Serial-Parallel Scheduling Problem

Alek Westover

July 15, 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 exhibit a scheduler that is 2-competitive for the symmetric-task case of this problem, a scheduler that is 4-competitive for the symmetric-cost-ratio case of this problem, and an algorithm that is 8-competitive for the general case of this problem. Further, we 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.

We also consider building an oracle to compute OPT, the optimal off-line strategy. We give a $O(2^n)$ time algorithm, where n is the number of tasks, that computes a strategy for OPT. We also show that constructing such an oracle is NP-hard.

1 Introduction

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.

2 Deterministic Scheduling Algorithms

In this section we present deterministic scheduling algorithms for this problem. We first analyze the competitive-ratio of simple algorithms for special cases of the problem, and then generalize these strategies to get an algorithm with low competitive-ratio in the general setting of the problem.

2.1 Symmetric-Tasks Case

TODO: hypothesis: we can actually do way better than just being 2-competitive is my guess...

First we consider a special case of the problem: the case where all tasks have identical serial and parallel works. Let the work of the serial implementations be 1, and let the work of the parallel implementations be $k \in [1, p]$[1]. Throughout this subsection it is implicit that any TAP we refer to consists of identical tasks with serial work 1 and parallel work k.

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 ***SGR*** (SGR stands for "simple greedy"), for scheduling in the symmetric-tasks case.

Algorithm 1 SGR

```
while True do
    if verge time then
        q ← number of ready tasks
        schedule ⌊q/p⌋ serial tasks to each processor
        q' ← q mod p
        if q' ≥ p/k then
            schedule q' tasks in serial
            giving each processor at most 1 task
        else
            schedule q' tasks in parallel
            distributing their work equally
```

In the rest of this subsection we analyze the competitive ratio of SGR.

Consider a TAP $\mathcal{T}$. Let v be the number of verge times; note that $v \leq n$ 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 SGR at time t_i. Let $T^{ALG}(q_1, \ldots, q_{v'})$ 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 $t_{v'}$.

First we analyze T^{OPT} and T^{CHILL} for the case where all the tasks arrive at a single time, i.e. $v = 1$. Let

$$T(q) = \lfloor q/p \rfloor + \min(1, (q \bmod p) \cdot k/p).$$

Claim 1.

$$T^{SGR}(q) = T(q).$$

Proof. SGR first schedules $\lfloor q/p \rfloor$ tasks in serial to each processor; completing these tasks requires time $1 \cdot \lfloor q/p \rfloor$. Then SGR either schedules $q \bmod p$ tasks in serial, which takes time 1, or schedules $q \bmod p$ tasks in parallel, which takes time $(q \bmod p)k/p$; in particular SGR chooses whichever of these options leads to lower awake time. Hence overall on this TAP SGR achieves awake time $T(q)$. □

Claim 2.

$$T^{OPT}(q) = T(q).$$

Proof. We now claim that OPT can do no better than this. This is obvious if $q \leq p$: if OPT schedules any task in serial then OPT has awake time at least 1, and if OPT schedules everything in parallel then OPT has awake time at least qk/p; for $q \leq p$ we have $T(q) = \min(1, qk/p)$, so OPT can do no better than SGR here. It is intuitively obvious that if there are $q > p$ tasks, then scheduling about $\lfloor q/p \rfloor$ tasks in serial to each processor like SGR does is the right strategy; we now formalize this.

Say OPT schedules x of the q tasks in serial, and schedules the remaining $q - x$ in parallel.

Say that an optimal assignment of the tasks, under the constraint that x of the tasks are scheduled in serial and $q - x$ are scheduled in parallel, achieves awake time M.

We claim that there must exist an optimal assignment of tasks such that each processor is assigned either $\lfloor x/p \rfloor$ or $\lceil x/p \rceil$ serial tasks. To prove this, we start from some optimal assignment, and modify it in such a way as to make the configuration "closer" to our desired configuration. Let $n_\sigma(\rho_i)$ be the number of serial tasks scheduled on processor ρ_i.

To formalize a notion of "closeness" we define a potential function ϕ of the assignment S of tasks:

$$\phi(S) = \sum_i \min(|n_\sigma(\rho_i) - \lfloor x/p \rfloor|, |n_\sigma(\rho_i) - \lceil x/p \rceil|).$$

Note that $\phi(S)$ is non-negative. We desire a configuration with $\phi(S) = 0$. Consider a configuration of

[1] Note that without loss of generality $k \in [1, p]$: if $k < 1$, 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 $k = 1$, similarly, if $k > p$ then the scheduler should never run the parallel task and we can replace the parallel implementation of the task with the serial implementation.

tasks achieving awake time M with $\phi(S) > 0$. We first apply the following procedure to the configuration: while the difference between the maximum work assigned to a processor and the minimum work assigned to a processor is more than 1 swap 1 unit of work from a processor with the maximum amount of work to a processor with the minimum amount of work. This swap cannot increase the range of works assigned to processors, and the process must eventually terminate. Now, while $\phi(S) > 0$ there must exists ρ_i, ρ_j with $n_\sigma(\rho_i) < \lfloor x/p \rfloor$ and $n_\sigma(\rho_j) > \lfloor x/p \rfloor$ or there exists ρ_i, ρ_j with $n_\sigma(\rho_i) < \lceil x/p \rceil$ and $n_\sigma(\rho_j) > \lceil x/p \rceil$. By construction the range of the works is at most 1; hence ρ_i has at least 1 unit of parallel work to have work within 1 of ρ_j despite having at least 2 fewer serial tasks than ρ_j. Then ρ_i gives this 1 unit of parallel work to ρ_j, and in exchange ρ_j gives ρ_i a serial task. This swapping operation decreases $\phi(S)$ by exactly 1, and does not change the amount of work assigned to each processor, which importantly means that the swap does not increase the range of the works. Hence, we can repeat this swapping process to eventually achieve a configuration with awake time M where each processor has $\lfloor x/p \rfloor$ or $\lceil x/p \rceil$ serial tasks.

□

Combining Claim 1 and Claim 2 we have

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

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

An ALG-gap is an interval of time of non-zero length where for all times in the interior of the interval 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 3. *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. □

Note that as an immediate consequence of Claim 3 we have that any ALG-gap is 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 4. *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 3 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 4, in order to bound SGR's competitive ratio, it suffices to consider TAPs without SGR-gaps. Note however that a TAP without SGR-gaps could still have OPT-gaps.

We conclude our analysis of the competitive-ratio of SGR 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: we consider SGR's competitive-ratio on a TAP without OPT-gaps.

Claim 5. *SGR is 2-competitive on any OPT-gap-free TAP.*

Proof. Since the TAP is OPT-gap-free we must have

$$T^{OPT}(q_1, \ldots, q_v) \geq T^{SGR}(q_1, \ldots, q_{v-1}). \tag{2}$$

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

$$T^{SGR}(q_1, \ldots, q_v) = \sum_{i=1}^{v} T^{SGR}(q_i). \tag{3}$$

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

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

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

$$\begin{aligned} T^{SGR}(q_1, \ldots, q_v) &\leq T^{OPT}(q_1, \ldots, q_v) + T^{OPT}(q_v) \\ &\leq 2T^{OPT}(q_1, \ldots, q_v), \end{aligned}$$

as desired. □

Proposition 1. *SGR 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 5, which says that if there are 0 OPT-gaps then SGR 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:

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

□

2.2 Symmetric-Cost-Ratios Case

Next we consider the case where there are different tasks with implementations that have different works, but with the restriction that the cost ratio of the parallel implementation to the serial implementation is some fixed value k.

The ideas in this section were inspired by extremely elegant analysis of an unrelated scheduling problem in [1].

Making a global definition of 1 unit of work is now difficult to do in a meaningful way, so we do not do this. Instead, at every verge time we define locally 1 unit of work to be the work of the serial implementation of the task with the serial implementation with the most work. Further, we partition the unscheduled ready tasks at a given verge time into sets called ***level-i*** sets based on the work of their serial implementation: the level-i set of tasks on a verge time is the unscheduled ready tasks that have serial implementation's with work in $[1/2^{i+1}, 1/2^i]$. We now define a ***virtual-task*** to be a collection of tasks. The work of the serial and parallel implementations of a virtual-task are the sums of the works of the serial and parallel implementations of the virtual-tasks constituent tasks.

We propose Algorithm 2, which we call ***LGR*** (LGR stands for "leveled greedy"), for scheduling in the symmetric-cost-ratio case.

Algorithm 2 LGR

while True **do**
  if verge time **then**
    Combine unscheduled ready tasks into virtual-tasks to maximize the number of level-0 virtual-tasks
    $q \leftarrow$ number of virtual-tasks
    if $q \geq p/k$ **then**
      schedule $\min(q, p)$ virtual-tasks in serial giving each processor at most 1 virtual-task
    else
      schedule a level-0 task in parallel distributing its work equally

We prove the following regarding LGR:

Proposition 2. *LGR is 4-competitive.*

Proof. **TODO:** hmm □

2.3 General Case

Now we are ready to consider the general case, i.e. we place no restrictions on the tasks in this Subsection. We use the definitions from Subsection 2.2 and Subsection 2.1. We propose Algorithm 3, which we call ***GGR*** (GGR stands for "general greed").

Algorithm 3 GGR

while True **do**
  if verge time **then**
    Combine unscheduled ready tasks into virtual-tasks to maximize the number of level-0 virtual-tasks
    $q \leftarrow$ number of virtual-tasks
    if $q \geq p/k$ **then**
      schedule $\min(q, p)$ virtual-tasks in serial giving each processor at most 1 virtual-task
    else
      schedule a level-0 task in parallel distributing its work equally

We claim the following regarding GGR:

Proposition 3. *GGR is 8-competitive with OPT.*

Proof. **TODO:** hmm □

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 there is some input on which OPT has awake time at most half of the awake time of the deterministic scheduler.

Note that the competitive ratio is trivially at least 1.

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 in the general case of the Serial-Parallel Scheduling Problem, or in fact in the symmetric-cost-ratio case of the problem, 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: I kinda want a bound on the symmetric-tasks case... At the moment I kind of am thinking that maybe there is no good bound for that case... **TODO:** better bound in the general case?

4 Randomized Scheduling Algorithms

Given a particular deterministic scheduling algorithm there will be some inputs on which the algorithm will perform poorly. By employing randomization these worst case inputs can be mitigated somewhat, at least in expectation.

We propose Algorithm 4, which we call ***RGR*** (RGR stands for "randomized greedy"), for this case.

Algorithm 4 RGR

```
while True do
    if verge time then
        sleep for a random amount of time, chosen uniformly at random from something, not really sure what
        q ← number of ready tasks
        if q ≥ p/k then
            schedule min(q, p) tasks in serial
            giving each processor at most 1 task
        else
            schedule one task in parallel
            distributing its work equally
```

Proposition 4. *The expectation of RGR's competitive ratio on any input is at least* 1.5.

Proof. **TODO:** hmmm. □

5 Recursion

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

6 Oracle for OPT

In this section we consider the question of creating an off-line scheduling algorithm. We propose Algorithm 5 for this.

Algorithm 5 OPT Oracle

```
minAwakeTime ← ∞
for I ∈ {0,1}^n do
    if I_i = 1 then
        Schedule task i in parallel when it arrives
    else
        Schedule task i in serial when it arrives
    In particular, first schedule serial tasks to minimize awake time
    And then sprinkle parallel tasks on top
    if awakeTime_I ≤ minAwakeTime then
        minAwakeTime ← awakeTime_I
```

In Proposition 5 we show that Algorithm 5, which we call OPT Oracle, is an Oracle for OPT. Before proving this, we establish the following Lemma:

Lemma 1. *Given tasks $\tau_1, \ldots, \tau_n$ with serial works $\pi(\tau_i)$ and parallel works $\sigma(\tau_i)$ that all arrive at the same time along with binary indicators $I(\tau_i) \in \{0, 1\}$ specifying that task τ_i must be scheduled in parallel if $I(\tau_i) = 1$ and serial if $I(\tau_i) = 0$, then to schedule the tasks to minimize awake time it suffices to first place the serial tasks, in order of*

Proof. Consider each of the n bins to have a bunch of serial tasks, and then a little bit of parallel frosting on top. Clearly given any arrangement of the serial tasks the best thing to do with the serial tasks is to maximally smush them on top of everything. **TODO:** And this works out better if the serial thing started out better. □

Proposition 5. *OPT Oracle actually computes OPT. OPT Oracle has running time $O(2^n)$.*

Proof. **TODO:** hmm □

Conjecture 1. *Constructing an oracle for OPT is NP-Hard.*

Proof. **TODO:** Find some input on which you can reduce the problem to 3-SAT. □

7 Conclusions

TODO: Greed is good. An interesting question is: the whole not doing preemption thing seems pretty dumb, what's going on?

References

[1] Davide Bilò, Luciano Gualà, Stefano Leucci, Guido Proietti, and Giacomo Scornavacca. Cutting bamboo down to size. *FUN*, 2020.