We investigate the classical scheduling problem $P \mid || C_{max}$ where a set of $n$ independent jobs has to be processed on $m \geq 2$ parallel and identical processors (machines) such that the makespan $C_{max}$ is minimized. If $n \leq m$ then any schedule that just assigns each job to a different machine would be optimal. Therefore we assume that $n \geq m$. We denote the minimum makespan across any schedule by $C^*_{max}$.

1 Lower Bounds

Let $A$ be any algorithm that solves $P \mid || C_{max}$ with makespan $C^A_{max}$. Let $J_k$, with processing time $p_k$ and completion time $C_k$, be the job that determines the makespan.

Lemma 1 (Lower Bound):

$$C^A_{max} \geq C^*_{max} \geq \max \left\{ C_k, \frac{\sum_{j=1}^n p_j}{m} \right\}$$

Proof. $\sum_{j=1}^n p_j$ is the total amount of processing that must be done across all machines. Then for $m$ machines,

$$\frac{\sum_{j=1}^n p_j}{m}$$

would be the average processing that needs to be done. By properties of the average, at least one machine must then process at least this average. Hence, there will be always at least one job $J_k$ with completion time

$$C_k \geq \frac{\sum_{j=1}^n p_j}{m}.\,$$

Lemma 2 (Lower Bound): $C^A_{max} \geq C^*_{max} \geq p_{max} \geq p_j$ (for any $j = 1, \ldots, n$).

Proof. A processor can process only one job at a time. This means, a job can be completed no faster than it’s given processing time. If $p_{max} = \max\{p_j\}$, then at least one machine must have $p_{max}$ processing load.

As a result, $\sum_{j=1}^n p_j/m$ and $p_{max}$ form lower bounds for the makespan of any algorithm solving $P\mid || C_{max}$ and clearly $C^A_{max} \geq C^*_{max} \geq \max \left\{ (\sum_{j=1}^n p_j)/m, \ p_{max} \right\}$. 

1
2 List Scheduling

Let LS be the List Scheduling algorithm: Always take the next job of the list and process it on that processor that has done the least amount of work so far (if there is more than one processor with the smallest load use that with the smallest index).

Lemma 3 (Upper Bound): \( C_{max}^{LS} \leq \sum_{j=1}^{n} p_j \).

**Proof.** This bound is obviously true as List Scheduling would never use just one processor (actually at least \( m \) processors as we assume that \( n \geq m \)). \( \square \)

Lemma 4 (Upper Bound): \( C_{max}^{LS} \leq (2 - \frac{1}{m}) C^{\ast}_{max} \).

**Proof.** Let \( J_k \) denote the job that completes last across all machines, say on machine \( P_i \). This job was assigned according to the List Scheduling procedure when machine \( P_i \) had the least load, so that \( C_{max}^{LS} = C_k = s_k + p_k \), where \( s_k \) is the starting time of job \( J_k \). This starting time \( s_k \) is equal to the amount of load on machine \( P_i \) when \( J_k \) was assigned to it. We now use ”\( \leq \)” relations: \( s_k \), the starting time of the job \( J_k \), cannot be larger than the sum of the processing times of all of the other jobs divided by the number of machines, \( m \).

\[
s_k \leq \frac{\sum_{j=1}^{n} p_j, j \neq k}{m} = \frac{\left(\sum_{j=1}^{n} p_j\right) - p_k}{m} = \frac{\sum_{j=1}^{n} p_j}{m} - \frac{p_k}{m}.
\]

So we come to

\[
C_{max}^{LS} = C_k = s_k + p_k \\
\leq \frac{\sum_{j=1}^{n} p_j}{m} - \frac{p_k}{m} + p_k \\
\leq \frac{\sum_{j=1}^{n} p_j}{m} + p_k \left(1 - \frac{1}{m}\right)
\]

Using the Lower Bounds (Lemma 1 and Lemma 2), it follows that

\[
C_{max}^{LS} \leq C_{max}^{\ast} + C_{max}^{\ast} \left(1 - \frac{1}{m}\right) \\
\leq \left(2 - \frac{1}{m}\right) C_{max}^{\ast}
\]

\( \square \)
Lemma 5 (Tightness of the Upper Bound): There are problem instances for arbitrary values of \( m \) where the upper bound \( (2 - \frac{1}{m}) C^\ast_{\text{max}} \) derived in the previous Lemma is tight.

Proof. To achieve a tight worst-case for the makespan by List Scheduling with \( m \) processors, we choose \( n = m(m - 1) + 1 \) jobs as follows: One job ("‘Type A’") has processing time \( m \), \( m \cdot (m - 1) \) jobs ("‘Type B’") have processing times 1. List Schedule: "‘Type B’"-jobs followed by "‘Type A’"-job. Optimum Schedule: "‘Type A’"-job followed by "‘Type B’"-jobs.

Example: The tight worst-case example for \( m = 3 \) processors is displayed in Figure 1.

![Figure 1: Tight worst-case example for \( m = 3 \) processors](image)

3 LPT

Let LPT be the Longest Processing Time first algorithm: Start sorting the jobs in non-increasing order. Apply then List Scheduling on this sorted set of jobs.

Lemma 6 (Upper Bound): \( C^{\text{LPT}}_{\text{max}} \leq \sum_{j=1}^{n} p_j \).

Proof. This bound is obviously true as LPT would never use just one processor (actually at least \( m \) processors as we assume that \( n \geq m \)). \( \square \)

In order to prove the tight upper bound \( C^{\text{LPT}}_{\text{max}} \leq (4/3 - 1/(3m))C^\ast_{\text{max}} \), we start to prove four Corollaries. Let \( J_k \) (with processing time \( p_k \)) be again that job that determines the makespan (in this case of the LPT-schedule), i.e. \( C_k = C^{\text{LPT}}_{\text{max}} \). Let \( T_1 \) be the original job set and \( T_2 \) be the truncated job set where all jobs that come after job \( J_k \) in the LPT-order are deleted from the original job set. Doing this, we assure that \( J_k \) is actually not only the last job that is scheduled (in the LPT-order) but also that it is the smallest job.

Corollary 1

\[
\frac{C^{\text{LPT}}_{\text{max}}(T_1)}{C^\ast_{\text{max}}(T_1)} \leq \frac{C^{\text{LPT}}_{\text{max}}(T_2)}{C^\ast_{\text{max}}(T_2)}
\]
Proof. The makespans of the original LPT list and the truncated list are the same (note that we just take out those jobs that come after \( J_k \) and have no influence on the makespan). The optimum makespan can definitely not get larger when we have to schedule less jobs. As the numerators remain the same and the denominator of the \( T_2 \)-schedule is not larger than the denominator of the \( T_1 \)-schedule, the result follows immediately. 

Corollary 2

\[
C_{\text{max}}^{LPT} \leq C_{\text{max}}^* + p_k \left( 1 - \frac{1}{m} \right)
\]

Proof. In a similar way as in the List Scheduling prove above, we come to:

\[
C_{\text{max}}^{LPT} = C_k = s_k + p_k \\
\leq \frac{\sum_{j=1}^{n} p_j}{m} - \frac{p_k}{m} + p_k \\
\leq \frac{\sum_{j=1}^{n} p_j}{m} + p_k \left( 1 - \frac{1}{m} \right) \\
\leq C_{\text{max}}^* + p_k \left( 1 - \frac{1}{m} \right)
\]

Corollary 3 If \( C_{\text{max}}^* < 3 \cdot p_k \), then each processor has at most two jobs.

Proof. We show: If there is a processor that has at least three jobs, then \( C_{\text{max}}^* \geq 3 \cdot p_k \). This is obviously true as all other jobs have processing times that are greater than or equal to \( p_k \). 

Corollary 4 If each processor has at most two jobs, then LPT is always optimal.

Proof. It is obvious that for \( n \leq m + 1 \) LPT must be optimal (if \( n = m + 1 \), then the two smallest jobs are scheduled on \( P_m \), and in all other cases there is only one job scheduled on each processor). Let’s look at the case \( n = 2m \). In that case, each processor must have exactly two jobs (as the current finishing times of the processors are non-increasing, we would never let one of the processors that have less load than \( P_1 \) just schedule one job). We assume w.l.o.g. that on processor \( P_1 \) there is the largest job \( J_1 \) with length \( p_1 \) scheduled. Then it is clear that in an optimal schedule \( J_1 \) has to be combined with \( J_k \) (remember: \( J_k \) is the smallest job that is scheduled and it determines the makespan, i.e. \( C_k = C_{\text{max}}^{LPT} \)). If we would interchange \( J_k \) with another job, then the makespan would only get larger and will not remain optimal. The same is true if we have \( n < 2m \) jobs. In this case, we combine \( J_k \) always with \( J_{2m-n+1} \). And again, interchanging would only increase the makespan. 

4
**Example:** The Example in Figure 2 shows $m = 3$ processors with $n = 2m = 6$ and $n = 2m - 1 = 5$ jobs.

![Figure 2: Example for $m = 3$ processors with $n = 2m = 6$ and $n = 2m - 1 = 5$ jobs](image)

**Lemma 7 (Upper Bound):** $C_{LPT}^{max} \leq \left(\frac{4}{3} - \frac{1}{3m}\right)C_{max}^*$.

**Proof (by contradiction).** Assume that there exists a job set $T_1 = \{J_1, \ldots, J_n\}$ that contradicts Lemma 7, i.e. $C_{LPT}^{max}(T_1) > \left(\frac{4}{3} - \frac{1}{3m}\right)C_{max}^*(T_1)$. Let $T_2$ be the truncated job set where all jobs that come after job $J_k$ in the LPT-order are deleted from the original job set.

**Conclusion 1:** It follows from

$$C_{LPT}^{max}(T_1) > \left(\frac{4}{3} - \frac{1}{3m}\right)C_{max}^*(T_1)$$

that (because of $\frac{1}{3m} < \frac{1}{3}$)

$$C_{LPT}^{max}(T_1) > C_{max}^*(T_1).$$

As the optimum makespan of the truncated job set cannot be larger than the optimal makespan of the original job set, i.e.

$$C_{max}^*(T_2) \leq C_{max}^*(T_1),$$

we come to

$$C_{LPT}^{max}(T_1) > C_{max}^*(T_2).$$

**Conclusion 2:** It follows from

$$C_{LPT}^{max}(T_1) > \left(\frac{4}{3} - \frac{1}{3m}\right)C_{max}^*(T_1)$$

that

$$\frac{4}{3} - \frac{1}{3m} < \frac{C_{LPT}^{max}(T_1)}{C_{max}^*(T_1)}.$$
and so (by Corollary (a))
\[
\frac{4}{3} - \frac{1}{3m} < \frac{C_{\text{LPT}}^{\text{max}}(T_2)}{C_{\text{max}}^{\star}(T_2)}
\]
and (by Corollary (b))
\[
\frac{4}{3} - \frac{1}{3m} < \frac{C_{\text{max}}^{\star}(T_2) + pk \left(1 - \frac{1}{m}\right)}{C_{\text{max}}^{\star}(T_2)}
\]
or
\[
\frac{4}{3} - \frac{1}{3m} < 1 + \frac{pk \left(1 - \frac{1}{m}\right)}{C_{\text{max}}^{\star}(T_2)}.
\]
This yields to
\[
\frac{1}{3} - \frac{1}{3m} < \frac{pk \left(1 - \frac{1}{m}\right)}{C_{\text{max}}^{\star}(T_2)}
\]
or
\[
\frac{m - 1}{3m} < \frac{pk \left(1 - \frac{1}{m}\right)}{C_{\text{max}}^{\star}(T_2)} \cdot \frac{m - 1}{m}
\]
and so to
\[
C_{\text{max}}^{\star}(T_2) < 3 \cdot pk.
\]
It follows from Corollary (c) that in this case each processor must have at most two jobs, and from Corollary (d) it follows that if each processor has at most two jobs, then LPT is always optimal, i.e.
\[
C_{\text{max}}^{\text{LPT}}(T_1) = C_{\text{max}}^{\star}(T_2).
\]
But this is a contradiction to Conclusion 1 and therefore the assumption that there exists a job set \(T_1 = \{J_1, \ldots, J_n\}\) that contradicts Lemma 7, i.e. \(C_{\text{max}}^{\text{LPT}}(T_1) > \left(\frac{4}{3} - \frac{1}{3m}\right)C_{\text{max}}^{\star}(T_1)\), cannot be true.

Lemma 8 (Tightness of the Upper Bound): There are problem instances for arbitrary values of \(m\) where the upper bound \(\left(\frac{4}{3} - \frac{1}{3m}\right)C_{\text{max}}^{\star}\) derived in the previous Lemma is tight.

Proof. To achieve a tight worst-case for the makespan by LPT Scheduling with \(m\) processors, we choose \(n = 2m + 1\) jobs as follows: 2 jobs of size \(2m - 1\), 2 jobs of size \(2m - 2\), \ldots, 2 jobs of size \(m + 1\), 3 jobs of size \(m\).

Example: The Example in Figure 3 shows a tight worst-case example for \(m = 3\) processors.
4 Complexity

Lemma 1 (Complexity): $P_2 || C_{\text{max}}$ is NP-hard.

Proof. (From this result it follows that also $Pm || C_{\text{max}}$ for an arbitrary $m \geq 2$ must be NP-hard.) We prove this by reducing PARTITION to the decision version of the SCHEDULING problem $P_2 || C_{\text{max}}$.

PARTITION: Given a list of $n$ positive integers $s_1, s_2, \ldots, s_n$ and a value $b = \sum_{j=1}^{n} s_j / 2$, does there exist a subset $J \subset I = \{1, \ldots, n\}$ such that
\[ \sum_{j \in J} s_j = b = \sum_{j \notin J} s_j \]

Remark: Partition is NP-hard in the ordinary sense, i.e. the problem cannot be optimally solved by an algorithm with polynomial time complexity but with an algorithm of time complexity $O((n \cdot \max s_j)^k)$.

SCHEDULING: Given $n$ jobs with processing times $p_j$ where $j \in \{1, 2, \ldots, n\}$ and a number $k$, the decision version of the scheduling problem $P_2 || C_{\text{max}}$ is to check if there is a schedule with makespan not more than $k$.

Step 1: $P_2 || C_{\text{max}}$ is in NP
\[ s_k \leq \frac{\sum_{j \in I \setminus J} p_j}{m} \] If the schedule of the jobs for each of the 2 machines is given, it can be verified in polynomial time that the completion times $C_1, \ldots, C_n$ of all jobs $J_1, \ldots, J_n$ are less than or equal to $T$. Thus, $P_2 || C_{\text{max}}$ belongs to NP.

Remark 1: Clearly, any $C_k$ has a binary encoding which is bounded by a polynomial in the input length of the problem.

Remark 2: Every decision problem solvable in polynomial time belongs to NP. If we have such a problem $P$ and an algorithm which calculates for each input $x$ the answer $h(x) \in \{\text{yes, no}\}$ in a polynomial number of steps, then this answer $h(x)$ may be used as a certificate. This certificate can be verified by the algorithm. Thus $P$ is also in NP which implies $P \subseteq \text{NP}$.

Step 2: PARTITION reduces to SCHEDULING: PARTITION $\propto$ SCHEDULING

Remark 3: For two decision problems $P$ and $Q$, we say that $P$ reduces to $Q$ (denoted $P \propto Q$) if there exists a polynomial-time computable function $g$ that transforms inputs for $P$ into inputs for $Q$ such that $x$ is a yes-input for $P$ if and only if $g(x)$ is a yes-input for $Q$. A yes-answer for a decision problem can be verified in polynomial time (this is not the case for the no-answer).
We are going to show that the PARTITION is reducible to SCHEDULING:

1. The input of the SCHEDULING problem can be computed in polynomial time given the input of the PARTITION problem:
We must transform in polynomial time the input for PARTITION into an instance of SCHEDULING such that there is a solution for PARTITION if and only if there is a schedule with \( C_{\text{max}} \leq k \) for a suitable value \( k \). This is easy: We just set \( k = b \) and define the SCHEDULING problem as follows: Consider the jobs \( J_j \) with \( p_j = s_j \) for \( j = 1, \ldots, n \). We choose \( k = b \) as the threshold for the corresponding decision problem.

2. a. If PARTITION has a solution, then the decision version of SCHEDULING has a solution:
If PARTITION has a solution, then there exists an index set \( J \subseteq \{1, \ldots, n\} \) such that \( \sum_{i \in J} s_i = b \). In this case the schedule \( p_{i \in J} \) on \( P_1 \) and \( p_{i \notin J} \) on \( P_2 \) solves the decision version of problem \( P_2 || C_{\text{max}} \).

2. b. If the decision version of SCHEDULING has a solution, then PARTITION has a solution:
If the decision version of SCHEDULING has a solution, then the jobs with processing times \( p_1, \ldots, p_n \) are scheduled on \( P_1 \) (all \( p_{i \in J} \) and \( P_2 \) (all \( p_{i \notin J} \)) such that the makespan is less than or equal to \( k \). In this case the PARTITION \( \sum_{j \in J} s_j = b = k = \sum_{j \in l \setminus J} s_j \) solves the PARTITION problem.

Since SCHEDULING is in NP, and since we can reduce the NP-complete problem PARTITION to SCHEDULING in polynomial time, SCHEDULING must also be NP-complete. \( \square \)