their approach is to first compute (in non-polynomial time) an optimal schedule, $OPT$, and then return either $OPT$ or $OPT^{mirror}$, each with probability $\frac{1}{2}$. Here $OPT^{mirror}$ is essentially a reflection of $OPT$ over its max completion time, though the details vary for each specific case. Truthfulness follows from the fact that each job's expected completion time depends only on the makespan of $OPT$, and the job's execution time – so all adversaries will aim to make $OPT$ as good as possible, and thus act truthfully. Moreover, since $OPT^{mirror}$ has at most twice the makespan of $OPT$, the algorithm is $\frac{3}{2}$-competitive.

The authors of [2] also consider the distinction between the weak and strong models of completion. In the weak model of computation, a job which pads its runtime with garbage work needs to wait for the garbage work to finish before being considered complete. In the strong model, the job is considered complete once all the real work is finished. We assume the weak model of completion in our discussion for the sake of clarity, but all of our results can be trivially generalized to the strong model as well.

In [1], Angel et al. investigate the problem of minimizing the sum of weighted completion times on possibly many machines, where each client reports their job's running time and possibly weight, with the goal of being completed as early as possible. Angel's model is different from the basic adversary model in that jobs can both overreport and *underreport* their running time – for example, a job with running time 10 can claim that it only takes time 1 to process.

They begin with the case where the weights of the jobs are all public information, and therefore the jobs can't lie about their weights. Angel et al. demonstrate that if preemption – that is, running one job for some time, then deciding to switch to a second job before completing the first – is not allowed, then we cannot achieve any competitive ratio with the optimal solution in the case where the true running times and weights are all public. The intuitive reason for this is quite simple: any job can claim an arbitrarily small running time, and we may discover only after starting the job that the true running time was enormous, and that we should have run that job much later. This is easily corrected when preemption is allowed, because our algorithm can heavily punish jobs that claim a runtime less than the true runtime, by preempting the job after running for the claimed runtime. The algorithm then runs the jobs in order of their claimed runtimes, shortest first. With this algorithm it is easy to see that all jobs are incentivized to report their true running times, and it is therefore truthful, and yields the same solution as the optimum in the case where all true running times are public.

To modify their algorithm for the case where job weights are also private, Angel et al. show that the algorithm must accept payments from each job depending on their claimed weight in order to keep their algorithm truthful in this case.

In our work, we do not consider the case of weights; we assume all jobs have the same weight, so our minimization objective is simply the sum of completion times. As in Angel et al, jobs are still allowed to claim a runtime which is longer than their true runtime, since this assumption is very realistic – a job can simply pad itself with arbitrary garbage work at the end to achieve a longer runtime. The key difference between our work and that of Angel et al. is essentially that we modify the mechanism by which a job can pretend to have a shorter runtime than its true runtime, in order to be more faithful to reality, as described in Section 1.

# 3 A Truthful Offline Algorithm

From here on, we will view the jobs themselves as the selfish adversaries. So instead of saying e.g. "A client splits their job into parts," we will say e.g. "The job splits itself into parts to decrease its completion time."

We begin with a discussion of the offline case, in which all jobs arrive at the same time (i.e. all release times are 0). The goal of our algorithm is to minimize the sum of completion times. Each individual job's objective is to minimize (selfishly) its own expected completion time, i.e., the last completion time of its component sub-jobs.

**Theorem 1.** *There is a randomized truthful algorithm for this problem.*

We propose the following randomized algorithm, which we will refer to as the random stack algorithm. We pick a random job, choosing each job with probability proportional to its stated running time, and process that job *last*. We then iterate this to choose the second-to-last, third-to-last, etc, jobs. In this way we build our job processing queue in reverse order, i.e. we build a stack. At each step, a job is chosen to be pushed onto the stack with probability proportional to its stated runtime. Once the stack is built, we run the jobs by iteratively popping and running the topmost job in the stack.

*Claim.* Our proposed algorithm is truthful. That is, no job is incentivized to report a running time longer than its true running time, and no job is incentivized to split itself into sub-jobs.

*Proof of claim.* It is obvious that no job is incentivized to claim a runtime longer than its true runtime, since this would only increase the job's chance of being pushed into the stack at each step.

So it suffices to show that no job $J$ can, by splitting itself into two or more sub-jobs, decrease the expected time by which all of its sub-jobs will finish.

Observe that job-splitting does not change the sum total of running times across the sub-jobs associated with the original job $J$, nor does it change the total running time of the entire list of jobs. Observe also that the completion time of $J$ is determined by when the first of $J$'s sub-jobs is placed into the stack. Before any of $J$'s sub-jobs are pushed into the stack, at each step, the probability that one of $J$'s sub-jobs is pushed into the stack is proportional to the total running time of all of $J$'s sub-jobs. But this total is just the running time of the original job $J$. Therefore, no matter how it's split into sub-jobs, $J$ cannot change the probability that one of its sub-jobs is chosen to go last at each step. Therefore, holding all other jobs constant, no job $J$ can decrease its expected completion time by splitting itself into sub-jobs.

Finally, a job $J$ cannot benefit from lying in multiple ways (i.e. splitting into sub-jobs and then increasing the length of some of those sub-jobs), because we can iteratively remove one lie at a time using our above arguments without increasing the job's expected completion time. This completes the proof of our algorithm's truthfulness. □

*Remark.* In fact, we have proven an even stronger result. Namely, if a job $J$ splits itself up into sub-jobs of the same total length, then not only does it not change its expected completion time, it does not even change the *distribution* over its possible completion times.

## 3.1 Analysis of Competitive Ratio

Recall that we wish to minimize the sum of completion times. The optimal algorithm for this objective simply runs all jobs in order from shortest to longest. This greedy algorithm fails to be truthful because all

jobs are incentivized to split themselves into infinitesimally small sub-jobs.

**Theorem 2.** *Our proposed algorithm achieves a competitive ratio of at most 2. That is, its expected sum of completion times is at most twice that of the greedy algorithm.*

*Proof.* Suppose there are $n$ jobs $J_1, J_2, \ldots, J_n$ with associated positive runtimes $t_1 \leq t_2 \leq \cdots \leq t_n$.

First we analyze the optimal non-truthful greedy algorithm. This processes jobs in the order $J_1, \ldots, J_n$, yielding a total waiting time of $t_1 + 2t_2 + 3t_3 + \cdots + nt_n$. We will rewrite this sum as $\sum_{b \geq a} \min(t_a, t_b)$.

We now turn to our randomized algorithm. According to our algorithm, a job $J_a$ is placed into the stack with probability proportional to $t_a$ at each step, while a job $J_b$ is pushed into the stack with probability proportional to $t_b$ at each step. Therefore, given that one of $J_a$ or $J_b$ was chosen in a certain step, the probability that $J_a$ was chosen is $\frac{t_a}{t_a+t_b}$. So our algorithm will run job $J_b$ before job $J_a$ with probability $\frac{t_a}{t_a+t_b}$. Therefore, the expected time that job $J_a$ needs to wait while job $J_b$ runs is equal to this fraction multiplied by $t_b$, the running time of $J_b$. This is an expected wait of $\frac{t_a t_b}{t_a+t_b}$. Therefore, the total expected time that job $J_a$ waits before it can be started is $\sum_{b \neq a} \frac{t_a t_b}{t_a+t_b}$. We add $t_a$ to get $J_a$'s expected completion time, $t_a + \sum_{b \neq a} \frac{t_a t_b}{t_a+t_b}$. Summing this expression over all jobs gives our expected sum of completion times:

$$\begin{aligned}
\mathbb{E}\left[\text{total waiting time}\right] &= \sum_a t_a + \sum_{a \neq b} \frac{t_a t_b}{t_a + t_b} \\
&= \sum_{b \geq a} \frac{2 t_a t_b}{t_a + t_b}.
\end{aligned}$$

Interestingly, this is just a sum of harmonic means. To each term of the above sum, apply the inequality $\frac{xy}{x+y} \leq \min(x, y)$, valid for all positive real numbers $x$ and $y$. The result: our algorithm's expected sum of completion times is at most $\sum_{b \geq a} 2 \min(t_a, t_b)$. This is exactly twice the optimal sum of completion times. Thus, our algorithm is 2-competitive. □

It is not immediately clear what is the worst-case for our algorithm. We conjecture that our algorithm has competitive ratio better than 2, but we could not prove a better bound. The worst case we found is a competitive ratio of slightly more than 1.5 for the set of job sizes $\{\frac{1}{n^2} | 4 \leq n < 4000\}$. Determining the true competitive ratio would be an interesting further exploration.

## 3.2 Random Variable Formulation

Finally, we present an interesting alternative formulation of our algorithm. The constraint on truthfulness is essentially that any group of jobs with the same sum of runtimes should have the same expected waiting time for the last job in the group to complete. So it would be nice if for each job $J$ we could define a continuous random variable $X_J$, such that for any set $S$ of jobs with the same sum of runtimes, $\max_{J \in S}(X_J)$ is distributed identically. Then we could simply sample each random variable, and run jobs in order of their value of $X_J$.

A simple choice is to define $P(X_J \leq c) = e^{-tf(c)}$ for any real number $c$, where $t$ is the running time of job $J$, and $f$ is a decreasing function of $c$ with $f(c) \to 0$ as $c \to \infty$. Since this random variable depends only on the job's runtime, we will henceforth denote such random variables as $X_t$ for a job with runtime $t$.

Surprisingly, this new formulation is equivalent to our original random stack algorithm. To see this, first consider the case where all job sizes are integers. Then both formulations are equivalent to splitting every job of size $k$ into $k$ jobs of length 1, putting these length-1 jobs in random order, and then running the original jobs in order of their last length-1 job in the random ordering. (One can see that the random variable construction is equivalent to this using the property that $X_k$ is identically distributed to $\max(\underbrace{X_1, X_1, \ldots, X_1}_{k})$.) One can scale this argument to show the equivalence for rational job sizes as well, from which a continuity argument proves it for all real job sizes.

# 4 The Online Setting

We turn now to the online case. In this model, each job $J$ has a processing time $t$ and a release time $r$. It can lie about either of them, and it can break itself up into smaller jobs. Additionally, we only become aware of a job's existence at its (declared) release time, and must make our scheduling decisions without any knowledge of which jobs will be released in the future.

## 4.1 An Online Algorithm Allowing Preemption

In the easiest version of the problem, we allow preemption: that is, our schedule is permitted to pause a job before it finishes, and then resume it later. Like in the offline case, there is a natural greedy approach: always work on the job with the least remaining processing time required. This gives an optimal solution. However, this algorithm is not truthful, as jobs have strong incentives to break themselves up.

How can we fix this? We will use the random variable formulation of our offline algorithm from Section 3.2, generalized to the online case. We will choose $f(c) = \frac{1}{c}$, yielding $\Pr[X_t \leq c] = e^{-t/c}$.

**Algorithm.** Whenever we receive a job $J$ with length $t$, assign it a *priority* $x_J$ sampled from the random variable $X_t$ with distribution defined by $\Pr[X_t \leq c] = e^{-t/c}$.
Always work on the uncompleted job with smallest priority. So when the algorithm receives a new job, it may preempt the currently running job.
We only need to make scheduling decisions when a job is released and when a job completes.

Before we dive into the truthfulness and competitiveness of this algorithm, let us note three things about it. First, if all release times are 0, the problem is exactly the same as the offline case, and our algorithm is exactly the same as well! Secondly, though our algorithm's priority queue is very similar to that of the (optimal) greedy algorithm, there is a subtle difference: in our case, priority is never recalculated. Even if a job is almost finished, it does not obtain a better priority, unlike in the greedy algorithm. Finally, like the greedy approach, our algorithm uses $O(n \log n)$ time to schedule $n$ jobs, because one can implement it with a priority queue.

We now show that our algorithm is truthful.

**Theorem 3.** *The proposed online algorithm is truthful.*

*Proof.* Jobs have no incentive to lie about their release times. Suppose job $J_i$, with true release time $r_i$, is considering claiming a release time $s_i > r_i$. This lie does not change the distribution of its priority (since it's generated from a random variable based only on its reported runtime). Moreover, it may cause job $J_i$ to miss out on some machine time that it would have received between $r_i$ and $s_i$, and its treatment from $s_i$ onwards is exactly the same in both cases. Thus, all jobs will provide accurate release times.

Jobs also have no incentive to lie about their processing time. For any fixed relative ordering of the other jobs, job $J_i$'s completion time depends only on its position in that ordering. Getting a smaller priority simply makes more blocks of time available to job $J_i$ if it needs them, and thus can only decrease its completion time. And as in the offline case, claiming a higher processing time is strictly worse for a job's priority: essentially, for any job $J$, declaring a longer processing time strictly increases the probability that $X_J > c$ for all positive reals $c$. So all jobs are incentivized to provide accurate processing times.

Finally, jobs have no incentive to break themselves up. Let us fix a job $J$. Job $J$ can choose to remain in one piece, with release time $R$ and processing time $T$. Call this Scenario 1. Alternatively, in Scenario 2, job $J$ breaks itself up into $m$ pieces $j_1, j_2, \cdots, j_m$ with processing times $t_1 + \cdots + t_m = T$ and all with release time $R$. As discussed in Section 3.2, the value of the priority $x_J$ is distributed in the same way as $\max(x_{j_1}, x_{j_2}, \ldots, x_{j_m})$. So it suffices to show that, given $x_{j_1} \leq x_{j_2} \leq \cdots \leq x_{j_m} = x_J$, and holding priority values of all other jobs constant, Scenario 1 and Scenario 2 will result in the same completion time for job $J$ or all of its sub-jobs respectively.

The key point is as follows: at any point in time, the total remaining runtime of jobs with priority value less than or equal to $x_J$ must be exactly the same in either scenario. (Recall that we have fixed the priority values of all jobs other than $J$, and $x_J$ is the priority of $J$ in Scenario 1 or the max across any of its sub-jobs in Scenario 2.) This is because that total only changes in two ways: it continuously decreases at a rate of 1 while we process jobs, and it increases whenever we receive a job with priority $\leq x_J$. These changes are the same in Scenario 1 and Scenario 2.

Therefore, if our algorithm eventually completes all jobs with priority less than or equal to $x_J$, it must do so at the same time $T_{\text{fin}}$ in either scenario. And since our algorithm completes the jobs of priority value $x_J$ only after all jobs with smaller priority, both Scenario 1 and Scenario 2 have completion time $T_{\text{fin}}$ when $x_{j_1} \leq x_{j_2} \leq \cdots \leq x_{j_m} = x_t$.

Furthermore, it's clear that jobs cannot benefit from lying in a multiple ways (e.g. giving a longer runtime, while also breaking itself up), because we can iteratively remove one lie at a time using our above arguments without increasing the job's expected completion time.

This completes the proof of our online algorithm's truthfulness. □

*Remark.* As in the offline case, we have actually proved a statement stronger than truthfulness. Namely, if a job $J$ splits itself into sub-jobs of the same total length, all with the same release time as $J$, then not only does $J$ not change its expected completion time, it does not even change the *distribution* of its completion time.

## 4.2 Analysis of Competitive Ratio

**Theorem 4.** *Our proposed online algorithm is 3-competitive when the objective is the sum of completion times.*

*Proof.* Let OPT denote the sum of completion times in the best possible schedule we could have made. We bound OPT in two different ways. First, we know that the completion time of job $J_i$ is at least $r_i + t_i$, so OPT is at least $\sum(r_i + t_i)$. Second, we know that lowering the release times only decreases the value of the optimal solution, so we consider lowering them all to 0. Then the optimal solution is to process the jobs in order of required runtime, which takes $\sum t_i + \sum_{i<j} \min(t_i, t_j)$ as shown in Section 3.1. Thus OPT is lower bounded by this quantity as well.

To bound the expected performance of our algorithm, we observe that once a job is released at time $r_i$, only jobs with lower priority can preempt it. Thus,

$$C_i \leq t_i + r_i + \sum_{j \text{ where } J_j \text{ has priority lower than } J_i} t_j.$$

Thus the expected value of $C_i$ can be computed just as in the offline case in Section 3.1, so the expected value of $\sum C_i$ is $\sum(r_i + t_i) + \sum_{i<j} \frac{2t_i t_j}{t_i + t_j}$. Observe that this bound is tight when all $r_i$ are 0 (in which case our algorithm is exactly equivalent to the offline version). Putting everything together, we see:

$$\mathbb{E}\left[\sum C_i\right] \leq \sum(r_i + t_i) + \sum_{i<j} \frac{2t_i t_j}{t_i + t_j} \leq \text{OPT} + 2\sum_{i<j} \min(t_i, t_j) \leq 3(\text{OPT}).$$

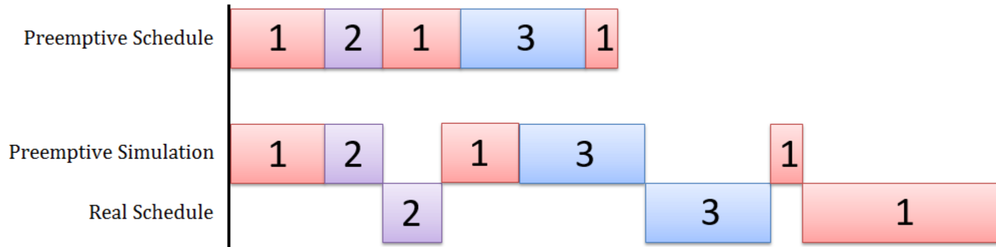
We conclude that our algorithm is 3-competitive, as claimed. □

Therefore, we have presented an online, truthful, polynomial-time, 3-competitive algorithm for $1 \,|r_i\,| \sum C_j$ in the case where preemption is allowed.

## 4.3 Modification For Non-Preemptive Case

Now, what happens if we do not allow preemption? We use a trick that converts any $k$-competitive algorithm using preemption to a $2k$-competitive algorithm that does not require preemption, thereby achieving a 6-competitive online algorithm that does not require preemption.

Consider any online preemptive schedule $S_0$. We will convert it to a non-preemptive schedule $S_1$ using at most twice as much time, as follows. We run a simulation of $S_0$. Whenever the simulated $S_0$ completes a job $J_i$ of time $T_i$, we pause the simulation, and complete $J_i$ for real. Then we unpause the simulation. Note that this algorithm can clearly be done online. See the diagram below for an example.




Finally, we add the detail that if a job is released at time $r$, then even though our non-preemptive algorithm becomes aware of the job at real time $r$, our algorithm doesn't take this information into account until the simulation reaches time $r$. Essentially we are artificially shifting the release time back to keep consistency with $S_0$.

**Lemma.** *$S_1$ is always at worst 2-competitive compared to $S_0$.*

*Proof.* We'll use the phrase "imaginary time" to mean the time within the simulation. For example, if the simulation runs for time 3, pauses for any amount of time, and then runs for time 4, we'll say that it is currently imaginary time 7, regardless of how much time has passed in the real world.

It suffices to show that the real time at which a job completes in $S_1$ is at most twice the imaginary time at which it completes, since the imaginary time is the time at which $S_0$ would have completed the job. But observe that the difference between the real and imaginary completion time for a job $J_i$ is just the runtime of $J_i$, plus the sum of runtimes for all jobs that completed before $J_i$ in $S_0$. It's clear that this difference is greater than or equal to the completion time of $J_i$ in $S_0$, because $J_i$ cannot complete in time less than its own runtime plus the sum of runtimes of jobs that are completed before it in $S_0$. (It can, however, complete in longer time, if some idle time was required due to release times, or if some other jobs were started but not completed.) Therefore, the difference between the real and imaginary completion time for a job $J_i$ is less than or equal to the imaginary completion time of $J_i$. Thus, every job has real completion time at most twice its imaginary completion time. The result follows. □

**Lemma.** *Our algorithm remains truthful.*

*Proof.* This follows easily from the fact that the difference between real and imaginary time for the completion time of a job $J_i$ is just the runtime of $J_i$, plus the sum of runtimes for all jobs that completed before $J_i$ in $S_0$, as described above. Jobs minimize this difference in time by minimizing the expected total runtime of jobs that run before them. But observe that this quantity is minimized by minimizing the job's declared release time and runtime, and is unaffected by splitting, as shown in the proof of Theorem 3. Therefore, the expected difference in real and imaginary time is minimized exactly when the job's expected completion time is minimized in the preemptive case, which is what the jobs were originally minimizing anyway. So the non-preemptive algorithm remains truthful. □

This demonstrates that our modification of a preemptive schedule to a non-preemptive schedule worsens our competitive ratio by at most a factor of 2 while remaining truthful. Therefore, using this trick, we can modify our preemptive 3-competitive algorithm to achieve a non-preemptive 6-competitive algorithm which is online and truthful, as claimed.

### 4.4 An Equivalent Random Stack-Based Formulation

Finally, we present the equivalent formulation of our online algorithm, obtained by generalizing our original random stack algorithm (which randomly places jobs in a stack with probability proportional to their size).

We store the order of all the jobs received so far (without discarding jobs from storage after completion). How do we insert a newly-released job into this order? Suppose that we have received $k$ jobs already, numbered $J_1, J_2, \cdots, J_k$, and given them the order $\sigma(1), \sigma(2), \cdots, \sigma(k)$. Suppose we now receive job $J_{k+1}$, with processing time $t_{k+1}$. To insert it, we do the following:

for each $i$ from $k$ downto 1:

with probability $\dfrac{t_{k+1}}{(t_{\sigma(1)} + t_{\sigma(2)} + \cdots + t_{\sigma(i)}) + t_{k+1}}$, insert job $k+1$ after job $\sigma(i)$ and return

insert job $k+1$ before job $\sigma(1)$ and return

Observe that by inserting a job into the stack this way, we match the probabilities from the offline case. That is to say, the distribution of orderings that the offline algorithm produces for given jobs $J_1, J_2, \cdots, J_k$ is the same as the distribution that the online algorithm produces when inserting the $J_i$ one by one. This can be seen by a trivial inductive argument. Again, note that we never remove completed jobs from the ordering (even though doing so would still preserve the algorithm's truthfulness), so that we get probabilities which match the offline setting.

Since the online generalizations of our two offline algorithms generate the same distributions of final orderings as their equivalent offline versions, the two formulations remain equivalent in the online case. However, note that in the random variable formulation of the online case, there is no need to keep track of previously completed jobs in order to preserve the same distribution of orderings as in the offline case; therefore, we can simply discard completed jobs from storage. Moreover, the natural priority queue implementation of the random variable formulation runs in $O(n \log n)$ time, while the online version of the equivalent random stack formulation as described here would require $O(n^2)$ time.

## 5 Lower Bounds with LPs

In Section 3, we showed a 2-competitive algorithm for the offline setting. Now, we will prove a lower bound:

**Theorem 5.** *There is no* 1.1292*-competitive algorithm in the offline setting.*

We will do this by formulating the problem as an LP, and then proving the following result by computer:

*Claim.* There is no 1.1292-competitive algorithm for our problem, even if the input jobs are all positive integers that sum to at most 10.

### 5.1 Formulating the LP with no artificial waiting

In the offline model, we receive a multiset of job lengths, which we will usually denote by $J$ throughout this section. For example, $J = \{1, 1, 6\}$ means receiving two jobs of length 1 and one job of length 6. We can process that jobset in any of three different orders: $(1, 1, 6)$, $(1, 6, 1)$, and $(6, 1, 1)$. This ignores strategies that insert artificial waiting time, e.g. "Process the length-6 job, wait 5 seconds, then process both length-1 jobs." We'll later show how to handle strategies that wait in between jobs, but for now, let's only consider strategies that never artificially wait.

Define $P_{(1,6,1)}$ to be the probability that, when given the jobset $\{1, 1, 6\}$, we choose to process them in the order $(1, 6, 1)$. Define $P_T$ for other tuples $T$ analogously.

We claim that we can express the problem "Find the best competitive ratio $R_{\text{ratio}}$" as an LP in the variables $P_T$ and $R_{\text{ratio}}$.

### 5.2 The constraints

First of all, probabilities are nonnegative and sum to 1:

$$\text{For every jobset } J, \quad \sum_{\text{orders } T \text{ of } J} P_T = 1. \text{ Also, all } P_T \geq 0.$$

Secondly, the algorithm must be $R_{\text{ratio}}$-competitive:

$$\text{For every jobset } J, \quad \sum_{\text{orders } T \text{ of } J} [\text{total waiting time for } T] \cdot P_T \leq R_{\text{ratio}} \cdot [\text{optimal waiting time for } J]$$

Finally, the algorithm must be truthful. For any jobset $J$ and any job length $j$, the client submitting $j$ cannot do better by splitting $j$ into any number of sub-jobs and/or padding extra runtime onto their jobs. That is, for every $J$ and every $j_1 + \cdots + j_k \geq j$,

$$\sum_{\substack{\text{orders } T \\ \text{of } J+\{j\}}} [\text{time until } T \text{ processes } j] \cdot P_T \quad \leq \quad \sum_{\substack{\text{orders } T \text{ of} \\ J+\{j_1,\ldots,j_k\}}} [\text{time until } T \text{ processes every } j_i] \cdot P_T.$$

This completes the LP for the case when there is no waiting in between jobs.[1]

### 5.3 Why wait? A motivating example

At first it seems useless to waste time by waiting. However, consider the following. Suppose you always receive the jobset $\{1, 2\}$. It is tempting to always process it in the order $(1, 2)$. But this is not truthful, because then the length-2 job can improve its completion time by splitting into two length-1 jobs.

If the length-2 job splits into two length-1 jobs, then you receive the jobset $\{1, 1, 1\}$. When you process these jobs, by symmetry, the length-2 job has a $1/3$ chance of having completion time 2, and a $2/3$ chance of having completion time 3. So, its expected completion time is $8/3$.

From this, one can calculate that to be truthful, one must process the jobs $\{1, 2\}$ in the order $(2, 1)$ at least $1/3$ of the time. It follows that the expected total waiting time for the jobset $\{1, 2\}$ must be at least $13/3$. Thus, the competitive ratio of any truthful algorithm on the input $\{1, 2\}$ must be at least $\frac{13/3}{4} = 13/12$.

...

Or maybe not!

What if we do this?

• When given the jobset $\{1, 2\}$, we process them in the order $(1, 2)$ with probability $5/6$, and otherwise process them in the order $(2, 1)$.

• When given the jobset $\{1, 1, 1\}$, we process two of the jobs, *wait for time 1/4*, and then process the last one.

One can check that this algorithm is truthful (for those two inputs) and has competitive ratio $25/24$. Wasting time actually helps!!

### 5.4 Handling algorithms that wait

The above LP does not handle algorithms that add artificial waiting times in between jobs.

It seems difficult to formulate an LP taking waiting into account. Our original LP exploits the fact that any jobset $J$ can only be processed in finitely many ways. So the set of mixed strategies for $J$ is simply the

[1] There is a bit of trickery with the last constraint. What is the "time until $(1, 6, 1)$ processes 1"? It is either 1 or 8, depending on which length-1 job we process first. We take the convention that equal-length jobs are indistinguishable, thus the expected time until $(1, 6, 1)$ processes 1 is 4.5. This symmetry issue can get confusing but adds no theoretical difficulty, so we recommend ignoring it on a first read of this paper.

convex polytope of all such configurations. But with waiting, there are uncountably many ways to process any nonempty jobset.

Surprisingly, we will show that waiting can still be handled with an LP.

Given any jobset $J$ and sub-multiset $S \subseteq J$, define $E_{J,S}$ to be the expected time, if we receive jobset $J$, for all jobs in $S$ to complete. Truthfulness is then equivalent to the condition that for all jobsets $J$ and all $j = j_1 + \cdots + j_k$, we have $E_{J+\{j\},\{j\}} \leq E_{J+\{j_1,\ldots,j_k\},\{j_1,\ldots,j_k\}}$. Without the possibility of artificial waiting, $E_{J,S}$ is a linear combination of the probabilities $P_T$. This made the original LP easy to write.

To handle waiting, we will show how to express $E_{J,S}$ as a linear combination of the $P_T$ and a function of the waiting times.

For any jobset $J$, any ordering $T$ of $J$, and any $0 < i < |T|$, define $w_{T,i}$ to be the expected time, conditioned on us receiving jobset $J$ and choosing to process it in order $T$, that the algorithm waits between jobs $i-1$ and $i$. Also define $w_{T,0}$ to be the expected time the algorithm waits before processing the first job in $T$.

Then each $E_{J,S}$ takes the form

$$
\begin{aligned}
E_{J,S} &= \sum_{\text{orders } T \text{ of } J} P_T \cdot \mathbb{E}\left[\text{time until } T \text{ processes every job in } S\right] \\
&= \sum_{\text{orders } T \text{ of } J} P_T \cdot \left(\text{time spent completing jobs before } S \text{ finishes} + \mathbb{E}\left[\text{time spent waiting before } S \text{ finishes}\right]\right) \\
&= \sum_{\text{orders } T \text{ of } J} P_T \cdot \left(\text{constant depending on } T + \text{sum of the } w_{T,i} \text{ that appear before the last job in } S\right).
\end{aligned}
$$

Observe that this last expression expresses $E_{J,S}$ as a linear combination of variables of the form $P_T$ and $(P_T w_{T,i})$. The only constraint on the variables $(P_T w_{T,i})$ is that they are nonnegative.

Using this method, we can solve our problem by writing an LP in the variables $P_T$, $(P_T w_{T,i})$, $E_{J,S}$, and $R_{\text{ratio}}$.

### 5.5 Results of the LP

All of these constraints define an LP in the variables $P_T$ and $R_{\text{ratio}}$. To find the best competitive ratio for our problem, one need only minimize $R_{\text{ratio}}$ in this LP. Unfortunately, the LP is infinite-dimensional, so we cannot solve it. Instead, we'll do the best we can: we'll pick finitely many jobsets $J$, and relax the LP to include only the constraints with those jobsets. Solving the relaxed LP, we thereby obtain a lower bound on $R_{\text{ratio}}$.

We considered only the jobsets $J$ where the job lengths are positive integers which sum to $\leq 10$. Even for only these jobsets, the LP has 2959 variables. Given the extreme computational difficulty of this method, it seems difficult to explore jobsets of total length much more than 10 without additional tricks.

Using the program lp_solve [3], we obtained the LP lower bound $R_{\text{ratio}} \geq 1.1292\ldots$. Therefore, there is no truthful algorithm for parallel adversaries in the offline case with competitive ratio 1.1292 or better, as this is less than the optimal $R_{\text{ratio}}$. This completes the proof of Theorem 5.

Of course, any lower bound on the offline setting also applies to the online setting.

## 6 Future Work

Our paper leaves several questions unanswered.

What is the best competitive ratio for parallel adversaries in the offline setting? In the online setting? The gap between our upper and lower bounds is quite large, especially for the online setting, so we anticipate that further progress can be made.

In addition, as we discussed, our competitive analysis of our algorithms may not be tight. Does our offline algorithm have a competitive ratio better than 2? Does our online algorithm have a competitive ratio better than 6?

We focused our efforts on the single-machine setting. Is there a good definition of truthful for parallel adversaries when the job processing server has multiple machines? In this setting, splitting jobs up into smaller pieces is often helpful for scheduling them, so our definition of "lying" no longer corresponds to undesirable behavior.

We could also analyze the case where jobs have different weights, where the objective becomes sum of *weighted* completion times. Jobs' weights could be either public or private; in the latter case, jobs would be able to lie about their weights, as in [1].

Finally, all of our algorithms share the undesirable feature that clients can decrease the expected completion time for a job by submitting multiple copies of it. In many practical cases, this is not a problem, since the user has to pay for all the additional copies. Nonetheless, the question remains: can we design algorithms that are also truthful with respect to such *copying adversaries*?

# References


[1] Eric Angel, Evripidis Bampis, Fanny Pascual, and Nicolas Thibault. *Truthfulness for the Sum of Weighted Completion Times*, pages 15–26. Springer International Publishing, Cham, 2016.

[2] Eric Angel, Evripidis Bampis, and Nicolas Thibault. Randomized truthful algorithms for scheduling selfish tasks on parallel machines. *Theoretical Computer Science*, 414(1):1–8, 2012.

[3] Michel Berkelaar and Jeroen Dirks et al. Lpsolve. http://lpsolve.sourceforge.net/5.5/.

[4] Noam Nisan and Amir Ronen. Algorithmic mechanism design. In *Proceedings of the thirty-first annual ACM symposium on Theory of computing*, pages 129–140. ACM, 1999.