Serial-Parallel Scheduling Problem

Alek Westover

August 20, 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 $\Omega(p^n)$. 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 inefficient algorithm 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 $\tau_1, \dots, \tau_n$ (n unknown ahead of time). We have p processors $\rho_1, \dots, \rho_p$. 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 $t(\tau_1), \dots, t(\tau_n)$. 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 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.

1.3 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.

1.4 Results

We show that, very surprisingly, 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 a 2-approximation algorithm for the single-arrival-time off-line problem with running time $O(n)$.

We also prove several impossibility results. We show that no deterministic scheduler can have a competitive ratio smaller than 1.25 in general. Even with randomization, we show that for any randomized algorithm there is some input on which the algorithm achieves competitive ratio at least 1.0625 with high probability.

We also consider a generalization of the scheduling problem where tasks can have dependencies. Here we show XXX.

2 Minimizing Awake Time on TAPs

In this section we give a simple deterministic scheduling algorithm — that does not use preemption — for minimizing awake time. We show that our algorithm is 2-competitive with OPT on all TAPs. Our algorithm is theoretically interesting, but not efficient. We also give an efficient algorithm for 2-approximating our algorithm.

Note that without loss of generality we may 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.

We say that a time is a ***verge*** time for our algorithm if at this time no processors have work assigned to them, and there is at least one ready task.

We propose Algorithm 1, which we call ***BAT*** (BAT is short for "batch"), as a scheduling algorithm.

A ***single-time-TAP*** is a TAP where all tasks arrive at a single time. FROST is an algorithm that

Algorithm 1 BAT

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

yields a schedule achieving minimal awake time, i.e. the same awake time as OPT, on single-time-TAPs. In Lemma 1 we establish that Algorithm 2 implements FROST, hence showing that BAT can actually be computed. Algorithm 2 exactly computes FROST, but is very slow. We also give an algorithm THRESH in Algorithm 3 for 2-approximating FROST in linear time. We remark that it is not obvious that an oracle for OPT can be computed in finite time, even in this special case: there are an uncountably infinite number of possible scheduling strategies that OPT can choose from. Nevertheless, we show how to consider a finite search space that a search can actually be performed on, at least in the special case of single-time-TAPs. Before considering FROST, and approximations to FROST, we analyze the competitive ratio of BAT assuming that we have ORACLE_R, an algorithm that is R-competitive with FROST for some $R \leq O(1)$; ORACLE_1 corresponds to FROST while ORACLE_2 corresponds to THRESH.

Consider a TAP $\mathcal{T}$. Let ℓ be the number of verge times for $\mathcal{T}$; note that $\ell \leq n$ which in particular is finite. Let t_i be the i-th verge time, and define t_0 (not a verge time) to be $-\infty$ for convenience. Let $T^{ALG}(a, b)$ denote the awake time of a scheduling algorithm ALG on the truncation of the TAP $\mathcal{T}$ that only consists of tasks arriving at times in $(a, b]$.

Note that all tasks that arrive at times in $(t_0, t_1]$ arrive at time t_1. By definition of ORACLE_R we have that ORACLE_R is R-competitive with OPT on single-time-TAPs, i.e.

$$T^{BAT}(t_0, t_1) \leq R \cdot T^{OPT}(t_0, t_1). \tag{1}$$

An ***ALG-gap*** is an interval of time I of non-zero length where for all times in the interior of I ALG has completed every task that has arrived thus far. Additionally, for an interval to be an ALG-gap the interval must contain no other intervals which are also ALG-gaps (i.e. it is a "maximal" interval satisfying our conditions). We say that a TAP is ***ALG-gap-free*** if it contains no ALG-gaps.

Now we prove an obvious property of OPT.

Claim 1. *If there is a scheduling algorithm ALG that completes all tasks by time t_* then OPT finishes all tasks by time t_*.*

Proof. Say that ALG completes all tasks by time t_*. Let $t_0 < t_*$ be the most recent time that OPT has completed all tasks that arrive before time t_0. If OPT has not finished all tasks by time t_* then it was acting sub-optimally, as it could steal the strategy that ALG used on $[t_0, t_*]$ to achieve lower awake time. In particular, for any tasks that arrive in $[t_0, t_*]$ OPT could schedule them as ALG schedules them. □

As an immediate consequence of Claim 1 we have that any ALG-gap is a subset of an OPT-gap.

Decomposing TAPs into gap-free subsets of the TAP is very useful. Part of the reason for this is the following fact:

Claim 2. *If an algorithm ALG achieves competitive ratio r on ALG-gap-free TAPs, then ALG achieves competitive ratio r on arbitrary TAPs.*

Proof. We define an equivalence relation $\tau_i \sim \tau_j$ to mean that no ALG-gap separates τ_i, τ_j. $\sim$ partitions the tasks into sets $I_1, \ldots, I_m$. Let $s(I_k) = \min_{\tau \in I_k}(t(\tau)), f(I_k) = \max_{\tau \in I_k}(t(\tau))$. We clearly have

$$T^{ALG}(t_1, t_n) = \sum_{k=1}^{m} T^{ALG}(s(I_k), f(I_k)).$$

By Claim 1, ALG-gaps are subsets of OPT-gaps. Hence, we also have

$$T^{OPT}(t_1, t_n) = \sum_{k=1}^{m} T^{OPT}(s(I_k), f(I_k)).$$

On the truncation of the TAP to a gap-free set of tasks ALG is r-competitive with OPT by assumption, so we have

$$T_{I_k}^{ALG}(s(I_k), f(I_k)) \leq r \cdot T_{I_k}^{OPT}(s(I_k), f(I_k)).$$

Summing this, we get

$$T^{ALG}(t_1, t_n) \leq r \cdot T^{OPT}(t_1, t_n),$$

as desired. □

By Claim 2, in order to bound BAT's competitive ratio, it suffices to consider TAPs without BAT-gaps. Note however that a TAP without BAT-gaps could still have OPT-gaps.

We conclude our analysis of the competitive ratio of BAT in Theorem 1 with an inductive argument on the number of OPT-gaps in the TAP. First we establish the base case for the argument in Proposition 1: we consider BAT's competitive ratio on an OPT-gap-free TAP.

Proposition 1. *BAT is* $(R+1)$*-competitive on OPT-gap-free TAPs.*

Proof. For an OPT-gap-free TAP (which is of course also a BAT-gap-free TAP) we must have

$$T^{OPT}(t_1, t_n) \geq t_{n-1} - t_1 = T^{BAT}(t_1, t_{n-1}), \quad (2)$$

because some tasks arrive at time t_1, and some tasks arrive after t_{n-1}, so for both BAT and OPT to have no gaps their awake times must satisfy the specified constraints.

Because BAT finishes all q_i tasks that arrive at time t_i by time t_{i+1} we can actually always decompose $T^{BAT}(t_1, t_n)$ as

$$T^{BAT}(t_1, t_n) = \sum_{i=1}^{n} T^{BAT}(t_{i-1}, t_i). \quad (3)$$

By Equation (3), and Equation (1) we thus have

$$T^{BAT}(t_1, t_n) \leq T^{BAT}(t_1, t_{n-1}) + R \cdot T^{OPT}(t_{n-1}, t_n). \quad (4)$$

Hence by Equation (2) and Equation (4) we have

$$\begin{aligned} T^{BAT}(t_1, t_n) &\leq T^{OPT}(t_1, t_n) + R \cdot T^{OPT}(t_{n-1}, t_n) \\ &\leq (R+1) T^{OPT}(q_1, \ldots, q_\ell), \end{aligned}$$

as desired. □

Theorem 1. *BAT is* $(R+1)$*-competitive.*

Bill's proof of this:

Let an OPT-batch be a maximal set of tasks done by OPT. Let a BAT-batch be one of BAT'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.

Proof. By Claim 2 without loss of generality we consider TAPs without ALG-gaps.

The proof is by strong induction on the number of OPT-gaps. The base case of our induction is established in Proposition 1, which says that if there are 0 OPT-gaps then BAT is 2-competitive.

Consider a TAP that has more than $k > 0$ OPT gaps, assume that for any TAP with fewer than k OPT gaps BAT is 2-competitive with OPT.

Let the first OPT-gap of the TAP start at time t_*. Let j be an index such that $t_j < t_* < t_{j+1}$.

Using our inductive hypothesis we have:

$$\begin{aligned} &T^{OPT}(t_1, t_n) \\ &= T^{OPT}(t_1, t_*) + T^{OPT}(t_*, t_n) \\ &\geq \frac{1}{R+1} \left(T^{BAT}(t_1, t_*) + T^{BAT}(t_*, t_n) \right) \\ &= \frac{1}{R+1} T^{BAT}(q_1, \ldots, q_\ell). \end{aligned}$$

Rearranging we get the desired bound:

$$T^{BAT}(q_1, \ldots, q_\ell) \leq (R+1) T^{OPT}(q_1, \ldots, q_\ell).$$

□

Now we analyze Algorithm 2, which we call FROST, or ORACLE_1.

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 larger than $\Omega(p^n)$, which is extremely large. Nevertheless the existence of the algorithm is interesting. Later we give an algorithm for ORACLE_2 that has much more reasonable running time.

We now prove that FROST actually does compute a schedule with awake time the same as that of OPT.

Lemma 1 (Frosting Lemma)**.** *FROST is an oracle for OPT on TAPs where all tasks arrive at a single time.*

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 of these. For this configuration it is clearly impossible to achieve lower awake time than by spreading the

Algorithm 2 FROST (i.e. ORACLE$_1$)

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

```
minAwakeTime ← ∞
bestSchedule ← schedule everything in serial on ρ_1
for I ∈ {0,1}^n do
    x ← Σ_{i=1}^n I_i
    for J ∈ {1,...,p}^x do
        j ← 0
        for i ∈ {1,2,...,n} do
            if I_i = 1 then
                j ← j + 1
                schedule task τ_i in serial on ρ_{J_j}
        m ← max_{ρ_i}(work(ρ_i))
        w ← Σ_{ρ_i} (m − work(ρ_i))
        f ← Σ_{ρ_i}(1 − I_i)π(ρ_i)
        if f ≥ 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) ≤ minAwakeTime then
            minAwakeTime ← awakeTime(I, J)
            bestSchedule ← schedule(I, J)
```

Algorithm 3 THRESH

Input: tasks $\tau_1, \dots, \tau_n$ 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 $\rho_1, \dots, \rho_p$ that achieves awake time at most twice the optimal awake time.

```
w_σ ← Σ_i σ(τ_i)
w_π ← 0
a ← 0, i_* = 0
for i ∈ [n] do
    w_σ ← w_σ − σ(τ_i)
    w_π ← w_π + π(τ_i)
    if i < n then
        a_0 = w_σ/p + σ(τ_{i+1}) + w_π/p
    else
        a_0 = w_π/p
    if a_0 ≤ a then
        i_* ← i
        a ← a_0
Schedule tasks τ_1, ..., τ_{i_*} in parallel, distributing
their work equally among the processors.
for k ∈ {i_* + 1, ..., n} do
    Schedule τ_k in serial on processor ρ_{1+(k mod p)}
```

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. First, 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. 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 more efficiently compute the best schedule for the case where all tasks arrive at the start. In fact, we consider making an algorithm to get a constant approximation of this, which will still give an algorithm with constant competitive ratio.

We propose Algorithm 3, which we call THRESH, as a simple way to 2-approximate FROST.

Put simply 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 requires space $\Theta(1)$ beyond the input to implement, and has running time $\Theta(n)$, given that the input is pre-sorted.

In order to remove the assumption that the input is pre-sorted, we can sort the tasks at the start of the algorithm, using heap sort or radix sort. If we simply use heap sort then we clearly will have running time $\Theta(n \log n)$ but still have $O(1)$ space requirement, and have competitive ratio 2 with FROST. On the other hand, if we use radix sort, it is conceivable that we could do better. First, if the tasks can be assumed to have serial works that can only take on $2^{O(1)}$ possible values, then the radix sort can be performed in linear time, in-place, resulting in a 2-competitive algorithm that uses $O(1)$ memory and $O(n)$ time. On the other hand, if such an assumption cannot be made, then if better running time is still desired, then it can be achieved at a cost to the competitive-ratio. Let 1 be the largest serial work of any task. If we sort the tasks based on the key of which bucket $[1/2^i, 1/2^{i-1}]$ the task's serial work falls in, except saying that any tasks with work less than $1/2^{\lg(np)}$ fall in the same bucket, then there are only $\log n + \log p$ distinct keys, so sorting can be done much faster. If we are willing to spend $\Theta(n+\log(np))$ space, then the sorting can be done by counting sort in time $\Theta(n+\log(np))$. On the other hand, using radix sort we could do the sort in time $\Theta(n\log(np))$ using only $\Theta(1)$ auxiliary space. Since the tasks are only approximately sorted we can only guarantee 4-competitiveness in this case: this is clear by noting that increasing the serial works of all tasks by a factor of 2 would double the best attainable awake time.

Now we show why THRESH is 2-competitive. Consider OPT's strategy. 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.

Now we give an algorithm that is 3-competitive with FROST, which *does not use the size of the parallel implementations of tasks.* We present this algorithm in Algorithm 4, which we call ***JUSTRUNIT***.

We remark on the very interesting property of JUSTRUNIT: it does not need to know $\pi(\tau_i)$ until after running τ_i in parallel, in which case it could of course have measured $\pi(\tau_i)$. JUSTRUNIT results in a strategy very similar to that given by THRESH. It differs slightly though, which results in the slightly worse competitive-ratio of 3 versus FROST.

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

Algorithm 4 JUSTRUNIT

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 3-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 $\rho_{1+(i \bmod p)}$
  $i \leftarrow i+1$

2.1 Lower Bound on Deterministic Algorithms for Minimizing Awake Time

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 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 $(1+\epsilon)$-competitive scheduling algorithm for this problem, which would arguably be much better than BAT. However, in this subsection we establish 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, to give an improved lower bound of about 1.36 on competitive ratio. As this is asymptotically not interesting, and much messier, we decide to not give this better argument.

2.2 Lower Bound against Randomized Algorithms

We might that there is a randomized algorithm that gets a competitive ratio substantially better than any deterministic algorithm can, for example maybe there is a randomized algorithm that is $(1+\epsilon)$-competitive on any input with high probability, or a randomized algorithm with expected competitive ratio at most $(1+\epsilon)$ on any input. 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 $n \le mp$, and thus $m \ge n/p$. 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.

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 greedily in batches AUG-B awake time is 4-competitive with SmallOPT's awake time. We define a ***B start time*** to be a time when B has no work and there are tasks that have already arrived with cost ratio larger than $\sqrt{p}$ that B is about to schedule. We define a ***B finish time*** to be a time when B finishes a batch of tasks. Note that B finish times and B start times certainly can coincide. Let T_t^{start} denote the difference between the t-th finish time and the t-th start time, i.e. the amount of time that greedy bin-packing takes to process the t-th batch of tasks. Let T_t^{during} denote the amount of time that greedy bin-packing takes to process the tasks that arrived between the t-th start time and the t-th finish time. Clearly AUG-B awake time is at most

$$\sum_t \left(T_t^{start} + T_t^{during}\right).$$

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 greedily 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

On the other hand, greedy bin-packing is 2-competitive with OPT for serial batches, so SmallOPT's awake time is at least

$$\sum_t \max(T_t^{start}/2, T_t^{during}/2).$$

Hence 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.

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 response 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. *This is OPT, or at least constant-competitive with OPT.*

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.