Serial-Parallel Scheduling Problem

Alek Westover

July 28, 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 mean response time, or total awake time.

First we consider the problem of minimizing awake time. We show that, very surprisingly, the general 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 the single-arrival-time case of the off-line problem; the only exact algorithm that we found for this has exponential running time. However, we can get a constant approximation algorithm for the problem with quasi-polynomial running time in n.

Next we consider the problem of minimizing mean response time. We show XXX.

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.

Also, we look at the problem when the tasks are allowed to do recursion, i.e. they can spawn multiple tasks. In this case we prove XXX.

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 sequence of tasks with their associated parallel and serial implementations 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.

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.

Another interesting metric to consider is ***mean response time***, the average over the tasks of the time between when the task arrives and when the task is finished. Again we can consider the competitive ratio of a scheduler relative to this metric.

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 [**?**], 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 [**?**] Im et al exhibit an algorithm keeps the average flow time small.

In [**?**] 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

First we consider the problem of minimizing awake time. We show that, very surprisingly, the general 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 the single-arrival-time case of the off-line problem; the only exact algorithm that we found for this has exponential running time. However, we can get a constant approximation algorithm for the problem with quasi-polynomial running time in n.

Next we consider the problem of minimizing mean response time. We show XXX.

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.

Also, we look at the problem when the tasks are allowed to do recursion, i.e. they can spawn multiple tasks. In this case we prove XXX.

2 An Algorithm for Optimizing Awake Time

In this section we give a simple deterministic scheduling algorithm — that does not use preemption — for minimizing awake time; note that we only consider the metric of awake time in this section. We show that our algorithm is 2-competitive with OPT (in terms of awake time) on all TAPs.

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 ***GR*** (GR is short for "greedy"), as a scheduling algorithm.

Algorithm 1 GR

```
while True do
    if verge time then
        schedule tasks as directed by ORACLE-1
```

A ***single-time-TAP*** is a TAP where all tasks arrive at a single time. ORACLE-1 is an algorithm that 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 ORACLE-1, which we specify in Algorithm 2, is an oracle for OPT on single-time-TAPs, hence showing that GR can actually be computed. 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 analyzing ORACLE-1 we analyze the competitive ratio of GR assuming the correctness of ORACLE-1.

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 time that is a verge time, let q_i be the number of ready tasks for GR at time t_i. Let $T^{ALG}(q_1, \ldots, q_{\ell'})$ 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 before $t_{\ell'}$.

By construction of ORACLE-1 we have that for single-time-TAPs OPT and GR achieve the same awake time, i.e.

$$T^{OPT}(q) = T^{GR}(q). \tag{1}$$

We remark GR "locally" schedules optimally, which is why we refer to GR as "greedy".

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 partition the tasks based on arrival time, splitting the tasks on the ALG-gaps. That is, we split the tasks into groups so that two tasks τ_i, τ_j are in the same group if and only if there are no gaps in between the arrival times of τ_i and τ_j. We can define an interval of time I_i for each of these ALG-gap-free subsets of the TAP, where I_i is defined so that all tasks in the i-th group start and finish at times contained in the interval I_i.

Let $T^{OPT}_{I_i}$ and $T^{ALG}_{I_i}$ denote the awake time of OPT and ALG on interval I_i. Because I_i is ALG-gap-free we have $T^{ALG} = \sum_i T^{ALG}_{I_i}$. Further, recall that by Claim 1 any ALG-gap is also an OPT-gap, so $T^{OPT} = \sum_i T^{OPT}_{I_i}$. Hence from our assumption that ALG is r-competitive on gap-free TAPs, such as the subset of the TAP on the interval I_i, we have $T^{ALG}_{I_i} \leq r \cdot T^{OPT}_{I_i}$ for all i. Summing we get $T^{ALG} \leq r \cdot T^{OPT}$, as desired. □

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

We conclude our analysis of the competitive ratio of GR in Proposition 1 with an inductive argument on the number of OPT-gaps in the TAP. First we establish the base case for the argument in Claim 3: we consider GR's competitive ratio on an OPT-gap-free TAP.

Claim 3. *GR is 2-competitive on OPT-gap-free TAPs.*

Proof. For an OPT-gap-free TAP we must have

$$T^{OPT}(q_1, \ldots, q_\ell) \geq T^{GR}(q_1, \ldots, q_{\ell-1}). \tag{2}$$

Because GR finishes all q_i tasks that arrive at time t_i by time t_{i+1} we can actually always decompose $T^{GR}(q_1, \ldots, q_\ell)$ as

$$T^{GR}(q_1, \ldots, q_\ell) = \sum_{i=1}^{\ell} T^{GR}(q_i). \tag{3}$$

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

$$T^{GR}(q_1, \ldots, q_\ell) = T^{GR}(q_1, \ldots, q_{\ell-1}) + T^{OPT}(q_\ell). \tag{4}$$

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

$$\begin{aligned} T^{GR}(q_1, \ldots, q_\ell) &\leq T^{OPT}(q_1, \ldots, q_\ell) + T^{OPT}(q_\ell) \\ &\leq 2T^{OPT}(q_1, \ldots, q_\ell), \end{aligned}$$

as desired. □

Proposition 1. *GR is 2-competitive.*

Proof. The proof is by strong induction on the number of OPT-gaps. The base case of our induction is established in Claim 3, which says that if there are 0 OPT-gaps then GR is 2-competitive.

Consider a TAP that has more than 0 OPT gaps; say that its first OPT-gap starts at time t_*. Let j be the largest index such that verge time $t_j < t_*$.

Using our inductive hypothesis we have:

$$\begin{aligned} &T^{OPT}(q_1, \ldots, q_\ell) \\ &\geq T^{OPT}(q_1, \ldots, q_j) + T^{OPT}(q_{j+1}, \ldots, q_\ell) \\ &\geq \frac{1}{2}\left(T^{GR}(q_1, \ldots, q_j) + T^{GR}(q_{j+1}, \ldots, q_\ell)\right) \\ &= \frac{1}{2}T^{GR}(q_1, \ldots, q_\ell). \end{aligned}$$

□

Now we analyze Algorithm 2, which we call ORACLE-1.

Algorithm 2 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 ← ∞
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)
```

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 ORACLE-1 for n tasks is much larger than $(2p)^n$ which is extremely large. Nevertheless the existence of our algorithm is interesting; we reduce the running time of our algorithm later.

We now prove that ORACLE-1 actually does compute a schedule with awake time the same as that of OPT.

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

Proof. Consider the configuration that OPT chooses. ORACLE-1 considers a configuration of tasks with the same assignment of serial tasks at some point, because ORACLE-1 brute force searches through all of these. 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 ORACLE-1. □

This existence proof is nice and simple. 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.

Here is a way to 8-approximate ORACLE-1 in time $2^{\log^2 n}$; note that quasipolynomial time is much better than super-exponential time.

Define 1 unit of work to be the work of the largest serial implementation. Round the serial works to be powers of 1/2. Round the cost ratios (work efficiencies) to be powers of 2.

Now we make a key observation: there are now only $\lg n \cdot \lg p$ different types of tasks! Because we can ignore tasks with less than $1/n = 1/2^{\lg n}$ work in their serial implementation because in total they can't contribute more than 1 unit of work and of course the cost ratios are in $[1, p]$.

But this isn't so many types of tasks! Say there are t_i of task i, then the time it takes to brute force try all ways of doing some tasks in serial and some in parallel is

$$\prod t_i.$$

By AM-GM, and using $n \gg p$,

$$\prod_{i=1}^{\lg p \lg n} t_i \leq \left(\frac{1}{\lg p \lg n} \sum t_i\right)^{\lg p \lg n} = \left(\frac{n}{\lg p \lg n}\right)^{\lg p \lg n} \leq 2^{\log^2 n \lg p}.$$

We can speed this up even further. Consider a $\lg n \times \lg p$ matrix with $A[i,j]$ being the number of tasks that have serial work $1/2^i$ and cost ratio 2^j. Consider a row of the matrix A. Clearly we should schedule at least $p\left\lfloor \sum_j A[i,j]/p \right\rfloor$ tasks with serial work $1/2^i$ in serial; by doing so we incur no inefficiency! So now our algorithm for computing ORACLE-1 goes as follows: compute the matrix A. Schedule tasks and update A accordingly; this will result in every row-sum of A being at most $p-1$. This has already improved our running time, by simply trying every quantity that could be scheduled in serial for each serial work size (note that within a row of the matrix it is always better to make the tasks scheduled in serial be the tasks with higher cost ratios). This strategy would have running time $O(p^{\lg n})$ or equivalently $O(n^{\lg p})$.

But we can make it even faster! We claim that we can 3-approximate OPT in this special case where all tasks are modified to have power of $1/2$ serial works and power of 2 cost ratios, and where we ignore all tasks with work below $1/n$, if we know what OPT's awake time is. Say that OPT's awake time is d. Then our strategy is to schedule any task with serial work greater than d in parallel, and to schedule any task with serial work less than d in serial. OPT must also have scheduled all tasks with serial work greater than d in parallel, so doing those tasks requires at most time d. We can upper bound the amount of time it takes to do the serial tasks by $d+d/2+d/4+\cdots = 2d$, by remembering that there are at most $p-1$ of each type of serial task. Hence overall we have bounded our awake time by $3d$. Of course we don't know what OPT's awake time is, however, we can simply iterate over the $\lg n$ possible serial work thresholds and try each one; at the end we take the best of these.

In general this strategy is guaranteed to 24-approximate OPT.

There is some sort of trade-off curve between how good the approximation is and the running time of our algorithm. However, at least asymptotically, it is satisfactory to have an $O(1)$-approximation algorithm for OPT with such good running time, in particular running time $O(n+(\lg p)(\lg n)^2)$ which is just $O(n)$.

3 Lower Bounds

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

3.1 Deterministic Algorithms for Minimizing Awake Time

It is clear that GR is not $(2-\epsilon)$-competitive for any $\epsilon > 0$. We might hope to achieve a $(1+\epsilon)$-competitive scheduling algorithm for this problem. 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 1 and Table 2 we specify two sets of tasks. For each time we give a list of which tasks arrive in the format $(\sigma, \pi) \times m$ where σ, π are the serial and parallel works of a task and m is how many of this type of task arrive at this time.

Table 1:

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

Table 2:

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 2 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 1, then the algorithm has again acted sub-optimally. In particular, for the schedule given in Table 1, 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.

3.2 Randomized Algorithms For Minimizing Awake Time

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 1 and Table 2; 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 1 at time $10i$ if $I_i = 1$ and the TAP from Table 2 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.

3.3 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 Recursion

TODO: First we must formalize this problem. Like what does this even mean?

5 An Algorithm for Optimizing Response Time

In this section we construct an algorithm that achieves low response time. As shown in Section 3, our algorithm necessarily uses preemption. We propose Algorithm 3, which we call PR(α)

ok, so according to Bill this problem has already been solved

Algorithm 3 PR(α)

while True **do**

OPT-schedule $\leftarrow$ how OPT would schedule tasks if all tasks were preempted and no more tasks come.

projected-schedule $\leftarrow$ what if we just wait

if OPT-schedule response time $\leq \alpha \cdot$ projected response time **then**

Run OPT-schedule