When to Give up on a Parallel Implementation

Nathan S. Sheffield
MIT, Cambridge, MA, USA

Alek Westover
MIT, Cambridge, MA, USA

Abstract

In the Serial Parallel Decision Problem (SPDP), introduced by Kuszmaul and Westover [SPAA'24], an algorithm receives a series of tasks online, and must choose for each between a serial implementation and a parallelizable (but less efficient) implementation. Kuszmaul and Westover describe three decision models: (1) **Instantly-committing** schedulers must decide on arrival, irrevocably, which implementation of the task to run. (2) **Eventually-committing** schedulers can delay their decision beyond a task's arrival time, but cannot revoke their decision once made. (3) **Never-committing** schedulers are always free to abandon their progress on the task and start over using a different implementation. Kuszmaul and Westover gave a simple instantly-committing scheduler whose total completion time is 3-competitive with the offline optimal schedule, and proved two lower bounds: no eventually-committing scheduler can have competitive ratio better than $\phi \approx 1.618$ in general, and no instantly-committing scheduler can have competitive ratio better than 2 in general. They conjectured that the three decision models should admit different competitive ratios, but left upper bounds below 3 in any model as an open problem.

In this paper, we show that the powers of instantly, eventually, and never committing schedulers are distinct, at least in the "massively parallel regime". The massively parallel regime of the SPDP is the special case where the number of available processors is asymptotically larger than the number of tasks to process, meaning that the *work* associated with running a task in serial is negligible compared to its *runtime*. In this regime, we show (1) The optimal competitive ratio for instantly-committing schedulers is 2, (2) The optimal competitive ratio for eventually-committing schedulers lies in $[1.618, 1.678]$, (3) The optimal competitive ratio for never-committing schedulers lies in $[1.366, 1.500]$. We additionally show that our instantly-committing scheduler is also 2-competitive outside of the massively parallel regime, giving proof-of-concept that results in the massively parallel regime can be translated to hold with fewer processors.

2012 ACM Subject Classification Theory of computation → Online algorithms

Keywords and phrases Scheduling, Multi-Processor, Online-Algorithms

Digital Object Identifier 10.4230/LIPIcs.ITCS.2025.87

Related Version *arxiv*: `https://arxiv.org/abs/2408.16092`

1 Introduction

1.1 Background

Many computational tasks can be performed quickly in parallel over a large number of processors – but such parallel implementations may be less work-efficient than a serial implementation on a single processor, requiring substantially more total computation time across all machines. When several different tasks must be completed in as little total time as possible, this trade-off between work and time can necessitate running different tasks in different modes: small tasks can be done in serial to save work, while large tasks must be parallelized to prevent their serial runtimes from dominating the overall computation.

To formalize this problem, Kuszmaul and Westover introduced the **Serial Parallel Decision Problem (SPDP)** [16]. In their model, each task has exactly two possible implementations: an "embarrassingly parallel" implementation which can be worked on by

multiple machines at once (where the rate of progress on the implementation is proportional to the number of processors assigned to it), and a serial implementation which can only be worked on by a single processor at a time. If all tasks are available at time 0, it is easy to efficiently determine the optimal strategy: all jobs with serial completion time smaller than some threshold can be run in serial, and the larger tasks must be run in parallel. The model becomes interesting when previously-unknown tasks are allowed to arrive at arbitrary times, and one wishes to minimize the competitive ratio between the total completion time of an *online* algorithm compared to the offline optimal completion time.

Kuszmaul and Westover define three distinct versions of this model, parameterized by the degree to which the online scheduler is able to reverse its decisions.

1. An **instantly-committing** scheduler must choose an implementation for each task as soon as the task arrives, and is not allowed to revisit this choice.
2. An **eventually-committing** scheduler may delay choosing an implementation, but must choose one irrevocably before assigning its work to a processor.
3. A **never-committing** scheduler can, at any time, discard all as-yet completed work on an implementation and re-start the task with the other implementation.

The distinction between the eventually- and never-committing models is motivated by potential practical concerns: if a task involves mutating an input in memory, it may not be feasible to cancel an implementation once it begins running. Westover and Kuszmaul present an instantly-committing scheduler achieving competitive ratio 3, and show competitive ratio lower bounds of 2 and $\phi \approx 1.618$ in the instantly-committing and eventually-committing models, respectively (for deterministic schedulers). They conjecture that the ability to delay or cancel choices should allow for more competitive online algorithms, but leave open the problem of finding better competitive ratio upper bounds than 3.

1.2 This Work

In this work, we consider Kuszmaul and Westover's SPDP when the number of available processors is much larger than the number of tasks, noting that all of their upper and lower bounds hold in this parameter regime. This is a particularly simple setting, since the work associated to a serial implementation is now negligible compared to its completion time – running a task in serial means accepting a lower bound on completion time, but requires essentially no work. We can think of this setting as an unrelated-machines scheduling problem with an unlimited number of identical "slow" machines, and a single unrelated "fast" machine, representing a massively parallel implementation of the task across many processors – note that this could also describe scenarios with a literal fast machine, such as a single piece of accelerated hardware. Although we consider the number of processors to be large, we won't assume that tasks complete instantly when run on all the processors; instead, we'll let the total parallel work of each tasks be proportional to p.

Our main results are tight bounds on the competitive ratio of instantly-committing schedulers in this regime, and separations between the strength of all 3 models. Our results are summarized in Table 1. Note that we focus on deterministic schedulers (although we briefly discuss randomized schedulers in the appendix).

In each case, the upper bound comes from a simple heuristic in which the algorithm compares its projected completion time to its current estimate of the optimal completion time. More precisely, at each time t, our scheduler computes an offline optimal strategy "$\mathsf{opt}(t)$" on the truncation of the task sequence to tasks that arrive before time t, and makes decisions based on the completion time of $\mathsf{opt}(t)$.

Table 1 Main Results. * = this work.

Model	Lower Bound	Upper Bound
Instantly-Committing Schedulers	2 [16]	**2** *
Eventually-Committing Schedulers	1.618 [16]	**1.678** *
Never-Committing Schedulers	**1.366** *	**1.5** *

Our main technical contribution is the analysis of the schedulers. Working with $\mathsf{opt}(t)$ is challenging, because the schedules $\mathsf{opt}(t)$ and $\mathsf{opt}(t')$ can be quite different for $t \neq t'$. For instantly-committing schedulers we use an invariant-based approach to bound, at all times, the work taken by our scheduler in terms of the minimum work and completion time among all schedulers. For eventually- and never-committing schedulers this approach is no longer feasible: there is no well defined notion of the "work taken" by our scheduler, because the scheduler may not have committed to a decision yet. Instead, these analyses rely on choosing a couple of critical times to observe the state of our scheduler and $\mathsf{opt}(t)$, and then establishing a dichotomy: either (1) "real fast tasks" (tasks that opt runs on the fast machine) arrive quickly, in which case our scheduler prioritizes real fast tasks on the fast machine and will run most other tasks on slow machines, or (2) real fast tasks arrive slowly, in which case our scheduler never falls too far behind opt, despite making suboptimal use of the fast machine.

In addition to these results, we show that with some effort our instantly-committing scheduler can be adapted to work for any number of processors, fully resolving the question of the optimal competitive ratio of instantly-committing schedulers in the general SPDP, and giving a proof-of-concept that results for a large number of processors can be adapted to hold when the work associated with serial tasks is also a concern.

1.3 Related Work

There is a long line of work studying the phenomenon of work-inefficient parallel implementations in multi-processor scheduling. Typically, the models of limited parallelism considered involve one of three types of jobs:

1. **Rigid** jobs, which come with a number p_i specifying a fixed number of processors the job must be run on at each timestep of its execution.
2. **Moldable** jobs, where the scheduler may choose the (fixed) number of processors the job is run on, and the amount of work scales depending on this choice according to some **speedup curve**.
3. **Malleable** jobs, which like moldable jobs have an associated speedup curve, but where the job may be assigned to different numbers of processors at different timesteps (as opposed to the scheduler choosing a fixed value at the start of the task's runtime).

In each of these cases, there is interest in minimizing the total completion time (**makespan**) in both the offline setting – where problems tend to be NP-hard, but may have approximation algorithms [25, 20, 18, 24, 23] – and the online setting, where the goal is to minimize competitive ratio [8, 7, 4, 14, 9, 29, 28]. Kuszmaul and Westover's Serial Parallel Decision Problem is related to this line of work, but doesn't quite fit into the usual framework – in their model, instead of dealing with an arbitrary speedup curve, there is a single binary decision between a completely serial and perfectly parallelizable implementation.

As noted, the massively-parallel regime of the SPDP considered in this paper can be naturally viewed as a scheduling problem with an unlimited number of identical "slow" machines, and a single unrelated "fast" machine. Standard scheduling problems in the

unrelated machines model have also been well-studied, in terms of both offline approximation algorithms and hardness results [13, 17, 22, 26, 19, 21, 10, 15, 6], and online algorithms [3, 2, 22, 5, 1, 11, 12, 30]. We note, however, that since we treat "slow" machines as an unbounded resource, and there is only a single fast machine available, most of the typical difficulties of multi-processor scheduling problems do not arise. In particular, unlike a typical load-balancing problem where NP-hardness follows from a standard set-partition reduction, the One-Fast-Many-Slow Decision Problem (without dependencies) is easily solvable offline, simply by putting all tasks which finish below a certain threshold on distinct slow machines.

1.4 Open Questions

We leave three main open questions as directions for future work.

▶ **Question 1.** What are the optimal competitive ratios for eventually/never-committing schedulers?

In Section A we identify barriers, showing that improving on our eventually/never-committing schedulers will require substantially different algorithms – but we suspect that such improvements may be possible.

▶ **Question 2.** Are randomized schedulers more powerful than deterministic schedulers?

In the main body of the paper we consider only deterministic schedulers; however, for many online problems randomized algorithms can do substantially better than deterministic ones. In Section C we give some lower bounds against randomized schedulers, but these bounds are weaker than those known for deterministic schedulers.

▶ **Question 3.** Is there a general transformation between schedulers for the massively parallel regime (i.e. the One-Fast-Many-Slow Decision Problem) and the general SPDP?

The fundamental difficulty of the SPDP is deciding between implementations which take a lot of work, and implementations which take a lot of time. This tradeoff is absolute in the massively parallel regime, since the large number of processors means the amount of work associated with a serial implementation is negligible, whereas in the general SPDP it is possible for all processors to be saturated with serial implementations to run. Intuitively, one might expect that having work associated to the serial implementations only makes the problem *easier*, since it makes the tradeoff less dramatic – indeed, Kuszmaul and Westover's competitive ratio lower bounds become weaker when the number of processors is small. So, one might hope that algorithms in the massively parallel regime can be generically translated to limited-processor settings. Formalizing this connection is an interesting direction for future research.

2 Preliminaries

2.1 The One-Fast-Many-Slow Decision Problem

In this section we formally define the **One-Fast-Many-Slow Decision Problem**, where the goal is to distribute work between a single **fast machine** and an unlimited number of **slow machines**. An instance of the problem is a **Task Arrival Process (TAP)** $\mathcal{T} = (\tau_1, \ldots, \tau_n)$, where each task τ_i consists of a tuple (σ_i, π_i, t_i) indicating runtime on a slow machine, runtime on the fast machine, and arrival time, respectively, such that $t_1 \leq \cdots \leq t_n$ and such that

$\sigma_i \leq \pi_i$. A valid schedule associates at most one task to each machine at each point in time[1] such that no work is done on any task before its arrival time, each task runs on at most 1 machine, and each task τ_i is either run for a total of σ_i time on some slow machine, or a total of π_i time on the fast machine. The **completion time** (also known as **makespan**) of the schedule is the time when the last task is finished.

We will be interested in online algorithms for this problem. An online scheduler learns about each task only at its arrival time, and at each time t must already have fixed the prefix of the schedule on times less than t. We define three distinct models for how these online decisions are made:

1. For each task τ, an **instantly-committing** scheduler must fix at τ's arrival time the machine that τ will run on.
2. An **eventually-committing** scheduler need not fix a machine for any task until that task begins running.
3. A **never-committing** scheduler is an eventually-committing scheduler with the additional power to, at any time, "cancel" a task from the schedule, erasing all work previously done on the task and allowing it to be re-assigned to a new machine.

In each case, we are interested in minimizing the **competitive ratio** of an online scheduler, which is the supremum over all TAPs of the ratio of the online scheduler's completion time to the completion time of an optimal scheduler on that TAP.

2.2 Connection to the SPDP

In Kuszmaul and Westover's Serial Parallel Decision Problem, a scheduler must allocate work to p equally-powerful processors, where each task is specified by the work of the serial implementation, the work of the parallel implementation, and the runtime. The scheduler must choose whether to run each task in serial or parallel, and then must assign the resulting work to the p processors, where parallel work can run on multiple processors at once but serial work cannot.

We can define the **massively parallel** regime of this problem to be the limit as the number of processors becomes large compared to the number of tasks. Letting n be the number of tasks, if $n \leq \varepsilon p$ then restricting the serial implementations to run on only the first n many processors, and the parallel implementations to run on only the last $p - n$ many processors, the completion time can increase by at most a $\frac{1}{1-\varepsilon}$ factor. This corresponds directly to the One-Fast-Many-Slow Decision Problem: we think of each of these serial processors as a "slow machine", noting that since we have as many processors as we have tasks there are effectively an unlimited number of processors. We think of the parallel processors collectively as a "fast machine", noting that we can assume without loss of generality that, at any point in time, all parallel processors are running the same parallel implementation.

2.3 Notation

We now introduce our notation for describing and analyzing schedulers. For algorithm alg and TAP $\mathcal{T}$, we let $\mathsf{C}_{\mathsf{alg}}^{\mathcal{T}}$ be the completion time of alg on $\mathcal{T}$. Let $\mathcal{T}^t$ be the truncation of TAP $\mathcal{T}$ consisting of the tasks τ_i with $t_i \leq t$. When $\mathcal{T}$ is clear from context we will write $\mathsf{C}_{\mathsf{alg}}^{t}$ to denote $\mathsf{C}_{\mathsf{alg}}^{\mathcal{T}^t}$, and we will write $\mathsf{C}_{\mathsf{alg}}$ to denote $\mathsf{C}_{\mathsf{alg}}^{\infty} = \mathsf{C}_{\mathsf{alg}}^{\mathcal{T}}$. We will also use $\widetilde{\mathsf{C}}_{\mathsf{alg}}$ to denote the completion time of the fast machine – that is, the final time when the fast machine has work.

[1] In order for the notions like "amount of work performed on τ_i" to be well-defined, we must additionally mandate that a schedule be measurable. Alternatively, one can assume that time is discretized into appropriately fine timesteps.

It will be useful to be able to talk about the optimal completion time of a prefix of the TAP. Define the schedule $\mathsf{opt}(t)$ to be a schedule for $\mathcal{T}^t$ with minimal completion time. Note that $\mathsf{opt}(t)$ is only defined as an offline strategy, but that an online algorithm can compute it (efficiently!) at time t, thus obtaining a lower bound on $\mathsf{C}_{\mathsf{opt}}$, which will be useful to inform the algorithm's future decisions. For ease of notation, we'll often abbreviate $\mathsf{C}^t_{\mathsf{opt}(t)}$ as C^t.

There may be many sets of decisions which result in the optimal completion time; as opposed to letting $\mathsf{opt}(t)$ be an arbitrary such scheduler, it will be useful to fix a canonical one, which we will do by letting $\mathsf{opt}(t)$ run as many tasks in serial as possible.

▶ **Scheduler 4.** The scheduler $\mathsf{opt}(t)$, defined on $\mathcal{T}^t$, makes decisions as follows:

- If τ_i has $\sigma_i + t_i \leq \mathsf{C}^t$, run τ_i on a slow machine when it arrives.
- Otherwise, run τ_i on the fast machine. Prioritize tasks with larger $\sigma_i + t_i$, and break ties by taking tasks with smaller i.

Finally, we let $[n] = \{1, \ldots, n\}$, and for a set J of tasks we will write π_J to denote $\sum_{j \in J} \pi_j$.

3 A 2-Competitive Instantly-Committing Scheduler

In this section we present and analyze a 2-competitive instantly-committing scheduler. Kuszmaul and Westover showed that a competitive ratio of $(2 - \varepsilon)$ is impossible for instantly-committing schedulers, so our scheduler is optimal. The scheduler, which we call ins ("*ins*tantly commiting") is defined in Scheduler 5.

▶ **Scheduler 5.** When task τ_i arrives:

- If $\sigma_i + t_i > 2\mathsf{C}^{t_i}$ run τ_i on the fast machine, with the fast machine processing tasks in order of arrival.
- Otherwise run τ_i on a slow machine.

We analyze ins by showing inductively that $\widetilde{\mathsf{C}}_{\mathsf{ins}}$ (the completion time of the fast machine) is small compared to the work and completion time of *any* other schedule. For length n TAP $\mathcal{T}$, scheduler alg, and $i \in [n]$, we define the quantity $\mathsf{K}^t_{\mathsf{alg}}$ to be the sum of π_j for all tasks $\tau_j \in \mathcal{T}^t$ that alg runs on the fast machine. The key to analyzing ins is the following lemma.

▶ **Lemma 6.** *Fix a length n TAP. For all $i \in [n]$, and for all instantly-committing schedulers* alg*,*

$$\widetilde{\mathsf{C}}^{t_i}_{\mathsf{ins}} \leq \mathsf{C}^{t_i}_{\mathsf{alg}} + \mathsf{K}^{t_i}_{\mathsf{alg}}. \tag{1}$$

Proof. We prove the lemma by induction on i. For $i = 1$ the claim is trivial. Now, fix $i \in [n-1]$, alg and assume the lemma for i and for all alg'; we will prove the lemma for $i+1$, alg.

If ins runs τ_{i+1} on a slow machine then $\widetilde{\mathsf{C}}^{t_{i+1}}_{\mathsf{ins}} = \widetilde{\mathsf{C}}^{t_i}_{\mathsf{ins}}$, and $\mathsf{C}^{t_i}_{\mathsf{alg}} + \mathsf{K}^{i}_{\mathsf{alg}} \leq \mathsf{C}^{t_{i+1}}_{\mathsf{alg}} + \mathsf{K}^{i+1}_{\mathsf{alg}}$. Thus, the invariant Equation (1) is maintained. We always have $\widetilde{\mathsf{C}}^{t_{i+1}}_{\mathsf{ins}} \leq \widetilde{\mathsf{C}}^{t_i}_{\mathsf{ins}} + \pi_{i+1}$, so if alg runs τ_{i+1} on the fast machine then the invariant Equation (1) is also maintained, since then $\mathsf{C}^{t_{i+1}}_{\mathsf{alg}} + \mathsf{K}^{t_{i+1}}_{\mathsf{alg}} \geq \mathsf{C}^{t_i}_{\mathsf{alg}} + \mathsf{K}^{t_i}_{\mathsf{alg}} + \pi_{i+1} \geq \widetilde{\mathsf{C}}^{t_i}_{\mathsf{ins}} + \pi_{i+1}$ by the inductive hypothesis.

The final case to consider is when alg runs τ_{i+1} on a slow machine, while ins runs τ_{i+1} on the fast machine. From the definition Scheduler 5 of ins, the fact that ins ran τ_{i+1} on the fast machine implies

$$2\mathsf{C}^{t_{i+1}} < \sigma_{i+1} + t_{i+1}. \tag{2}$$

On the other hand, alg ran τ_{i+1} on the fast machine. Thus,

$$\sigma_{i+1} + t_{i+1} \leq \mathsf{C}_{\mathsf{alg}}^{t_{i+1}}. \tag{3}$$

Now, we use the invariant for $(i, \mathsf{opt}_{t_{i+1}})$ to bound $\widetilde{\mathsf{C}}_{\mathsf{ins}}^{t_{i+1}}$. We have:

$$\widetilde{\mathsf{C}}_{\mathsf{ins}}^{t_{i+1}} \leq \widetilde{\mathsf{C}}_{\mathsf{ins}}^{t_i} + \pi_{i+1} \leq \mathsf{K}_{\mathsf{opt}(t_{i+1})}^{t_i} + \mathsf{C}_{\mathsf{opt}(t_{i+1})}^{t_i} + \pi_{i+1}. \tag{4}$$

Because of Equation (2) we know that $\mathsf{opt}(t_{i+1})$ must run τ_{i+1} on the fast machine. So, we have

$$\mathsf{K}_{\mathsf{opt}(t_{i+1})}^{t_i} + \mathsf{C}_{\mathsf{opt}(t_{i+1})}^{t_i} + \pi_{i+1} = \mathsf{K}_{\mathsf{opt}(t_{i+1})}^{t_{i+1}} + \mathsf{C}_{\mathsf{opt}(t_{i+1})}^{t_i} \leq 2\mathsf{C}^{t_{i+1}}. \tag{5}$$

Stringing together the above inequalities Equation (4), Equation (5), Equation (2), and Equation (3), we get

$$\widetilde{\mathsf{C}}_{\mathsf{ins}}^{t_{i+1}} < \mathsf{C}_{\mathsf{alg}}^{t_{i+1}}.$$

Thus, the invariant Equation (1) holds. ◀

Using Lemma 6 it is easy to show that ins is 2-competitive.

▶ **Theorem 7.** ins *is a 2-competitive instantly-committing scheduler.*

Proof. By Lemma 6 we have $\widetilde{\mathsf{C}}_{\mathsf{ins}}^{t_n} \leq 2\mathsf{C}_{\mathsf{opt}}$. Thus, ins finishes using the fast machine before time $2\mathsf{C}_{\mathsf{opt}}$. Any task that ins runs on a slow machine must have $\sigma_i + t_i \leq 2\mathsf{C}_{\mathsf{opt}}$, so these tasks finish before $2\mathsf{C}_{\mathsf{opt}}$ as well. ◀

4 A 1.678-Competitive Eventually-Committing Scheduler

In this section we present and analyze a ξ-competitive eventually-committing scheduler, where $\xi \approx 1.678$ is the real root of the polynomial $2x^3 - 3x^2 - 1$. Kuszmaul and Westover gave a lower bound of $\phi \approx 1.618$ on the competitive ratio of any eventually-committing scheduler and conjectured that this lower bound is tight. Our scheduler represents substantial progress towards resolving Kuszmaul and Westover's conjecture, improving on their previous best algorithm which had a competitive ratio of 3. Our scheduler, which we call eve ("*eve*ntually committing"), is defined in Scheduler 8.

▶ **Scheduler 8.** At time t:

- If task τ_i, which has arrived but not yet been started, has $\sigma_i + t \leq \xi\mathsf{C}^t$, then start τ_i on a slow machine.
- Maintain up to one **active** task at a time. The fast machine is always allocated to the active task.
- When there is no active task, but there are unstarted tasks present, choose as the new active task the unstarted task with the largest $\sigma_i + t_i$ value (breaking ties arbitrarily).

▶ **Theorem 9.** eve *is a ξ-competitive eventually-committing scheduler.*

Proof. Fix TAP $\mathcal{T}$. Let $\widetilde{\mathsf{C}}_{\mathsf{eve}}$ denote the time when eve completes the last task run on the fast machine. If τ_i is run on a slow machine at any time t, then τ_i finishes before $\xi\mathsf{C}^t \leq \xi\mathsf{C}_{\mathsf{opt}}$. Thus, it suffices to show that $\widetilde{\mathsf{C}}_{\mathsf{eve}} \leq \xi\mathsf{C}_{\mathsf{opt}}$.

For any $x \in [0, \mathsf{C}_{\mathsf{opt}}]$, let $R(x)$ be the first time that an online algorithm becomes aware that the optimal schedule requires at least x completion time – that is, $R(x) = \inf\{t : \mathsf{C}^t \geq x\}$. Let $\mathcal{A}$ ("actual") be the set of tasks that opt runs on the fast machine, and $\mathcal{F}$ ("fake") be the set of tasks that opt runs on a slow machine but eve runs on the fast machine. We can bound the sizes and arrival times of tasks in $\mathcal{F}$ as follows.

▷ Claim 10. All tasks $\tau_i \in \mathcal{F}$ arrive before time $R(\mathsf{C}_{\mathsf{opt}}/\xi)$, and have $\pi_i < \sigma_i/\xi$.

Proof. All tasks $\tau_i \in \mathcal{F}$ are run on the fast machine by eve, and on slow machines by opt. In particular this means

$$\xi \mathsf{C}^{t_i} < \sigma_i + t_i \leq \mathsf{C}_{\mathsf{opt}} \leq \xi \mathsf{C}^{R(\mathsf{C}_{\mathsf{opt}}/\xi)}.$$

Thus, $t_i < R(\mathsf{C}_{\mathsf{opt}}/\xi)$. To show $\pi_i < \sigma_i/\xi$, note that $\pi_i + t_i \leq \mathsf{C}^{t_i} < \frac{\sigma_i + t_i}{\xi}$. ◁

To analyze when tasks in $\mathcal{F}$ get run it will be useful to partition $\mathcal{F}$ into $\mathcal{F}_{\mathsf{big}} = \{\tau_i \in \mathcal{F} \colon \sigma_i + t_i > \mathsf{C}_{\mathsf{opt}}/\xi\}$ and $\mathcal{F}_{\mathsf{small}} = \{\tau_i \in \mathcal{F} \colon \sigma_i + t_i \leq \mathsf{C}_{\mathsf{opt}}/\xi\}$. Now we show that, without loss of generality, eve does not start any tasks in $\mathcal{F}_{\mathsf{small}}$ too late.

▷ Claim 11. If eve starts a task $\tau \in \mathcal{F}_{\mathsf{small}}$ at any time $t \geq R(\mathsf{C}_{\mathsf{opt}}/\xi)$, then $\widetilde{\mathsf{C}}_{\mathsf{eve}} \leq \xi \mathsf{C}_{\mathsf{opt}}$.

Proof. Note that no task $\tau_i \in \mathcal{F}_{\mathsf{small}}$ can be started after time $\mathsf{C}_{\mathsf{opt}}$: since $\mathsf{C}_{\mathsf{opt}} + \sigma_i \leq \mathsf{C}_{\mathsf{opt}} + \mathsf{C}_{\mathsf{opt}}/\xi < \xi \mathsf{C}_{\mathsf{opt}}$, any task $\tau_i \in \mathcal{F}_{\mathsf{small}}$ present but not already running at time $\mathsf{C}_{\mathsf{opt}}$ would be run on a slow machine. Let t_* be the last time after $R(\mathsf{C}_{\mathsf{opt}}/\xi)$ when eve starts a task $\tau \in \mathcal{F}$. If t_* does not exist the claim is vacuously true. In light of our previous observation, $t_* < \mathsf{C}_{\mathsf{opt}}$. Let τ_i be the task that eve starts at time t_*. Because eve prioritizes making tasks with larger $\sigma_j + t_j$ values active, at time t_* there are no tasks $\tau \in \mathcal{F}_{\mathsf{big}} \cup \mathcal{A}$ present. After time t_*, no more tasks from $\mathcal{F}$ can arrive by Claim 10, and at most $\mathsf{C}_{\mathsf{opt}} - t_*$ work in $\mathcal{A}$ can arrive because opt must be able to complete this work. Thus,

$$\widetilde{\mathsf{C}}_{\mathsf{eve}} \leq t_* + (\mathsf{C}_{\mathsf{opt}} - t_*) + \pi_i = \mathsf{C}_{\mathsf{opt}} + \pi_i. \tag{6}$$

Now, because $\tau_i \in \mathcal{F}_{\mathsf{small}}$ we have $\pi_i \leq \mathsf{C}_{\mathsf{opt}}/\xi^2$; using this in Equation (6) we find $\widetilde{\mathsf{C}}_{\mathsf{eve}} \leq (1 + 1/\xi^2)\mathsf{C}_{\mathsf{opt}} \leq \xi \mathsf{C}_{\mathsf{opt}}$. ◁

This means that we can assume that, after time $R(\mathsf{C}_{\mathsf{opt}}/\xi)$, the only tasks that eve runs on the fast machine are $\mathcal{A}$, $\mathcal{F}_{\mathsf{big}}$, and whatever the active task was at time $R(\mathsf{C}_{\mathsf{opt}}/\xi)$. We call the active task at time $R(\mathsf{C}_{\mathsf{opt}}/\xi)$, if one exists, the **stuck** task, denoted τ_s. We split into cases depending on how large this stuck task is.

Case 1. There is no stuck task.

In this case, we in fact have $\widetilde{C}_{\mathsf{eve}} \leq \mathsf{C}_{\mathsf{opt}}$. Since there is no active task at time $R(\mathsf{C}_{\mathsf{opt}}/\xi)$, there are no tasks present but not started on slow machines. By Claim 10, eve will run all tasks $\tau \notin \mathcal{A}$ arriving after time $R(\mathsf{C}_{\mathsf{opt}}/\xi)$ on slow machines. Thus, at all time steps $t \in [R(\mathsf{C}_{\mathsf{opt}}/\xi), \mathsf{C}_{\mathsf{opt}}]$, eve either has no active task on the fast machine, or has some $\tau \in \mathcal{A}$ as the active task on the fast machine, so eve completes $\mathcal{A}$ at least as quickly as opt.

Case 2. There is a stuck task, with $\sigma_s + t_s > \mathsf{C}_{\mathsf{opt}}/\xi$.

Define $\mathcal{A}_{\text{early}} = \{\tau_i \in \mathcal{A} : t_i < R(\mathsf{C}_{\mathsf{opt}}/\xi)\}$ and $\mathcal{A}_{\text{late}} = \mathcal{A} \setminus \mathcal{A}_{\text{early}}$. Let $t < R(\mathsf{C}_{\mathsf{opt}}/\xi)$ be a time when all tasks in $\{\tau_s\} \cup \mathcal{F}_{\mathsf{big}} \cup \mathcal{A}_{\text{early}}$ have already arrived; such a time must exist by Claim 10. Observe that $\mathsf{opt}(t)$ runs all tasks in $\{\tau_s\} \cup \mathcal{F}_{\mathsf{big}} \cup \mathcal{A}_{\text{early}}$ on the fast machine due to $\mathsf{C}^t < \mathsf{C}_{\mathsf{opt}}/\xi$. This further implies that $\pi_{\mathcal{F}_{\mathsf{big}} \cup \{\tau_s\} \cup \mathcal{A}_{\text{early}}} \leq \mathsf{C}_{\mathsf{opt}}/\xi$. Also $\pi_{\mathcal{A}_{\text{late}}} \leq \mathsf{C}_{\mathsf{opt}} - R(\mathsf{C}_{\mathsf{opt}}/\xi)$, simply because opt must complete the work on tasks $\mathcal{A}_{\text{late}}$ after these tasks arrive. By Claim 11 we may assume without loss of generality that, after time

$R(\mathsf{C}_{\mathsf{opt}}/\xi)$, eve is always running a task from $\{\tau_s\} \cup \mathcal{A} \cup \mathcal{F}_{\mathsf{big}}$. Thus,

$$\begin{aligned}\widetilde{\mathsf{C}}_{\mathsf{eve}} &\leq R(\mathsf{C}_{\mathsf{opt}}/\xi) + \pi_{\mathcal{A}\cup\mathcal{F}_{\mathsf{big}}\cup\{\tau_s\}} \\ &= R(\mathsf{C}_{\mathsf{opt}}/\xi) + \pi_{\mathcal{A}_{\mathrm{early}}\cup\mathcal{F}_{\mathsf{big}}\cup\{\tau_s\}} + \pi_{\mathcal{A}_{\mathrm{late}}} \\ &\leq R(\mathsf{C}_{\mathsf{opt}}/\xi) + \mathsf{C}_{\mathsf{opt}}/\xi + (\mathsf{C}_{\mathsf{opt}} - R(\mathsf{C}_{\mathsf{opt}}/\xi)) \\ &\leq \xi\mathsf{C}_{\mathsf{opt}}\end{aligned}$$

Case 3. There is a stuck task, with $\sigma_s + t_s \leq C_{\mathsf{opt}}/\xi$.

This case will be the most difficult to handle of the three. It will be useful to focus now on the tasks of $\mathcal{F}_{\mathsf{big}}$ that arrive after the stuck task is started. Let t_* be the time when eve starts running τ_s, and let $\mathcal{F}_{\mathsf{big}}{}' = \{\tau_i \in \mathcal{F}_{\mathsf{big}} : t_i \geq t_*\}$ be the fake tasks arriving after t_*. We first observe that, if no such tasks arrive, eve performs very well.

▷ Claim 12. If $\mathcal{F}_{\mathsf{big}}{}' = \varnothing$ then $\widetilde{\mathsf{C}}_{\mathsf{eve}} \leq \xi\mathsf{C}_{\mathsf{opt}}$.

Proof. At time $t_\star$ no tasks $\tau_i \in \mathcal{A}\cup\mathcal{F}_{\mathsf{big}}$ can be present, since all such tasks have $\sigma_i + t_i > \mathsf{C}_{\mathsf{opt}}/\xi$, so eve would prioritize running them on the fast machine instead of the stuck task. By Claim 11, we know that after time t_* eve will always be running tasks from $\{\tau_s\} \cup \mathcal{A} \cup \mathcal{F}_{\mathsf{big}}{}'$. The total work on tasks from $\mathcal{A}$ that arrives after time $t_\star$ is at most $\mathsf{C}_{\mathsf{opt}} - t_\star$, so if $\mathcal{F}_{\mathsf{big}}{}' = \varnothing$ we have

$$\widetilde{\mathsf{C}}_{\mathsf{eve}} \leq t_\star + \pi_s + \mathsf{C}_{\mathsf{opt}} - t_\star \leq \sigma_s/\xi + \mathsf{C}_{\mathsf{opt}} \leq \xi\mathsf{C}_{\mathsf{opt}}. \qquad \triangleleft$$

By Claim 12 we may assume $\mathcal{F}_{\mathsf{big}}{}' \neq \varnothing$. So, let $\sigma_{\min} = \min_{\tau_i \in \mathcal{F}_{\mathsf{big}}{}'}(\sigma_i)$; we will be able to control how much work arrives in the TAP by the fact that eve never decides to run the task $\tau_i \in \mathcal{F}_{\mathsf{big}}{}'$ with $\sigma_i = \sigma_{\min}$ on a slow machine. Split $\mathcal{A}$ into $\mathcal{A}_{\mathrm{early}} = \{\tau \in \mathcal{A} : t_\star \leq t_i < R(\sigma_{\min})\}$ and $\mathcal{A}_{\mathrm{late}} = \{\tau \in \mathcal{A} : t_i \geq \max(t_\star, R(\sigma_{\min}))\}$ (note that we use a different threshold to define earliness here than we did in case 2). First we need the following analogue of Claim 11.

▷ Claim 13. If eve starts a task $\tau \in \mathcal{F}_{\mathsf{big}}{}'$ at any time $t \in [R(\mathsf{C}_{\mathsf{opt}}/\xi), \mathsf{C}_{\mathsf{opt}}]$, then $\widetilde{\mathsf{C}} \leq \xi\mathsf{C}_{\mathsf{opt}}$.

Proof. Let t_* be the last time in $[R(\mathsf{C}_{\mathsf{opt}}/\xi), \mathsf{C}_{\mathsf{opt}}]$ when eve starts a task $\tau \in \mathcal{F}'$. If t_* does not exist the claim is vacuously true. Let τ_i be the task that eve starts at time t_*. Because eve prioritizes making tasks with larger $\sigma_j + t_j$ values active, at time t_* there are no tasks $\tau \in \mathcal{A}$ present. After time t_* at most $\mathsf{C}_{\mathsf{opt}} - t_*$ work in $\mathcal{A}$ can arrive because opt must be able to complete this work. Recalling that $\pi_{\mathcal{F}_{\mathsf{big}}{}'} \leq \mathsf{C}_{\mathsf{opt}}/\xi$, we have

$$\widetilde{\mathsf{C}}_{\mathsf{eve}} \leq t_* + \pi_{\mathcal{F}_{\mathsf{big}}{}'} + (\mathsf{C}_{\mathsf{opt}} - t_*) \leq \xi\mathsf{C}_{\mathsf{opt}}. \qquad \triangleleft$$

▷ Claim 14. $t_\star + \pi_{\mathcal{F}_{\mathsf{big}}{}'} + \pi_{\mathcal{A}_{\mathsf{early}}} < (\sigma_{\min} + R(\sigma_{\min}))/\xi$.

Proof. Fix a time $t < R(\sigma_{\min})$ after all tasks in $\mathcal{F}_{\mathsf{big}}{}' \cup \mathcal{A}_{\mathsf{early}}$ have arrived, and fix a task $\tau_i \in \mathcal{F}_{\mathsf{big}}{}'$ with $\sigma_i = \sigma_{\min}$. First, note that by Claim 13 we can assume that eve has not started τ_i by time t. Thus, eve is free to start τ_i on a slow machine, but chooses not to. This implies

$$\xi\mathsf{C}^t < \sigma_i + t < \sigma_{\min} + R(\sigma_{\min}). \tag{7}$$

We also observe that $\mathsf{opt}(t)$ must run $\mathcal{F}_{\mathsf{big}}{}' \cup \mathcal{A}_{\mathsf{early}}$ on the fast machine, since running any of them on the slow machine would finish after time $\sigma_{\min}$. Thus,

$$t_\star + \pi_{\mathcal{F}_{\mathsf{big}}{}'} + \pi_{\mathcal{A}_{\mathsf{early}}} \leq \mathsf{C}^t. \tag{8}$$

Combining Equation (8) and Equation (7) gives the desired statement. $\triangleleft$

The other observation we make is that $R(\sigma_{\min})$ cannot happen too early.

▷ Claim 15. $R(\sigma_{\min}) > (\xi - 1)\sigma_{\min}$.

Proof. Let $\tau_i \in \mathcal{F}_{\mathsf{big}}{}'$ be a task with $\sigma_i = \sigma_{\min}$. Note that $t_i \leq R(\sigma_{\min})$ by Claim 10. Then, by Claim 13 we may assume without loss of generality that at time $R(\sigma_{\min})$ eve is not running τ_i, and does not choose to start τ_i on a slow machine. So,

$$R(\sigma_{\min}) + \sigma_{\min} > \xi \mathsf{C}^{R(\sigma_{\min})} \geq \xi\sigma_{\min}.$$ ◁

Now we conclude the theorem.

▷ Claim 16. $\widetilde{\mathsf{C}}_{\mathsf{eve}} \leq \xi \mathsf{C}_{\mathsf{opt}}$.

Proof. Note that $\pi_{\mathcal{A}_{\text{late}}} \leq \mathsf{C}_{\mathsf{opt}} - R(\sigma_{\min})$. Also, note that since τ_s was not put on a slow machine immediately upon arrival, we must have $\pi_s \leq \sigma_s/\xi \leq \mathsf{C}_{\mathsf{opt}}/\xi^2$. Then, applying Claim 14 and Claim 15 we have

$$\begin{aligned}
\widetilde{\mathsf{C}}_{\mathsf{eve}} &\leq \pi_s + t_\star + \pi_{\mathcal{F}_{\mathsf{big}}{}'} + \pi_{\mathcal{A}_{\mathsf{early}}} + \pi_{\mathcal{A}_{\text{late}}} \\
&\leq \mathsf{C}_{\mathsf{opt}}/\xi^2 + (\sigma_{\min} + R(\sigma_{\min}))/\xi + \mathsf{C}_{\mathsf{opt}} - R(\sigma_{\min}) \\
&\leq (1/\xi^2 + (1 + \xi - 1)/\xi + 1 - (\xi - 1))\mathsf{C}_{\mathsf{opt}} \\
&= (3 + 1/\xi^2 - \xi)\mathsf{C}_{\mathsf{opt}} \\
&= \xi \mathsf{C}_{\mathsf{opt}}.
\end{aligned}$$ ◁

◀

▶ Remark 17. The simple nature of the lower bound in Proposition 32, along with the fact that eve gets a competitive ratio quite close to ϕ might leave the impression that ϕ is clearly the correct competitive ratio, and a slightly better analysis of (a natural variant of) eve would be ϕ-competitive. However, this is not the case: in Section A, we show that no **non-procrastinating** eventually-committing scheduler can achieve competitive ratio better than ξ, where a scheduler is called non-procrastinating if, whenever tasks are present, it always runs at least one task. Thus, if the competitive ratio of eve can be improved upon, doing so will require a substantially different scheduler: one which occasionally decides to do nothing at all despite work being present.

5 A 1.5-Competitive Never-Committing Scheduler

In this section we analyze never-committing schedulers. First we give a simple lower bound.

▶ **Proposition 18.** *Fix $\varepsilon > 0$. There is no deterministic $((1 + \sqrt{3})/2 - \varepsilon)$-competitive never-committing scheduler.*

Proof. We may assume $\varepsilon < .001$. Let $\psi = (\sqrt{3} - 1)/2$. Let $\mathcal{T} = \Big((1, 2\psi, 0), (\infty, 1 - \psi, \psi)\Big)$; that is, τ_1 has $\sigma_1 = 1, \pi_1 = 2\psi, t_1 = 0$, and τ_2 has $\sigma_2 = \infty, \pi_2 = 1 - \psi, t_2 = \psi$. Let $\mathcal{T}' = \Big((1, 2\psi, 0)\Big)$ be the same TAP without the second task. We have $\mathsf{C}^{\mathcal{T}'}_{\mathsf{opt}} = 2\psi$, since opt just runs the single task on the fast machine, and $\mathsf{C}^{\mathcal{T}}_{\mathsf{opt}} = 1$, since opt can run τ_1 on a slow machine and τ_2 on the fast machine.

Suppose that alg is a $(\psi + 1 - \varepsilon)$-competitive scheduler. On TAP $\mathcal{T}'$, at time $\psi - \varepsilon/2$, we claim alg must be running τ_1 on the fast machine. If not, then alg's completion time must be at least $\min(\sigma_1, \psi - \varepsilon/2 + \pi_1) = 1$, with the branch of the min depending on whether τ_1 is ever moved to the fast machine – but this gives competitive ratio $1/(2\psi) = 1 + \psi$.

Before time ψ, it is impossible to distinguish between $\mathcal{T}$ and $\mathcal{T}'$. Thus, alg must be running τ_1 on the fast machine at time $\psi - \varepsilon/2$ on TAP $\mathcal{T}$. Now, we have $\mathsf{C}^{\mathcal{T}}_{\mathsf{alg}} \geq \min(\sigma_1 + \psi - \varepsilon/2, \pi_1 + \pi_2) = 1 + \psi - \varepsilon/2$, with the branch of the min depending on whether τ_1 is ever moved to a slow machine – but this gives competitive ratio $(1 + \psi - \varepsilon/2)/1$. Thus, alg is not actually $(\psi + 1 - \varepsilon)$-competitive. ◀

Now we give a 1.5-competitive never-committing scheduler, which we call nev ("*nev*er committing"). Note that this competitive ratio is smaller than the lower bound of $\phi \approx 1.618$ known for eventually committing schedulers, so this demonstrates a separation between the strengths of schedulers in the two models.

▶ **Scheduler 19.** At time t:

- If task τ_i has $\sigma_i + t \leq 1.5\mathsf{C}^t$ but is not currently running on a slow machine, start τ_i on a slow machine, cancelling its fast machine implementation if necessary.
- Let $\mathcal{P}$ be the set of τ_i that have arrived and are not running on a slow machine. Choose $\tau_i \in \mathcal{P}$ maximizing $\sigma_i + t_i$, breaking ties by choosing the task with the smaller i. Run τ_i on the fast machine during this time step.

▶ **Theorem 20.** nev *is a* 1.5*-competitive never-committing scheduler.*

Proof. Fix TAP $\mathcal{T}$. Let $\widetilde{C}_{\mathsf{nev}}$ denote the final time when nev has work on the fast machine. Observe that if nev ever runs τ_i on a slow machine, then nev finishes τ_i before time $1.5\mathsf{C}_{\mathsf{opt}}$. Thus, to show that nev is 1.5-competitive it suffices to show $\widetilde{C}_{\mathsf{nev}} \leq 1.5\mathsf{C}_{\mathsf{opt}}$.

Let $\mathcal{A} = \{\tau_i : \sigma_i + t_i > \mathsf{C}_{\mathsf{opt}}\}$ be the set of tasks that opt *actually* runs on the fast machine.

▷ Claim 21. nev never runs a task $\tau \in \mathcal{A}$ on the fast machine after time $\mathsf{C}_{\mathsf{opt}}$.

Proof. nev always allocates the fast machine to the present task with the largest value of $\sigma_i + t_i$ among tasks that aren't running on slow machines. Thus, whenever there are tasks from $\mathcal{A}$ that aren't running on slow machines, nev will run one such task on the fast machine. opt is able to complete all tasks in $\mathcal{A}$ on the fast machine by time $\mathsf{C}_{\mathsf{opt}}$. Thus, nev completes or starts on slow machines all tasks $\tau \in \mathcal{A}$ before time $\mathsf{C}_{\mathsf{opt}}$. ◁

This means that the only way to have $\widetilde{\mathsf{C}}_{\mathsf{eve}} > \mathsf{C}_{\mathsf{opt}}$ is if there are tasks with $\sigma_i + t_i \leq \mathsf{C}_{\mathsf{opt}}$ that have yet to be completed at time $\mathsf{C}_{\mathsf{opt}}$; we assume that this is the case for the remainder of the proof. Let $\Pi(x)$ be the total amount of work nev performs on the fast machine after time $\mathsf{C}_{\mathsf{opt}}$ across all tasks with $\sigma_i + t_i \in [x, 1.5x]$. For any $x \leq \mathsf{C}_{\mathsf{opt}}$, let $R(x) = \inf\{t \mid \mathsf{C}^t \geq x\}$ be the first time an online algorithm becomes aware that the optimal schedule requires x completion time; the following key claim allows us to bound this left-over work $\Pi(x)$ in terms of $R(x)$.

▷ Claim 22. For all x, we have $\Pi(x) \leq x - R(x)$.

Proof. Let $\mathcal{J}_x$ denote the set of tasks with $\sigma_i + t_i \leq 1.5x$ that nev runs on the fast machine at some time after $\mathsf{C}_{\mathsf{opt}}$. First, note that all $\tau_i \in \mathcal{J}_x$ must have $t_i < R(x)$, or else τ_i would be placed on a slow machine upon arrival. Choose ε sufficiently small, such that no tasks arrive between times $R(x) - \varepsilon$ and $R(x)$. Since $R(x) - \varepsilon < R(x)$, we know $\mathsf{C}^{R(x)-\varepsilon} < x$, and so $\mathsf{opt}(R(x) - \varepsilon)$ must run all tasks with $\sigma_i + t_i \geq x$ on the fast machine. In order for $\mathsf{opt}(R(x) - \varepsilon)$ to finish these tasks before time $\mathsf{C}^{R(x)-\varepsilon} < x$, $\mathsf{opt}(R(x) - \varepsilon)$ must have at most $x - R(x) + \varepsilon$ fast work remaining across all such tasks.

Now, by the same argument as in Claim 21, because nev prioritizes tasks with $\sigma_i + t_i \geq x$ over tasks with $\sigma_i + t_i < x$ on the fast machine whenever they are present (and not yet started on slow machines), nev has at most $x - R(x) + \varepsilon$ work remaining on tasks in $\mathcal{J}_x$ at time $R(x) - \varepsilon$. Because no more tasks from $\mathcal{J}_x$ arrive after this time, we have $\Pi(x) \leq x - R(x) + \varepsilon$ as well. The claim held for all $\varepsilon > 0$, and so taking $\varepsilon \to 0$ we have $\Pi(x) \leq x - R(x)$. ◁

We now give an observation to control $R(x)$. Let $\tau_{i_\star}$ be the task, among all tasks that nev runs on the fast machine after time $\mathsf{C}_{\mathsf{opt}}$, with the smallest value of $\sigma_i + t_i$. Let $\lambda = \sigma_{i_\star} + t_{i_\star}$.

▷ Claim 23. For all $x \geq \lambda$, we have $R(x) > 1.5x - \lambda$.

Proof. First, note that $t_{i_\star} \leq R(\lambda) \leq \lambda$ or else nev would start $\tau_{i_\star}$ on a slow machine upon arrival. Now, because nev doesn't start $\tau_{i_\star}$ on a slow machine at time $R(x) > t_{i_\star}$, we have $R(x) + \sigma_{i_\star} > 1.5x$. ◁

To prove the theorem, it will now suffice to branch into two cases, based on how large λ is.

Case 1. $\lambda \geq (2/3)\mathsf{C}_{\mathsf{opt}}$.

In this case, since $1.5\lambda \geq \mathsf{C}_{\mathsf{opt}}$, by Claim 21 all left-over work at time $\mathsf{C}_{\mathsf{opt}}$ comes from tasks with $\sigma_i + t_i \in [\lambda, 1.5\lambda]$. By Claim 22, the total amount of such work is at most $\lambda - R(\lambda)$. Then, by Claim 23, we know $R(\lambda) > .5\lambda$. Together, this implies that the total amount of leftover work is at most $.5\lambda \leq .5\mathsf{C}_{\mathsf{opt}}$.

Case 2. $\lambda < 2\mathsf{C}_{\mathsf{opt}}/3$.

First note that $\lambda \geq \mathsf{C}_{\mathsf{opt}}/2$ or else $\tau_{i_\star}$ would be started on a slow machine at time $\mathsf{C}_{\mathsf{opt}}$. So, all left-over work at time $\mathsf{C}_{\mathsf{opt}}$ comes either from tasks with $\sigma_i + t_i \in [\lambda, 1.5\lambda]$, or from tasks with $\sigma_i + t_i \in [1.5\lambda, 1.5^2\lambda]$. By Claim 22, we can therefore bound the total amount of leftover work by $(\lambda - R(\lambda)) + (1.5\lambda - R(1.5\lambda))$. Now, by Claim 23, this quantity can be at most $(\lambda - .5\lambda) + (1.5\lambda - 1.25\lambda) = .75\lambda$. Since $\lambda < 2\mathsf{C}_{\mathsf{opt}}/3$, this is at most $.5\mathsf{C}_{\mathsf{opt}}$. ◀

▶ Remark 24. In Section A we show that Scheduler 19 is optimal among never-committing schedulers that never cancel implementations running on slow machines. This shows that improving on Scheduler 19 will require a substantially different scheduler.

6 Extending Beyond the Massively Parallel Regime

In Theorem 7, we have shown that Scheduler 5 is a 2-competitive instantly-committing scheduler in the Massively Parallel regime of the SPDP. In this section, we will show that in fact, Scheduler 5 is a 2-competitive scheduler even outside of the Massively Parallel regime, although the analysis is slightly more complicated. This result is interesting in its own right, resolving an open question from [16]. However, we think that the main virtue of this proof is that it serves as a proof-of-concept that results from the conceptually simpler Massively Parallel regime can be adapted to apply to the general SPDP: we conjecture that all upper bounds holding in the massively parallel regime should also hold in the general SPDP.

First, recall the setup of the SPDP problem. The input is a sequence of triples $\tau_i = (\sigma_i, \pi_i, t_i)$, where π_i is the work of the parallel implementation of task τ_i, and σ_i is the work of the serial implementation of task τ_i. At each time step, the scheduler allocates its processors to the jobs, giving at most 1 processor to each serial job.

Scheduler 5 is not a defined scheduler in the SPDP, because we specify the decisions for which tasks to run, but do not specify how to schedule the tasks. We extend ins to the general SPDP as follows:

▶ **Scheduler 25.** When task τ_i arrives:
- If $\sigma_i + t_i > 2\mathsf{C}^{t_i}$ parallelize τ_i.
- Otherwise, serialize τ_i.

At every timestep, if there are x serial jobs present, then ins schedules the jobs by allocating a processor to each of the $\min(p, x)$ serial jobs with the most remaining work (or an arbitrary set of p jobs with maximum remaining work if there are more than p serial jobs with maximum remaining work), and then allocating any remaining processors to an arbitrary parallel job (if a parallel job is present).

Now we analyze ins. We say ins is **saturated** at time t if ins has no idle processors at time t.

▶ **Lemma 26.** *If* ins *is unsaturated right before finishing, then* $\mathsf{C}_{\mathsf{ins}} \leq 2\mathsf{C}_{\mathsf{opt}}$.

Proof. We claim that if ins is unsaturated at time t, then for each task τ_i present at time t, τ_i has been run on every time step since it arrived. Suppose this is not the case. Then, there must have been some time step before time t when there were at least p serial jobs with at least as much remaining work as τ_i. But then τ_i will finish at the same time as these other jobs, contradicting the fact that ins is unsaturated at time t. Thus, if ins is unsaturated at time t, then $t \leq \sigma_i + t_i$ for some i such that ins ran τ_i in serial. Thus, $t \leq 2\mathsf{C}_{\mathsf{opt}}$, as desired. ◀

By virtue of Lemma 26 it suffices to consider the case that ins is saturated immediately before finishing. Let t_* be the final time in $[0, \mathsf{C}_{\mathsf{ins}})$ when ins is unsaturated (we set $t_* = 0$ if ins is always saturated). Let $i_* \in [n]$ be the smallest i such that $t_i \geq t_*$; in fact we will have $t_{i_*} = t_*$, since in order to transition from being unsaturated to being saturated, some tasks must arrive. For integer $i \in [i_*, n]$, let $\mathsf{K}^i_{\mathsf{alg}}$ denote the sum of π_j for each τ_j with $i_* \leq j \leq i$ that alg runs in parallel; If alg is an instantly-committing scheduler then $\mathsf{K}^i_{\mathsf{alg}}$ can be computed at time t_i. Now we prove an analogue of Lemma 6.

▶ **Lemma 27.** *Fix a length n TAP. For all $i \in [i_*, n]$, and for all instantly-committing schedulers* alg*,*

$$\mathsf{K}^i_{\mathsf{ins}} \leq (\mathsf{C}^{t_i}_{\mathsf{alg}} - t_*)p + \mathsf{K}^i_{\mathsf{alg}}. \tag{9}$$

Proof. We prove Equation (9) by induction on i. The base case is $i = i_*$. If alg takes π_i work here, then we have $\mathsf{K}^i_{\mathsf{ins}} \leq \mathsf{K}^i_{\mathsf{alg}}$ (and $\mathsf{C}^{t_i}_{\mathsf{alg}} \geq t_*$) so the invariant will hold. If instead alg runs τ_i in serial, then we'll have $\mathsf{C}^{t_i}_{\mathsf{alg}} - t_* \geq \sigma_i \geq \pi_i/p$, in which case we have $\mathsf{K}^i_{\mathsf{ins}} \leq (\mathsf{C}^{t_i}_{\mathsf{alg}} - t_*)p$ so the invariant holds. This establishes the base case.

Now, assume Equation (9) for $i \in [i_*, n)$; we prove Equation (9) for $i + 1$. If alg takes at least as much work as ins on τ_{i+1}, i.e., $\mathsf{K}^{i+1}_{\mathsf{alg}} - \mathsf{K}^i_{\mathsf{alg}} \geq \mathsf{K}^{i+1}_{\mathsf{ins}} - \mathsf{K}^i_{\mathsf{ins}}$, then Equation (9) for (i, alg) implies Equation (9) for $(i + 1, \mathsf{alg})$. It remains to consider the case when alg runs τ_{i+1} in serial, but ins runs τ_{i+1} in parallel. Here, ins thought τ_{i+1} was too large to serialize large, so:

$$\mathsf{C}^{t_{i+1}}_{\mathsf{alg}} \geq \sigma_{i+1} + t_{i+1} > 2\mathsf{C}^{t_{i+1}}. \tag{10}$$

One consequence of Equation (10) is that $\mathsf{opt}(t_{i+1})$ parallelizes τ_{i+1}; a corollary of this is that Equation (9) holds for $(i+1, \mathsf{opt}(t_{i+1}))$. Thus, applying Equation (9) for $(i+1, \mathsf{opt}(t_{i+1}))$ and using Equation (10) gives:

$$\mathsf{K}_{\mathsf{ins}}^{i+1} \leq (\mathsf{C}^{t_{i+1}} - t_*)p + \mathsf{K}_{\mathsf{opt}(t_{i+1})}^{i+1} \leq (2\mathsf{C}^{t_{i+1}} - 2t_*)p \leq (\mathsf{C}_{\mathsf{alg}}^{t_{i+1}} - t_*)p,$$

so the invariant holds in this case as well. ◀

In the Massively Parallel regime Theorem 7 followed immediately from Lemma 6. Slightly more work is required in the general setting, but Lemma 27 is still very useful.

▶ **Theorem 28.** ins *is a 2-competitive instantly-committing scheduler in the SPDP.*

Proof. Recall from Lemma 26 that we need only consider the case that ins ends saturated, and recall the definition of t_*. For any scheduler alg, let $\mathsf{B}_{\mathsf{alg}}$ denote the work that alg has left immediately before time t_*, and let $\mathsf{K}_{\mathsf{alg}}$ be work that alg takes on tasks τ_i with $t_i \geq t_*$. Because ins ends saturated, we have

$$\mathsf{C}_{\mathsf{ins}} = t_* + (\mathsf{K}_{\mathsf{ins}} + \mathsf{B}_{\mathsf{ins}})/p.$$

Applying Lemma 27 gives

$$t_* + (\mathsf{K}_{\mathsf{ins}} + \mathsf{B}_{\mathsf{ins}})/p \leq \mathsf{C}_{\mathsf{opt}} + (\mathsf{K}_{\mathsf{opt}} + \mathsf{B}_{\mathsf{ins}})/p. \tag{11}$$

So, to conclude, it suffices to show that $\mathsf{B}_{\mathsf{ins}} + \mathsf{K}_{\mathsf{opt}} \leq p\mathsf{C}_{\mathsf{opt}}$. Let S be the set of tasks that ins has present immediately before time t_*. Let $W = \sum_{\tau_i \in S} \sigma_i$. Clearly $\mathsf{B}_{\mathsf{ins}} \leq W$. On the other hand, opt must take at least W work on the tasks S, and can have made at most pt_* progress on these tasks by time t_*. Thus,

$$\mathsf{B}_{\mathsf{ins}} \leq W \leq pt_* + \mathsf{B}_{\mathsf{opt}}.$$

Therefore,

$$\mathsf{B}_{\mathsf{ins}} + \mathsf{K}_{\mathsf{opt}} \leq pt_* + \mathsf{B}_{\mathsf{opt}} + \mathsf{K}_{\mathsf{opt}} \leq p\mathsf{C}_{\mathsf{opt}}.$$

Using this in Equation (11) gives $\mathsf{C}_{\mathsf{ins}} \leq 2\mathsf{C}_{\mathsf{opt}}$. ◀

References

1. S Anand, Naveen Garg, and Nicole Megow. Meeting deadlines: How much speed suffices? In *Automata, Languages and Programming: 38th International Colloquium, ICALP 2011, Zurich, Switzerland, July 4-8, 2011, Proceedings, Part I 38*, pages 232–243. Springer, 2011. doi:10.1007/978-3-642-22006-7_20.
2. James Aspnes, Yossi Azar, Amos Fiat, Serge Plotkin, and Orli Waarts. On-line routing of virtual circuits with applications to load balancing and machine scheduling. *Journal of the ACM (JACM)*, 44(3):486–504, 1997. doi:10.1145/258128.258201.
3. Baruch Awerbuch, Yossi Azar, Edward F Grove, Ming-Yang Kao, P Krishnan, and Jeffrey Scott Vitter. Load balancing in the l_p norm. In *Proceedings of IEEE 36th Annual Foundations of Computer Science*, pages 383–391. IEEE, 1995. doi:10.1109/SFCS.1995.492494.
4. Brenda S Baker and Jerald S Schwarz. Shelf algorithms for two-dimensional packing problems. *SIAM Journal on Computing*, 12(3):508–525, 1983. doi:10.1137/0212033.
5. Ioannis Caragiannis. Better bounds for online load balancing on unrelated machines. In *Proceedings of the nineteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 972–981, 2008. URL: http://dl.acm.org/citation.cfm?id=1347082.1347188.

6 Shichuan Deng, Jian Li, and Yuval Rabani. Generalized unrelated machine scheduling problem. In *Proceedings of the 2023 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2898–2916. SIAM, 2023. doi:10.1137/1.9781611977554.CH110.

7 Richard A. Dutton and Weizhen Mao. Online scheduling of malleable parallel jobs. In *Proceedings of the 19th IASTED International Conference on Parallel and Distributed Computing and Systems*, PDCS '07, pages 136–141, USA, 2007. ACTA Press.

8 R. L. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics*, 17(2):416–429, 1969. doi:10.1137/0117039.

9 Shouwei Guo and Liying Kang. Online scheduling of malleable parallel jobs with setup times on two identical machines. *European Journal of Operational Research*, 206(3):555–561, November 2010. doi:10.1016/j.ejor.2010.03.005.

10 Anupam Gupta, Amit Kumar, Viswanath Nagarajan, and Xiangkun Shen. Stochastic load balancing on unrelated machines. *Mathematics of Operations Research*, 46(1):115–133, 2021. doi:10.1287/MOOR.2019.1049.

11 Varun Gupta, Benjamin Moseley, Marc Uetz, and Qiaomin Xie. Stochastic online scheduling on unrelated machines. In *Integer Programming and Combinatorial Optimization: 19th International Conference, IPCO 2017, Waterloo, ON, Canada, June 26-28, 2017, Proceedings 19*, pages 228–240. Springer, 2017. doi:10.1007/978-3-319-59250-3_19.

12 Varun Gupta, Benjamin Moseley, Marc Uetz, and Qiaomin Xie. Greed works—online algorithms for unrelated machine stochastic scheduling. *Mathematics of operations research*, 45(2):497–516, 2020. doi:10.1287/MOOR.2019.0999.

13 Ellis Horowitz and Sartaj Sahni. Exact and approximate algorithms for scheduling nonidentical processors. *J. ACM*, 23(2):317–327, April 1976. doi:10.1145/321941.321951.

14 Johann L Hurink and Jacob Jan Paulus. Online algorithm for parallel job scheduling and strip packing. In *Approximation and Online Algorithms: 5th International Workshop, WAOA 2007, Eilat, Israel, October 11-12, 2007. Revised Papers 5*, pages 67–74. Springer, 2008. doi:10.1007/978-3-540-77918-6_6.

15 Sungjin Im and Shi Li. Improved approximations for unrelated machine scheduling. In *Proceedings of the 2023 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2917–2946. SIAM, 2023. doi:10.1137/1.9781611977554.CH111.

16 William Kuszmaul and Alek Westover. Scheduling jobs with work-inefficient parallel solutions. In *Proceedings of the 36th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 101–111, 2024. doi:10.1145/3626183.3659960.

17 Jan Karel Lenstra, David B Shmoys, and Éva Tardos. Approximation algorithms for scheduling unrelated parallel machines. *Mathematical programming*, 46:259–271, 1990. doi:10.1007/BF01585745.

18 Walter Ludwig and Prasoon Tiwari. Scheduling malleable and nonmalleable parallel tasks. In *Proceedings of the fifth annual ACM-SIAM symposium on Discrete algorithms*, pages 167–176, 1994. URL: http://dl.acm.org/citation.cfm?id=314464.314491.

19 Marco Molinaro. *Stochastic ℓ_p Load Balancing and Moment Problems via the L-Function Method*, pages 343–354. SIAM, 2019. doi:10.1137/1.9781611975482.22.

20 Gregory Mounie, Christophe Rapine, and Dennis Trystram. Efficient approximation algorithms for scheduling malleable tasks. In *Proceedings of the eleventh annual ACM symposium on Parallel algorithms and architectures*, SPAA '99, pages 23–32, New York, NY, USA, 1999. Association for Computing Machinery. doi:10.1145/305619.305622.

21 Daniel R. Page, Roberto Solis-Oba, and Marten Maack. Makespan minimization on unrelated parallel machines with simple job-intersection structure and bounded job assignments. *Theoretical Computer Science*, 809:204–217, 2020. doi:10.1016/j.tcs.2019.12.009.

22 Andreas S. Schulz and Martin Skutella. Scheduling unrelated machines by randomized rounding. *SIAM Journal on Discrete Mathematics*, 15(4):450–469, 2002. doi:10.1137/S0895480199357078.

23 John Turek, Walter Ludwig, Joel L. Wolf, Lisa Fleischer, Prasoon Tiwari, Jason Glasgow, Uwe Schwiegelshohn, and Philip S. Yu. Scheduling parallelizable tasks to minimize average response time. In *Proceedings of the sixth annual ACM symposium on Parallel algorithms and architectures*, SPAA '94, pages 200–209, New York, NY, USA, 1994. Association for Computing Machinery. doi:10.1145/181014.181331.

24 John Turek, Uwe Schwiegelshohn, Joel L Wolf, and Philip S Yu. Scheduling parallel tasks to minimize average response time. In *Proceedings of the fifth annual ACM-SIAM symposium on Discrete algorithms*, pages 112–121, 1994. URL: http://dl.acm.org/citation.cfm?id=314464.314485.

25 John Turek, Joel L Wolf, and Philip S Yu. Approximate algorithms scheduling parallelizable tasks. In *Proceedings of the fourth annual ACM symposium on Parallel algorithms and architectures*, pages 323–332, 1992. doi:10.1145/140901.141909.

26 Nodari Vakhania, Jose Hernandez, and Frank Werner. Scheduling unrelated machines with two types of jobs. *International Journal of Production Research*, 52:1–9, February 2014. doi:10.1080/00207543.2014.888789.

27 Andrew Chi-Chin Yao. Probabilistic computations: Toward a unified measure of complexity. In *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, pages 222–227. IEEE Computer Society, 1977.

28 Deshi Ye, Danny Z. Chen, and Guochuan Zhang. Online scheduling of moldable parallel tasks. *Journal of Scheduling*, 21(6):647–654, 2018. Publisher: Springer. URL: https://ideas.repec.org//a/spr/jsched/v21y2018i6d10.1007_s10951-018-0556-2.html, doi:10.1007/S10951-018-0556-2.

29 Deshi Ye, Xin Han, and Guochuan Zhang. A note on online strip packing. *Journal of Combinatorial Optimization*, 17(4):417–423, 2009. doi:10.1007/S10878-007-9125-X.

30 Xiaoyan Zhang, Ran Ma, Jian Sun, and Zan-Bo Zhang. Randomized selection algorithm for online stochastic unrelated machines scheduling. *Journal of Combinatorial Optimization*, pages 1–16, 2022.

A Barriers Against Improved Schedulers

In this section we show that the schedulers of Section 4 and Section 5 are optimal among natural restricted classes of schedulers. This highlights what changes must be made to the schedulers in order to have hopes of achieving better competitive ratios.

First we show that among **non-procrastinating** eventually-committing schedulers (i.e., eventually-committing schedulers with the property that whenever tasks are present, they will run at least one task), the scheduler Scheduler 8 is optimal.

▶ **Proposition 29.** *Fix $\varepsilon > 0$. Let $\xi \approx 1.677$ denote the real root of the polynomial $2x^3 - 3x^2 - 1$. There is no deterministic $(\xi - \varepsilon)$-competitive non-procrastinating eventually-committing scheduler.*

Proof. It suffices to consider the case that $\varepsilon < .001$. Fix a non-procrastinating eventually-committing scheduler alg. Assume towards contradiction that alg is $(\xi - \varepsilon)$-competitive. We now describe a TAP $\mathcal{T}$ on which $\mathsf{C}_{\mathsf{alg}} \geq \xi \mathsf{C}_{\mathsf{opt}}$. The TAP starts with $\tau_1 = (1/\xi, 1/\xi^2, 0)$. Next, let $\tau_2 = (1 - \varepsilon^2, 1/\xi - \varepsilon^2, \varepsilon^2)$. Finally, at each time $t \in [\xi + 1/\xi - 2, 1 - \varepsilon^2] \cap (\mathbb{N}\varepsilon^2)$, give a task $\tau = (\infty, \varepsilon^2, t)$.

We now argue that alg must run all the tasks on the fast machine. Because alg is a non-procrastinating $(\xi - \varepsilon)$-competitive scheduler, alg must instantly start τ_1 on the fast machine (in case there are no tasks after τ_1). Now we argue that alg runs τ_2 on the fast machine as well. Suppose that alg starts τ_2 on a slow machine at some time t with

$$1 - \varepsilon^2 + t > (\xi - \varepsilon)\mathsf{C}^t. \tag{12}$$

Then, alg would not be $(\xi - \varepsilon)$-competitive on the truncated TAP $\mathcal{T}^t$. Thus, alg must not start τ_2 on a slow machine at any time t satisfying Equation (12). We now show that Equation (12) holds for all $t \geq \varepsilon$, thus proving that alg must run τ_2 on the fast machine. For $t \in [\varepsilon^2, \xi + 1/\xi - 2)$ we have $\mathsf{C}^t \leq 1/\xi$, and $1 - \varepsilon^2 + t \geq 1$, so Equation (12) holds. For $t \geq \xi + 1/\xi - 2$ we have

$$\mathsf{C}^t \leq \min(1, t - \xi + 2 + \varepsilon^2).$$

Thus, it suffices to show:

$$1 - \varepsilon^2 + t > (\xi - \varepsilon)\min(1, t - \xi + 2 + \varepsilon^2). \tag{13}$$

To show Equation (13) it suffices to check Equation (13) for $t = \xi - 1 - \varepsilon^2$ (by monotonicity of the inequality on either side of $t = \xi - 1$). At $t = \xi - 1 - \varepsilon^2$ Equation (13) is:

$$\xi - 2\varepsilon^2 > \xi - \varepsilon,$$

which is true because $\varepsilon < .001$.

We have now shown that alg runs all tasks in $\mathcal{T}$ on the fast machine. Thus, (by definition of ξ)

$$\mathsf{C}_{\mathsf{alg}} \geq 1/\xi^2 + 1/\xi - \varepsilon^2 + 1 - (1/\xi + \xi - 2) - \varepsilon^2 = \xi - 2\varepsilon^2.$$

However, $\mathsf{C}_{\mathsf{opt}} \leq 1$. This contradicts the assumption that alg is $(\xi - \varepsilon)$-competitive. ◀

Now, we show that the 1.5-competitive scheduler of Section 5 is optimal among never-committing schedulers that don't cancel tasks on slow machines.

▶ **Proposition 30.** *Let* alg *be a deterministic never-committing scheduler that never cancels serial tasks. Then, for any $\varepsilon > 0$, there is a TAP $\mathcal{T}$ with $n \leq O(1)$ on which* alg *has is not $(1.5 - \varepsilon)$-competitive.*

Proof. It suffices to consider the case that $\varepsilon < .001$. The TAP is defined as follows. First, $\tau_1 = (2, 1, 0)$. Then, for each time $t \in [\varepsilon^2, 1 - \varepsilon^2] \cap \mathbb{N}\varepsilon^2$, a task $\tau = (\infty, \varepsilon^2, t)$ arrives. We will show that if alg starts τ_1 on a slow machine at any time t then alg is not $(1.5 - \varepsilon)$-competitive on $\mathcal{T}^t$. We show this by considering two cases.

Case 1. alg starts τ_1 on a slow machine at time $t \in [0, 1]$.
If alg does this, then $\mathsf{C}^{\mathcal{T}^t}_{\mathsf{alg}} \geq 2 + t$. However, $\mathsf{C}^t \leq t + 1 + \varepsilon^2$. Thus,

$$\mathsf{C}^{\mathcal{T}^t}_{\mathsf{alg}}/\mathsf{C}^t \geq \frac{2 + t}{t + 1 + \varepsilon^2} \geq \frac{3}{2 + \varepsilon^2} > 1.5 - \varepsilon.$$

So alg cannot start τ_1 on a slow machine at this time.

Case 2. alg starts τ_1 on a slow machine at time $t \geq 1$.
If alg does this, then $\mathsf{C}_{\mathsf{alg}} \geq 2 + t$. However, $\mathsf{C}_{\mathsf{opt}} \leq 2$. Thus,

$$\mathsf{C}_{\mathsf{alg}}/\mathsf{C}_{\mathsf{opt}} \geq 1.5.$$

In conclusion, alg must run τ_1 on the fast machine. But then

$$\mathsf{C}_{\mathsf{alg}} \geq 3 - \varepsilon^2 > (1.5 - \varepsilon)\mathsf{C}_{\mathsf{opt}} = (1.5 - \varepsilon)2,$$

a contradiction. ◀

B Lower Bounds from [16]

In this section we state, for the reader's convenience, the lower bounds from [16] against instantly- and eventually- committing schedulers.

▶ **Proposition 31** (Kuszmaul, Westover [16]). *Fix $\varepsilon > 0$. There is no deterministic $(2-\varepsilon)$-competitive instantly-committing scheduler.*

Proof. Consider an n-task TAP where for each $i \in [n]$, the i-th task has $\sigma_i = 2^i, \pi_i = 2^{i-1}$, and the arrival times are all very close to 0. For each $i \in [n]$, it is possible to handle the first i tasks in the TAP with completion time 2^{i-1}. Thus, a $(2-\varepsilon)$-competitive scheduler cannot afford to run task τ_i on a slow machine. So, a $(2-\varepsilon)$-competitive scheduler must run all tasks on the fast machine, giving completion time at least $2^n - 1$ on this TAP, while $\mathsf{C}_{\mathsf{opt}} \leq 2^{n-1}$. For large enough n this implies that the scheduler is not actually $2-\varepsilon$ competitive. ◀

▶ **Proposition 32** (Kuszmaul, Westover [16]). *Fix $\varepsilon > 0$. There is no deterministic $(\phi-\varepsilon)$-competitive eventually-committing scheduler, where $\phi \approx 1.618$ is the golden ratio.*

Proof. Suppose that alg is a $(\phi-\varepsilon)$-competitive eventually-committing scheduler. Let $\tau_1 = (\phi, 1, 0)$; if there are no further tasks, alg must run τ_1 on the fast machine, starting at some time $t_0 \leq 1/\phi$. Let $\tau_2 = (\infty, \phi - t_0, t_0)$. On this TAP, $\mathsf{C}_{\mathsf{opt}} = \phi$, while $\mathsf{C}_{\mathsf{alg}} \geq \phi + 1 = \phi^2$. So alg is not $(\phi-\varepsilon)$-competitive. ◀

C Randomized Lower Bounds

In this section we give lower bounds against randomized schedulers. Our main tool is Yao's minimax principle [27], which allows us to prove a lower bound on the competitive ratio by exhibiting a distribution over TAPs, and showing that any deterministic scheduler has poor expected cost on a random TAP drawn from the distribution.

▶ **Proposition 33.** *For any $\varepsilon > 0.03$, there is no $(5/3-\varepsilon)$-competitive instantly-committing scheduler, even with randomization.*

Proof. Fix $N = 25$. For $k \in \mathbb{N}$, define $\mathcal{T}_k$ to be a length k TAP with $\sigma_i = 2^{i+1}, \pi_i = 2^i$. Let $\mathcal{D}$ denote the following distribution over TAPs: choose $k \in [N]$ uniformly randomly, and then output TAP $\mathcal{T}_k$. By brute force enumeration of all possible deterministic instantly-committing strategies, one can show that no such strategy is 1.637-competitive on this TAP. ◀

▶ **Proposition 34.** *For any $\varepsilon > 0$, there is no $((3+\sqrt{3})/4-\varepsilon)$-competitive eventually-committing scheduler, even with randomization.*

Proof. In the proof of Proposition 18 we defined two TAPs, and showed that no deterministic eventually-committing scheduler is $((1+\sqrt{3})/2-\varepsilon)$-competitive on both of the TAPs. One can show that if we randomly choose between the two TAPs of Proposition 18, there is no deterministic eventually-committing scheduler with expected competitive ratio $(1+\sqrt{3})/4 - \varepsilon$. ◀