Scheduling Jobs with Work-*Inefficient* Parallel Solutions

William Kuszmaul*
Harvard University
Cambridge, MA, USA
william.kuszmaul@gmail.com

Alek Westover
Massachusetts Institute of Technology
Cambridge, MA, USA
alekw@mit.edu

ABSTRACT

This paper introduces the *serial-parallel decision problem*. Consider an online scheduler that receives a series of tasks, where each task has both a parallel and a serial implementation. The parallel implementation has the advantage that it can make progress concurrently on multiple processors, but the disadvantage that it is (potentially) work-inefficient. As tasks arrive, the scheduler must decide for each task which implementation to use.

We begin by studying *total awake time*. We give a simple *decide-on-arrival* scheduler that achieves a competitive ratio of 3 for total awake time—this scheduler makes serial/parallel decisions immediately when jobs arrive. Our second result is an *parallel-work-oblivious* scheduler that achieves a competitive ratio of 6 for total awake time—this scheduler makes all of its decisions based only on the size of each serial job and without needing to know anything about the parallel implementations. Finally, we prove a lower bound showing that, if a scheduler wishes to achieve a competitive ratio of $O(1)$, it can have at most one of the two aforementioned properties (decide-on-arrival or parallel-work-oblivious). We also prove lower bounds of the form $1 + \Omega(1)$ on the optimal competitive ratio for any scheduler.

Next, we turn our attention to optimizing *mean response time*. Here, we show that it is possible to achieve an $O(1)$ competitive ratio with $O(1)$ speed augmentation. This is the most technically involved of our results. We also prove that, in this setting, it is not possible for a parallel-work-oblivious scheduler to do well.

In addition to these results, we present tight bounds on the optimal competitive ratio if we allow for arrival dependencies between tasks (e.g., tasks are components of a single parallel program), and we give an in-depth discussion of the remaining open questions.

CCS CONCEPTS

• **Theory of computation** → **Parallel algorithms**.

KEYWORDS

Scheduling, Parallel, Work-Inefficient, Competitive-Analysis

*William Kuszmaul is funded by the Rabin Postdoctoral Fellowship in Theoretical Computer Science at Harvard University. Large parts of this research were completed while William was a PhD student at MIT, where he was funded by a Fannie and John Hertz Fellowship and an NSF GRFP Fellowship. William Kuszmaul was also partially sponsored by the United States Air Force Research Laboratory and the United States Air Force Artificial Intelligence Accelerator and was accomplished under Cooperative Agreement Number FA8750-19-2-1000. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the United States Air Force or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation herein.

1 INTRODUCTION

There are many tasks τ for which the best parallel algorithms are work inefficient. This can leave engineers with a surprisingly subtle choice: either implement a serial version $\tau^{\circledcirc}$ of the task, which is work efficient but has no parallelism, or implement a parallel version $\tau^{\shortparallel}$ of the task, which is work inefficient but has ample parallelism. The serial version $\tau^{\circledcirc}$ of the task takes some amount of time σ to execute on a single processor; the parallel version $\tau^{\shortparallel}$ takes time $\pi \geq \sigma$ to execute on a single processor, but can be scaled to run on any number $k \leq p$ processors with a k-fold speedup. Which version of the task should the engineer implement?

If the task is running in isolation on a p-processor system, and assuming that $\pi/p \leq \sigma$, then the answer is trivial: use the parallel implementation $\tau^{\shortparallel}$. But what if the system is shared with many other tasks that are arriving/completing over time? Intuitively, the engineer should use $\tau^{\shortparallel}$ if the system can afford to allocate at least π/σ processors to the task, and should use $\tau^{\circledcirc}$ otherwise. But this choice is complicated by two factors, since the number of processors that the system can afford to allocate to τ may both (1) change over time as τ executes and (2) depend on whether *other* tasks τ' were executed using *their* serial or parallel implementations. The second factor, in particular, creates complicated dependencies—the right choice for one task depends on what choices have been and will be made for others.

In this paper, we propose an alternative perspective: What if the engineer implements both a serial and parallel version of each task, and then leaves it to the *scheduler* to decide which version to use?

Formally, we define the ***serial-parallel decision problem*** as follows: a set $\mathcal{T} = \{\tau_1, \dots, \tau_n\}$ of tasks arrive over time. Each task $\tau_i \in \mathcal{T}$ arrives at some start time t_i, and comes with two implementations: a serial implementation $\tau_i^{\circledcirc}$ with work σ_i and a parallel implementation $\tau_i^{\shortparallel}$ with work $\pi_i > \sigma_i$. In order for the scheduler to begin executing a task τ_i, it must choose (irrevocably) between which of the two implementations to use. If the serial job $\tau_i^{\circledcirc}$ is chosen, then the job can execute on up to one processor at a time (the processor can change), and τ_i completes once σ_i work has been performed on $\tau_i^{\circledcirc}$. If the parallel job $\tau_i^{\shortparallel}$ is chosen, then the job can execute on up to p processors over time (now both the set of processors and the number of processors can change), and τ_i completes once the total work performed on $\tau_i^{\shortparallel}$ (by all processors) reaches π_i.

Two natural objectives for the scheduler are to minimize the ***mean response time*** (MRT), which is the average amount of time between when a task arrives and when it completes; and the ***total awake time***, which is the total amount of time during which there is at least one job executing. We emphasize that, in both cases, our task is fundamentally to solve an online *decision* problem. If the scheduler was told for each task τ_i whether to use $\tau_i^{\circledcirc}$ or $\tau_i^{\shortparallel}$, then the problem would become straightforward. But actually making these decisions is potentially difficult.

Results

In addition to formalizing the *Serial-Parallel Scheduling Problem*, we give upper and lower bounds for how well online schedulers can perform on both awake time and average completion time.

Optimizing awake time. Our first result is a very simple scheduler that optimizes awake time with a competitive ratio of 3. This is a ***decide-on-arrival scheduler***, meaning that it makes its serial/parallel decisions immediately when a job arrives. We also give lower bounds preventing a competitive ratio of $1 + \varepsilon$: we show that any deterministic scheduler must incur competitive ratio at least $\phi - o(1) \approx 1.62$; and that any deterministic *decide-on-arrival* scheduler must incur competitive ratio at least $2 - o(1)$; and that even *randomized* schedulers must incur competitive ratios at least $(3 + \sqrt{3})/4 - o(1) \approx 1.18$.

Our second result studies what we call ***parallel-work-oblivious schedulers***, which are schedulers that get to know the serial work of each job but *not* the parallel work. We show that, even in this setting, it is possible to construct a 6-competitive scheduler for awake time. On the other hand, we show that there is a fundamental tension between *decide-on-arrival* and *parallel-work-oblivious* schedulers: any scheduler that achieves both properties has competitive ratio at least $\Omega(\sqrt{p})$.

Optimizing mean response time. Next, we turn our attention to mean response time (MRT). We construct an online scheduler that achieves a competitive ratio of $O(1)$ for MRT using $O(1)$ speed augmentation. This is our most technical result, and is achieved through three technical components. First, in Section 6.1, we prove two technical lemmas for comparing the optimal schedule for a set of serial jobs to the optimal schedule for perfectly scalable versions of the same jobs. Then, in Section 6.2 we build on this to construct a scheduler that is $O(1)$ competitive *if* it is permitted to sometimes cancel parallel tasks and restart them as serial ones. This cancellation 'superpower' would seem to make the decision problem significantly easier (as decisions are no longer irrevocable). However, in Section 6.3, we show how to take our $O(1)$-competitive scheduler (with cancellation) and transform it into a $O(1)$-competitive scheduler (without cancellation).

Our MRT scheduler is neither a decide-on-arrival scheduler nor an parallel-work-oblivious scheduler. We prove that, if a scheduler is decide-on-arrival and uses $O(1)$ speed augmentation, then it must incur competitive ratio $\Omega(p^{1/4})$.

Extending to Tasks with Dependencies. In Section 7, we extend our model to support arrival dependencies between tasks: Each task τ_i has a set $D_i \subseteq [n]$ of other tasks that must complete *before* τ_i can arrive. Dependencies must be acyclic, but besides that, they can be arbitrary.

In this setting, it is helpful to think of the tasks as representing components of a *single parallel program*. Each component has both a serial and parallel implementation, that the runtime scheduler can choose between. The goal is to minimize the completion time of the entire program—this corresponds to the awake time objective function.

In Section 7, we show that the optimal online competitive ratio for this problem (even for randomized schedulers) is $\Theta(\sqrt{p})$. The upper bound holds for any set of dependencies, and the lower bound holds even when the dependencies form a tree (this means that the lower bound applies, for example, even to fork-join parallel programs [3]).

Open questions. Finally, in Section 10, we conclude with a discussion of open questions.

2 RELATED WORK

There is a vast literature on multiprocessor scheduling problems. For an excellent (but now somewhat dated) survey, see [4]. Past work has often categorized sets of jobs $J = \{j_1, \ldots, j_n\}$ as being composed of jobs j_i which are either rigid, moldable, or malleable. Both ***rigid*** and ***moldable*** jobs have the property that once a job j_i begins on some number p_{j_i} of processors, it must continue on that same set of p_{j_i} processors without interruption until completion. Rigid and moldable jobs differ in that for rigid jobs the number p_{j_i} of processors which task j_i is to be run on is pre-specified, whereas for moldable jobs p_{j_i} may be chosen by the scheduler. Finally, if the number (and set) of processors on which a job is executed is permitted to vary over time, then the job is said to be ***malleable***. In the contexts of moldable and malleable jobs, the jobs often come with ***speedup curves*** determining how quickly the job can make progress on a given number of processors. If the speedup curve is proportional to the number of processors on which the job runs (as is the case for the parallel jobs associated with the tasks described in this paper) then the job is said to be ***perfectly scalable***.

Much of the work in this area focuses on optimizing awake time, which as discussed earlier, is the total amount of time during which any jobs are present. Here, there has been a great deal of work on both offline schedulers [12, 14–17] and online schedulers [1, 5, 9–11, 18, 19].

For moldable jobs with arbitrary speedup curves, Ye, Chen, and Zhang [18] show an online competitive ratio of $O(1)$ for awake time. Of special interest to this paper would be the speedup curve where job j_i takes time σ_i to complete on 1 processor and time π_i/k to complete on $k > 1$ processors. In this case, the scheduler's commitment to a fixed number of processors would also implicitly represent a commitment to running the job in serial or parallel. One difference between this and the problem studied here is that the scheduler (and the OPT to which it is compared) are *non-preemptive*: they are required to execute the tasks on a fixed set of processors (without interruption). Nonetheless, it is not too difficult to show that Ye, Chen, and Zhang's algorithm actually *does* yield an $O(1)$ competitive awake-time algorithm for our problem—we emphasize, however, that this algorithm is neither decide-on-arrival nor parallel-work-oblivious, and would achieve a quite large constant competitive ratio. Interestingly, in the context of mean response time (MRT), we show in Section 9 that preemption is necessary: in the context of our serial-parallel decision problem, any online scheduler that rigidly assigns jobs to fixed numbers of processors will necessarily incur a worst-case competitive ratio of $\omega(1)$ for MRT (even with $O(1)$ resource augmentation).

There is of course also a great deal of interest MRT, i.e., the average amount of time between when tasks arrive and when they are completed. Besides work on offline approximation algorithms

[15, 16], most of the major successes in this area have been for malleable jobs [6–8]. The seminal result in this area is due to Edmonds [7], who considered malleable jobs with arbitrary sub-linear nondecreasing speedup curves, and showed that the so-called EQUI scheduler, which divides the processors evenly among all of the jobs present (using time sharing if there are more than p jobs), achieves a competitive ratio of $O(1)$ with $2 + \varepsilon$ speed augmentation (subsequent work only requires speed augmentation $1 + \varepsilon$ with different schedulers [6, 8]). A remarkable feature of the EQUI scheduler is that it is *oblivious* to the precise speedup curve of each job. In Section 9, we show that such a scheduler is not possible in our setting—any oblivious online scheduler for MRT (or, even any parallel-work-oblivious scheduler) must incur a worst-case competitive ratio of $\omega(1)$.

The tasks studied in this paper do not fit neatly into the rigid/moldable/malleable framework. They represent instead a direction that until now seems to have been unexplored: deciding for each task between the two extremes of (1) a fast algorithm with no parallelism and (2) a slower algorithm with ample parallelism and perfect scalability. Interestingly, several parts of the analysis end up making use of EQUI as an *analytical tool*. Thus, for completeness, we restate Theorem 1.1 of [8] (with parameters $\beta = 1$ and $\varepsilon = 1$) below:

Theorem 2.1 (Theorem 1.1 of [8]). EQUI *with* 3 *speed augmentation is* $O(1)$ *competitive with* OPT *for MRT on any set* J *of malleable jobs with arbitrary (nondecreasing sublinear) speedup curves.*

3 PRELIMINARIES

Problem Specification. In this section we introduce our terminology and notation for describing the problem. A ***task*** is some computation that must be performed. Tasks can be performed using a serial or parallel ***job*** implementing the task. In the ***Serial-Parallel Scheduling Problem*** a scheduler receives tasks $\mathcal{T} = \{\tau_1, \tau_2, \ldots, \tau_n\}$ over time, with n unknown beforehand in the on-line case. Task τ_i has an associated serial job $\tau_i^{\circledcirc}$ with work σ_i and an associated parallel job $\tau_i^{\shortparallel}$ with work π_i. Finally, task τ_i arrives at time t_i with $t_1 \le t_2 \le \cdots \le t_n$. Thus one should think of τ_i as being determined by a triple (σ_i, π_i, t_i).

The scheduler must decide whether to perform each task τ_i using the serial or parallel implementation. By default the scheduler need not decide which implementation to run for a task exactly when the task arrives, sometimes it may be beneficial to wait before starting a task. We also consider the alternative model where the scheduler must decide on arrival which implementation to use.

For convenience, we treat time steps as being small enough that time is essentially continuous. At each time step, the scheduler allocates its p processors to the unfinished jobs present. In a given time step multiple processors can be allocated to a parallel job, but only a single processor can be allocated to each serial job (and, of course, some jobs may be allocated 0 processors). Sometimes it is convenient to treat a job as being assigned to a fractional number of processors; this can be accomplished by time-sharing the processor over multiple steps. A serial job $\tau_i^{\circledcirc}$ finishes once it has been allocated σ_i time (not necessarily contiguously) on a processor (not necessarily the same one over time). A parallel job $\tau_i^{\shortparallel}$ finishes once it has been allocated π_i total time on processors—i.e., the integral over time of the number of processors allocated to $\tau_i^{\shortparallel}$ reaches π_i.

We refer to a set of tasks $\mathcal{T} = \{\tau_1, \tau_2, \ldots, \tau_n\}$ specifying an instance of the Serial-Parallel Scheduling Problem as a ***task arrival process*** or ***TAP***. We use $[n]$ to denote $\{1, 2, \ldots, n\}$, and $\mathcal{T}_i^j$ to denote $\{\tau_i, \ldots, \tau_j\}$

Objective. We say that a task is ***alive*** if it has arrived but not yet been completed. We consider two objective functions:

- minimize ***mean response time*** (MRT): the average amount of time that tasks are alive. If task τ_i finishes at time f_i, then the MRT is $\frac{1}{n}\sum(f_i - t_i)$. It is equivalent, but generally more convenient, to work with a scaled version of MRT called ***total response time*** (TRT)—the sum of the amounts of time that tasks are alive. We denote the TRT of a scheduler ALG on a TAP $\mathcal{T}$ by $\mathsf{TRT}_{\mathsf{ALG}}^{\mathcal{T}}$.
- minimize ***awake time*** (T): the total amount of time that there are alive tasks. If there are tasks alive on intervals $[a_1, b_1] \sqcup [a_2, b_2] \sqcup \cdots$, then the awake time is $\sum(b_i - a_i)$. We denote the awake time of a scheduler ALG on TAP $\mathcal{T}$ by $\mathsf{T}_{\mathsf{ALG}}^{\mathcal{T}}$.

We measure a scheduler's performance by comparison to the optimal off-line scheduler OPT, who can see the sequence of tasks in advance. The ***competitive ratio*** of a scheduler ALG on TAP $\mathcal{T}$ is the ratio of its performance to OPT's, e.g. $\mathsf{TRT}_{\mathsf{ALG}}^{\mathcal{T}} / \mathsf{TRT}_{\mathsf{OPT}}^{\mathcal{T}}$ for TRT. More generally, we will say ALG is ***k competitive*** if the competitive ratio of ALG is bounded by k on all TAPs. Sometimes we will also say ALG is k competitive with another scheduler ALG′, meaning that the MRT (or awake time) of ALG is never more than a factor of k larger than the MRT (or awake time, respectively) of ALG′.

Finally, when comparing two schedulers ALG_1 and ALG_2 in the context of MRT we will often assume ***c speed augmentation*** for some $c \in O(1)$. This means that ALG_1 gets to use processors that are c times faster than those used by ALG_2. Let $c \cdot \mathcal{T}$ denote the TAP $\mathcal{T}$ but with every job's work multiplied by c; similarly define $c \cdot J$ for a set of jobs J as the jobs from J with work multiplied by c. Then, the statement ALG_1 with c speed augmentation is $O(1)$ competitive with ALG_2 on TAP $\mathcal{T}$ (or jobs J) can be written as

$$\mathsf{TRT}_{\mathsf{ALG}_1}^{\mathcal{T}} \le O(\mathsf{TRT}_{\mathsf{ALG}_2}^{c \cdot \mathcal{T}}).$$

Our goal is to create a scheduler that is $O(1)$ competitive with OPT, potentially with use of $O(1)$ speed augmentation in the case of MRT.

Technical Details. We emphasize that our focus is on schedulers that are allowed to ***preempt*** running tasks, i.e. pause running tasks and continue later on a potentially different set of processors. For MRT, in particular, preemption is provably necessary—we show in Section 9 that a non-preemptive scheduler cannot be $O(1)$-competitive for MRT. It is sometimes theoretically helpful to consider schedulers that are allowed to ***cancel*** running tasks, i.e. stop a running task, erasing all progress made on the task, and restart the task using a different implementation (e.g., a parallel task can be cancelled and restarted as a serial task). Our final schedulers *will not* require cancelling. However, in designing/analyzing our scheduler, it will be helpful to be able to imagine what would have happened if cancelling were possible. We say "ALG, ***with use of cancelling***" to denote that a scheduler ALG has been augmented with the ability to cancel.

We consider only TAPs where the ***cost ratio*** π_i/σ_i satisfies $\pi_i/\sigma_i \in [1, p]$ for all tasks τ_i. If $\pi_i/\sigma_i < 1$ then the scheduler clearly should never run τ_i in serial so we can replace the serial implementation with the parallel implementation to get cost ratio 1. Similarly, if $\pi_i/\sigma_i > p$ then the scheduler should never run τ_i in parallel and we can replace the parallel implementation with the serial implementation to get cost ratio p.

Throughout the paper we will assume $p \geq \Omega(1)$ is at least a sufficiently large constant. We are more interested in the asymptotic behavior of our schedulers as a function of p than the behavior on small values of p.

4 A 3-COMPETITIVE AWAKE-TIME SCHEDULER THAT MAKES DECISIONS ON ARRIVAL

The scheduler we describe in this section belongs to a simple class of schedulers called ***decide-on-arrival*** schedulers. Whenever a decide-on-arrival scheduler receives a task it must immediately make an irrevocable decision about whether to run the serial or parallel job associated with the task. Our scheduler will run its chosen serial jobs via ***most-work-first***, i.e., run up to p of the serial jobs with the most remaining work at each time step and run a parallel job on any remaining processors. Clearly this is an optimal method for scheduling any given set of jobs. Thus, in the decide-on-arrival model the Serial-Parallel Scheduling Problem is really not a scheduling problem but rather a decision problem: once the scheduler chooses which job to run for each task it is clear how to schedule the jobs. We define OPT to also be a decide-on-arrival scheduler. This is without loss of generality: OPT does not benefit from delaying its decisions because OPT has all the information in advance. Thinking of OPT as a decide-on-arrival scheduler will simplify the terminology. We now define specialized notation that is helpful in describing the decisions of decide-on-arrival schedulers:

Definition 4.1. A scheduler ALG is ***saturated*** on time step t if ALG has no idle processors at time t. We say that ALG is ***balanced*** after the arrival of n_0 tasks if ALG would be saturated immediately before completing these n_0 tasks, assuming no further tasks arrive. In other words, being balanced means that ALG's current set of jobs could be distributed so that, assuming no additional tasks arrive, all processors would be in use at each time step until ALG has completed all the currently present jobs. If ALG is not balanced we say that ALG is ***jagged***.

We say that ALG ***incurs*** work W on a task τ if it takes a job requiring W work to complete. We say that ALG ***incurs*** work W on a TAP $\mathcal{T}$ if ALG does W total work to handle the tasks in $\mathcal{T}$.

We now present our simple 3-competitive scheduler.

Theorem 4.2. *There is a 3-competitive decide-on-arrival scheduler for awake time.*

Proof. We propose the scheduler BAL which is always balanced. Whenever a task τ arrives

- If taking $\tau^{\circledcirc}$ would cause BAL to become jagged BAL takes $\tau^{\shortparallel}$.
- Otherwise BAL takes $\tau^{\circledcirc}$.

To analyze BAL it suffices to analyze TAPs where BAL always has at least one uncompleted task present at any time from the start of time until the completion of the final task. Thus, the awake time of BAL is the same as its completion time. OPT's awake time may be less than OPT's completion time; we call the intervals of time when OPT has already completed all arrived tasks ***gaps***. We call the maximal intervals of time when OPT has uncompleted tasks which have already arrived ***OPT-awake-intervals***.

Our main technical lemma is the following:

Lemma 4.3. *Fix some* OPT*-awake-interval I. Let $\mathcal{T}(I)$ denote the set of tasks that arrive during I and let $\mathcal{T}'$ denote the tasks that arrive before I. Assume that immediately before the start of I* BAL *has B work present (and is of course balanced). Let $\mathsf{C}_{\mathsf{ALG}}$ for* ALG $\in$ {BAL, OPT} *denote the time from the start of I until when* ALG *would complete on the TAP $\mathcal{T}(I) \cup \mathcal{T}'$. Then*

$$\mathsf{C}_{\mathsf{BAL}} \leq B/p + 3\,\mathsf{C}_{\mathsf{OPT}}\,.$$

Proof. Let $\mathcal{T}(I,i)$ denote the first i tasks in $\mathcal{T}(I)$. Let $\mathsf{K}^i_{\mathsf{OPT}}$ denote the work that OPT incurs on $\mathcal{T}(I,i)$. Let $\mathsf{C}^i_{\mathsf{ALG}}$ for ALG $\in$ {BAL, OPT} denote completion time of ALG on the TAP $\mathcal{T}' \cup \mathcal{T}(I,i)$ minus the start time of the OPT-awake-interval I. We will prove the lemma by induction on i, i.e., how many of the tasks in $T(I)$ to consider. One complication with the proof is that OPT's schedule on $\mathcal{T}(I,i)$ can be very far from optimal; for instance, OPT might schedule a very large task $\tau \in \mathcal{T}(I,i)$ in serial if it knows that future tasks in $\mathcal{T}(I)$ cause τ to not bottleneck completion time. Nonetheless with an appropriately constructed invariant we can control the evolution of the work profiles of BAL and OPT as we increase i.

In particular, we will inductively show the following claim: For each $i \in \{0, 1, \ldots, |\mathcal{T}(I)|\}$ we have the invariant:

$$\mathsf{C}^i_{\mathsf{BAL}} \leq B/p + 2\,\mathsf{C}^i_{\mathsf{OPT}} + \mathsf{K}^i_{\mathsf{OPT}}/p. \tag{1}$$

When $i = 0$ we have $\mathsf{C}^0_{\mathsf{BAL}} = B/p$ and the invariant is satisfied. Now we assume the invariant is true for some $i > 0$ and consider how the addition of the the $(i+1)$-th task τ' impacts the invariant. We use σ', π' to denote the serial and parallel works of τ'. If BAL chooses an implementation of τ' requiring $K \in \{\sigma', \pi'\}$ work and OPT chooses an implementation of τ' requiring at least K work then

$$\mathsf{C}^{i+1}_{\mathsf{BAL}} - \mathsf{C}^i_{\mathsf{BAL}} = K/p \leq \mathsf{K}^{i+1}_{\mathsf{OPT}}/p - \mathsf{K}^i_{\mathsf{OPT}}/p$$

and so the invariant for i implies the invariant for $i+1$.

The only remaining case is if OPT runs τ' in serial while BAL runs τ' in parallel. However, in this case τ' must have relatively large serial work. In particular, if we let t'_0 denote the time at the beginning of I and t'_{i+1} denote arrival time of τ' then $\sigma' > \mathsf{C}^i_{\mathsf{BAL}} + t'_0 - t'_{i+1}$ or else BAL would have chosen to run τ' in serial. Thus we have

$$\mathsf{C}^{i+1}_{\mathsf{BAL}} = \mathsf{C}^i_{\mathsf{BAL}} + \pi'/p \leq \mathsf{C}^i_{\mathsf{BAL}} + \sigma' \leq t'_{i+1} - t'_0 + 2\sigma' \leq 2(t'_{i+1} - t'_0 + \sigma'). \tag{2}$$

On the other hand, OPT ran τ' in serial and does not have any gaps between t'_0, t'_{i+1} because these times occur during the same OPT-awake-interval I. Thus

$$\mathsf{C}^{i+1}_{\mathsf{OPT}} \geq t'_{i+1} - t'_0 + \sigma'.$$

Combined with Equation (2) this gives

$$\mathsf{C}^{i+1}_{\mathsf{BAL}} \leq 2\,\mathsf{C}^{i+1}_{\mathsf{OPT}},$$

and so the invariant holds in this case as well. This completes the proof of the inductive claim.

To conclude the lemma we use $i = |\mathcal{T}(I)|$ in the inductive claim, which gives:

$$\mathsf{C}_{\mathsf{BAL}} \le B/p + 2\,\mathsf{C}_{\mathsf{OPT}} + \mathsf{K}_{\mathsf{OPT}}^{|\mathcal{T}(I)|}/p. \quad (3)$$

Using the fact $\mathsf{K}_{\mathsf{OPT}}^{|\mathcal{T}(I)|}/p \le \mathsf{C}_{\mathsf{OPT}}$ in Equation (3) completes the proof of the lemma. □

To finish our analysis of BAL we inductively show that BAL is 3-competitive on any prefix of the OPT-awake-intervals. As a base-case we can take the first OPT-awake-interval. Applying Lemma 4.3 with $B = 0$ shows that BAL is 3-competitive on the first OPT-awake-interval. The inductive step is:

Corollary 4.4. *Let $\mathcal{T}^{(i)}$ denote the set of tasks that arrive during the first i OPT-awake-intervals. Assume that BAL is 3-competitive with OPT on $\mathcal{T}^{(i)}$. Then BAL is 3-competitive with OPT on $\mathcal{T}^{(i+1)}$.*

Proof. Let I denote the $(i+1)$-th OPT-awake-interval and let $\mathcal{T}(I)$ denote the set of tasks that arrive during I. Let W be the work performed by BAL before I and let B denote the work that BAL has remaining immediately before the start of I. By assumption we have

$$(B + W)/p \le 3\,\mathsf{T}_{\mathsf{OPT}}^{\mathcal{T}^{(i)}}. \quad (4)$$

Let $\mathsf{C}_{\mathsf{ALG}}$ for $\mathsf{ALG} \in \{\mathsf{BAL}, \mathsf{OPT}\}$ denote the completion time of ALG on the tasks $\mathcal{T}^{(i+1)}$ measured from the start of I; i.e., if ALG completes $\mathcal{T}^{(i+1)}$ at time t_{ALG} and I starts at time t_I then $\mathsf{C}_{\mathsf{ALG}} = t_{\mathsf{ALG}} - t_I$. By Lemma 4.3 we have

$$\mathsf{C}_{\mathsf{BAL}} \le B/p + 3\,\mathsf{C}_{\mathsf{OPT}}. \quad (5)$$

Combining Equation (4), Equation (5) gives

$$\begin{aligned} \mathsf{T}_{\mathsf{BAL}}^{\mathcal{T}^{(i+1)}} &= W/p + \mathsf{C}_{\mathsf{BAL}} \\ &\le W/p + 3\,\mathsf{C}_{\mathsf{OPT}} + B/p \\ &\le 3(\mathsf{T}_{\mathsf{OPT}}^{\mathcal{T}^{(i)}} + \mathsf{T}_{\mathsf{OPT}}^{\mathcal{T}(I)}) \\ &= 3\,\mathsf{T}^{\mathcal{T}^{(i+1)}}. \end{aligned}$$

□

Using Corollary 4.4 on all the OPT-awake-intervals shows that BAL is 3-competitive on $\mathcal{T}$.

□

5 A 6-COMPETITIVE AWAKE-TIME SCHEDULER THAT IS PARALLEL-WORK OBLIVIOUS

Now we turn our attention to designing a ***parallel-work-oblivious*** scheduler, which is a scheduler that is not allowed to see the parallel works of each task. We show that it is still possible to construct an $O(1)$-competitive scheduler for awake time in this model; this contrasts with MRT where Proposition 9.1 asserts that knowledge of the parallel works is necessary to achieve a good competitive ratio for MRT. In particular, we show:

Theorem 5.1. *There is a deterministic parallel-work-oblivious 6-competitive scheduler for awake time.*

We call our scheduler UNK. Whenever there are idle processors, UNK takes any not-yet-started (but already arrived) task τ_i and:

- If τ_i arrived more than σ_i time ago UNK runs $\tau_i^{\circledcirc}$.
- Otherwise UNK runs $\tau_i^{\shortparallel}$.

At each time step UNK allocates a processor to each of the at-most-p running serial jobs. There will be at most one parallel task running at a time. If there is a running parallel job UNK allocates any processors not being used to run serial jobs to this single running parallel job. At any time some tasks may have arrived but not yet been started; that is, UNK is not a decide-on-arrival scheduler.

Now we analyze UNK using two lemmas. As in Section 4 we say UNK is saturated on a time step if all p processors are in use, and unsaturated otherwise. Also, as in the previous section, it suffices to analyze TAPs where UNK always has at least one uncompleted task present at any time from the start of time until the completion of the final task. We will make this assumption wlog for the rest of the section, meaning that the awake time of UNK is equal to the completion time.

Lemma 5.2. *UNK is unsaturated at most 1/2 of the time.*

Proof. Let $W_1, W_2, \ldots$ denote the maximal intervals of time when UNK is saturated; we call these ***saturated intervals***. Similarly we refer to maximal intervals where UNK is unsaturated as ***unsaturated intervals***. Observe that, whenever we are in an unsaturated interval, every task that has arrived so far either must have already completed or must be currently running in serial. That is, during an unsaturated interval there are never parallel jobs running and there are never jobs sitting around without being started.

For each saturated interval W_i let W_i' denote an interval of the same length as W_i but shifted to start exactly at the end of interval W_i. We claim that $\bigcup_i W_i'$ covers all unsaturated intervals.

Fix some unsaturated interval $[a, b]$. Let τ_j be the serial task with the most remaining work present at time a. Let W_i be the saturated interval when τ_j was started. We will show that $[a, b] \subseteq W_i'$. Let t denote the time when τ_j is started. Observe that UNK must be saturated for all of $[t_j, t]$ or else τ_j would have been started in parallel. Thus, $[t_j, t] \subseteq W_i$. Further, observe that $t - t_j \ge \sigma_j$, because UNK must wait σ_j time before starting τ_j in serial. In particular this means that $|W_i| \ge \sigma_j$.

Next, we claim that at all times in $[t, b]$, UNK allocates a processor to τ_j. Indeed, suppose that at some time step in $[t, b]$ τ_j was not allocated a processor. This would mean that there are p other serial tasks with at least as much remaining work as τ_j. But in this case UNK would remain saturated until τ_j is completed, contradicting the fact that UNK is not saturated during times $[a, b]$. Thus, $b \le t + \sigma_j$, and so $[a, b] \subseteq [t, t + \sigma_j]$. Finally, recall that $|W_i| \ge \sigma_i$ so $[a, b] \subseteq [t, t + |W_i|]$ and consequently $[a, b] \subseteq W_i'$.

Of course $\left|\bigcup_i W_i'\right| \le \sum_i |W_i|$. Thus UNK is unsaturated at most 1/2 of the time.

□

Lemma 5.3. *The amount of time that UNK is saturated is at most $3\,\mathsf{T}_{\mathsf{OPT}}$.*

Proof. For each task τ, let A_τ be the interval of time between when τ arrives and when OPT completes τ. Let $A = \bigcup_{\tau \in \mathcal{T}} A_\tau$ be the set of times when OPT has uncompleted tasks, and let B denote the set of times when OPT has no uncompleted tasks present. We divide tasks τ into four categories [1]:

(1) UNK runs $\tau^{\circledcirc}$.
(2) UNK runs $\tau^{\parallel}$ starting at some time after the end of A_τ.
(3) UNK runs $\tau^{\parallel}$ entirely within time steps in A.
(4) UNK starts $\tau^{\parallel}$ during A_τ, but part of τ's execution time by UNK occurs during B.

We now analyze the performance of UNK on each category of task.

Claim 1. *UNK's total work on tasks of category (1) and (2) is at most* $p\,\mathsf{T}_{\mathsf{OPT}}$.

Proof. For each task τ_i of category (1) OPT must have incurred at least σ_i work; this is true of all tasks. For each task τ_i of category (2) τ_i has not yet been available for σ_i time steps when UNK starts τ_i in parallel. Thus, for OPT to have already finished τ_i by the time that UNK starts τ_i OPT must have also run τ_i in parallel. Thus, OPT incurs work π_i for task τ_i. Therefore, OPT incurs at least as much work as UNK on all tasks of categories (1) and (2). The amount of work that OPT performs is at most $p\,\mathsf{T}_{\mathsf{OPT}}$, which then bounds UNK's work on tasks of category (1) and (2). ◻

Claim 2. *UNK's work on tasks of category (3) is at most* $p\,\mathsf{T}_{\mathsf{OPT}}$.

Proof. Tasks of category (3) all run during time steps in A, so the work spent on such tasks is at most $p \cdot |A| = p\,\mathsf{T}_{\mathsf{OPT}}$. ◻

Claim 3. *UNK's work on tasks of category (4) is at most* $p\,\mathsf{T}_{\mathsf{OPT}}$.

Proof. Let $W_1^{\mathsf{OPT}}, W_2^{\mathsf{OPT}}, \ldots$ denote the (maximal) intervals of time when OPT has uncompleted tasks. For each W_i^{OPT} there is at most one task τ_{k_i} that UNK starts in parallel during W_i^{OPT} whose execution time overlaps with B: this is because UNK only runs a single parallel task at a time. Let K denote the set of category (4) tasks. For each category (4) task $\tau_{k_i} \in K$, the corresponding W_i^{OPT} must have size at least π_{k_i}/p because OPT completes task τ_{k_i} during W_i^{OPT}. Thus,

$$\mathsf{T}_{\mathsf{OPT}} = \sum_i |W_i^{\mathsf{OPT}}| \geq \sum_{\tau_{k_i} \in K} \pi_{k_i}/p. \tag{6}$$

The right hand side of Equation (6) is UNK's work on the category (4) tasks, giving the desired bound. ◻

Combining the previous three claims, the total work completed by UNK during saturated steps is at most $3p\,\mathsf{T}_{\mathsf{OPT}}$. At each saturated step p units of work are performed. Thus, the total number of saturated time steps is at most $3\,\mathsf{T}_{\mathsf{OPT}}$. ◻

Combined, the previous lemmas prove Theorem 5.1.

6 MINIMIZING MRT

In this section we present our main result: a scheduler that, with $O(1)$ speed augmentation, is $O(1)$ competitive for MRT (or equivalently TRT).

[1] The categories are not mutually exclusive. If a task τ falls in multiple categories we over-charge UNK for τ's work.

6.1 Two Technical Lemmas

In our analyses, it will be helpful to compare two settings: one in which a set of jobs J must be executed with every job in serial, and the other in which the same set of jobs J must be executed but where every job is perfectly scalable (i.e., $\pi_i = \sigma_i$).

Lemma 6.1. *Let $J_{\circledcirc}$ be a set of serial jobs with arbitrary arrival times. Let $J_{\parallel}$ be jobs of the same work as jobs in $J_{\circledcirc}$ but that are perfectly scalable. Then*

$$\mathsf{TRT}_{\mathsf{OPT}}^{J_{\circledcirc}} \leq O\left(\mathsf{TRT}_{\mathsf{OPT}}^{O(1)\cdot J_{\parallel}} + \sum_{j_i \in J_{\circledcirc}} \mathrm{work}(j_i)\right).$$

Proof. We prove the lemma by constructing a scheduler SSS that achieves mean response time at most

$$O\left(\mathsf{TRT}_{\mathsf{OPT}}^{12 J_{\parallel}} + \sum_{j_i \in 12 J_{\circledcirc}} \mathrm{work}(j_i)\right).$$

The ***Silly-Serious*** scheduler SSS operates in 2 modes: ***silly mode***, where there are less than p unfinished jobs alive, and ***serious mode***, where there are at least p unfinished jobs alive. Define ***serious intervals*** and ***silly intervals*** to be maximal contiguous sets of time where SSS is in the respective modes. When discussing SSS, we will assume that it has access to $2p$ processors (rather than just p). This can be simulated using a factor-of-2 speed augmentation and time sharing.

During silly mode SSS schedules each job on a single processor. As new jobs arrive, once the total number of jobs present reaches p, SSS enters serious mode. As a boundary condition, we consider any jobs that arrive in that moment to have arrived *during* the serious interval. We refer to the jobs that arrive during the serious interval as ***scary*** (for this serious interval).

During a given serious interval, SSS uses $2p$ total processors: it puts the at-most-p non-scary jobs from the prior silly interval onto p processors, and it schedules the scary jobs via EQUI on the other p processors.

We remark that there may be fewer than p scary jobs, in which case EQUI may want to allocate multiple processors to a single job. In this case SSS does not actually schedule the (serial) job on multiple processors but simply runs it on its own processor. Because the jobs that SSS is running are serial only, we can think of them as having flat speedup curves, i.e. they require the same amount of time to run regardless of how many processors they are run on. So, when EQUI tries to run a job on multiple processors, the progress is the same as if we were to run it on a single processor. Thus we can think of SSS as faithfully simulating EQUI on the scary jobs.

We bound the TRT incurred by SSS with 6 speed augmentation by partitioning the TRT into three parts and bounding each part.

We can bound the total TRT incurred by SSS during silly intervals by $\sum_{j_i \in J_{\circledcirc}} \mathrm{work}(j_i)$, because every job has a dedicated processor at all times during a silly interval. Thus we will focus the rest of the proof on serious intervals.

Now we fix a serious interval $I = [t_a, t_b]$ to analyze. Let $X_{\circledcirc}$ be the scary jobs for I, but with each job's work truncated to be the amount of work that the job completes during I. Similarly, let $Y_{\circledcirc}$ be the non-scary jobs that run during I, but with each of their works also reduced to be the work completed by that job during I. We claim the following chain of inequalities holds for the jobs in the serious interval I:

$$\begin{aligned}\mathsf{TRT}_{\mathsf{SSS}}^{X_{\circledcirc}\cup Y_{\circledcirc}} &\le \mathsf{TRT}_{\mathsf{EQUI}}^{2(X_{\circledcirc}\cup Y_{\circledcirc})} &(7)\\ &= \mathsf{TRT}_{\mathsf{EQUI}}^{2(X_{\shortparallel}\cup Y_{\shortparallel})} &(8)\\ &\le \mathsf{TRT}_{\mathsf{OPT}}^{6(X_{\shortparallel}\cup Y_{\shortparallel})} &(9)\\ &\le \mathsf{TRT}_{\mathsf{OPT}}^{12X_{\shortparallel}} + \sum_{j_i\in 12Y_{\shortparallel}} \mathrm{work}(j_i). &(10)\end{aligned}$$

Note that the first expression $\mathsf{TRT}_{\mathsf{SSS}}^{X_{\circledcirc}\cup Y_{\circledcirc}}$ represents the TRT for SSS during serious interval I.

Inequality (7)**:** SSS's treatment of $X_{\circledcirc}\cup Y_{\circledcirc}$ on $2p$ processors (which it is granted via factor-of-2 speed augmentation) is at least as good as running EQUI on $X_{\circledcirc}\cup Y_{\circledcirc}$ with p processors, because SSS runs EQUI on $X_{\circledcirc}$ and then separately allocates one processor to each job in $Y_{\circledcirc}$.

Inequality (8)**:** We denote by $X_{\shortparallel}, Y_{\shortparallel}$ perfectly scalable versions of the jobs in $X_{\circledcirc}, Y_{\circledcirc}$. Since there are at least p jobs at all times during a serious interval, EQUI's treatment of $2(X_{\circledcirc}\cup Y_{\circledcirc})$ and EQUI's treatment of $2(X_{\shortparallel}\cup Y_{\shortparallel})$ are actually the same: it equally partitions the processors amongst the available jobs, potentially using time sharing; crucially EQUI never assigns more than 1 processor to any job because there are a sufficiently large number of jobs.

Inequality (9)**:** By Theorem 2.1 EQUI on $2(X_{\shortparallel}\cup Y_{\shortparallel})$ is $O(1)$ competitive with OPT on $6(X_{\shortparallel}\cup Y_{\shortparallel})$.

Inequality (10)**:** OPT on $6(X_{\shortparallel}\cup Y_{\shortparallel})$ using p processors is at least as good as OPT on $12(X_{\shortparallel}\cup Y_{\shortparallel})$ using $2p$ processors. This, in turn, is at least good as the TRT incurred by OPT on $12X_{\shortparallel}$ using p processors plus the TRT incurred by OPT on $12Y_{\shortparallel}$ using p processors, i.e., $\mathsf{TRT}_{\mathsf{OPT}}^{6X_{\shortparallel}} + \mathsf{TRT}_{\mathsf{OPT}}^{6Y_{\shortparallel}}$. Finally, since $\mathsf{TRT}_{\mathsf{OPT}}^{12Y_{\shortparallel}} \le \sum_{j_i\in 12Y_{\shortparallel}} \mathrm{work}(j_i)$, we get (10) as desired.

Total TRT incurred by serious intervals: We can now bound the total TRT incurred by serious intervals as follows. Let $I_1, I_2, \ldots$ be the serious intervals, and define $X_{\shortparallel}^{(1)}, X_{\shortparallel}^{(2)}, \ldots$ and $Y_{\shortparallel}^{(1)}, Y_{\shortparallel}^{(2)}, \ldots$ so that $X_{\shortparallel}^{(k)}$ is $X_{\shortparallel}$ for interval I_k and $Y_{\shortparallel}^{(k)}$ is $Y_{\shortparallel}$ for interval I_k. The jobs in each $X_{\shortparallel}^{(k)}$ and each $Y_{\shortparallel}^{(k)}$ represent portions of jobs from $J_{\shortparallel}$. Note that, although a given job from $J_{\shortparallel}$ could appear in multiple $Y_{\shortparallel}^{(k)}$'s, each job appears in at most one $X_{\shortparallel}^{(k)}$ since each job can be scary in at most one serious interval.

By the inequalities above, we have that the total response time spent in serious intervals is at most

$$\sum_k \left(\mathsf{TRT}_{\mathsf{OPT}}^{12X_{\shortparallel}^{(k)}} + \sum_{j_i\in 12Y_{\shortparallel}^{(k)}} \mathrm{work}(j_i) \right).$$

Since each job in $J_{\shortparallel}$ appears in at most one $X_{\shortparallel}^{(k)}$, we have that

$$\sum_k \mathsf{TRT}_{\mathsf{OPT}}^{12X_{\shortparallel}^{(k)}} \le \mathsf{TRT}_{\mathsf{OPT}}^{12J_{\shortparallel}}.$$

Moreover, since the jobs in the $Y_{\shortparallel}^{(k)}$'s each represent disjoint portions of the jobs in $J_{\shortparallel}$, we have that

$$\sum_k \sum_{j_i\in 12Y_{\shortparallel}^{(k)}} \mathrm{work}(j_i) \le \sum_{j_i\in 12J_{\shortparallel}} \mathrm{work}(j_i) \le \mathsf{TRT}_{\mathsf{OPT}}^{12J_{\shortparallel}}.$$

Thus the total response time spent in serious intervals is at most $2\,\mathsf{TRT}_{\mathsf{OPT}}^{12J_{\shortparallel}}$, which completes the proof. ◻

It will also be helpful to consider the setting in which we are comparing a set of perfectly scalable jobs to a set of serial jobs that arrive slightly later.

Lemma 6.2. *Let $J = \{j_1, \ldots, j_n\}$ be a set of perfectly scalable jobs, where job j_i has work $2w_i$ and arrival time t_i. Let $K = \{k_1, \ldots, k_n\}$ be a set of serial jobs where job k_i has work at most w_i and arrival time $t_i + w_i$. Then,*

$$\mathsf{TRT}_{\mathsf{OPT}}^{K} \le O\left(\mathsf{TRT}_{\mathsf{OPT}}^{O(1)\cdot J} + \sum_i w_i\right).$$

Proof. Define J' to be the set of jobs J, but where each job is serial rather than perfectly scalable. Then

$$\mathsf{TRT}_{\mathsf{OPT}}^{K} \le \mathsf{TRT}_{\mathsf{OPT}}^{J'},$$

since when scheduling a serial job with $2w_i$ work, we would rather the job have w_i less work than have the same job show up at time w_i earlier. Finally, by Lemma 6.1,

$$\mathsf{TRT}_{\mathsf{OPT}}^{J'} \le O\left(\mathsf{TRT}_{\mathsf{OPT}}^{12\cdot J} + \sum_i w_i\right).$$

This completes the proof. ◻

6.2 A Cancelling Scheduler

In this subsection we define and analyze a scheduler CANC to prove Theorem 6.3. Note that CANC uses cancelling, i.e. can kill tasks and restart them with a different implementation; in Theorem 6.6 we remove the need for cancelling.

Theorem 6.3. *There is an online scheduler which, using $O(1)$ speed augmentation **and cancelling**, is $O(1)$ competitive for* MRT.

We now describe the operation of CANC on TAP $\mathcal{T}$. CANC starts by defining a set of "***relaxed jobs***" J' which incorporate the serial and parallel jobs from $\mathcal{T}$ into their speed-up curves; CANC will simulate running jobs J' as a subroutine to determine how to schedule $\mathcal{T}$. In particular, for each task $\tau_i \in \mathcal{T}$ we form a relaxed job $j_i' \in J'$ with total work $2\sigma_i$ and the following speedup curve:

- j_i' receives no speedup on $x < \pi_i/\sigma_i$ processors.
- j_i' receives speedup $x\cdot\sigma_i/\pi_i$ on $x \ge \pi_i/\sigma_i$ processors.

When describing CANC we will assume that it has access to $2p$ processors; this can be simulated using a factor-of-2 speed augmentation. CANC schedules $\mathcal{T}$ as follows:

- CANC maintains a pool of p processors for running parallel jobs and a pool of p processors for running serial jobs.
- Initially tasks are placed in the parallel pool and will be run with their parallel implementation.
- CANC manages the parallel pool by simulating EQUI on the relaxed jobs j_i' and then actually running $\tau_i^{\shortparallel}$ during the simulated execution slots for j_i'.
- Whenever a task τ_i been in the parallel pool for time at least σ_i, CANC cancels task τ_i (which was running as job $\tau_i^{\shortparallel}$) and restarts τ_i as job $\tau_i^{\circledcirc}$ in the serial pool.

- CANC manages the serial pool with the EQUI strategy.

First we must establish that CANC is a valid schedule, i.e. each task τ_i is completed by CANC. This is a concern because CANC computes a schedule for the relaxed jobs, and assumes that the actual tasks are completed by running during the time slots of their corresponding relaxed job.

Proposition 6.4. CANC *completes all tasks.*

Proof. Tasks placed in the serial pool are clearly completed. We proceed to argue that tasks never placed in the serial pool are finished in the parallel pool. Consider some job j_i' that finishes in the parallel pool, i.e. finishes in time less than σ_i; this corresponds to a task τ_i that is never placed in the serial pool because its corresponding relaxed job finishes in the parallel pool. We say that j_i' executes in "parallel mode" when executing on at least π_i/σ_i processors, and in "serial mode" otherwise.

For each $x \in [p]$, define f_x to be the amount of time that j_i' spends executing on (exactly) x processors. Then the progress completed by job j_i' is

$$\sum_{x<\pi_i/\sigma_i} f_x + \sum_{x>\pi_i/\sigma_i} f_x \cdot x \cdot \sigma_i/\pi_i.$$

Job j_i' completes once it has made $2\sigma_i$ progress. At least half of the progress on j_i' must have been made in parallel mode, because there is insufficient time to achieve σ_i progress in serial mode. But this implies that

$$\sum_{x>\pi_i/\sigma_i} f_x \cdot x \cdot \sigma_i/\pi_i \geq \sigma_i$$

and thus that

$$\sum_{x>\pi_i/\sigma_i} f_x \cdot x \geq \pi_i.$$

This implies that $\tau_i^{\shortparallel}$, which also spends f_x time on x processors for each $x \in [p]$, successfully completes. □

The reason that we refer to relaxed jobs as "relaxed" is because there is a sense in which they are strictly easier to schedule than $\mathcal{T}$. We formalize this in the following lemma.

Proposition 6.5.

$$\mathsf{TRT}_{\mathsf{OPT}}^{J'} \leq \mathsf{TRT}_{\mathsf{OPT}}^{2\mathcal{T}}.$$

Proof. A schedule for completing $2\mathcal{T}$ can be used to to perform J' by running j_i' in the time slot for $2\tau_i$. □

We now bound the cost of CANC, thereby proving Theorem 6.3.

Proof of Theorem 6.3. Let $J_{\circledcirc}^1$ denote the serial jobs that end up in the serial pool. Let $J_{\shortparallel}^2$ denote the jobs in $J_{\circledcirc}^1$ but modified to be perfectly scalable. And let $J_{\circledcirc}^2$ denote the jobs in $J_{\circledcirc}^1$ but with the arrival time of each job j_i' arrival time delayed by σ_i (i.e., delayed to be the time at which j_i' is placed in the serial pool by CANC). CANC's TRT is bounded by:

$$\mathsf{TRT}_{\mathsf{CANC}}^{\mathcal{T}} \leq \mathsf{TRT}_{\mathsf{EQUI}}^{J'} + \mathsf{TRT}_{\mathsf{EQUI}}^{J_{\circledcirc}^2}.$$

By Theorem 2.1, this is at most

$$\mathsf{TRT}_{\mathsf{OPT}}^{3J'} + \mathsf{TRT}_{\mathsf{OPT}}^{3J_{\circledcirc}^2}.$$

By Lemma 6.2,

$$\mathsf{TRT}_{\mathsf{OPT}}^{3J_{\circledcirc}^2} \leq O\left(\mathsf{TRT}_{\mathsf{OPT}}^{O(1)\cdot J_{\shortparallel}^1} + \sum_{j \in J_{\circledcirc}^2} \mathrm{work}(j)\right).$$

Since $\mathsf{TRT}_{\mathsf{OPT}}^{O(1)\cdot J_{\shortparallel}^1} \leq \mathsf{TRT}_{\mathsf{OPT}}^{O(1)\cdot J'}$, it follows that

$$\mathsf{TRT}_{\mathsf{CANC}}^{\mathcal{T}} \leq O\left(\mathsf{TRT}_{\mathsf{OPT}}^{O(1)\cdot J'} + \sum_{j \in J_{\circledcirc}^2} \mathrm{work}(j)\right).$$

In other words, defining $\mathcal{T}'$ to be the set of tasks in $\mathcal{T}$ that CANC runs in serial mode, we have

$$\mathsf{TRT}_{\mathsf{CANC}}^{\mathcal{T}} \leq O\left(\mathsf{TRT}_{\mathsf{OPT}}^{O(1)\cdot J'} + \sum_{\tau_i \in \mathcal{T}'} \sigma_i\right).$$

Notice, however, that by design, $\mathsf{TRT}_{\mathsf{CANC}}$ only runs a task τ_i in serial mode if, when we run EQUI on J', the job j_i' incurs at least σ_i response time. Thus

$$\sum_{\tau_i \in \mathcal{T}'} \sigma_i \leq \mathsf{TRT}_{\mathsf{EQUI}}^{J'},$$

which by Theorem 2.1 implies that

$$\sum_{\tau_i \in \mathcal{T}'} \sigma_i \leq O\left(\mathsf{TRT}_{\mathsf{OPT}}^{O(1)\cdot J'}\right).$$

Thus

$$\mathsf{TRT}_{\mathsf{CANC}}^{\mathcal{T}} \leq O\left(\mathsf{TRT}_{\mathsf{OPT}}^{O(1)\cdot J'}\right),$$

which by Proposition 6.5 completes the proof. □

6.3 A Non-Cancelling Scheduler

Now we show how to convert CANC from Theorem 6.3 to a non-cancelling scheduler. This is the most technically difficult section of the paper.

Theorem 6.6. *There is an online scheduler that, with $O(1)$ speed augmentation and* ***without use of cancelling****, is $O(1)$ competitive for* MRT.

Up to a factor of 2 in speed augmentation, we can assume without loss of generality that every σ_i and π_i is a power of two—to simplify our exposition, we shall make this wlog assumption throughout the rest of the section.

We say that a task τ_k is of ***type*** $(2^j, 2^i)$ if $\log \sigma_k = i$ and $\log \pi_k = i + j$. In other words, the job has parallelism 2^j and serial work 2^i. We also partition the jobs into ***parallelism classes***, where the 2^j ***parallelism class*** consists of tasks satisfying $\pi_k/\sigma_k = 2^j$.

Our first lemma shows that we can modify the scheduler CANC from Theorem 6.3 to obtain a "just as good" scheduler which only runs one task of each type in parallel at a time.

Lemma 6.7. *There exists an online scheduler B that,* ***with cancelling*** *and $O(1)$ speed augmentation is $O(1)$ competitive for* MRT. *Furthermore, B guarantees that at most one task of each type is run via its parallel implementation at any time. Moreover, for any task*

τ_i that B completes in parallel, B is guaranteed to complete that task within time σ_i of the task arriving.

Proof. Recall that CANC runs EQUI on tasks in the parallel pool until their serial time has elapsed. If a task τ_k is in the parallel pool for longer than σ_k, CANC cancels τ_k and restarts it in the serial pool via $\tau_k^{\circledcirc}$.

Our task is to construct B so that

$$\mathsf{TRT}_B^{\mathcal{T}} \leq \mathsf{TRT}_{\mathsf{CANC}}^{\mathcal{T}}. \tag{11}$$

B runs CANC on the parallel pool, except it concentrates all of CANC's work on each task type into a single task of that type. That is, if CANC allocates k processors to tasks of type $(2^j, 2^i)$ in the parallel pool on a certain time step, then B will allocate k processors to the current running task of type $(2^j, 2^i)$ (if B has a task of this type). The scheduler B also copies CANC's cancellation behavior as follows: when CANC cancels a task of type $(2^j, 2^i)$, B attempts to cancel a task of the same type that is not currently running, and then restart that task in the serial pool; if there is only one task of the type $(2^j, 2^i)$ running in B, then B cancels the running task and restarts it in the serial pool; and finally, if there are no tasks of this type in B, then B does nothing and places a *fake* task of type $(2^j, 2^i)$ in the serial pool. This behavior ensures that the serial pool for B receives tasks of exactly the same types (and at exactly the same times) as the serial pool for CANC.

The point of this construction is that, at any given moment the number of tasks of each type that B has either completed or evicted from the parallel pool is trivially guaranteed to be at least as large as that of CANC. The serial pools of CANC and B are identical (with the fake tasks included). Hence the number of tasks alive for B at any given moment is at most as large as in CANC. So B achieves MRT at least as good as CANC on $\mathcal{T}$. □

Now we present a *non-cancelling* scheduler C that, with $O(1)$ speed augmentation, is $O(1)$ competitive with B for TRT.

Up to a factor of 4 speed augmentation, we can assume that C has $4p$ processors. We will make this assumption (without loss of generality) throughout the rest of the proof and keep the speed augmentation implicit. Thus, for the rest of the proof, we assume both that for every task $\tau \in \mathcal{T}$, σ_i and π_i are powers of two, and that C is given $4p$ processors.

To clarify our exposition, when discussing B, we will make a distinction ***parallel work*** (i.e., work on parallel jobs) and ***serial work*** (i.e., work on serial jobs) performed by B. Note that the parallel work on a job in a time interval $[a, b]$ is defined as the integral over $[a, b]$ of the number of processors allocated to the job at each point in time.

As we run the scheduler C, we will also simulate B running $3\mathcal{T}$ with p processors. The scheduler C will attempt to use p of its processors to copy B's behavior. Of course, as B is a cancelling scheduler, C will not be able to precisely copy B. The challenge will be to somehow achieve an MRT competitive with B's MRT, but without cancelling.

Call a task in C ***vested*** if it has actually started executing in parallel in C. C can copy B's behavior except for when B cancels a vested task (to be restarted in serial). Whenever B cancels a vested task τ, the task τ enters ***ballistic mode***. Whenever a task τ enters ballistic mode, we say that its parallelism class enters ***emergency mode*** (although the class may already be in emergency mode due to other tasks in the class already being in ballistic mode). When a parallelism class is in emergency mode, all of the parallel work that B performs is allocated by C to the *smallest* ballistic task in the parallelism class (we will see later that there are no ties here, but as we have not proven that yet, assume ties are broken arbitrarily).

Note that non-ballistic tasks lose work in C compared to B when their parallelism class is in emergency mode (i.e., when a non-ballistic task τ's parallelism class is in emergency mode, it is possible that C does work on τ while B does not). If a non-ballistic task τ_i loses q total parallel work to some ballistic task τ_k, then we say that τ_k ***stole*** q work from τ_i. We emphasize that this stolen work is not queued up to be done later by τ_i, it is just lost. Thus it is possible for a task τ to finish running in parallel in B *without finishing in* C. If this happens, and τ is already vested in C, then the task τ also enters ballistic mode; otherwise, if τ is not yet vested in C, then the task τ enters what we call ***semi-ballistic mode***. Thus, a task τ enters ballistic mode if it is already vested and is then either cancelled *or completed* by B; and a task τ enters semi-ballistic mode if B completes it in parallel, but if, at that point in time, C has not even vested it.

The tasks in semi-ballistic mode are executed as *serial jobs* on p processors using EQUI. Finally, the remaining $2p$ processors are allocated by C as follows: C allocates $p/2^i$ processors to each parallelism class 2^i. Whenever the parallelism class is in emergency mode, those processors are allocated as extra processors to the smallest ballistic task in the class. This completes the description of C.

Now let us turn to the analysis of C. Let $\mathcal{T}_0$ denote the set of tasks that enter ballistic mode and $\mathcal{T}_1$ denote the set of tasks that enter semi-ballistic mode.

Lemma 6.8. *Each task $\tau_i \in \mathcal{T}_0$ spends at most $2\sigma_i$ time in ballistic mode.*

Proof. Whenever τ_i in parallelism class 2^j is actually *executing* in ballistic mode (i.e., it is the smallest ballistic task from its parallelism class), it is given at least $p/2^j$ processors. Thus it spends at most $\pi_i/(p/2^j) = \sigma_i$ time executing in ballistic mode. Additionally, τ_i may spend time in ballistic mode waiting on other (smaller) ballistic tasks to complete.

Once a parallelism class enters emergency mode, it stops vesting new tasks. Let t be the time at which τ_i entered ballistic mode, and let t' be the most recent time $t' \leq t$ at which the 2^j parallelism class entered emergency mode. Any tasks in parallelism class 2^j that are ballistic at the same time as τ_i must have been running in B at time t'. There can be at most one such task (including τ_i) of each parallel power-of-two serial size $\sigma' \leq \sigma_i$. Since each task in ballistic mode of some size σ' spends at most σ' time actually executing in ballistic mode, the total time that τ_i spends waiting on smaller ballistic tasks to finish is at most

$$\sum_{r=1}^{\infty} \sigma_i/2^r \leq \sigma_i.$$

This complete the proof. □

Lemma 6.9. *The total response time incurred by the tasks $\mathcal{T}_1$ while in semi-ballistic mode in C is at most*

$$O\left(\mathsf{TRT}_{\mathsf{OPT}}^{O(1)\cdot\mathcal{T}} + \sum_{\tau_i\in\mathcal{T}_0}\sigma_i\right).$$

Proof. For each job $\tau_i \in \mathcal{T}_1$, define x_i to be a serial job of size $3\sigma_i$ whose arrival time is the time at which τ_i goes semi-ballistic in C; and define y_i to be a perfectly scalable job of size $18\sigma_i$ whose arrival time is simply t_i. Let $X = \{x_i \mid \tau_i \in \mathcal{T}_1\}$ and let $Y = \{y_i \mid \tau_i \in \mathcal{T}_1\}$. The total response time incurred by the tasks $\mathcal{T}_1$ while in semi-ballistic mode in C is at most $\mathsf{TRT}_{\mathsf{EQUI}}^{X}$. By Theorem 2.1, this is at most $\mathsf{TRT}_{\mathsf{OPT}}^{3X}$. By Lemma 6.2, this is at most

$$O\left(\mathsf{TRT}_{\mathsf{OPT}}^{18Y} + \sum_{\tau_i\in\mathcal{T}_0}\sigma_i\right).$$

Since $\mathsf{TRT}_{\mathsf{OPT}}^{18Y} \le \mathsf{TRT}_{\mathsf{OPT}}^{O(1)\cdot\mathcal{T}}$, the lemma follows. □

The only way that a task can be present in C but not in B is if the task is either in ballistic or semi-ballistic mode. It follows by Lemmas 6.7, 6.8, and 6.9 that

$$\mathsf{TRT}_{C}^{\mathcal{T}} \le O\left(\mathsf{TRT}_{\mathsf{OPT}}^{O(1)\cdot\mathcal{T}} + \sum_{\tau_i\in\mathcal{T}_0\cup\mathcal{T}_1}\sigma_i\right). \tag{12}$$

Thus, to complete the analysis of C, it suffices to bound

$$\sum_{\tau_i\in\mathcal{T}_0\cup\mathcal{T}_1}\sigma_i.$$

This is achieved through a charging argument in the following lemma.

Lemma 6.10.

$$\sum_{\tau_i\in\mathcal{T}_0\cup\mathcal{T}_1}\sigma_i \le O(\mathsf{TRT}_{\mathsf{OPT}}^{3\mathcal{T}}).$$

Proof. If a task τ enters ballistic or semi-ballistic mode because B placed τ into serial mode (i.e., τ was either canceled by B or was never even run in parallel mode by B), then call τ ***easy***. By the analysis of CANC the sum of the serial lengths of the easy tasks is $O(\mathsf{TRT}_{\mathsf{OPT}}^{O(1)\cdot\mathcal{T}})$.

Call the other tasks that enter ballistic or semi-ballistic mode ***hard***. The hard tasks are the ones that went ballistic or semi-ballistic only because of other ballistic jobs stealing their parallel processing times—this causes the task to complete in the parallel pool for B without completing for C.

Each easy task τ_i is paid $2\sigma_i$ tokens upfront. Whenever any task τ_i in C has its parallel work (that B wishes to perform on it) stolen by a ballistic task τ_k, the ballistic task τ_k pays tokens to the task τ_i proportionally to the work that is stolen: if both tasks are in the 2^j parallelism class, and task τ_k stole q parallel processing time from task τ_i (here we are defining parallel processing time to be the integral over time of the number of processors that τ_k stole from τ_i), then τ_k pays τ_i a total of $q/2^j$ tokens. Finally, whenever a task τ_i enters either ballistic or semi-ballistic mode, the task pays σ_i tokens to the scheduling algorithm.

Since the sum of the serial lengths of the easy tasks is $O(\mathsf{TRT}_{\mathsf{OPT}}^{O(1)\cdot\mathcal{T}})$, the number of tokens paid to easy tasks upfront is $O(\mathsf{TRT}_{\mathsf{OPT}}^{O(1)\cdot\mathcal{T}})$. On the other hand, the number of tokens paid by ballistic and semi-ballistic tasks to the scheduler is

$$\sum_{\tau_i\in\mathcal{T}_0\cup\mathcal{T}_1}\sigma_i.$$

Thus, to complete the proof, it suffices to show that no task has a negative number of tokens when it completes.

For each task τ_i that goes ballistic or semi-ballistic in some parallelism class 2^j, the total number of tokens that it ever *spends* is at most σ_i (paid to the scheduler) plus $\pi_i/2^j = \sigma_i$ (paid to other tasks that τ_i stole work from while being ballistic). Thus it suffices to show that every task τ_i that goes ballistic or semi-ballistic *earns* at least $2\sigma_i$ tokens during its lifetime.

If the task τ_i is easy, then it trivially receives $2\sigma_i$ tokens upfront. Otherwise, if a non-easy task τ_i in some parallelism class 2^j goes ballistic or semi-ballistic, then it must have had at least $2\pi_i$ parallel work stolen from it by ballistic tasks in its parallelism class (because B completed $3\pi_i$ parallel work on the task, but C completed less than π_i). This means that the task was paid at least $2\pi_i/2^j = 2\sigma_i$ tokens, which completes the proof. □

Combining Lemma 6.10 with (12), we have completed the proof of Theorem 6.6.

7 GENERALIZATION: TAPS WITH DEPENDENCIES

We now consider a generalization of the serial-parallel decision problem in which tasks can have dependencies—a given task τ_i will not arrive until all of the other tasks on which it depends are complete. For this section, we focus exclusively on optimizing awake time—note that, if the tasks correspond to components of a parallel program, the awake time corresponds to the completion time of the parallel program.

A ***DTAP*** $\mathcal{D}$ (TAP with dependencies) is a set of tasks τ_i specified by σ_i, π_i, t_i along with an associated set $D_i \subset [n]$ (potentially empty) of tasks that must be completed before task τ_i can be started. That is, task τ_i becomes available only when the time t satisfies $t > t_i$ and furthermore all tasks $\tau_j \in D_i$ have already been completed. Of course, the dependency structure must form a DAG, or else it is impossible to run all tasks. We are interested in an online scheduler, which in this case means that the scheduler does not know anything about task τ_i until the task becomes available to run.

The following propositions establish a tight bound of $\Theta(\sqrt{p})$ on the optimal competitive ratio achievable by an online scheduler. The lower bound holds even for the case where the DTAP dependencies are required to form a tree.

Proposition 7.1. *For any (potentially randomized) online scheduler* ALG*, there exists a DTAP where* ALG*'s awake-time competitive ratio is $\Omega(\sqrt{p})$ with high probability in p.*

Proof. For convenience, we will discuss the DTAP as being randomly chosen from a set of DTAPs. Of course, if all schedulers ALG have competitive ratio $\Omega(\sqrt{p})$ with high probability for randomly chosen DTAPs from a class, then there exists some DTAP in the class for which a given ALG is very likely to perform poorly on.

We consider a class of DTAPs which consist of $\lfloor\sqrt{p}\rfloor$ ***levels***. Each level of these DTAPs consists of $\lfloor\sqrt{p}\rfloor$ tasks with serial work 1 and parallel work $\sqrt{p}$. Of the $\lfloor\sqrt{p}\rfloor$ tasks on each level exactly one randomly chosen task spawns $\lfloor\sqrt{p}\rfloor$ more tasks which form the next level. In particular, this single task is the sole dependency for all tasks on the next level. All the tasks in the DTAP have $t_i = 0$, so each task arrives immediately once all the tasks it depends on are completed.

OPT, knowing the dependencies, could first run all the spawning tasks via their parallel implementations to unlock all tasks after time $\lfloor\sqrt{p}\rfloor\sqrt{p}/p \le 1$. Next, OPT can schedule the remaining $\lfloor p\rfloor^2 - \lfloor\sqrt{p}\rfloor$ serial tasks via their serial implementations. Doing so OPT achieves awake time of at most 2.

However, a scheduler ALG that is unaware of the dependencies will likely require much longer on this DTAP. If ALG is not willing to run more than $1/2$ of the tasks in a level via parallel implementations then there is at least a $1/2$ chance that it requires time 1 to pass the level due to running the spawning task in serial. However, if the ALG is willing to run at least $1/2$ of the tasks in parallel then in expectation it requires at least $1/4$ time to uncover the spawning task. Either way, with constant probability ALG spends $\Omega(1)$ time on each level. Thus, with high probability in p the scheduler requires $\Omega(\sqrt{p})$ total time to complete the DTAP. □

Proposition 7.2. *There exists an online DTAP scheduler that is $O(\sqrt{p})$ competitive for awake-time.*

Proof. Fix a DTAP $\mathcal{D}$ with n tasks. It suffices to consider a DTAP where our scheduler always has at least one available uncompleted task, i.e., the case where its awake time and completion time are the same.

We say that a task τ_i is ***fairly-parallel*** if $\pi_i/p < \sigma_i/\sqrt{p}$, and ***not-very-parallel*** otherwise. The TURTLE scheduler runs fairly-parallel tasks in parallel and not-very-parallel tasks in serial. TURTLE schedules the available tasks as follows:

- Whenever there is an available fairly-parallel task allocate all processors to a fairly-parallel task.
- If all available tasks are not-very-parallel, and there are k such tasks, then allocate a processor to each of the $\min(p, k)$ present jobs with the largest remaining serial works.

Now we analyze the performance of the TURTLE scheduler. First we consider the time that TURTLE spends running fairly-parallel tasks. Let $\mathcal{D}_{\shortparallel} \subseteq \mathcal{D}$ be the fairly-parallel tasks. The time that TURTLE spends running tasks in $\mathcal{D}_{\shortparallel}$ is

$$\sum_{i|\tau_i\in\mathcal{D}_{\shortparallel}} \pi_i/p \le \sum_{i|\tau_i\in\mathcal{D}_{\shortparallel}} \sigma_i/\sqrt{p} \le \sqrt{p}\sum_{i\in[n]}\sigma_i/p \le \sqrt{p}\,\mathsf{T}_{\mathsf{OPT}}^{\mathcal{D}}.$$

Thus, in order to prove $\mathsf{T}_{\mathsf{TURTLE}}^{\mathcal{D}} \le O(\sqrt{p})\,\mathsf{T}_{\mathsf{OPT}}^{\mathcal{D}}$ it remains to bound the time that TURTLE spends executing not-very-parallel tasks. Let $\mathcal{D}'$ be a new DTAP where we set the size of all fairly-parallel tasks to 0. Observe that the time that TURTLE spends executing not-very-parallel tasks is the same in both $\mathcal{D}, \mathcal{D}'$ because in $\mathcal{D}$ we always preempt not-very-parallel tasks if fairly-parallel tasks are available and run fairly-parallel tasks. Thus, it suffices to analyze $\mathsf{T}_{\mathsf{TURTLE}}^{\mathcal{D}'}$.

Let S be the amount of time that TURTLE on $\mathcal{D}'$ is saturated (i.e., has at least p tasks) and U be the amount of time that TURTLE is unsaturated. Let $\mathsf{T}_{\infty}^{\mathcal{D}'}$ be the time that it would take to perform $\mathcal{D}'$ by running each task in serial on its own processor (i.e., imagining that we had infinitely many processors). Observe that

$$U \le \mathsf{T}_{\infty}^{\mathcal{D}'} \le \sqrt{p}\,\mathsf{T}_{\mathsf{OPT}}^{\mathcal{D}'}$$

where the second inequality is due to the fact that tasks in $\mathcal{D}'$ are not-very-parallel; in particular, if we ran each task in serial on its own $\sqrt{p}$-speed-augmented processor then the tasks would certainly complete no later than OPT. We also have

$$S \le \sum_{i\in[n]}\sigma_i/p \le \mathsf{T}_{\mathsf{OPT}}^{\mathcal{D}'}$$

because because total work is a lower bound on OPT's awake time. Combining our bounds on U, S we have we have

$$\mathsf{T}_{\mathsf{TURTLE}}^{\mathcal{D}'} \le O(\sqrt{p}) \cdot \mathsf{T}_{\mathsf{OPT}}^{\mathcal{D}'}.$$

□

8 AWAKE-TIME LOWER BOUNDS

In this section we present several lower bounds for awake time.

Proposition 8.1. *No deterministic online scheduler can have competitive ratio smaller than $\phi - 1/p$.*[2]

Proof. Fix a deterministic scheduler ALG. Consider a TAP with $\sigma_1 = \phi, \pi_1 = p$. If ALG schedules τ_1 in serial ALG fails to be better than ϕ competitive in the case that no more tasks arrive. If ALG waits at least $1/\phi$ time before scheduling τ_1 then ALG also fails to be better than ϕ competitive if no more tasks ever arrive. If instead ALG schedules τ_1 in parallel at some time $t_0 < 1/\phi$ and then $p-1$ un-parallelizable tasks arrive right after t_0 with $\sigma_i = \phi - t_0$, then OPT achieves awake time ϕ due to having chosen to run every task in serial while ALG's awake time is at least

$$\begin{aligned} t_0 + \frac{p + (p-1)(\phi - t_0)}{p} &\ge t_0 + \frac{p + p(\phi - t_0)}{p} - \phi/p \\ &= 1 + \phi - \phi/p \\ &= \phi^2 - \phi/p. \end{aligned}$$

So ALG's competitive ratio is at least $\phi - 1/p$. □

Now we consider the decide-on-arrival model. We can prove a stronger lower bound in the decide-on-arrival model than the Proposition 8.1, and we conjecture that this separation is real.

Proposition 8.2. *No deterministic decide-on-arrival scheduler has competitive ratio better than $2 - \Omega\left(\frac{\log p}{p}\right)$.*

Proof. Fix a deterministic scheduler ALG. Let $k = \lfloor\log p\rfloor$. Consider the TAP $\mathcal{T}_1^p$ where $\sigma_i = 2^i, \pi_i = 2^{i-1}p$ for $i \in [k]$. In $\mathcal{T}_1^k$ tasks run twice as fast as their serial run time if they are fully parallelized. Furthermore task τ_i is twice as large as task τ_{i-1}. The arrival times for tasks in this TAP are separated by infinitesimal amounts of time. The final $p-k$ tasks of $\mathcal{T}_1^p$ are un-parallelizable tasks with $\sigma_i = 2^k$ which all arrive instantly after τ_k.

[2] $\phi \approx 1.618$ denotes the golden ratio, i.e. the positive root of $x + 1 = x^2$.

First we consider the truncation of $\mathcal{T}_1^p$ to $\mathcal{T}_1^j$ for some $j \le k$. OPT will complete $\mathcal{T}_1^j$ by running tasks $\tau_1, \tau_2, \ldots, \tau_{j-1}$ in serial and task τ_j in parallel. Then, OPT's awake time on $\mathcal{T}_1^j$ is

$$\frac{1}{p}\left(2+4+\cdots+2^{j-1}+2^{j-1}p\right) \le (1+2/p)2^{j-1}. \qquad (13)$$

Now assume that ALG runs τ_j in serial. Then τ_j's awake time is at least 2^j, which by Equation (13) means that ALG has competitive ratio at least $2-\Omega(1/p)$ on $\mathcal{T}_1^j$. That is, for ALG to have any chance of achieving competitive ratio better than $2-\Omega(1/p)$, ALG must schedule the first k tasks in parallel.

Now we consider the performance of ALG on $\mathcal{T}_1^p$ assuming that ALG schedules the first k tasks in parallel. Here ALG will have awake time at least

$$\frac{1}{p}\left(1p+2p+\cdots+2^{k-1}p+(p-k)\cdot 2^k\right) \ge 2^k(2-k/p)-1. \qquad (14)$$

On the other hand, OPT will run all p tasks in serial and complete in time 2^k. Thus, ALG's awake time is at least $(2-\Omega(k/p))$-times larger than OPT's on $\mathcal{T}_1^p$. Plugging in our choice $k=\Theta(\log p)$ gives the desired bound on ALG's competitive ratio. □

Although the lower bound of Proposition 8.1 does not obviously translate to randomized schedulers we can still show a $1+\Omega(1)$ lower bound for such schedulers as follows.

Proposition 8.3. *Let* RAND *be a randomized scheduler. There exists a TAP on which* RAND *has competitive ratio at least* $\frac{3+\sqrt{3}}{4}-\Theta(1/p)$ *with probability arbitrarily close to* 1.

Proof. In $\mathcal{T}_A$ a single task with $\sigma_1=\sqrt{3}+1$, $\pi_1=2p$ arrives at time 0. TAP $\mathcal{T}_B$ starts the same as $\mathcal{T}_A$, but in $\mathcal{T}_B$ $p-1$ additional maximally un-parallelizable tasks arrive at time 1 with serial work $\sqrt{3}$. A simple calculation shows that no deterministic algorithm can achieve competitive ratio better than $\frac{1+\sqrt{3}}{2}-\Theta(1/p)$ on both $\mathcal{T}_A$ and $\mathcal{T}_B$. Furthermore RAND cannot have better than a $1/2$ chance of performing well on both $\mathcal{T}_A$ and $\mathcal{T}_B$. Thus, RAND's expected competitive ratio on at least one of these TAPs is at least $\frac{1+\frac{1+\sqrt{3}}{2}}{2}-\Theta(1/p)=\frac{3+\sqrt{3}}{4}-\Theta(1/p)$.

Let $s_1, s_2, \ldots, s_n$ be a random string of A's and B's. Now we form a TAP $\mathcal{T}$ by placing $\mathcal{T}_{s_i}$ at time $10i$. On $\mathcal{T}$ RAND handles each choice between $\mathcal{T}_A, \mathcal{T}_B$ correctly with probability at most $1/2$. Thus, for any $\varepsilon > 0$ if we make the sequence sufficiently long (i.e., take n large enough) then RAND has arbitrarily low probability of handling more than a $(1/2+\varepsilon)$-fraction of the $\mathcal{T}_A, \mathcal{T}_B$ choices correctly. Thus, RAND has competitive ratio at least at least $\frac{3+\sqrt{3}}{4}-\Theta(1/p)$ with probability arbitrarily close to 1. □

Now we consider parallel-work-oblivious schedulers. We show that, even if all tasks arrive at a single time, there is a lower bound of $2-o(1)$ on the optimal competitive ratio.

Proposition 8.4. *There is no deterministic parallel-work-oblivious scheduler that achieves competitive ratio better than* $2-\Omega(1/p)$ *for awake time, even in the single-arrival-time setting.*

Proof. Fix a deterministic parallel-work-oblivious scheduler ALG. Consider a TAP with two tasks τ_1, τ_2 both of serial size 1. Technically there are many different ways that ALG can handle τ_1, τ_2. We will reduce the space of possible strategies that ALG can employ by showing that certain strategies are dominant over other strategies. Combined with some case analysis this will allow us to show that there must be some TAP on which ALG performs poorly.

Without loss of generality ALG instantly starts (at least) one of the tasks. If ALG starts a serial job at time 0 then ALG has competitive ratio at least $\Omega(p)$ on the TAP where τ_1, τ_2 are perfectly scalable. Thus, it suffices to consider the case where ALG starts by running one of the tasks in parallel; call this instantly started task τ_1.

Without loss of generality ALG runs τ_1 on all processors at each time step until it starts τ_2. If τ_1 completes before ALG starts τ_2 then without loss of generality ALG instantly starts τ_2 in parallel.

Now we can describe ALG as follows: at each time $t \in [0,1]$ until τ_1 finishes ALG can decide whether to start τ_2 at time t and which implementation of τ_2 to use when starting it. Let $x_{\text{ALG}} \in [0,1]$ be the earliest time when ALG is willing to start τ_2 even if τ_1 is not yet completed by this time. If ALG chooses to schedule τ_2 in parallel at time x_{ALG} then there is a TAP on which ALG has competitive ratio at least 2: namely, the TAP where τ_1, τ_2 are both completely un-parallelizable. Thus, it suffices to consider the case where ALG would choose to run τ_2 in serial at time x_{ALG} assuming that τ_1 has not yet finished by time x_{ALG}.

By now we have substantially simplified the description of ALG. In particular, we have shown that ALG is completely parameterized by a single value $x_{\text{ALG}} \in [0,1]$. Given x_{ALG}, we have reduced to the case where ALG's strategy is

(1) Run $\tau_1^{\parallel}$ from the start.
(2) If τ_1 finishes before time x_{ALG} start $\tau_2^{\parallel}$ immediately once τ_1 finishes.
(3) Otherwise, start $\tau_2^{\circledcirc}$ at time x_{ALG}.

To conclude we consider two cases on the value of x. If $x_{\text{ALG}} < 1-1/p$ then ALG performs poorly on a TAP where $\pi_1/p = x+1/p$ and $\pi_2/p = 1/p$. Indeed, for this TAP OPT has awake time $x+2/p$ whereas ALG has awake-time at least $x+1$. Thus that ALG's competitive ratio here is at least

$$\frac{x+1}{1+2/p} \ge 2-\Omega(1/p)$$

If instead $x_{\text{ALG}} \ge 1-1/p$ then ALG performs poorly on the TAP where τ_1, τ_2 are both completely un-parallelizable. In this case OPT achieves awake time 1 by running the tasks in serial from the start whereas ALG has awake time at least $2-1/p$.

Thus, regardless of x_{ALG}, ALG has competitive ratio at least $2-\Omega(1/p)$ on some TAP. □

Finally, we show that if we simultaneously restrict to the decide-on-arrival model and the parallel-work-oblivious model then the scheduler cannot perform well.

Proposition 8.5. *Any deterministic scheduler* ALG *that is both decide-on-arrival and parallel-work-oblivious must have competitive ratio* $\Omega(\sqrt{p})$.

Proof. Consider the following two TAPs:

(a) $\lceil\sqrt{p}\rceil$ identical scalable tasks arrive at the start.
(b) $\lceil\sqrt{p}\rceil$ identical un-scalable tasks arrive at the start.

To ALG these two TAPs look identical. However, if ALG decides to run any task in serial then it's competitive ratio on (a) is $\Omega(\sqrt{p})$.

Otherwise, if ALG decides to run all tasks in parallel then its competitive ratio on (b) is $\Omega(\sqrt{p})$. Note that this simple decomposition into two cases is made possible by the assumption that ALG is a decide-on-arrival scheduler. □

9 MRT LOWER BOUNDS

In this section we prove two lower bounds on schedulers for MRT. These bounds show that any $O(1)$ competitive scheduler must (1) make decisions at least partially based on the values of $\{\pi_i\}$; and (2) must be willing to vary the number of processors assigned to a given job over time. Thus these properties cannot be relaxed in the schedulers presented in the previous sections.

We remark that our first lower bound applies not just to schedulers that achieve worst-case competitive ratios, but also to schedulers that use randomization in order to achieve a bounded *expected* competitive ratio.

Proposition 9.1. *Fix an online scheduler* ALG *that is oblivious to the parallel works of tasks, and fix some $c \in \Theta(1)$. There exists a TAP on which the expected competitive ratio of* ALG *with c speed augmentation is at least $\Omega(p^{\frac{1}{4}})$ for* MRT.

Proof. Consider a TAP with $\sqrt{p}+p^{1/4}$ tasks, all with serial work 1. Suppose that $\sqrt{p}$ of the tasks have parallel work 1, and that (a random subset of) $p^{1/4}$ of the tasks have parallel work p. Call these the ***cheap*** and ***expensive*** tasks, respectively. All of the tasks arrive at time 0.

If the cheap tasks are run in parallel, and then the expensive tasks are run in serial, then the TRT will be $O(\sqrt{p}\cdot\frac{1}{\sqrt{p}}+p^{1/4}\cdot 1) = p^{1/4}$.

Now suppose for contradiction that ALG also achieves $O(p^{1/4})$ TRT using $O(1)$ speed augmentation. Let δ be the expected fraction of the tasks that ALG runs in serial. The expected number of cheap jobs that are executed in serial is $\delta\sqrt{p}$. Thus, in order for ALG to be $O(1)$-competitive (even with $O(1)$ speed augmentation), we would need $\delta \le O(p^{-1/4})$. But this means that the expected number of expensive jobs that are executed in parallel is at least $(1-\delta)p^{1/4} \ge \Omega(p^{1/4})$. If k expensive jobs are executed in parallel, their TRT will be at least $\Omega(k^2)$. By Jensen's inequality, the expected TRT of expensive jobs executed in parallel is therefore at least $\Omega((p^{1/4})^2) = \Omega(\sqrt{p})$. This contradicts the assumption that ALG achieves TRT $O(p^{1/4})$. □

Proposition 9.2. *Consider an online scheduler* ALG *that is non-preemptive (i.e., the number of processors that it assigns to each task is fixed for the full duration of time that the task executes). Fix some $c \in \Theta(1)$, and any $R \in \mathbb{N}$. There exists a TAP $\mathcal{T}$ on which the worst-case competitive ratio of* ALG *with c speed augmentation is $\Omega(R)$ for* MRT. *That is,* ALG *with performs arbitrarily worse than* OPT.

Proof. As we are interested in the worst-case competitive ratio of ALG, we may assume without loss of generality that ALG is deterministic.

Let p_0 be the maximum number of processors that ALG ever simultaneously gives work over all input TAPs. We claim that there is some input TAP on which ALG does arbitrarily poorly compared to OPT in terms of mean response time.

Consider a sequence of tasks $\mathcal{T}$ that causes ALG to choose to have p_0 processors in use at some point in time t. Without loss of generality, we may assume that all p_0 processors that are in use at time t each have at least 1 remaining work. Let h be the TRT that OPT would incur on $\mathcal{T}$. Now, suppose that at time t, we have $\lceil R\cdot h\rceil$ additional tasks arrive, each with 0 work.

The TRT of OPT on this TAP is $0+h$, whereas the TRT of ALG on this TAP is at least $h+1\cdot R\cdot h$, since all $\lceil R\cdot h\rceil$ of the new tasks will have to wait for time at least 1 before ALG begins them. Hence ALG's competitive ratio on this TAP is at least R. □

10 OPEN QUESTIONS

We conclude by discussing open questions and conjectures.

Questions about awake time. We conjecture that Proposition 8.1 should be tight.

Conjecture 10.1. *There exists a deterministic ϕ-competitive scheduler for awake time.*

More concretely, let us propose a scheduler GoldenAlg that we suspect is ϕ competitive. For simplicity of exposition we assume that OPT's awake time is its completion time on the TAP in question, i.e., OPT does not have "gaps" of time when there are no available uncompleted jobs. It is straightforward to adapt GoldenAlg to handle TAPs with OPT gaps because GoldenAlg will simulate OPT and in particular will know when these gaps are. With this knowledge an appropriately modified version of GoldenAlg would be ϕ competitive on arbitrary TAPs assuming that the version of GoldenAlg described below is ϕ competitive on TAPs without gaps.

Let OPT_n be an optimal offline schedule for the first n tasks $\mathcal{T}_1^n$. Note that $\mathrm{OPT}_n, \mathrm{OPT}_{n+1}$ may make very different decisions, which is part of the challenge for an online scheduler. The GoldenAlg scheduler makes decisions by comparing to OPT_n.

GoldenAlg maintains a ***serial pool*** of tasks that it has decided to run in serial, and a ***parallel pool*** of tasks that it has tentatively decided to run in parallel. GoldenAlg manages the serial pool by running the jobs with the most remaining work first at each time step. Crucially, GoldenAlg only runs a single task from the parallel pool at a time, so all but one of the tasks in the parallel pool have not actually been started. Thus, if GoldenAlg desires, it can freely move one of these not-yet-started tasks to the serial pool. Tasks default to the parallel pool but GoldenAlg greedily moves tasks to the serial pool as soon as it is sure that this will not instantly cause the awake time to be too large. Formally, if the awake time incurred so far is t, if n tasks have arrived so far, and if τ_i is some not-yet-started task in the parallel pool, then GoldenAlg uses the following logic: If $\sigma_i + t < \phi\cdot \mathsf{T}^n_{\mathrm{OPT}_n}$, move τ_i to the serial pool. Finally, if there is no parallel job running but there are tasks present in the parallel pool GoldenAlg schedules the earliest arrived task (if any) from the parallel pool in parallel. At each time step this parallel job is allocated any processors not allocated to serial jobs.

GoldenAlg is clearly ϕ-competitive on any TAP that results in GoldenAlg being jagged. However, TAPs that result in GoldenAlg being balanced are more difficult. An inductive argument seems challenging because we can re-arrange our work. A more direct combinatorial argument seems more promising but has eluded us.

We also pose open questions regarding the optimal competitive ratios for different types of awake-time schedulers. We begin by considering decide-on-arrival schedulers:

QUESTION 1. *Does there exist a deterministic decide-on-arrival* 2*-competitive scheduler for awake time?*

Such a scheduler would imply that Proposition 8.2 is tight. More broadly, as noted in Section 8, we conjecture that there should be a separation between arbitrary deterministic schedulers and deterministic decide-on-arrival schedulers.

Next, we consider parallel-work-oblivious schedulers:

QUESTION 2. *What is the optimal awake-time competitive ratio achievable by a deterministic parallel-work-oblivious scheduler?*

We suspect that, for Question 2, the optimal competitive ratio is $4 - o(1)$.

Finally, we consider randomized schedulers:

CONJECTURE 10.2. *Is the optimal awake-time competitive ratio achievable by a randomized scheduler better than the optimal competitive ratio achievable by a deterministic scheduler?*

Note, in particular, that if our lower bounds are tight, then a separation should exist: the optimal competitive ratio for deterministic schedulers would be $\phi \approx 1.62$, and the optimal competitive ratio for randomized ones would be $(3 + \sqrt{3})/4 \approx 1.18$.

Questions about mean response time. In the context of optimizing MRT, there are even more basic questions that remain open.

We conjecture that speed augmentation is necessary to achieve a competitive ratio of $O(1)$:

CONJECTURE 10.3. $1 + \Omega(1)$ *speed augmentation is necessary in an* $O(1)$*-competitive scheduler.*

Although we have shown that (deterministic) parallel-work-oblivious schedulers perform poorly on MRT, it remains open whether decide-on-arrival schedulers can do well.

QUESTION 3. *Can a decide-on-arrival scheduler, with* $O(1)$ *speed augmentation, achieve* $O(1)$ *competitive ratio?*

Finally, if algorithms for this problem are to be made practical, then one important direction to focus on is simplicity. This motivates the following question:

QUESTION 4. *Is there a simpler scheduler that is still* $O(1)$ *competitive for* MRT *with* $O(1)$ *speed?*

An anonymous reviewer suggested one possible direction that would could try, which is to (1) virtually simulate running both jobs at once, and then (2) actually run whichever finishes first (with speed augmentation). The challenge that arises from this approach is that, by the time we finally decide which job to run, we may be significantly past the task's original arrival time. Thus, to make this approach work, one would likely need a much stronger version of Lemma 6.2, allowing one to analyze settings in which every job has its arrival time delayed based on its simulated completion time. It is conceivable that, in order to prove such a lemma, we could take technical inspiration from work on other scheduling problems (see, e.g., Theorem 3.1 of [2] or Theorem 5 of [13]).

Questions about offline scheduling algorithms. We conclude with two open problems about *offline* scheduling algorithms.

The first question concerns the scheduling of DTAPS (i.e., TAPS with job-arrival dependencies) to optimize awake time. Our results in Section 7 establish that the optimal online competitive ratio is $\Theta(\sqrt{p})$ for this problem. However, in some settings, even an *offline* scheduler could be useful, for example, if a compiler understands the parallel structure of the program that it is compiling and needs to decide for each component of the program whether to compile a serial version or a parallel version.

QUESTION 5. *Is there a polynomial-time offline algorithm that produces an* $O(1)$*-competitive DTAP schedule?*

Our final question concerns the offline scheduling of TAPS. We know from Section 8 that any online awake-time scheduler must incur a competitive ratio of $1 + \Omega(1)$. But what is the best competitive ratio achievable by a polynomial-time offline scheduler?

CONJECTURE 10.4. *Constructing an optimal (offline) schedule for awake time is NP-hard.*

REFERENCES

[1] Brenda S Baker and Jerald S Schwarz. 1983. Shelf algorithms for two-dimensional packing problems. *SIAM J. Comput.* 12, 3 (1983), 508–525.

[2] Nikhil Bansal, Ravishankar Krishnaswamy, and Viswanath Nagarajan. 2010. Better scalable algorithms for broadcast scheduling. In *Automata, Languages and Programming: 37th International Colloquium, ICALP 2010, Bordeaux, France, July 6-10, 2010, Proceedings, Part I 37*. Springer, 324–335.

[3] Robert D Blumofe, Christopher F Joerg, Bradley C Kuszmaul, Charles E Leiserson, Keith H Randall, and Yuli Zhou. 1995. Cilk: An efficient multithreaded runtime system. *ACM SigPlan Notices* 30, 8 (1995), 207–216.

[4] P. Dutot, G. Mounié, and D. Trystram. 2004. Scheduling Parallel Tasks Approximation Algorithms. https://www.semanticscholar.org/paper/Scheduling-Parallel-Tasks-Approximation-Algorithms-D

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

[6] Roozbeh Ebrahimi, Samuel McCauley, and Benjamin Moseley. 2018. Scheduling Parallel Jobs Online with Convex and Concave Parallelizability. *Theory Comput Syst* 62, 2 (Feb. 2018), 304–318. https://doi.org/10.1007/s00224-016-9722-0

[7] Jeff Edmonds. 2000. Scheduling in the dark. *Theoretical Computer Science* 235, 1 (March 2000), 109–141. https://doi.org/10.1016/S0304-3975(99)00186-3

[8] Jeff Edmonds and Kirk Pruhs. 2009. *Scalably Scheduling Processes with Arbitrary Speedup Curves*. Vol. 8. https://doi.org/10.1145/1496770.1496845 Journal Abbreviation: ACM Transactions on Algorithms Pages: 692 Publication Title: ACM Transactions on Algorithms.

[9] R. L. Graham. 1969. Bounds on Multiprocessing Timing Anomalies. *SIAM J. Appl. Math.* 17, 2 (1969), 416–429. https://doi.org/10.1137/0117039 arXiv:https://doi.org/10.1137/0117039

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

[11] Johann L Hurink and Jacob Jan Paulus. 2008. 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*. Springer, 67–74.

[12] Walter Ludwig and Prasoon Tiwari. 1994. Scheduling malleable and nonmalleable parallel tasks. In *Proceedings of the fifth annual ACM-SIAM symposium on Discrete algorithms*. 167–176.

[13] Benjamin Moseley, Anirban Dasgupta, Ravi Kumar, and Tamás Sarlós. 2011. On scheduling in map-reduce and flow-shops. In *Proceedings of the twenty-third annual ACM symposium on Parallelism in algorithms and architectures*. 289–298.

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

[15] John Turek, Walter Ludwig, Joel L. Wolf, Lisa Fleischer, Prasoon Tiwari, Jason Glasgow, Uwe Schwiegelshohn, and Philip S. Yu. 1994. Scheduling parallelizable tasks to minimize average response time. In *Proceedings of the sixth annual ACM symposium on Parallel algorithms and architectures (SPAA*

'94). Association for Computing Machinery, New York, NY, USA, 200–209. https://doi.org/10.1145/181014.181331

[16] John Turek, Uwe Schwiegelshohn, Joel L Wolf, and Philip S Yu. 1994. Scheduling parallel tasks to minimize average response time. In *Proceedings of the fifth annual ACM-SIAM symposium on Discrete algorithms*. 112–121.

[17] John Turek, Joel L Wolf, and Philip S Yu. 1992. Approximate algorithms scheduling parallelizable tasks. In *Proceedings of the fourth annual ACM symposium on Parallel algorithms and architectures*. 323–332.

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

[19] Deshi Ye, Xin Han, and Guochuan Zhang. 2009. A note on online strip packing. *Journal of Combinatorial Optimization* 17, 4 (2009), 417–423.