Serial-Parallel Scheduling Problem

Alek Westover

July 13, 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.

We prove that no deterministic scheduler can have a competitive ratio smaller than 2.

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.

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 $\tau_1, \dots, \tau_n$ (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 sequence of tasks with their associated parallel and serial implementations works and with their associated arrival times is called a ***task arrival plan***. 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 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 the scheduler can choose to run the task in parallel, in which case the scheduler can distribute the tasks 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 exhibit three scheduling algorithms that guarantee small competitive ratios. We start with looking at special cases of the game, and build on the strategies from the special cases to get algorithms that work in more general settings.

2.1 Symmetric-Tasks Case

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 task arrival plan 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 ***GRD*** (which stands for "greedy"), for scheduling in the symmetric-tasks case.

Algorithm 1 GRD

```
while True do
    if verge time then
        $q \leftarrow$ number of ready tasks
        if $q \geq p/k$ then
            schedule $\min(q,p)$ tasks in serial
            giving each processor at most 1 task
        else
            schedule $q$ tasks in parallel
            distributing work equally
```

In the rest of this subsection we analyze the competitive-ratio of GRD.

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 had no work. If OPT has work at 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 cannot steal all of ALG. □

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 at time t_i, let $\Delta_i t = t_{i+1} - t_i$, let $\Delta_i q = q_{i+1} - q_i$. Define $t_0 = -\infty, t_{v+1} = +\infty$; these are not verge times, but are merely defined for convenience. We call the interval (t_i, t_{i+1}) for $i \in \{0, 1, \dots, v\}$ ***valley interval i***.

Let $T^{OPT}(q_1, \dots, q_{v'})$ and $T^{GRD}(q_1, \dots, q_{v'})$ denote the awake time of OPT and GRD respectively on the truncation of the task arrival plan that only consists of tasks arriving at times before verge time $t_{v'}$ where the task arrival plan is such that GRD has q_i ready tasks at verge time t_i for all $i \leq v'$. Let

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

We claim that when all work arrives before the first verge time T^{OPT} and T^{GRD} are the same, and in particular are T.

Claim 2.

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

Proof. If work only arrives before a single verge time, then all the work arrives at the same time.

GRD first schedules p tasks in serial for $\lfloor q/p \rfloor$ verge times, which takes time $1 \cdot \lfloor q/p \rfloor$. Then on the final verge time GRD 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 GRD chooses whichever of these options leads to lower awake time. Hence overall on this task arrival plan GRD achieves awake time $T(q)$.

We now claim that OPT can do no better than this. **This is pretty clear.** We remark that in this sense GRD is greedy: it "locally" schedules optimally. □

An ALG-gap is a interval of time of non-zero length on which the scheduling algorithm ALG has completed all tasks, but more tasks have yet to arrive.

Claim 3. *If an algorithm ALG can achieve competitive ratio r on any task arrival plan with no ALG-gaps then it achieves competitive-ratio r on arbitrary task arrival plans.*

Proof. If a task arrival plan has ALG-gaps, then we can partition the tasks based on arrival time, splitting the tasks into the same group if the interval of time book-ended by their arrival times has no ALG-gaps in it. In particular, we can define intervals $I_1, \dots, I_g$ of time such that there are no ALG-gaps on I_i for all $i \leq g$, and such that some tasks arrive in I_i. By Claim 1 an ALG-gap is also an OPT-gap. Let $T_{I_i}^{OPT}$ and $T_{I_i}^{ALG}$ denote the awake time of OPT and ALG on interval I_i. Note that clearly $T^{OPT} = \sum_i T_{I_i}^{OPT}$ and $T^{ALG} = \sum_i T_{I_i}^{ALG}$. Hence if $T_{I_i}^{ALG} \leq 2T_{I_i}^{OPT}$ for all i then $T^{ALG} \leq 2T^{OPT}$, as desired. □

[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.

By Claim 3, in order to bound GRD's competitive-ratio, it suffices to consider task arrival plans without GRD-gaps. Note however that a task arrival plan without GRD-gaps could still have OPT-gaps.

Claim 4. *GRD is* 2*-competitive on any task arrival plan with no OPT-gaps.*

Proof. Because **see my fairly convincing picture** we have

$$\begin{aligned} T^{OPT}(q_1,\dots,q_\ell) &\ge T^{GRD}(q_2,\dots,q_\ell) \\ &\ge T^{GRD} - T(q_1) \end{aligned}$$

Hence

$$\begin{aligned} T^{GRD}(q_1,\dots,q_\ell) &\le T^{OPT}(q_1,\dots,q_\ell) + T(q_1) \\ &\le 2T^{OPT}(q_1,\dots,q_\ell). \end{aligned}$$

□

Proposition 1. *GRD is* 2*-competitive.*

Proof. The proof is by induction on the number of OPT-gaps. The base case of our induction is established in Claim 4, which says that if there are 0 OPT-gaps then GRD is 2-competitive.

Consider a task sequence with its first OPT-gap at time t_*. Let ℓ be the largest index such that verge time $t_\ell < t_*$ □

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 ***LEVELGRD***, for scheduling in the symmetric-cost-ratio case.

Algorithm 2 LEVELGRD
```
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 $q \ge p/k$ 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 LEVELGRD:

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 ***GENERALGRD***.

Algorithm 3 GENERALGRD
```
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 $q \ge p/k$ 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 GENERALGRD:

Proposition 2. *GENERALGRD is* 8*-competitive with OPT.*

Proof. eh, how bad could it possibly be □

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 2. 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: do a completely different argument to get a better bound.

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 ***RANDGRD***, for this case.

Algorithm 4 RANDGRD

```
while True do
    if verge time then
        sleep for a random amount of time, chosen uniformly at random from something, not really sure what
        $q \leftarrow$ number of ready tasks
        if $q \geq p/k$ 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 3. *The expectation of RANDGRD's competitive-ratio on any input is at least* 1.5.

Proof. hmmm. ☐

5 Recursion

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

6 Conclusions

GRD is a pretty good algorithm. An interesting question is: GRD seems pretty dumb, why is it so good then? I'm not totally sure.

References

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