Serial-Parallel Scheduling Problem

Alek Westover

July 16, 2020

Abstract

There are many problems for which the best parallel algorithms have larger cost than the best serial algorithms. 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. The scheduler aims to minimize the total time that it has unfinished tasks. We analyze the competitive ratio of schedulers, i.e. the ratio of the time of a scheduler to the optimal time.

We construct a simple deterministic scheduler that does not use preemption that is 2-competitive for the scheduling problem. We also prove that no deterministic scheduler can have a competitive ratio smaller than 1.44 in general. We also exhibit a randomized scheduler that achieves expected competitive ratio at least 1.5.

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. Note that we consider time to be continuous, although our algorithms and bounds actually also apply if tasks must arrive and finish at discrete times; in fact having discrete time seems to hurt OPT much more than an off-line algorithm.

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. The scheduler attempts 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 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.

1.3 Related Work

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.

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 construct a simple deterministic scheduler that does not use preemption that is 2-competitive for the scheduling problem. We also prove that no deterministic scheduler can have a competitive ratio smaller than 1.44 in general. We also exhibit a randomized scheduler that achieves expected competitive ratio at least 1.5.

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 A Deterministic Scheduling Algorithm

In this section we present a simple deterministic scheduling algorithm that does not use preemption. We show that this algorithm is 2-competitive with OPT.

First we note that without loss of generality we may consider TAPs where the cost ratio $\pi(\tau)/\sigma(\tau)$ for any task τ is in $[1, p]$; if $\pi(\tau)/\sigma(\tau) < 1$, i.e. the parallel implementation has lower work than the serial implementation, then the scheduler clearly should never use the serial implementation of this algorithm, so we can replace the serial implementation with the parallel implementation and hence get cost ratio 1, similarly, if $\pi(\tau)/\sigma(\tau) > p$ then the scheduler should never run the parallel task and we can replace the parallel implementation of the task 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 are performing tasks 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 achieveing 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 brute force search can actually be performed on, at least in the special case of single-time-TAPs. Before analyzing ORACLE-1, which is a quite involved process, we analyze the competitive ratio of GR assuming the existence 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 important 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. We remark that OPT should not steal all of ALG. □

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 a TAP without OPT-gaps.

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.

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 at least $\sum_{i=1}^{n} n^{\binom{n}{i}} \geq 2^{2^n}$, which is pretty big. Nevertheless the existence of our algorithm is interesting, and the running time can almost certainly be reduced.

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,...,n}^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)
```

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. □

3 Lower-Bounds on Competitive Ratio

In this section we establish that it is impossible for a deterministic scheduler to get a competitive ratio lower than 1.44. That is, we show that for any deterministic algorithm ALG there is some input on which ALG has awake time at least 1.44 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	$(3, 3p/2) \times (p-1)$

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 would have awake time 4. In this case the competitive ratio of the algorithm is at least 2.

On the other hand, the algorithm could decide to run the task in parallel. If the algorithm decides to run the task in parallel, 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.

By optimizing this argument a bit we can get a stronger lower-bound of 1.44 on the competitive ratio (more specifically, we can get a lower bound of the positive root of the quadratic $x - 1/x = 3/4$ which is $(3 + \sqrt{73})/8 \in (1.44, 1.45)$).

TODO: hypothesis: can get a better bound, of 2, which would be tight

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.