Alek Westover

To Schedule in Parallel, or To Schedule in Serial, That is the Question?

1 TODO

- prove / disprove the claim "LNPU is 2-competitive in single-task version"
- prove / disprove the claim "HnH is 2-competitive in single-task version"
- prove / disprove the claim "LNPU is 2-competitive in multi-task version"
- prove / disprove the claim "HnH is 2-competitive in multi-task version"
- prove / disprove the claim "Threshold is $O(1)$-competitive in single-task version"
- generalize the threshold strategy
- recursion

2 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 aren't work efficient, they may have asymptotically greater work than the parallel version.

Imagine we have to perform n tasks $\tau_1, \ldots, \tau_n$ (n unknown ahead of time). Say that we have p processors $\rho_1, \ldots, \rho_p$. Each task τ_i has a parallel implementation with work π_i and a serial implementation with work σ_i. We will model time as a series of time steps. In particular, times are non-negative integers. The works π_i, σ_i are expressed in units of time-steps. The tasks will become available at some time steps $t_1, \ldots, t_n$. At time step t_i the scheduler must decide whether to run task τ_i using its parallel implementation or its serial implementation. Fix some time step t. Intuitively, if there are many i such that $t_i = t$, i.e. if many tasks arrive on a single time step, then the serial implementations of the tasks should be run because the scheduler can achieve parallelism accross the tasks. On the other hand, if there are not very many tasks that arrive in a given time step t it might be better for the scheduler to run the parallel versions of these tasks – even though they are possibly not work efficient, i.e. $\pi_i > \sigma_i$ – because then at least the scheduler can achieve parallelism within tasks.

Let $w_i(t)$ be the unfinished work assigned to the i-th processor at the start of the t-th time step. Let $w_i'(t)$ be the unfinished work assigned to the i-th processor after the scheduler has scheduled new work to processors on the t-th time step, but before the processors have done any work. Note that by definition processors all do 1 unit of work per time step (unless there is less than 1 unit of work to be done, in which case they just do all the available work). Hence $w_i(t+1) = \max(0, w_i'(t) - 1)$.

Let $\max_i w_i(t)$ be called the backlog on time step t. There is unfinished work on a time step t if $\sum_{i=1}^{p} w_i(t) > 0$, or you could say $\max_i w_i(t) > 0$. Let the ***awake time*** of the scheduler be the number of time steps that it has unfinished work. 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. Specifically, the ***competitive ratio*** of a scheduler is the ratio of its awake time to the awake time of OPT on the same input.

Results. We prove some bounds on this.

3 Special Case: A Single Task

First we consider a special case: where there is a single task that must be performed many times. We prove the following result about this case:

Proposition 1. *Let $\tau_1, \ldots, \tau_n$ be task all with k as the running time of their serial implementations, and with $k \cdot r$ as the running time of their parallel implementations. Then there is a scheduling algorithm that is $O(\sqrt{k})$-competitive with OPT in terms of backlog. Furthermore, any such algorithm for this problem is at least $\Omega(\sqrt{k})$ times slower than OPT on some input.*

Proof. The algorithm proceeds in 2 steps. Consider some time step t. First the algorithm schedules the serial implementation of

$$\left\lfloor \frac{(\max_j w_j(t)) - w_i(t)}{k} \right\rfloor$$

tasks to every processor ρ_i, or if this is not possible, it does an arbitrary appropriately large subset of these assignments. This pretty much doesn't matter, because dumping stuff into processors that are more than a task ahead of the most behind processor a) obviously doesn't increase the backlog, and b) it's not like you wanna do parallel versions of these things.

So by now it's pretty clear that any scheduler might as well first doing this.[1]

Anyways, what about if $m > 0$ tasks are left after the tasks have been packed in? In this case, the algorithm does chooses what to do by comparing m to a threshold T. If $m < T$ then the scheduler says ok thats not too many tasks, lets schedule their parallel implementations, scheduling them to basically try to equalize all the fills. So like in particular, find the largest x such that you can raise the x least full cups to all have uniform fill and then raise these cups to as high as you can. On the other hand, if $m \geq T$ then the scheudler is like, ok that's enough tasks that we can just get parallelism accross tasks, let's just schedule them all in serial, scheduling them to the m least busy processors.

It turns out that $T = \Theta\left(\frac{p}{r\sqrt{k}}\right)$ is a pretty decent choice of T.

OK so anyways, let's look at two nasty configuration of tasks to get some intuition for why $\Omega(\sqrt{k})$ can't be beat and why the alg described above achieves a $O(\sqrt{k})$-competitive ratio.

OK so one nasty thing that could happen is if there was a time step when $T + 1$ tasks arived, and then no tasks ever arived again. If we could see into the future we would be like, ok, that's not too many tasks, lets just schedule the paralallel implementations of the tasks. But we are stuck without the ability to see the future. Dang. So anyways, this is just above the threshold we decided on for when to run serial implementations of the tasks. So we do it, and it takes about k time steps. On the other hand OPT is going to do everything in parallel, so it finishes in time Tkr/p ish which is like $\sqrt{k}$ time steps.

Another nasty thing is if $T - 1$ tasks came on consecutive time steps for $p/(T-1)$ time steps. So at this point you might be like dang you should be aborting tasks and switching to serial implementations. Well this is a fair point. But ignoring that, OPT would get $p/T = r\sqrt{k}$ running time, wheras its gonna take our alg about kr time steps.

So intuitively, we've looked at the worst stuff that could happen. Which is why I kinda feel like you can't beat this, and that our alg achieves this.

Of course this isn't a proof.

In fact, I no longer believe the claim, at least if we are allowed to terminate and restart tasks!

I'll work on it.

Yeah so turns out that the strategy that I proposed above is **really not great**.

□

4 Single-Task Special Case, Now With Termination

Now we demonstrate a good strategy when you're allowed to terminate.

Note that OPT never terminates anything.

Proposition 2. *We can do pretty well.*

Proof. First we need two definitions. Call a processor ρ_i under-loaded at time step t if $w_i(t) < k$. Call a processor ρ_i bored at time step t if $w_i(t) < 1$.

The strategy is detailed in Algorithm 2.

Algorithm 3 is another pretty interesting strategy.

Algorithm 1 NonHarmfullTermination

procedure NonHarmfullTermination(t)
    while bored processors remain **and** there are procesors whose fill could be reduced by terminating a serial they are running, and distributing the parallel implementation's work amongst some processors without making those processors have $w_i'(t) > 1$ **do**
        Terminate the most full processor's serial task
        Distribute its work amongst bored processors ρ_i, without making any $w_i(t) > 1$
    end while
end procedure

Algorithm 2 Threshold it

procedure Move(t, m) ▷ Schedule m tasks (arrived plus stored) on time step t
    $b \leftarrow$ number of bored processors at time step t ▷ i.e. processors ρ_i with $w_i(t) < 1$
    if $m \geq b$ **then**
        Schedule b tasks in serial on the bored processors.
    else if $b > m \geq b/r$ **then**
        Schedule m tasks in serial on the bored processors that have the least amount of work.
        Run NonHarmfullTermination(t)
    else if $b/r > m \geq b/(kr)$ **then**
        Schedule kr tasks in parallel on the bored processors.
    else if $m < u/(kr)$ **then**
        Run NonHarmfullTermination(t)
    end if
end procedure

Algorithm 3 Leave no Parallel Task Undone

procedure Move(t, m) ▷ Schedule m tasks (arrived plus stored) on time step t
    $b \leftarrow$ number of bored processors at time step t ▷ i.e. processors ρ_i with $w_i(t) < 1$
    Schedule $\min(m, b)$ tasks in serial to the bored processors.
    Run NonHarmfullTermination(t)
end procedure

Algorithm 4 Half and Half

Choose half of the cups to be in the serial pile and half of the cups to be in the parallel pile.
procedure Move(t, m) ▷ Schedule m tasks (arrived plus stored) on time step t
    Schedule any tasks that come in to the serial pile cups (schedule like this such that you minimize backlog).
    Take serial tasks from the most full cups in the serial pile of cups and distribute them evenly amongst the parallel pile cups, until this would necessarily result in a cup having more than 1 as its average fill.
end procedure

Algorithm 5 Half and Half version 2

Choose half of the cups to be in the serial pile and half of the cups to be in the parallel pile.
procedure Move(t, m) ▷ Schedule m tasks (arrived plus stored) on time step t
    Schedule any tasks that come in to the serial pile cups (schedule like this such that you minimize backlog).
    Take serial tasks from the cups in the serial pile of cups **that have been waiting the longest** and distribute them evenly amongst the parallel pile cups, until this would necessarily result in a cup having more than 1 as its average fill.
end procedure

We claim that Algorithm 2, which we will refer to as THRESH, satisfies the following invariant:

$$\pi \tag{1}$$

We inductively prove (1). This will imply that it's $O(1)$-competitive. Or something like that maybe.
I think that Algorithm 4 and Algorihtm 3 are both 2-competitive.

□

Claim 1. *LNPU (leave no parallel task undone) is 2-competitive with OPT.*

Proof. First consider how LNPU deals with a pulse of $m < p$ tasks followed by nothing. (Note: if it gets more than p tasks, then it might as well just save those and get them k rounds later, because in this case its obviously going to schedule everything in serial)

If $m \geq p/r$ it of course schedules everything in serial, and finishes in k steps. Clearly you can't do better than this; if you scheduled everything in parallel from the start it would take k steps, so you might as well schedule something in serial.

If $m \leq p/(rk)$ it of course schedules everything in parallel and finishes in 1 time step. Obviously there is nothing better than this.

If $m = c \cdot p/r$ for some $c \in (1/k, 1)$ then either a) $c \geq 1/2$ in which case, we simply note that the algorithm obviously finishes in no more than k steps, which is of course no more than $2ck$, or b) $c \leq 1/2$ in which case there are at least $p/2$ processors that aren't getting serial tasks, so these will basically finish the tasks in $2ck$ time. OPT will finish this in ck time. Either way, LNPU is 2-competitive with OPT on this input.

Now, we analyze what would happen if there wasn't just a single pulse.

Intuitively LNPU does even better if it's not just a single pulse because then it gets to schedule more stuff in serial and gets more parallelism for doing so. Yippee!

More formally, ...

Now let's imagine it was more like $m_1, m_2, \ldots$ that came in.

□

Claim 2. *The Half and Half Alg (HnH) is 2-competitive with OPT when there's just a single task.*

Proof. □

5 Full Problem

Now we look at the problem in its full generality.

I feel like the alg from above is pretty good, but the "packing" step doesn't really make sense any more?

What are the equivalents of k, r here? Can we mandate an average task size or somehting? That's not super elegant...

The "Leave No Parallel Task Undone" strategy can be applied here without modifications! If I can prove upper-bounds on its competitive ratio in the single-task varaint of this game (which seems very likely possible, I haven't found any examples that prove anything more than a trivial $\Omega(1)$ lower bound on its competitve ratio!) then we'll be in great shape here! That is, I bet we'll be able to analyze this thing.

I also bet HnH is $O(1)$ competitive in both cases.

6 Random Thoughts

- there is a 2-competitive alg
- there is NO $(2 - \epsilon)$-competitive alg for $\epsilon > 0$

[1] For a while I was thinking about maybe even letting these guys rise to some $\epsilon > 0$ above the backlog. But then I was like, maybe lets not.