Serial-Parallel Scheduling Problem

Alek Westover

August 31, 2020

Abstract

There are many problems for which the best parallel algorithms have larger cost than the best serial algorithms, i.e. are not work-efficient. We consider a scheduler that is receiving many tasks with serial and parallel implementations that have potentially different costs. The scheduler can choose whether to run each task in serial or in parallel, and aims to either minimize total awake time, i.e. the amount of time that the scheduler has unfinished tasks, or average response time.

We show that, the off-line problem can essentially be reduced to the case where all tasks are available from the start, a setting in which the off-line and on-line algorithms are of course the same. In particular, we give a simple deterministic 2-competitive off-line scheduling algorithm, that does not even need to use preemption! The 2-competitive algorithm relies on solving a special case of the off-line problem where all tasks arrive at the same time. We give an exact solution to the off-line problem, but it has running time $(\Theta(p))^n$. We also give a 2-approximation algorithm for the single-arrival-time off-line problem with running time $O(n)$, assuming the tasks are pre-sorted. This yields a 4-competitive algorithm for the on-line problem. Further, we prove that there is no deterministic algorithm that gets competitive ratio better than 1.36 on all TAPs, and in fact, even with randomization we show that there is always some input on which the algorithm achieves competitive ratio at least 1.0625 with high probability.

Next we consider a generalization of the scheduling problem where tasks can have dependencies. Here we give a lower bound of $\Omega(\sqrt{p})$ against an on-line algorithm's performance. Further, we give an on-line algorithm that is $O(\sqrt{p})$-competitive. We also consider making an algorithm for the on-line problem. We give an algorithm with running time $(\Theta(p))^n$ to exactly compute OPT, and give a good approximation algorithm.

We also consider the problem of minimizing mean response time. This problem is fundamentally different from the problem of minimizing awake time: for example, preemption is crucial in an algorithm for minimizing response time, and the single-processor version of the minimum-awake-time problem is already non-trivial. We give good algorithms for this problem.

1 Introduction

1.1 Problem Specification

A parallel algorithm is said to be ***work-efficient*** if the work of the parallel algorithm is the same as the work of a serial algorithm for the same problem. Most implementations of parallel algorithms are not work-efficient, often having work that is a constant factor greater, or even asymptotically greater, than the work of the serial algorithm for the problem.

In the ***Serial-Parallel Scheduling Problem*** we have to perform n tasks $\tau_1, \ldots, \tau_n$ (n unknown ahead of time). We have p processors $\rho_1, \ldots, \rho_p$. Each task τ_i has a parallel implementation with work $\pi(\tau_i)$ and a serial implementation with work $\sigma(\tau_i)$. The tasks will become available at some times $t(\tau_1), \ldots, t(\tau_n)$. The set of tasks with their associated parallel and serial implementation's works and with their associated arrival times is called a ***task arrival plan*** or ***TAP*** for short.

The scheduler maintains a set of ***ready*** tasks, which are tasks that have become available but are not currently being run on any processor. At time $t(\tau_i)$ task τ_i is added to the set of ready tasks. At any time the scheduler can decide to schedule some (not already running) ready task, and can choose whether to run the task in serial, in which case the scheduler must choose a single processor to run the task on, or in parallel, in which case the scheduler can distribute the task's work arbitrarily among the processors. Intuitively, if there are many ready tasks then the scheduler should run the serial implementations of the tasks because the scheduler can achieve parallelism across the tasks. On the other hand, if there are not very many ready tasks it is probably better for the scheduler to run the parallel versions of the

tasks — even though they are possibly not work efficient, i.e. $\pi(\tau) > \sigma(\tau)$ — because by so doing at least the scheduler can achieve parallelism within tasks.

We also consider a generalization of the Serial-Parallel Scheduling Problem to the case where the task arrival times are not fixed, but rather some tasks may become available only after the completion of other tasks. Put another way, we consider a version of the problem where the tasks have dependencies. We refer to the set of tasks along with their associated parallel and serial implementation's works and their associated arrival times or dependency (i.e. what task they spawn after) as a ***DTAP*** (D stands for dependency).

Let the ***awake time*** of the scheduler be the duration of time over which the scheduler has unfinished tasks. One natural goal for the scheduler is to minimize awake time. We measure how well the scheduler is able to minimize its awake time by comparing its awake time to the awake time of the optimal off-line strategy, which we will denote OPT. Note that OPT is able to see the whole sequence of tasks in advance. The ***competitive ratio*** of a scheduler is the ratio of its awake time to the awake time of OPT on the same input.

1.2 Notation

We use $[n]$ to denote $\{1, 2, \ldots, n\}$.

1.3 Problem Motivation

Data-centers often get heterogeneous tasks. Being able to schedule them efficiently is a fundamental problem.

In the CILK programming language whenever a function is called we could let the scheduler choose between a serial and a parallel implementation of a task.

1.4 Related Work

Shortest Job First (SJF) is a pretty common idea for minimum response time. An algorithm called Shortest Remaining Processing Time (SRPT), and its variants are often useful for minimizing metrics like mean response time.

In [1], Berg et al study a related problem: many heterogeneous tasks come in, some which are elastic and exhibit perfect linear scaling of performance and some which are inelastic which must be run on a single processor, according to some stochastic assumptions, and they aim to minimize mean response time. They show that for some parameter settings the strategy "Inelastic First" is optimal.

In [3] Im et al exhibit an algorithm keeps the average flow time small.

In [2] Gupta et al prove some impossibility results about a problem somewhat similar to our problem.

Clearly related problems are widely studied. Our problem is novel however, and interesting.

1.5 Results

TODO: copy abstract here and reword it

2 Minimizing Awake Time on TAPs

In this section we show that the on-line scheduling problem can essentially be reduced to the off-line ***single-arrival-time*** case of the problem: the case where all tasks arrive at the start. In particular, we give an extremely simple deterministic scheduling algorithm, that in particular does not use preemption, for minimizing awake time, that makes use of an algorithm for the batch case of the off-line problem. If we use an algorithm that is R-competitive with OPT in the off-line batch case in our algorithm, then we get an on-line algorithm that is $2R$-competitive with OPT. We give an inefficient algorithm that accomplishes $R = 1$, and several more interesting and efficient algorithms with larger values of R.

2.1 On-line Reduction to Off-line

A ***single-time TAP*** is a TAP where all tasks arrive at the same time. In this subsection we show that algorithms that are R-competitive with OPT on single-time TAPs can be used to make an on-line algorithm that is $2R$-competitive with OPT.

Without loss of generality we consider TAPs where the cost ratio $\pi(\tau)/\sigma(\tau) \in [1, p]$ for all tasks τ; if $\pi(\tau)/\sigma(\tau) < 1$ then the scheduler clearly should never run τ in serial so we can replace the serial implementation with the parallel implementation to get cost ratio 1, similarly, if $\pi(\tau)/\sigma(\tau) > p$ then the scheduler should never run τ in parallel and we can replace the parallel implementation with the serial implementation to get cost ratio p.

A ***verge*** time for our algorithm is a time where no processors have work assigned to them, and there is at least one ready task. We propose Algorithm 1, which we call $\boldsymbol{BAT_R}$ (BAT is short for "batch"), as a scheduling algorithm for the on-line problem. In this subsection we analyze the competitive-ratio

of BAT_R assuming the existence of an algorithm for ORACLE_R; in Subsection 2.2 we show that ORACLE_R can be computed, and in fact can be efficiently computed for $R \geq 2$.

Algorithm 1 BAT_R

Let ORACLE_R be an algorithm that is R-competitive with OPT on single-time TAPs
if verge time **then**
    schedule tasks as directed by ORACLE_R

Theorem 1. *BAT_R is $2R$-competitive.*

Proof. Let an ***OPT-batch*** be a maximal set of tasks done by OPT. Let a ***BAT-batch*** be one of BAT_R's batches. Call a BAT-batch ***internal*** if it is a subset of an OPT-batch. Call a BAT-batch ***external*** if it is not a subset of an OPT-batch.

Consider an OPT-batch x. Let B_1 be the union of the internal BAT-batches contained in x, let B_2 be the BAT-batch that overlaps with the end of x (if such a batch exists), and let B_3 be the first BAT-batch that starts after x. Let T_1 be the time since the start of x to the start of B_2 and let T_2 be the time from the start of B_2 until the end of x. The amount of time that OPT spends on tasks from x is clearly $T_1 + T_2$. Now we bound how much time BAT spends on tasks from x. B_1 takes time T_1. B_3 spends time at most RT_2 on tasks from x. B_2 spends time at most $R(T_1 + T_2)$ on tasks from x. Adding these times up we get that BAT spends at most $2R(T_1+T_2)$ time on the tasks from x. Adding this up across all OPT-batches x, gives that BAT is $2R$-competitive, as desired. □

In fact, if we assume no properties of ORACLE_R beyond that it always gives a schedule that is R-competitive with OPT on single-time TAPs, the $2R$ competitive ratio is tight.

Proposition 1. *Let $R < p/2$. Fix $\varepsilon > 0$. There exists an $ORACLE_R$ such that BAT_R using this $ORACLE_R$ does not get competitive ratio $(2R - \varepsilon)$ on some TAP.*

Proof. Consider an ORACLE_R that, if it gets R tasks with serial work 1 and parallel work p, schedules the tasks one after another on ρ_1. Clearly this is R-competitive with OPT: OPT takes time 1, while ORACLE_R takes time R. Now consider a TAP where at time 0 and at time ε a set of R tasks with serial work 1 and parallel work p arrive. Clearly OPT achieves awake time $1 + \varepsilon$ on this TAP, while BAT_R, using the specified ORACLE_R, achieves awake time $2R$. Letting $\varepsilon' = \frac{\varepsilon}{1+\varepsilon}$ we see that the competitive ratio of BAT_R on this TAP is $2R-\varepsilon'$. Letting $\varepsilon'' = \varepsilon/2$, we have that BAT_R does not get competitive ratio $2R - \varepsilon''$ on this TAP. □

We remark that Proposition 1 does not imply that there is no ORACLE_R such that BAT_R is $(2R - \varepsilon)$-competitive for some $\varepsilon > 0$ on all TAPs. But it does imply that we would need extra assumptions on ORACLE_R to get such a result. It also means that our reduction transformation is tight.

2.2 Off-line Approximation Algorithms

In this Subsection we present algorithms for ORACLE_R for various values of R.

First we give Algorithm 2, which we call FROST, for ORACLE_1.

Algorithm 2 FROST (i.e. ORACLE_1)

Input: tasks $\tau_1, \ldots, \tau_n$ all with $t(\tau_i) = 0$
Output: a way to schedule the tasks to processors $\rho_1, \ldots, \rho_p$ that achieves minimal awake time

minAwakeTime $\leftarrow \infty$
bestSchedule $\leftarrow$ schedule everything in serial on ρ_1
for $I \in \{0,1\}^n$ **do**
    $x \leftarrow \sum_{i=1}^{n} I_i$
    for $J \in \{1, \ldots, p\}^x$ **do**
        $j \leftarrow 0$
        for $i \in \{1, 2, \ldots, n\}$ **do**
            if $I_i = 1$ **then**
                $j \leftarrow j + 1$
                schedule task τ_i in serial on ρ_{J_j}
        $m \leftarrow \max_{\rho_i}(\text{work}(\rho_i))$
        $w \leftarrow \sum_{\rho_i}(m - \text{work}(\rho_i))$
        $f \leftarrow \sum_{\rho_i}(1 - I_i)\pi(\rho_i)$
        if $f \geq w$ **then**
            make ρ_i have work $m + (f - w)/p$
        else
            distribute f units of work arbitrarily among ρ_i without increasing awake time
        if awakeTime$(I, J) \leq$ minAwakeTime **then**
            minAwakeTime $\leftarrow$ awakeTime(I, J)
            bestSchedule $\leftarrow$ schedule(I, J)

We remark that it is not obvious that an oracle for OPT can be computed in finite time, even in the single-arrival-time case: there are an uncountably infinite number of possible scheduling strategies that OPT can choose from. The key insight to decrease the search space to be finite is to notice that for any

method of distributing whichever tasks are chosen to run in serial, the parallel tasks may as well be redistributed afterwords, so long as doing so either results in not increasing the awake time, or results in all tasks having identical amounts of work. We can thus do a brute-force search over all the ways to assign some tasks to run in serial and to run on specific processors, and then put the parallel tasks on top "like frosting on a cake". We remark that the running time of FROST for n tasks is $\Theta(p^n)$, which is extremely large. Nevertheless the existence of the algorithm is interesting.

Now we prove that FROST is 1-competitive with OPT.

Lemma 1 (Frosting Lemma)**.** *FROST is an oracle for OPT on single-time TAPs.*

Proof. Consider the configuration that OPT chooses. FROST considers a configuration of tasks with the same assignment of serial tasks at some point, because FROST brute force searches through all such configurations. For this configuration it is clearly impossible to achieve lower awake time than by spreading the parallel tasks in the frosting method, hence OPT's awake time is at least that of FROST. □

We remark that it is straightforward to extend FROST to a full oracle for OPT. In particular, a full OPT oracle can be constructed as follows:

1. Perform a brute force search over every possible subset of the tasks to be the set of tasks scheduled in parallel, and do a brute force search over every way to assign the serial tasks to processors.

2. Now consider the gaps, i.e. times when all processors are idle under this current schedule. Add the parallel tasks, scheduling them so as not to decrease the size of any gaps unless it is impossible to do otherwise, in which case the parallel tasks should be added to make all processors have equal amounts of work extending into what used to be a gap.

We now consider how to efficiently approximate OPT on single-time TAPs. We propose Algorithm 3, which we call THRESH, as a simple way to 2-approximate OPT.

THRESH schedules the i_* tasks with largest serial work in parallel, distributing their work equally, and schedules the rest of the tasks in serial, sequentially giving out the tasks, for the optimal value of i_*. Clearly THRESH only requires space $\Theta(1)$ beyond the input to implement, and has running time $\Theta(n)$, given that the input is pre-sorted. We remark

Algorithm 3 THRESH

Input: tasks $\tau_1, \ldots, \tau_n$ with $\sigma(\tau_1) \geq \sigma(\tau_2) \geq \cdots \geq \sigma(\tau_n)$ all with $t(\tau_i) = 0$
Output: a way to schedule the tasks to processors $\rho_1, \ldots, \rho_p$ that achieves awake time at most twice the optimal awake time.

$w_\sigma \leftarrow \sum_i \sigma(\tau_i)$
$w_\pi \leftarrow 0$
$a \leftarrow 0$, $i_* = 0$
for $i \in [n]$ **do**
  $w_\sigma \leftarrow w_\sigma - \sigma(\tau_i)$
  $w_\pi \leftarrow w_\pi + \pi(\tau_i)$
  if $i < n$ **then**
    $a_0 = w_\sigma/p + \sigma(\tau_{i+1}) + w_\pi/p$
  else
    $a_0 = w_\pi/p$
  if $a_0 \leq a$ **then**
    $i_* \leftarrow i$
    $a \leftarrow a_0$
Schedule tasks $\tau_1, \ldots, \tau_{i_*}$ in parallel, distributing their work equally among the processors.
for $k \in \{i_* + 1, \ldots, n\}$ **do**
  Schedule τ_k in serial on processor $\rho_{1+(k \bmod p)}$

that a nice feature of THRESH is that it schedules parallel tasks on a constant number of processors; we allow OPT to schedule tasks on a variable number of processors (i.e. unevenly distribute the work of a parallel task). It is nice to not need to be able to unevenly distribute parallel tasks.

Proposition 2. *THRESH is* 2*-competitive with OPT on single-time TAPs.*

Proof. Let τ_{i_0} be the task with the largest serial work that OPT schedules in serial. Recalling that the tasks are sorted by serial work, for all $i < i_0$ we have that OPT chooses to schedule τ_i in parallel. Let

$$w_\sigma^{OPT} = \sum_{i \geq i_0} \sigma(\tau_i), w_\pi^{OPT} = \sum_{i < i_0} \pi(\tau_i).$$

OPT's awake time is obviously at least

$$(w_\pi^{OPT} + w_\sigma^{OPT})/p.$$

OPT's awake time is also obviously at least $\sigma(\tau_{i_0})$.

Without loss of generality we call the processor that gets the most work, which used to be called $\rho_{1+(i_* \bmod p)}$, ρ_1; in particular we circularly shift the labels of the processors so that the work that ρ_i gets is larger than the work that ρ_j gets for any $j > i$. Think about the sequential thing. Ignore $\sigma(\tau_{i_*})$. Then ρ_i

always has less work than all the other processors. Of course in reality ρ_1 has the most work of any processor. So we have that all processors have serial work at most

$$\sum_{i>i_*} \sigma(\tau_i)/p + \sigma(\tau_{i_*}).$$

Add on to that

$$\sum_{i\leq i_*} \pi(\tau_i)/p.$$

If $x \leq a+b$, and $y \geq a, y \geq b$, then obviously $x \leq 2y$. Thus, THRESH is 2-competitive with FROST. □

We now remark on the competitive-ratio of BAT_2 using THRESH. By Theorem 1 THRESH achieves competitive ratio at most 4 on all TAPs. However, Proposition 1 *does not* imply that THRESH must not be 3-competitive. In fact, we believe the opposite:

Conjecture 1. *BAT_2 using THRESH is 3-competitive with OPT.*

The competitive ratio of BAT_2 using THRESH is not (much) better than 3:

Proposition 3. *There exists a TAP such that BAT_2 using THRESH has competitive ratio larger than $3 - 3/p$.*

Proof. Consider the following TAP: at time 0 a completely unscalable task with serial work 1 arrives along with a fully scalable task with parallel work $p-2$, and then at time $1/(p-1)$ a completely unscalable task with serial work 1 arrives. BAT_2 using THRESH takes time $(p-2)/p + 1 + 1 = 3 - 2/p$ to complete these tasks, whereas OPT takes time $1 + 1/(p-1)$. The competitive ratio of BAT_2 here is thus

$$\frac{3-2/p}{1+1/(p-1)} = (3-2/p)(1-1/p) > 3 - 3/p.$$

□

However, we have not been able to find a TAP where BAT_2 using THRESH exhibits competitive ratio larger than 3, hence our conjecture.

The assumption that the input is pre-sorted might not be reasonable. If the tasks are not pre-sorted, then we must (at least approximately) sort them for our algorithm to achieve a good competitive-ratio. We now discuss several approaches for doing this sorting which will involve trade-offs in competitive ratio, running time, and space.

If we simply use heap sort to sort the tasks, then we will have increased running time of $\Theta(n \log n)$, but will retain our constant-space requirement and will still have competitive ratio 2.

If we use radix sort then we can possibly achieve better running time.

If the tasks can be assumed to have serial works that can only take on $O(1)$ possible values, then the radix sort can be performed in-place in linear time, which results in a linear-time 2-competitive algorithm.

If the serial works can take on more than $\omega(1)$ possible values, we can still achieve better than linearithmic running time by sacrificing competitive ratio. Let 1 be the largest serial work of any task. We sort the tasks based on which bucket $[1/2^i, 1/2^{i-1}]$ the task's serial work falls in, — i.e. rounding their serial works — except putting tasks with work less than $1/2^{\lg(np)}$ in a single bucket. Now there are only $\log(np)$ distinct keys, so sorting can be done faster.

If we are willing to spend $\Theta(n + \log(np))$ extra space, counting sort can sort the tasks (based on the rounded keys) in time $\Theta(n + \log(np))$. If we want to still only use $O(1)$ space, then by sequentially partitioning on each bit of the key's ranks we get an in-place algorithm with running time $O(n \log\log(np))$.

Since the tasks are only approximately sorted we cannot guarantee competitive ratio 2 with this method. However, we can guarantee that we are 8-competitive: with the exception of the tasks with work smaller than $1/(np)$ we have increased each task's size by at most a factor of 2; increasing the serial works of all tasks by a factor of 2 obviously doubles the awake time that we achieve. Now consider the tasks with works at most $1/(np)$. The sum of their works is at most $1/p$, and the largest task has work 1, and hence takes time at least $1/p$ to run.

Another interesting assumption to remove is the assumption that we know the works of the parallel implementations of tasks: it turns out that, if we are willing to sacrifice a constant-factor in the competitive ratio, we do not even need to know the parallel works of the tasks before running them! Now we give an algorithm that is $O(1)$-competitive with OPT, which *does not use the size of the parallel implementations of tasks*: it uses the size of the serial tasks to decide which tasks to run, and then is told how long parallel tasks took to run after running them. We present this algorithm in Algorithm 4, which we call ***RUN*** We remark that the same techniques discussed above for removing the assumption that tasks are pre-sorted in THRESH could be used in RUN.

Note that RUN accesses $\pi(\tau_i)$ only after running τ_i in parallel, after which it could of course have measured $\pi(\tau_i)$. RUN and THRESH give very similar strategies. RUN's strategy can be slightly worse than

Algorithm 4 RUN

Input: a list of tasks $\tau_1, \tau_2, \ldots, \tau_n$, sorted in descending order by $\sigma(\tau_i)$.
Output: a schedule that is $O(1)$-competitive with the schedule FROST would make

$w_\pi \leftarrow 0$
$i \leftarrow 1$
while $i \leq n$ **do**
  if $w_\pi > \sigma(\tau_i)$ **then**
    break
  Run task τ_i in parallel
  $w_\pi \leftarrow w_\pi + \pi(\tau_i)/p$
  $i \leftarrow i + 1$
while $i \leq n$ **do**
  Schedule τ_i in serial on $\rho_{1+(i \bmod p)}$
  $i \leftarrow i + 1$

THRESH's though, so RUN has a slightly worse competitive ratio than THRESH versus OPT.

Proposition 4. *RUN is 4-competitive with OPT.*

Proof. Let τ_{i_*} be the task with the largest serial work that RUN schedules in serial. Since

$$\sum_{i<i_*} \pi(\tau_i)/p > \sigma(\tau_{i_*})$$

it is clearly impossible for OPT to achieve awake time better than $\sigma(\tau_{i_*})$: if OPT schedules everything with serial work at least $\sigma(\tau_{i_*})$ in parallel OPT's awake time is at least $\sigma(\tau_{i_*})$, and if OPT schedules any tasks with serial work at least $\sigma(\tau_{i_*})$ in serial OPT's awake time is at least $\sigma(\tau_{i_*})$.

The running time of OPT is also of course at least

$$\sum_{i=1}^{n} \sigma(\tau_i)/p.$$

The running time of the parallel tasks for RUN is clearly at most $2\sigma(\tau_{i_*})$.

The running time for the parallel tasks of RUN is, by the same analysis presented in the analysis of THRESH, at most

$$\sum_{i>i_*} \sigma(\tau_i)/p + \sigma(\tau_{i_*}).$$

Combined we have the upper bound of

$$\sum_{i>i_*} \sigma(\tau_i)/p + 3\sigma(\tau_{i_*})$$

on the awake time of RUN. Using our lower bounds on the awake time of OPT we have that RUN is 4-competitive. □

By modifying RUN we can get a better competitive ratio. The new version of RUN will have a parameter $\eta \in (0, 1)$. The new version of RUN proceeds as follows:

- Run tasks in parallel (in order from tasks with largest serial work to smallest) until the amount of parallel work exceeds $p\eta\sigma(\tau_{i_*})$ where τ_{i_*} is the task currently running.
- Using bin-packing schedule in serial the tasks $\tau_{i_*+1}, \tau_{i_*+2}, \ldots, \tau_n$. Keep running τ_{i_*}, but redistribute its work so that it falls in the gaps in the bin-packing, if possible, and so that it falls evenly on top of the bins if placing it solely in the gaps in impossible.

Proposition 5. *RUN(η) is $\max(1 + 1/\eta, 2 + \eta)$-competitive.*

Proof. **TODO:** this proof is pretty sketchy, but has some interesting ideas. Let w_π^f be the final total parallel work performed by RUN(η), divided by p. Let τ_{i_*} be the task with the largest serial work that RUN(η) runs in parallel.

We consider 2 cases.

Case 1: $w_\pi^f \approx \eta\sigma(\tau_{i_*})$
OPT may choose to run all tasks τ_i with $i \leq i_*$ in parallel, like RUN(η) does. In this case we have

$$T^{OPT} \geq \eta\sigma(\tau_{i_*}) + \sum_{i>i_*} \sigma(\tau_i)/p.$$

On the other hand

$$T^{\mathrm{RUN}(\eta)} \leq \eta\sigma(\tau_{i_*}) + \sum_{i>i_*} \sigma(\tau_i)/p + \sigma(\tau_{i_*+1}) \leq (1+1/\eta)T^{OPT}.$$

OPT might also choose to run some task τ_i with $i \leq i_*$ in serial. This would make

$$T^{OPT} \geq \sigma(\tau_{i_*}).$$

On the other hand

$$T^{RUN} \leq \eta\sigma(\tau_{i_*}) + T^{OPT} \leq (1 + 1/\eta)T^{OPT},$$

TODO: because OPT can't beat our bin packing since its on these small tasks.

Case 2: $w_\pi^f \approx (\eta + 1)\sigma(\tau_{i_*})$ Here we have $T^{OPT} \geq \sigma(\tau_{i_*}), T^{OPT} \geq \sum_{i=1}^{n} \sigma(\tau_i)/p$. In this case the task can probably fit in the cracks of our bin-packing, if we assume somewhat good bin-packing this would mean

$$T^{\mathrm{RUN}(\eta)} \leq (1 + \eta)\sigma(\tau_{i_*}) + \sum_{i=1}^{n} \sigma(\tau_i)/p.$$

So this would be $(2 + \eta)$-competitive with OPT. □

Proposition 6. *RUN(η) is not $(1 + (1 - 2/p)/\eta)$-competitive, and RUN is not $(2 + \eta)(1 - 3/p)$-competitive.*

Proof. Consider the following TAP: at time 0 two perfectly scalable tasks arrive: one with serial work ηp, the other with serial work 1. Clearly OPT takes time $\eta + 1/p$ to do both tasks. On the other hand RUN(η) starts running the larger task in parallel, and by the end of it has done parallel work η. But this is at least $1 \cdot \eta$, so RUN(η) runs the other task in serial, hence achieveing awake time $1 + \eta$. The cost ratio here is

$$\frac{1+\eta}{1/p+\eta} \geq (1 - 1/(\eta p))(1 + 1/\eta) \geq 1 + \frac{1}{\eta}(1 - 2/p).$$

Now consider the following TAP: a task with serial work $1+1/p$ and parallel work $p(1+1/p)$ arrives along with $p+1$ perfectly scalable tasks with serial work 1. Clearly OPT has awake time $1+1/p+2/p$ by running the biggest task in serial along with $p-1$ tasks with serial work 1, and then after they all finish running the final 2 tasks in parallel across all the processors. On the other hand, RUN(η) runs the biggest task in parallel, and then runs the $p+1$ smaller tasks in serial so it takes time $\eta+1+1$. Hence RUN(η)'s competitive ratio on this TAP is

$$\frac{\eta+2}{1+1/p+2/p} \geq (\eta+2)(1-3/p).$$

□

Combined, Proposition 5 and Proposition 6 basically imply:

Corollary 1. *For $\eta = \frac{\sqrt{5}-1}{2}$, RUN($\eta$) is $\frac{3+\sqrt{5}}{2}$-competitive with OPT. For all η, RUN(η) is not $\left(\frac{3+\sqrt{5}}{2} - 1/p\right)$-competitive.*

Proof. It is clear that the competitive-ratio is minimized when

$$1 + 1/\eta = 2 + \eta.$$

Which happens at $\eta = \frac{\sqrt{5}-1}{2}$. □

While RUN(η) can do remarkably well for the case where the parallel works are not known, it is not clear that RUN(η) is optimal.

Proposition 7. *Any on-line scheduling algorithm for the variant of the game where the parallel works are not known is worse than $2(1 - 1/p)$-competitive on some TAP.*

Proof. Consider a TAP with $p + 1$ tasks all having identical serial work 1, with p tasks that have parallel work p and a single task that has parallel work 1. OPT clearly achieves awake time $1+1/p$, whereas no on-line algorithm can get awake time better than 2 in general. Hence the competitive ratio is

$$\frac{2}{1+1/p} \leq 2(1-1/p).$$

□

2.3 Arbitrary speedup curves

So far we have only considered the ***perfect*** speedup curve, i.e. $x \mapsto x$, where splitting a parallel task amongst x processors results in each processor getting work $\pi(\tau)/x$. Note that although we have been assuming that the parallel implementations scale perfectly, we have not been assuming that the parallel implementations are work-efficient; note that if all parallel implementations were work-efficient and scaled perfectly then obviously every task should be scheduled in parallel across all the processors. In this Subsection we consider a generalization of the problem to non-perfect speedup curves. However, we do still impose some reasonable restrictions on the speedup curves. The speedup curves we consider will be sublinear, monotonically increasing functions $f : [p] \to [1, p]$, with $f(1) = 1$. In the generalized problem, we denote by f_τ the speedup curve associated with τ, and still denote with $\sigma(\tau)$ and $\pi(\tau)$ the works of the serial and parallel implementations of τ. Now however, if the task is run equally split in parallel across k processors, the time it takes is $\pi(\tau)/f_\tau(k)$ per processor.

It is clear that Theorem 1 still applies: if we have a scheduling algorithm ORACLE$_R$ that is R-competitive with OPT in the off-line problem, then BAT$_R$ is $2R$-competitive with OPT in the on-line problem. The task thus becomes to find algorithms for ORACLE$_R$.

There is still a straightforward algorithm for ORACLE$_1$ in this case:

1. Perform a brute force search over every possible subset of the tasks to be the set of tasks scheduled in parallel, and do a brute force search over every way to assign the serial tasks to processors.

2. While the amount of work assigned to all the tasks is not even, we continue to assign parallel work as follows. Let k be the current number of processors with the minimum amount of work, let α, β be the minimum, and second lowest amounts of works assigned to any processor.

Find whichever task scales the best on k processors, and schedule $k(\beta - \alpha)$ parallel work from this task to these k processors, or all the work that the task has if the task has less than $k(\beta-\alpha)$ work. If the amount of work assigned to all the tasks becomes even, then simply schedule all the remaining work from the remaining parallel tasks evenly in parallel across all the processors.

Step 2 of this process is slightly more complex than previously, but the running time remains the same (asymptotically) as $(\Theta(p))^n$.

Now we consider how to make efficient algorithms for the non-perfect speedup curves case. We prove a striking lower bound, that indicates that fundamentally different strategies are needed in this case.

Proposition 8. *Any (off-line) scheduler that does not ever schedule a task on an intermediaite number of processors, i.e. a scheduler that always either assigns a task to a single processor or runs the task in parallel across all the processors, cannot achieve a competitive ratio better than $\sqrt{p}$ versus OPT.*

Proof. Consider a TAP where $\sqrt{p}$ identical tasks arrive at the beginning, with the following characteristics: $\sigma(\tau) = 1$, $\pi(\tau) = 1$, and

$$f_\tau(k) = \begin{cases} k & k \in [1, \sqrt{p}] \\ \sqrt{p} & k \in [\sqrt{p}, p]. \end{cases}$$

OPT clearly schedules all tasks in parallel on $\sqrt{p}$ processors, and hence achieves awake time $1/\sqrt{p}$.

A scheduler that schedules all tasks in serial achieves awake time 1. A scheduler that schedules all tasks in parallel across all the processors achieves awake time $\sqrt{p}/\sqrt{p} = 1$. Either way, a scheduling algorithm that does not consider scheduling tasks on intermediate numbers of processors has competitive ratio $\sqrt{p}$ on this input.

□

2.4 Lower Bound against Deterministic Algorithms

In this Subsection we prove several impossibility results, which show that we cannot hope to substantially improve our algorithms.

We can specify a TAP in a table with a list of which tasks arrive at each time; we use the compact notation $(\sigma, \pi) \times m$ to denote m identical tasks with serial works σ and parallel works π. It is clear that BAT_1 is not $(2 - \epsilon)$-competitive for any $\epsilon > 0$; consider, for example the TAP given in Table 1, on which BAT achieves awake time 2 and OPT achieves awake time $1 + \varepsilon$.

Table 1:

time	tasks
0	$(1, p) \times p/2$
ε	$(1, p) \times p/2$

We might hope to achieve a $(1 + \varepsilon)$-competitive scheduling algorithm for the on-line scheduling problem, which would be better than BAT_1 in terms of competitive ratio. However, we show that it is impossible for an off-line deterministic scheduler to get a competitive ratio lower than 1.25, even using preemption. That is, we show that for any deterministic algorithm ALG there is some input on which ALG has awake time at least 1.25 times greater than OPT.

In Table 2 and Table 3 we specify two TAPs.

Table 2:

time	tasks
0	$(4, 2p) \times 1$
1	$(3, 3p) \times (p - 1)$

Table 3:

time	tasks
0	$(4, 2p) \times 1$

Consider an arbitrary deterministic scheduling algorithm. If at time 0 the arriving tasks are $(4, 2p) \times 1$ (i.e. a single task arrives, with serial work 4 and parallel work $2p$) then the scheduler has two options: it can schedule this task in serial, or in parallel.

If no further tasks arrive, i.e. the task schedule is from Table 3 then OPT would have awake time 2 by distributing the tasks work equally amongst all processors, whereas a scheduler that ran the task in serial for all of the time that it was running the task during the first second after the task arrived would have awake time at least 3. In this case the competitive ratio of the algorithm is at least 1.5.

On the other hand, the algorithm could decide to not run the task in serial for any time during the first second after the task arrives. In this case, if and it turns out that the task schedule is from Table 2, then the algorithm has again acted sub-optimally. In particular, for the schedule given in Table 2, OPT schedules the task that arrives at time 0 in serial, and then schedules all the tasks that arrive at time 1 in serial as well, and hence achieves awake time 4. On the other hand, the awake time of an algorithm that did not schedule the task that arrived at time 0 in serial is at least 5: such a scheduler may either choose at time 1 to cancel the task from time 0 and

run it in serial, or the scheduler may choose to let the parallel implementation finish running. In this case the competitive ratio of the algorithm is 5/4.

Hence it is impossible for any deterministic algorithm to achieve a competitive ratio of lower than 1.25.

We remark that the numbers in this argument can clearly be optimized; doing so gives an improved lower bound of about 1.36 on competitive ratio.

2.5 Lower Bound against Randomized Algorithms

Often randomized algorithms can achieve improvements over deterministic algorithms, so we for example hope for a $(1+\epsilon)$-competitive randomized on-line scheduling algorithm that succeeds on any input with high probability. However, in this subsection we show that this is impossible.

In particular, we demonstrate a lower-bound of 1.0625 on the competitive ratio of any randomized off-line algorithm.

Recall the TAPs from Table 2 and Table 3; we will use these as sub-parts of our the TAP that we build to be adversarial for a randomized algorithm.

Fix some off-line randomized algorithm RAND. We say that an input TAP is ***bad*** for RAND if with high probability RAND's awake time on TAP is at least 1.0625 times that of OPT. We construct a class of TAPs, and show that some of the TAPs in this class must be bad for RAND.

Let $\mathcal{T}_I$, for some some binary string I, be the TAP consisting of the TAP from Table 2 at time $10i$ if $I_i = 1$ and the TAP from Table 3 if $I_i = 0$.

Consider a I chosen uniformly at random from $\{0,1\}^m$ for some parameter m. On each sub-tap RAND has at most a 1/2 chance of acting as OPT does, and at least a 1/2 chance of acting sub-optimally, in particular, from our analysis above showing that any deterministic algorithm has competitive ratio at least 1.25 on at least one of these inputs, RAND has at least a 1/2 chance of this happening. By a Chernoff Bound, with probability at least $1-e^{-\Omega(m)}$, on at least 1/4 of the sub-taps RAND has competitive ratio at least 1.25. Since there is no overlap, by design, of the sub-taps (by spacing them out), this means that overall the competitive ratio of RAND is at least $1 \cdot 3/4 + 1.25 \cdot 1/4 = 1.0625$.

Note that the number of tasks in such a TAP is less than mp, so $n \leq mp$, and thus $m \geq n/p$. Hence our result that holds with high probability in m holds with high probability in n/p too. Of course $n \gg p$ so this is pretty decent.

Because a randomly chosen TAP from this class of TAPs is bad for RAND with high probability in n/p, by the probabilistic method there is at least one TAP in this class of TAPs that is bad for RAND.

3 Minimizing Awake Time on DTAPs

3.1 Off-line $\Omega(\sqrt{p})$ Lower Bound

In the unknown dependencies it is impossible to get an $O(1)$-competitive algorithm with the optimal algorithm that knows the dependencies. In particular, we give a DTAP with $n = p$ tasks, that can be processed in time $O(1)$ by an algorithm that knows the dependencies, but that takes time $\Omega(\sqrt{n})$ to be processed in the worst case by any deterministic algorithm that does not know the dependencies. In fact, we can show any deterministic or randomized algorithm takes time $\Omega(\sqrt{n})$ to process this DTAP with high probability in n.

We now construct and analyze the DTAP. We will make $n = p$ tasks. The DTAP consists of $\sqrt{p}$ ***levels***. On each level of the DTAP the tasks are $(1, \sqrt{p}) \times \sqrt{p}$. One of these tasks spawns the $\sqrt{p}$ tasks at the next level of recursion upon completion.

A scheduler that knew the dependencies would first run all the spawning tasks using their parallel implementations. Doing so the emptier could unlock all the tasks in time $\sqrt{p}\sqrt{p}/p = 1$. Then there are less than p tasks, so the scheduler can schedule them all with their serial implementations, and finishes 1 unit of time later. In total this gives awake time 2. But if the scheduler does not know which are the spawning tasks, then it can't immediately do them. In the worst case such a scheduler will take 1 unit of time to uncover the dependencies: if the scheduler runs any task in serial it could take 1 unit of time, and if all tasks are run in parallel it could take $\sqrt{p}\sqrt{p}/p = 1$ unit of time. Taking 1 unit of time on each of the $\sqrt{p}$ levels means that this scheduler has awake time at least $\sqrt{p} = \sqrt{n}$.

By doing tasks in parallel until the spawning task is uncovered a randomized emptier can hope to do slightly better. But it turns out not better asymptotically. With high probability in n the scheduler takes time at least 1/2 on at least 1/4 of the levels. Hence the scheduler takes time at least $1/8\sqrt{n}$ with high probability in n, as desired.

3.2 Off-line $O(\sqrt{p})$-competitive algorithm

We now give an on-line scheduling algorithm in Algorithm 5, which we call ***AUG***, that is $O(\sqrt{p})$-competitive with the optimal off-line scheduling algorithm for the problem where there are unknown dependencies among the tasks. Recalling the $\Omega(\sqrt{p})$ lower bound on such an algorithm's competitive ratio, we have that our scheduler is asymptotically optimal in this situation.

Algorithm 5 AUG

$A = \{\rho_1, \rho_2, \ldots, \rho_{p/2}\}$
$B = \{\rho_{1+p/2}, \rho_{2+p/2}, \ldots, \rho_p\}$

$\mathcal{T} \leftarrow$ arrived tasks that haven't been scheduled yet
$\mathcal{T}_A \leftarrow \{\tau \in \mathcal{T} | \pi(\tau) \leq \sqrt{p} \cdot \sigma(\tau)\}$.
$\mathcal{T}_B \leftarrow \{\tau \in \mathcal{T} | \pi(\tau) > \sqrt{p} \cdot \sigma(\tau)\}$.

if B has no tasks running **then**
Sequentially (on tasks sorted in descending order by serial work) schedule all tasks $\tau \in \mathcal{T}_B$ on processors in B
Schedule all tasks $\tau \in \mathcal{T}_A$ in parallel on all processors in A

We will show that AUG with p processors is $O(1)$-competitive with OPT on $\sqrt{p}$ processors, which we refer to as ***SmallOPT***. We define the ***AUG-X awake time***, for $X = A, B$, to be the amount of time that X has unfinished tasks; we will show that AUG-X awake time is constant-competitive with SmallOPT's awake time for $X = A, B$.

First consider $X = A$. Recall that A gets tasks with cost ratio at most $\sqrt{p}$. SmallOPT can run these tasks in serial or in parallel. AUG runs these tasks in parallel on all processors in A. For a task τ, the time it takes AUG to run the task is $\pi(\tau)/(p/2) < \sigma(\tau)/(\sqrt{p}/2)$. The AUG-$A$ awake time is

$$\sum_{\tau \in \mathcal{T}_A} \frac{\pi(\tau)}{p/2} \leq \sum_{\tau \in \mathcal{T}_A} \frac{\sigma(\tau)}{\sqrt{p}/2}$$

where $\mathcal{T}_A$ denotes the set of tasks with cost ratio at most $\sqrt{p}$. On the other hand, the awake time of SmallOPT is trivially at least

$$\sum_{\tau} \frac{\sigma(\tau)}{\sqrt{p}}.$$

So AUG-A awake time is constant-competitive with SmallOPT's awake time.

Now consider $X = B$. Recall that B gets tasks with cost ratio larger than $\sqrt{p}$. SmallOPT must run these tasks in serial, and CHUNK chooses to run these tasks in serial as well, in particular on processors in B. Without loss of generality we restrict to considering DTAPs that consist only of tasks with cost ratio larger than least $\sqrt{p}$; considering DTAPs with other tasks increases SmallOPT's awake time without affecting CHUNK-B awake time. We claim that by scheduling tasks sequentially in batches AUG-B awake time is 4-competitive with SmallOPT's awake time. But it is clear that our algorithm is the same as BAT using THRESH: we are scheduling tasks in batches, and within the batch we are scheduling the same tasks as OPT does in serial, and scheduling them sequentially. BAT using THRESH was shown to be 4-competitive, so AUG-B awake time is 4-competitive with SmallOPT's awake time.

But of course AUG's awake time is at most the sum of AUG-A awake time and AUG-B awake time. So AUG's awake time is constant-competitive with SmallOPT's awake time.

SmallOPT is $(\sqrt{p})$-competitive with OPT since SmallOPT could just use $\sqrt{p}$ steps to simulate a step of OPT. Thus, because AUG is constant-competitive with SmallOPT, AUG is $O(\sqrt{p})$-competitive with OPT.

3.3 On-line Scheduler

Now we consider the off-line version of the dependency version of the scheduling problem. In particular, in this version of the problem the scheduler has full knowledge of all the specs of all the tasks and all the dependencies.

This is difficult even if we don't care about running time.

We give an algorithm to constant-approximate the optimal off-line scheduler in reasonable running time.

4 Minimizing Mean Response Time

In this section we consider a metric very different from awake time: mean response time.

4.1 Preemption is necessary for Minimizing Mean Response Time

In this paper we consider the metric of awake time. Another possible metric is mean response time. In this subsection we briefly demonstrate a major difference between the problem of minimizing mean re-

sponse time and minimizing awake time: Preemption is necessary for Minimizing Mean Response Time.

Consider a deterministic scheduling algorithm ALG that does not use preemption. Say that the max number of processors given work over all input TAPs is p_0. 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 that forces ALG to have p_0 processors in use, and let h_0 be the minimum amount of work on any processor with work. Say we have sent n_0 tasks so far. We choose n such that $n_0 = \epsilon n$, and now we send $(1-\epsilon)n$ tasks each with work $h_0/2$. OPT is presumably going to be preempting stuff to run these, so our competitive ratio is basically $\Theta(n)$, which is in particular trash.

4.2 Single-Processor

For a single-processor minimizing awake time is trivial: any strategy that always schedules a task when there are tasks and the processor is idle achieves minimal awake time. The sizes of the tasks do not matter at all. On the other hand, the single-processor case is already non-trivial for mean response time.

We propose the following algorithm, which we call SEER for the problem: if processor idle and there are tasks: schedule smallest task. Else: if new task arrives, predict the awake time for preemption or not, and do whichever looks better.

Claim 1. *SEER is OPT, or at least constant-competitive with OPT.*

Proof. This is just kind of obvious. But hard to prove. □

4.3 Multiprocessor-Processor

Now its harder. But we do basically the same thing. Just it's harder, especially hard to do it efficiently.

4.4 Dependencies

Basically we prove nasty lower bounds here.

References

[1] Benjamin Berg, Mor Harchol-Balter, Benjamin Moseley, Weina Wang, and Justin Whitehouse. Optimal resource allocation for elastic and inelastic jobs. *SPAA*, pages 75–87, 07 2020.

[2] Anupam Gupta, Sungjin Im, Ravishankar Krishnaswamy, Benjamin Moseley, and Kirk Pruhs. Scheduling heterogeneous processors isn't as easy as you think. *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1242–1253, 01 2012.

[3] Sungjin Im, Benjamin Moseley, Kirk Pruhs, and Eric Torng. Competitively scheduling tasks with intermediate parallelizability. *ACM Transactions on Parallel Computing*, 3:1–19, 07 2016.