# Scheduling Revisited

The previous analysis of the scheduling algorithm gave a makespan of

$$\frac{1}{m}\sum_{j\neq\ell} p_j + p_\ell$$

where $\ell$ is the last job to complete.

Together with the obervation that if each $p_i \geq \frac{1}{3}C^*_{\max}$ then LPT is optimal this gave a $4/3$-approximation.

# 16.2 Scheduling Revisited

Partition the input into long jobs and short jobs.

A job $j$ is called short if

$$p_j \leq \frac{1}{km} \sum_i p_i$$

**Idea:**

1. Find the optimum Makespan for the long jobs by brute force.
2. Then use the list scheduling algorithm for the short jobs, always assigning the next job to the least loaded machine.

We still have the inequality

$$\frac{1}{m}\sum_{j\neq\ell} p_j + p_\ell$$

where $\ell$ is the last job (this only requires that all machines are busy before time $S_\ell$).

If $\ell$ is a long job, then the schedule must be optimal, as it consists of an optimal schedule of long jobs plus a schedule for short jobs.

If $\ell$ is a short job its length is at most

$$p_\ell \leq \sum_j p_j/(mk)$$

which is at most $C^*_{\max}/k$.

Hence we get a schedule of length at most

$$\left(1 + \frac{1}{k}\right) C^*_{\max}$$

There are at most $km$ long jobs. Hence, the number of possibilities of scheduling these jobs on $m$ machines is at most $m^{km}$, which is constant if $m$ is constant. Hence, it is easy to implement the algorithm in polynomial time.

**Theorem 3**
*The above algorithm gives a polynomial time approximation scheme (PTAS) for the problem of scheduling $n$ jobs on $m$ identical machines if $m$ is constant.*

We choose $k = \lceil \frac{1}{\epsilon} \rceil$.

How to get rid of the requirement that $m$ is constant?

We first design an algorithm that works as follows:
On input of $T$ it either finds a schedule of length $(1 + \frac{1}{k})T$ or certifies that no schedule of length at most $T$ exists (assume $T \geq \frac{1}{m}\sum_j p_j$).

We partition the jobs into long jobs and short jobs:

- A job is long if its size is larger than $T/k$.
- Otw. it is a short job.

- We round all long jobs down to multiples of $T/k^2$.
- For these rounded sizes we first find an optimal schedule.
- If this schedule does not have length at most $T$ we conclude that also the original sizes don't allow such a schedule.
- If we have a good schedule we extend it by adding the short jobs according to the LPT rule.

After the first phase the rounded sizes of the long jobs assigned to a machine add up to at most $T$.

There can be at most $k$ (long) jobs assigned to a machine as otw. their rounded sizes would add up to more than $T$ (note that the rounded size of a long job is at least $T/k$).

Since, jobs had been rounded to multiples of $T/k^2$ going from rounded sizes to original sizes gives that the Makespan is at most

$$\left(1+\frac{1}{k}\right)T \enspace .$$

During the second phase there always must exist a machine with load at most $T$, since $T$ is larger than the average load. Assigning the current (short) job to such a machine gives that the new load is at most

$$T + \frac{T}{k} \leq \left(1 + \frac{1}{k}\right)T \ .$$

**Running Time for scheduling large jobs:** There should not be a job with rounded size more than $T$ as otw. the problem becomes trivial.

Hence, any large job has rounded size of $\frac{i}{k^2}T$ for $i \in \{k, \dots, k^2\}$. Therefore the number of different inputs is at most $n^{k^2}$ (described by a vector of length $k^2$ where, the $i$-th entry describes the number of jobs of size $\frac{i}{k^2}T$). This is polynomial.

The schedule/configuration of a particular machine $x$ can be described by a vector of length $k^2$ where the $i$-th entry describes the number of jobs of rounded size $\frac{i}{k^2}T$ assigned to $x$. There are only $(k+1)^{k^2}$ different vectors.

This means there are a constant number of different machine configurations.

Let $\mathrm{OPT}(n_1,\dots,n_{k^2})$ be the number of machines that are required to schedule input vector $(n_1,\dots,n_{k^2})$ with Makespan at most $T$.

**If $\mathrm{OPT}(n_1,\dots,n_{k^2}) \le m$ we can schedule the input.**

We have

$$\begin{aligned}
&\mathrm{OPT}(n_1,\dots,n_{k^2})\\
&\quad=\begin{cases}
0 & (n_1,\dots,n_{k^2}) = 0\\
1 + \min\limits_{(s_1,\dots,s_{k^2})\in C} \mathrm{OPT}(n_1-s_1,\dots,n_{k^2}-s_{k^2}) & (n_1,\dots,n_{k^2}) \gneqq 0\\
\infty & \text{otw.}
\end{cases}
\end{aligned}$$

where $C$ is the set of all configurations.

Hence, the running time is roughly $(k+1)^{k^2} n^{k^2} \approx (nk)^{k^2}$.

We can turn this into a PTAS by choosing $k = \lceil 1/\epsilon \rceil$ and using binary search. This gives a running time that is exponential in $1/\epsilon$.

Can we do better?

Scheduling on identical machines with the goal of minimizing Makespan is a strongly NP-complete problem.

**Theorem 4**

*There is no FPTAS for problems that are strongly NP-hard.*

- Suppose we have an instance with polynomially bounded processing times $p_i \le q(n)$
- We set $k := \lceil 2nq(n) \rceil \ge 2\,\mathrm{OPT}$
- Then

$$\mathrm{ALG} \le \left(1 + \frac{1}{k}\right)\mathrm{OPT} \le \mathrm{OPT} + \frac{1}{2}$$

- But this means that the algorithm computes the optimal solution as the optimum is integral.
- This means we can solve problem instances if processing times are polynomially bounded
- Running time is $\mathcal{O}(\mathrm{poly}(n,k)) = \mathcal{O}(\mathrm{poly}(n))$
- For strongly NP-complete problems this is not possible unless P=NP

## More General

Let $\text{OPT}(n_1, \dots, n_A)$ be the number of machines that are required to schedule input vector $(n_1, \dots, n_A)$ with Makespan at most $T$ ($A$: number of different sizes).

If $\text{OPT}(n_1, \dots, n_A) \leq m$ we can schedule the input.

$$
\begin{aligned}
&\text{OPT}(n_1, \dots, n_A) \\
&\quad = \begin{cases} 0 & (n_1, \dots, n_A) = 0 \\ 1 + \min\limits_{(s_1, \dots, s_A) \in C} \text{OPT}(n_1 - s_1, \dots, n_A - s_A) & (n_1, \dots, n_A) \gneq 0 \\ \infty & \text{otw.} \end{cases}
\end{aligned}
$$

where $C$ is the set of all configurations.

$|C| \leq (B+1)^A$, where $B$ is the number of jobs that possibly can fit on the same machine.

The running time is then $O((B+1)^A n^A)$ because the dynamic programming table has just $n^A$ entries.