1 Introduction

Lucas: Intro is partly AI-generated, temporary, and for internal use only

Efficient resource allocation and task management are fundamental challenges in modern computational systems, particularly in High-Performance Computing (HPC). As these systems grow in scale and complexity, effective scheduling algorithms become increasingly critical. Scheduling problems, that address these challenges, form a cornerstone of computer science and operations research. The significance of scheduling extends beyond theoretical computer science and is vital in enhancing productivity, minimizing costs, and maximizing resource utilization across various sectors. This paper delves into a specific subset of scheduling problems known as rigid scheduling, which presents unique challenges and opportunities for optimization in the context of HPC environments.

In rigid scheduling, we consider a set of n tasks $\mathcal{T}$ to be executed on P processors. Each task $\mathcal{T}_i$ is characterized by its execution time t_i and processor requirement $p_i \in [1, P]$. The tasks are interconnected in a directed acyclic graph (DAG), where edges represent precedence constraints. The objective is to assign start times to tasks to minimize the completion time.

This paper explores various versions of rigid scheduling, including:

- Offline scheduling, where complete knowledge of the instance is available at the start.
- Online clairvoyant scheduling, where tasks are revealed dynamically as their predecessors complete.
- Online semi-clairvoyant scheduling, which adds uncertainty to task execution times.

We introduce novel notation and concepts, such as task criticality, power levels, and categories, to analyze and develop efficient scheduling algorithms. Our main contributions include the following.

- Two new scheduling algorithms, Batch for the online clairvoyant setting, and Batchest for the online semi-clairvoyant setting;
- An analysis of the performance of these algorithms in the worst case;
- Lower bounds of the best performance of any scheduling algorithm in the worst case, close to the performance of our algorithms.

The remainder of this paper is organized as follows. Section 2 presents key definitions and notations. Section 3 summarizes the results obtained in this work. Section 4 focuses on offline scheduling, showing that the current state of the art is near-optimal in the worst case. Section 5 focuses on online clairvoyant scheduling, where Batch is introduced, and Section 6 on online non-clairvoyant scheduling and introduces Batchest. Those three last sections explores different settings, however the analyses are interconnected and the sections are not independents.

Lucas: Also made me realize e (margin of error) and f^∞ both uses e, maybe to be changed. More generally, there is a macro for nearly everything; you are free to change them. I know you're more experienced for this

Lucas: For t_i, p_i, I thought about adding them but decided not to since it's more common and the table is long. But I agree we can add them (same reason for n, P). For $\mathcal{P}(\mathcal{T})$ I forgot to add it indeed ! I checked all the other lines it is correct, it is defined where the table indicates.

Hongyang: Ok, let's review this later. But we should add in notation for everything, even if they are common.

For simplicity, we use $\log(x)$ to denote $\log_2(x)$ throughout the following sections. Table 1 provides a non-exhaustive list of non-trivial or new notations and vocabulary used throughout the paper.

2 Definitions and Notations

2.1 Rigid DAG Scheduling Model

An instance $\mathcal{I}$ is defined with the following components:

- A fixed DAG of n tasks to be scheduled. We denote those tasks as $\{\mathcal{T}_i\}$ for $i \in [1, n]$
- P processors available for processing tasks

Table 1: Table of notations and vocabulary

Notation	**Name**	**Definition**
$\mathcal{I}$	Instance	Section 2.1
$\mathcal{P}(\mathcal{T})$	Predecessors of a task	Section 2.1
E	Margin of Error (global)	Section 2.2
$\textsc{Lb}(\mathcal{I})$	Lower bound	Section 2.3
$\mathcal{C}(\mathcal{I})$	Critical Path	Section 2.3
$\mathcal{A}(\mathcal{I})$	Area	Section 2.3
$T_{\textsc{Alg}}$	Completion time of $\textsc{Alg}$	Section 2.3
$T_{\textsc{Opt}}$	Optimal completion time	Section 2.3
$R_{\textsc{Lb}}(\textsc{Alg})$	Ratio to lower bound	Section 2.3
$R_{\textsc{Opt}}(\textsc{Alg})$	Ratio to optimal	Section 2.3
t^*	Estimated length	Section 2.2
e	Margin of Error	Section 2.2
s^∞	Criticality (start time)	Definition 3
f^∞	Criticality (end time)	Definition 3
χ	Power level	Definition 4
λ	Longitude	Definition 4
ζ	Category	Definition 4
L_ζ	Set of all tasks of category ζ in $\mathcal{I}$	Definition 5
L_ζ	Value of a category ζ	Definition 5
$\mathcal{T}^\zeta$	Category tries	Definition 10
$L_P^i(K)$	A chain of task	Definition 1
$X_P^i(K)$	A graph of task	Definition 2
$Y_P^i(K)$	A graph of task	Definition 7
	Greedy (for heuristic)	Definition 9
	Offline scheduling	Section 2.2
	Online clairvoyant scheduling	Section 2.2
	Online semi-clairvoyant scheduling	Section 2.2

- Each task $\mathcal{T}_i$ has an execution time t_i and requires $p_i \in [1, P]$ processors. Alternatively, in the rest of this paper, for all task-related notations, we may drop the index i if the context is clear (e.g., a task $\mathcal{T}$'s execution time is t and requires p processors).

- A directed arrow exists between two tasks $\mathcal{T}_a$ and $\mathcal{T}_b$ if and only if $\mathcal{T}_b$ cannot start executing until task $\mathcal{T}_a$ is completed. In this case we say $\mathcal{T}_a$ is a predecessor of $\mathcal{T}_b$, and we denote for each task $\mathcal{T}_i$ as $\mathcal{P}(\mathcal{T}_i)$ the set of predecessors of $\mathcal{T}_i$.

The goal is to design a scheduling algorithm that allocates to each task $\mathcal{T}_i$ a start time $s_i \geq 0$ such that:

- A task may start only if all its predecessors are completed. More precisely, for all pairs of tasks ($\mathcal{T}_a$,$\mathcal{T}_b$) such that there is an arrow (or precedence constraint) from $\mathcal{T}_a$ to $\mathcal{T}_b$, $s_b \geq s_a + t_a$

- At all times, no more than P processors may be used. More precisely, for any time x, if we denote $\mathcal{J}(x) \subset \mathcal{I}$ the subset of tasks running at time x, i.e., for which $s_i < x < s_i + t_i$, then we must have $\forall x \in \mathbb{R}, \sum_{\mathcal{T}_i \in \mathcal{J}(x)} p_i \leq P$

Our objective is to minimize the completion time of the DAG, or the makespan, defined as $T(\mathcal{I}) = \max_{i \in \mathcal{I}}(s_i + t_i)$.

2.2 Different Variations of The Problem

We consider the following versions of the rigid DAG scheduling problem:

- **Offline scheduling**: The scheduler has complete knowledge of the instance at the start of the execution.

- **Online clairvoyant scheduling**: Before a task is ready for processing, the scheduler is unaware of its existence. In other words, a task becomes known only when all its predecessors are completed. When that happens, its predecessors in the graph, its execution time t_i, and its processor allocation p_i also become known.

In this setting, we allow **task termination**, e.g., the scheduler can decide to terminate a task at time t before its completion and choose another starting time $s_i > t$ later. The time required to terminate a task and relaunch it is considered 0 in this work.

- **Online semi-clairvoyant scheduling**: This is similar to the online clairvoyant setting, except that the length of the task t_i is unknown, but we are provided with an **estimated length** t_i^* and a **margin of error** $e_i \geq 1$ such that $t_i \in \left[\frac{t_i^*}{e_i}, e_i t_i^*\right]$. We call the **global margin of error** $E = \max_i e_i$. Note that the estimate becomes useless when $e_i \to \infty$, and the problem becomes **online non-clairvoyant scheduling**.

  Hongyang: Does this semi-clairvoyant setting allow task termination as well?

  Lucas: Yes, and although in clairvoyant settings the main version of the algorithm is without interruption, here it is required. Maybe we could specify no checkpoints?

  Hongyang: Since no task termination is needed for clairvoyant setting, maybe we can move the blue sentence to this section, and say only in the semi-clairvoyant setting we require task termination. I don't think we need to mention checkpointing.

  Lucas: The algorithm discussed in Section 5.2, Batch, doesn't require task termination. Although it's competitive ratio is near-optimal, in practice it is not an efficient algorithm. In Section 5.5 I discussed heuristics that could be used in practice, and most of them require task termination. If we don't do experiments, Section 5.5 is not really needed though, let's discuss it after as well (Section 5=clairvoyant scheduling)

2.3 Makespan Lower Bound and Worst-Case Ratios

Given an instance $\mathcal{I}$, a well-known lower bound on the makespan is

$$\textsc{Lb}(\mathcal{I}) = \max\left(\frac{\mathcal{A}(\mathcal{I})}{P}, \mathcal{C}(\mathcal{I})\right) \tag{1}$$

where

- $\mathcal{A}(\mathcal{I}) = \sum_{\mathcal{T}_i \in \mathcal{I}} t_i p_i$ is the **area** of the instance, therefore $\frac{\mathcal{A}(\mathcal{I})}{P}$ corresponds to the total execution time if we could always use all P processors
- $\mathcal{C}(\mathcal{I})$ is the **critical-path length**, which corresponds to the total execution time if we had an infinite number of processors and processed everything as soon as possible (ASAP).

Given an algorithm Alg and an instance $\mathcal{I}$, we denote $T_{\textsc{Alg}}(\mathcal{I})$ as the completion time of the schedule under Alg and $T_{\textsc{Opt}}(\mathcal{I})$ the shortest possible completion time of any schedule.

For a given algorithm Alg, we define:

- $R_{\textsc{Lb}}(\textsc{Alg}) = \sup_{\mathcal{I}}\left(\frac{T_{\textsc{Alg}}(\mathcal{I})}{\textsc{Lb}(\mathcal{I})}\right)$ as the worst-case ratio between the makespan for any instance $\mathcal{I}$ and the lower bound $\textsc{Lb}(\mathcal{I})$ for the same instance.
- $R_{\textsc{Opt}}(\textsc{Alg}) = \sup_{\mathcal{I}}\left(\frac{T_{\textsc{Alg}}(\mathcal{I})}{T_{\textsc{Opt}}(\mathcal{I})}\right)$ as the worst-case ratio between the makespan for any instance $\mathcal{I}$ and the optimal makespan $T_{\textsc{Opt}}(\mathcal{I})$.

We call the second ratio above the **approximation ratio** in the offline setting or the **competitive ratio** in the online settings.

Note that we clearly have $R_{\textsc{Lb}}(\textsc{Alg}) \geq R_{\textsc{Opt}}(\textsc{Alg})$, since $T_{\textsc{Opt}}(\mathcal{I}) \geq \textsc{Lb}(\mathcal{I})$ for any instance $\mathcal{I}$. Thus, the first ratio provides an even looser measure of the algorithm's performance than the approximation/competitive ratio. However, $R_{\textsc{Lb}}(\textsc{Alg})$ is used in most theoretical results about the problem because determining $T_{\textsc{Opt}}(\mathcal{I})$ for any given instance $\mathcal{I}$ is NP-complete. In contrast, $\textsc{Lb}(\mathcal{I})$ can be easily computed.

In some cases, when the size of the instances is unbounded, we can have $R_{\textsc{Lb}}(\textsc{Alg}) = \infty$ or $R_{\textsc{Opt}}(\textsc{Alg}) = \infty$. For this reason, we denote:

- $\{\mathcal{I}_n\}$ as the set of all instances where the graph has n tasks.
- $\{\mathcal{I}^P\}$ as the set of all instances where the system has P processors.

Then, for a given number of tasks n, we define $R_{\text{LB}}(\textsc{Alg}_n) = \sup_{\mathcal{I}\in\{\mathcal{I}_n\}}\left(\frac{T_{\textsc{Alg}}(\mathcal{I})}{\text{LB}(\mathcal{I})}\right)$. This represents the worst possible ratio between an algorithm's execution time and the lower bound for instances with n tasks. Similarly, we define $R_{\textsc{Opt}}(\textsc{Alg}_n) = \sup_{\mathcal{I}\in\{\mathcal{I}_n\}}\left(\frac{T_{\textsc{Alg}}(\mathcal{I})}{T_{\textsc{Opt}}(\mathcal{I})}\right)$ and for a given number of processors P, $R_{\text{LB}}(\textsc{Alg}^P) = \sup_{\mathcal{I}\in\{\mathcal{I}^P\}}\left(\frac{T_{\textsc{Alg}}(\mathcal{I})}{\text{LB}(\mathcal{I})}\right)$ and $R_{\textsc{Opt}}(\textsc{Alg}^P) = \sup_{\mathcal{I}\in\{\mathcal{I}^P\}}\left(\frac{T_{\textsc{Alg}}(\mathcal{I})}{T_{\textsc{Opt}}(\mathcal{I})}\right)$.

3 Summary of Results

In this work, we aim to design algorithms that minimize $R_{\text{LB}}(\textsc{Alg}_n)$. We also provide lower bounds on the best possible values obtainable for $R_{\text{LB}}(\textsc{Alg}_n)$ and $R_{\text{LB}}(\textsc{Alg}^P)$ in the offline setting, which also applies to the online settings, as well as lower bounds for $R_{\textsc{Opt}}(\textsc{Alg}_n)$ and $R_{\textsc{Opt}}(\textsc{Alg}^P)$ in the online settings.

Here is a summary of our results:

Lucas: You may ignore result (1) and (5) if you find it confusing, the main focus of this work is studying with n.It was just to highlight the fact that we may always have terrible processor utilization (even when the system is full of tasks awaiting) in rigid scheduling

- In the **offline setting**, for any $P > 0$ and $\mu > 0$, there exists an instance $\mathcal{I} \in \{\mathcal{I}^P\}$ whose optimal completion time satisfies $T_{\textsc{Opt}}(\mathcal{I}) > (P - \mu)\text{LB}(\mathcal{I})$. Therefore $\forall P > 0, \forall \textsc{Alg}, R_{\text{LB}}(\textsc{Alg}^P) \geq P$. (We actually have an equality if $\textsc{Alg}$ never leave the platform empty) (1)

- In the **offline setting**, for any $N > 0$, there exists an $n > N$ and an instance $\mathcal{I} \in \{\mathcal{I}_n\}$ whose optimal completion time satisfies $T_{\textsc{Opt}}(\mathcal{I}) > \frac{\log(n)}{2}\text{LB}(\mathcal{I})$. Therefore $\forall N > 0, \exists n > N, \forall \textsc{Alg}, R_{\text{LB}}(\textsc{Alg}_n) > \frac{\log(n)}{2}$. (2)

  Hongyang: Does this lower bound apply when tasks have bounded length as well?

  Lucas: No, and it would contradict our $6 + \log(M/m)$ result :-)

  Hongyang: Is there any upper bound result for this offline setting? I later see in your remarks that a cited paper has upper bound $2\log(n) + 2$, which is even worst than our online upper bound of $\log(n) + 3$. Maybe their analysis isn't very tight?

  Lucas: In fact, the problem they studied is slightly more general than ours, so my comment shouldn't remain like this. I believe their algorithm in our offline setting can be shown to be always within a factor $(\log(n) + 2)$ from the lower bound. It is actually quite similar to $\textsc{Batch}$ although the approach is different. Also it is not better than $\log(n)/2$-approx (compared to the optimal time)

- In the **online clairvoyant setting**, there exists an algorithm $\textsc{Batch}$ running in (low) polynomial time that never terminates tasks and satisfies $\forall n > 0, R_{\textsc{Opt}}(\textsc{Batch}_n) \leq R_{\text{LB}}(\textsc{Batch}_n) \leq 3 + \log(n)$. In other words, $\textsc{Batch}$ is $(3 + \log(n))$-competitive. Moreover, for any instance $\mathcal{I}$ such that the length of the task is bounded, i.e., $\forall i, m \leq t_i \leq M$, $\frac{T_{\textsc{Batch}}(\mathcal{I})}{\text{LB}(\mathcal{I})} \leq 6 + \log\left(\frac{M}{m}\right)$. (3)

- In the **online clairvoyant setting**, $\forall N > 0, \exists n > N, \forall \textsc{Alg}, R_{\textsc{Opt}}(\textsc{Alg}_n) > \frac{\log(n)-\log(\log(n))}{4}$. In other words, no algorithm (polynomial or not) may be $\left(\frac{\log(n)-\log(\log(n))}{4}\right)$-competitive for all n large enough (4)

- In the **online clairvoyant setting**, $\forall P > 0, \forall \textsc{Alg}, R_{\textsc{Opt}}(\textsc{Alg}^P) \geq \frac{P}{2}$. In other words, for any $\mu > 0$, no algorithm (polynomial or not) may be $\left(\frac{P}{2} - \mu\right)$-competitive for any P (5)

- In the **online semi-clairvoyant setting**, there exists an algorithm $\textsc{Batchest}$ running in (low) polynomial time that satisfies $\forall \mathcal{I}, \frac{T_{\textsc{Batchest}}(\mathcal{I})}{\text{LB}(\mathcal{I})} \leq 8 + \log(n) + \log(\log(E) + 1)$. In this bound, E is the largest margin of error from our estimator to the length of tasks. Moreover, for any instance $\mathcal{I}$ where the estimate of the tasks is bounded, i.e., $\forall i, m \leq \frac{t_i^*}{e_i} \leq t_i \leq t_i^* e_i \leq M$, then $\frac{T_{\textsc{Batchest}}(\mathcal{I})}{\text{LB}(\mathcal{I})} \leq 10 + \log\left(\frac{M}{m}\right)$. (6)

Results (2) and (3) demonstrate that in online clairvoyant settings, it is possible to have a polynomial algorithm that is nearly as close to the lower bound in the worst case as any algorithm with perfect knowledge of the instance and infinite time at its disposal for decision-making. This implies that a rigid instance can be impossible to schedule efficiently regardless of the instance's knowledge or the algorithm's quality. This can even be true for instances with a small longest path, and such examples are introduced in Section 4.

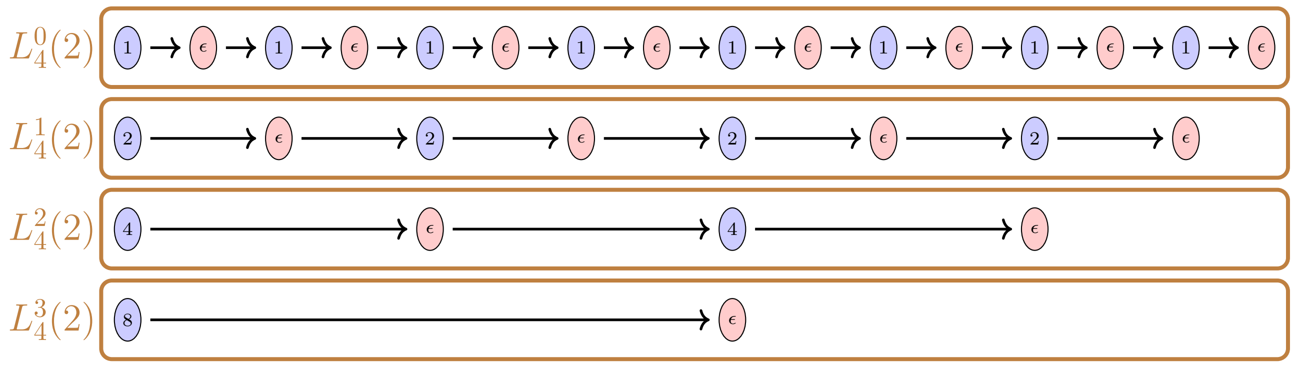


Figure 1: $X_4(2)$, minimum time of processing is $20 + 15\epsilon$

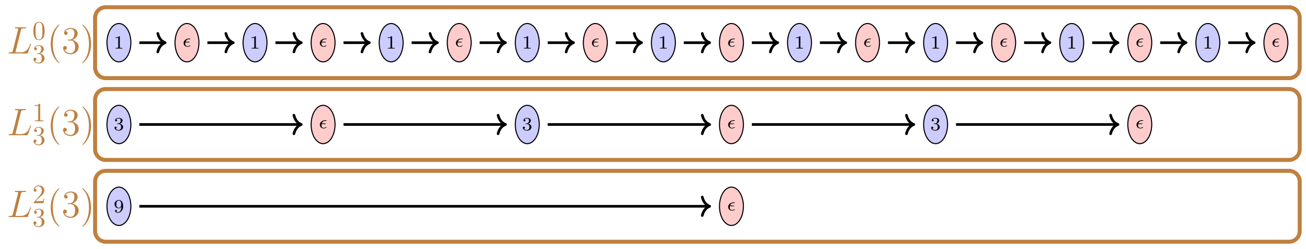


Figure 2: $X_3(3)$, minimum time of processing is $21 + 13\epsilon$

Results (3), (4), and (5) show that in online clairvoyant settings, we can always be penalized by our decisions and end up far from the best possible time. Furthermore, our algorithm almost achieves the best competitive ratio for a given n. The presentation of Batch and its analysis are provided in Section 5, as well as some extensions for practical use.

Finally, result (6) shows that even if the length of the task is not known, we can still design an algorithm such that $\log(n)$ is the dominating factor of the competitive ratio. The ratio is highly resilient to the margin of error since the additional term is in $\log(\log(E)+1)$ (nearly constant). This extension for semi-clairvoyant settings, named Batchest, is given in Section 6.

4 Offline Scheduling

In the next two theorems, we show:

- The lower bound may not be a good estimate of the optimal processing time
- There exist instances where any schedule has a very poor processor utilization, even when the longest path is very short (and therefore, there are always many tasks ready and waiting to be scheduled). This can be avoided if we are allowed to reduce the number of processors requested, assuming we don't have super-linear speedup (See Section 7)

More precisely, in this section, for any $\mu > 0$ and we will show the following:

- For any $P > 0$, there exists an instance $\mathcal{I}$ whose optimal completion time satisfies $T_{\text{Opt}}(\mathcal{I}) > (P - \mu)\text{Lb}(\mathcal{I})$
- For any $N > 0$, there exists an instance $\mathcal{I}'$ with $n > N$ tasks whose optimal completion time satisfies $T_{\text{Opt}}(\mathcal{I}') > \frac{\log(n)}{2}\text{Lb}(\mathcal{I}')$.

The rest of the section holds for any given $\mu > 0$ and $P > 0$. We will also use $\epsilon > 0$ and $K > 1$, which can be arbitrary in Definition 1, Definition 2, Equation 2 and Lemma 2. We will then use a special value of ϵ and K given μ and P to show the results given above in Theorem 1 and Theorem 2

Definition 1. *For $i \in [0, P-1]$, let $L_P^i(K)$ denote a linear chain consisting of $2K^{P-i-1}$ tasks, alternating a task of length K^i that requires a single processor and a task of length ϵ that requires all P processors.*

Definition 2. *Let $X_P(K)$ denote a graph of tasks that contains P distinct linear chains $L_P^i(K)$ for each $i \in [0, P-1]$.*

$X_4(2)$ is illustrated in Figure 1 and $X_3(3)$ is illustrated in Figure 2, where blue tasks require a single processor and red tasks require all P processors.

Lemma 1. *The lower bound of $X_P(K)$, which is the maximum of the critical path and the area, satisfies:*

$$\textsc{Lb}(X_P(K)) < K^{P-1} + PK^P\epsilon \tag{2}$$

Proof. For all i, the total area of a chain $L_P^i(K)$ is $\mathcal{A}(L_P^i(K)) = K^{P-i-1}(K^i + P\epsilon) = K^{P-1} + PK^{P-i-1}\epsilon$, so the area of $X_P(K)$ which is the sum of the area of all $L_P^i(K)$ can be computed as:

$$\begin{aligned}\mathcal{A}(X_P(K)) &= \sum_{i=0}^{P-1} \mathcal{A}(L_P^i(K)) \\ &= \sum_{i=0}^{P-1} (K^{P-1} + PK^{P-i-1}\epsilon) \\ &< P(K^{P-1} + PK^P\epsilon)\end{aligned}$$

Furthermore, the length of each chain $L_P^i(K)$ is $\mathcal{C}(L_P^i(K)) = K^{P-i-1}(K^i+\epsilon) = K^{P-1}+K^{P-i-1}\epsilon < K^{P-1}+K^P\epsilon$, therefore the longest path of $X_P(K)$ is $\mathcal{C}(X_P(K)) = \max_{0\leq i \leq P-1} \mathcal{C}(L_P^i(K)) < K^{P-1} + K^P\epsilon$.

We finally get $\textsc{Lb}(X_P(K)) = \max\left(\frac{\mathcal{A}(X_P(K))}{P}, \mathcal{C}(X_P(K))\right) < K^{P-1} + PK^P\epsilon$. □

Lemma 2. *The optimal time of processing of $X_P(K)$ satisfies:*

$$T_{\text{OPT}}(X_P(K)) > PK^{P-1} - (P-1)K^{P-2} \tag{3}$$

Proof. Because the tasks of length ϵ require all processors, it is impossible to process multiple blue tasks in a chain $L_P^i(K)$ during the processing of a single blue task in chain $L_P^j(K)$, for $j > i$. For example, in Figure 1, when processing the task of length 8, one can process at most one task of length 4, one task of length 2, and one task of length 1.

In the following, we are given an optimal schedule. At all times, either red or blue tasks must be processed; otherwise, the platform would do nothing, and time would be lost, which contradicts the schedule's optimality. Because red tasks require all processors, it is impossible to process red and blue tasks simultaneously. Therefore, the schedule alternates processing red tasks and blue tasks.

For this reason, we may define a segment as an interval of time $[a, b]$ in which only blue tasks are processed, and such that a red task was processed right before time a (or $a = 0$), and a red task will be processed right after time b (the last task executed is always red). There has to be at least K^{P-1} distinct segments because it is the number of blue tasks in $L_P^0(K)$, and a red task separates each of them. We denote as $t_1, t_2, \ldots, t_{K^{P-1}}$ the length of the K^{P-1} longest segments, arranged in decreasing order. We then have $T_{\text{OPT}}(X_P(K)) > \sum_{i=1}^{K^{P-1}} t_i$.

Clearly, $t_1 \geq K^{P-1}$ since the longest blue task must be processed without interruption. We also have $\forall i \leq K, t_i \geq K^{P-2}$ since at least K segments must be longer than the time required to process a task in $L_P^{P-2}(K)$, which contains K tasks of length K^{P-2} separated by red tasks (by Definition 1). Similarly, $\forall j \leq P-1, \forall i \leq K^{P-j-1}, t_i \geq K^j$.

Therefore,

$$\begin{aligned}
T_{\text{OPT}}(X_P(K)) &> \sum_{i=1}^{K^{P-1}} t_i \\
&= t_1 + \sum_{j=0}^{P-2} \sum_{i=K^{P-j-2}+1}^{K^{P-j-1}} t_i \\
&\geq t_1 + \sum_{j=0}^{P-2} \sum_{i=K^{P-j-2}+1}^{K^{P-j-1}} K^j \\
&= K^{P-1} + \sum_{j=0}^{P-2} (K^{P-j-1} - K^{P-j-2})K^j \\
&= K^{P-1} + \sum_{j=0}^{P-2} (K^{P-1} - K^{P-2}) \\
&= PK^{P-1} - (P-1)K^{P-2}
\end{aligned}$$

□

Theorem 1. *In the **offline setting**, for any $P > 0$ and $\mu > 0$, there exists an instance $\mathcal{I} \in \{\mathcal{I}^P\}$ whose optimal completion time satisfies $T_{\text{OPT}}(\mathcal{I}) > (P - \mu)\text{LB}(\mathcal{I})$.*

Proof. Using Equation 2 and Lemma 2, and choosing $K > \frac{2(P-1)}{\mu}$ and $0 < \epsilon < \frac{\mu}{2P^2K}$, we derive:

$$\begin{aligned}
\frac{T_{\text{OPT}}(X_P(K))}{\text{LB}(X_P(K))} &> \frac{PK^{P-1} - (P-1)K^{P-2}}{K^{P-1} + PK^P\epsilon} \\
&= \frac{P - \frac{P-1}{K}}{1 + PK\epsilon} \\
&> \frac{P - \frac{\mu}{2}}{1 + \frac{\mu}{2P}} \\
&= \frac{P\left(1 + \frac{\mu}{2P}\right) - \frac{\mu}{2} - \frac{\mu}{2}}{1 + \frac{\mu}{2P}} \\
&> P - \mu
\end{aligned}$$

□

This theorem shows that even when the critical path length is less than the area, it is possible to design an instance where no more than $1 + \mu$ processors may be utilized on average for μ arbitrarily close to 0 and P arbitrarily large. It suggests that the rigid scheduling framework is not very resilient, and one should build schedulers that allow a certain flexibility in processor allocation (e.g., moldable scheduling). This is stated in the following corollary:

Corollary 1. $\forall P > 0, \min_{\text{ALG}}(R_{\text{LB}}(\text{ALG}^P)) = P$.

Proof. Any algorithm ALG that never leaves all processors idle has to verify, for any instance $\mathcal{I}$, $T_{\text{ALG}}(\mathcal{I}) \leq \mathcal{A}(\mathcal{I}) \leq P\text{LB}(\mathcal{I})$, therefore we have $\exists \text{ALG}, R_{\text{LB}}(\text{ALG}^P)) = P$.

Hongyang: I need to discuss this proof with Lucas in a meeting. Maybe it should be stated in the same style as the next corollary.

The other inequality directly comes from Theorem 1: $\forall \mu > 0, \forall P > 0, \forall \text{ALG}, \exists \mathcal{I} \in \{\mathcal{I}\}^P, \frac{T_{\text{ALG}}(\mathcal{I})}{\text{LB}(\mathcal{I})} \geq \frac{T_{\text{OPT}}(\mathcal{I})}{\text{LB}(\mathcal{I})} \geq P - \mu$, therefore $\forall \text{ALG}, R_{\text{LB}}(\text{ALG}^P) = \sup_{\mathcal{I} \in \{\mathcal{I}^P\}} \left(\frac{T_{\text{ALG}}(\mathcal{I})}{T_{\text{OPT}}(\mathcal{I})}\right) \geq P$, hence the minimum is well defined and equal to P. □

Theorem 2. *In the **offline setting**, for any $N > 0$, there exists an $n > N$ and an instance $\mathcal{I} \in \{\mathcal{I}_n\}$ whose optimal completion time satisfies $T_{\text{OPT}}(\mathcal{I}) > \frac{\log(n)}{2}\text{LB}(\mathcal{I})$.*

Proof. The total number of tasks in $X_P(K)$, by Definition 1 and Definition 2, is $\sum_{i=0}^{P-1} 2K^{P-i-1} = 2\sum_{j=0}^{P-1} K^j = 2\frac{K^P-1}{K-1}$. With $K=2$, we find $n = 2^{P+1}-2$ and $P = \log(n+2)-1$, so we just need to choose $P > \log(N+2)-1$, to have $n > N$. Then we use Equation 2 and Lemma 2, with $0 < \epsilon < \frac{\log(n+2)-\log(n)}{2P\log(n+2)}$, and derive:

$$\begin{aligned}
\frac{T_{\text{OPT}}(X_P(K))}{\text{LB}(X_P(K))} &\geq \frac{PK^{P-1}-(P-1)K^{P-2}}{K^{P-1}+PK^P\epsilon} \\
&\geq \frac{P-\frac{P-1}{2}}{1+2P\epsilon} \\
&= \frac{P+1}{2+4P\epsilon} \\
&= \frac{\log(n+2)}{2+4P\epsilon} \\
&= \frac{\log(n+2)(1+2P\epsilon)}{2(1+2P\epsilon)} - \frac{2P\epsilon\log(n+2)}{2(1+2P\epsilon)} \\
&> \frac{\log(n+2)}{2} - \frac{2P\epsilon\log(n+2)}{2} \\
&> \frac{\log(n+2)}{2} - \frac{\log(n+2)-\log(n)}{2} \\
&= \frac{\log(n)}{2}
\end{aligned}$$

□

Corollary 2. $\forall N > 0, \exists n > N, \forall \text{ALG}, R_{\text{LB}}(\text{ALG}_n) > \frac{\log(n)}{2}$.

Proof. The inequality directly comes from Theorem 2: $\forall N > 0, \forall \text{ALG}, \exists n > N, \exists \mathcal{I} \in \{\mathcal{I}\}_n, \frac{T_{\text{ALG}}(\mathcal{I})}{\text{LB}(\mathcal{I})} \geq \frac{T_{\text{OPT}}(\mathcal{I})}{\text{LB}(\mathcal{I})} > \frac{\log(n)}{2}$, hence the result. □

Remark 1. *An algorithm presented here https://ieeexplore.ieee.org/document/8665753 can be adapted to our problem, and the resulting schedule's length is always within a factor* $2\log(n)+2$ *from the lower bound; thus, the margin for improvement is little at least when comparing to this bound. In the next section, we will show that even in online settings, we achieve a worst-case ratio* $\frac{T(\mathcal{I})}{\text{LB}(\mathcal{I})}$ *ratio relatively close to* $\frac{\log(n)}{2}$*, as we obtained* $\log(n)+3$.

5 Online clairvoyant scheduling (completion time known, graph unknown)

5.1 Definitions

5.1.1 Criticality and Category

Lucas: To change

Before presenting the algorithm, we will start with two definitions. They introduce a few attributes of a task within its instance $\mathcal{I}$. An example will illustrate these attributes at the end of the subsection.

Definition 3. *Given a task* $\mathcal{T}$ *of length* t*, we define its* ***earliest start time*** *as* s^∞*, which indicates the time the task would be launched in an ASAP schedule with an unlimited number of processors. It also represents the longest path length from the start of the graph to this task. Similarly, we define the* ***earliest finish time*** *of the task as* $f^\infty = s^\infty + t$*, indicating the time in which the task could be completed in an ASAP schedule. We further refer to* (s^∞, f^∞) *as the* ***criticality****, indicating the interval in which the task will be scheduled in an ASAP schedule. As* $\mathcal{C}(\mathcal{I})$ *corresponds to the longest path in the graph, or the minimum completion time with an unbounded number of processors of an ASAP schedule, we have* $\mathcal{C}(\mathcal{I}) = \max_j f_j^\infty$*.*

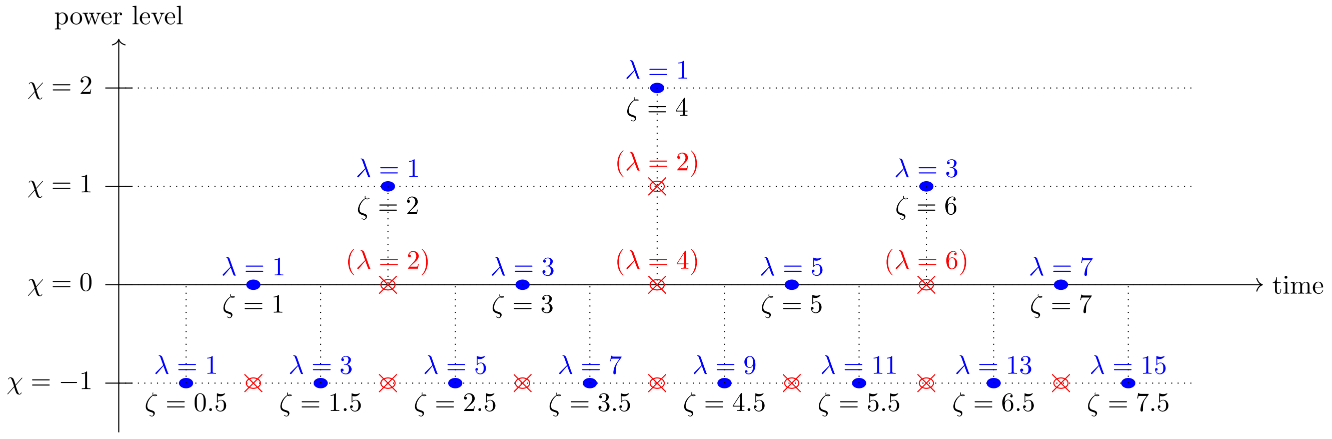


Figure 3: Graphical representation of some possible values of category ζ, power level χ and longitude λ.

Lemma 3. *Given a task $\mathcal{T}$ and its list of predecessors $\mathcal{P}(\mathcal{T})$, we have:*

$$s^\infty = \begin{cases} \max_{\mathcal{T}_j \in \mathcal{P}(\mathcal{T})} f_j^\infty, & \text{if } \mathcal{P}(\mathcal{T}) \neq \emptyset \\ 0, & \text{otherwise} \end{cases} \tag{4}$$

Proof. This is a direct induction: if the scheduler is ASAP and there are infinite processors, tasks will be launched as soon as their last predecessor completes. Alternatively, a direct induction shows that this represents the longest path length from the start of the graph to $\mathcal{T}$. □

Definition 4. *Given a task $\mathcal{T}$ and its criticality (s^∞, f^∞), we define its **power level** χ as:*

$$\chi = \max\{\chi \in \mathbb{Z} : \exists \lambda \in \mathbb{N}, s^\infty < \lambda 2^\chi < f^\infty\} \tag{5}$$

*where λ is the associated **longitude**. Given the power level and longitude of the task, we further define $\zeta = \lambda 2^\chi$ as the task's **category**. Lemma 4 shows that all these values are unique and well-defined for any given task.*

Figure 3 illustrates some values of λ, χ and ζ. Blue points represent category values with odd λ's, whereas red points represent category values with even λ's. The category of a task corresponds to the highest point within (s^∞, f^∞). We can see that such a point always corresponds to an odd λ (since red points always have a blue point directly above) and is unique since if two consecutive λ's are acceptable, one will be red and therefore have another point directly above. This is formally stated in the next lemma.

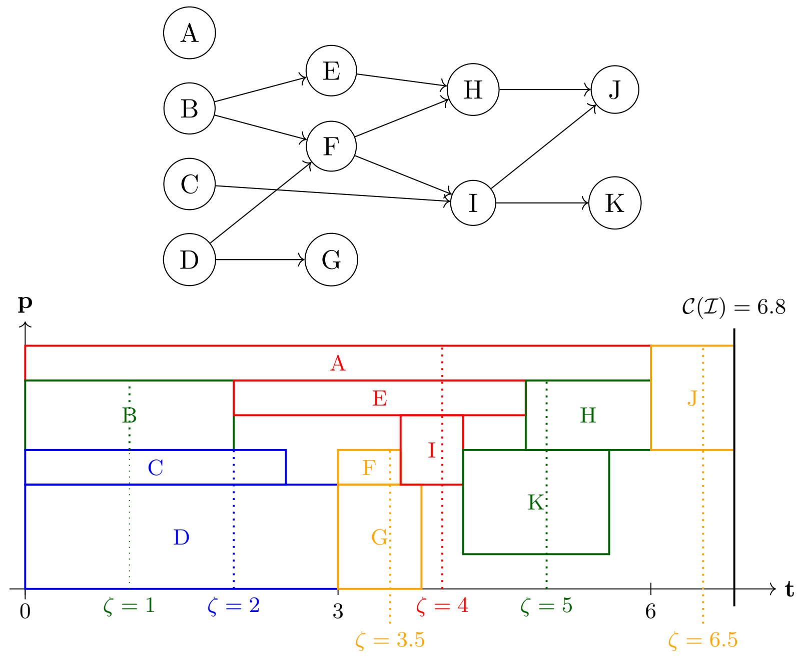
Lemma 4. *Given a task $\mathcal{T}$ of criticality (s^∞, f^∞), its power level χ, longitude λ and category ζ are all unique and well-defined. Furthermore, λ is odd and satisfies:*

$$(\lambda - 1)2^\chi \leq s^\infty < \zeta = \lambda 2^\chi < f^\infty \leq (\lambda + 1)2^\chi$$

Proof. Clearly, if $2^\chi \geq f^\infty$, then no $\lambda \geq 1$ may satisfy $\lambda 2^\chi < f^\infty$. Furthermore, if we take an $X \in \mathbb{Z}$ such that $2^X < f^\infty - s^\infty$, then with $\lambda = \left\lfloor \frac{s^\infty}{2^X} \right\rfloor$, $\lambda 2^X \leq s^\infty < (\lambda+1)2^X \leq s^\infty + 2^X < f^\infty$, which shows that $X \in \{\chi \in \mathbb{Z} : \exists \lambda \in \mathbb{N}^*, s^\infty < \lambda 2^\chi < f^\infty\}$. Therefore, this set is non-empty and has a finite number of values larger than X (as they have to be integers between X and $\log(f^\infty)$), so it admits a unique maximum and the power level χ is well-defined.

We then consider a λ that satisfies $s^\infty < \zeta = \lambda 2^\chi < f^\infty$. If λ was even, then $\left(\chi + 1, \frac{\lambda}{2}\right)$ would be an acceptable pair. In Figure 3, this corresponds to the point directly above. This contradicts the definition of χ; thus, λ must be odd. If we had $s^\infty < (\lambda - 1)2^\chi$, then because $(\lambda - 1)$ is even, the pair $(\chi + 1, \frac{\lambda - 1}{2})$ would be acceptable, contradicting the definition of χ. In Figure 3, this corresponds to the closest point in the top-left direction. Therefore, we must have $(\lambda - 1)2^\chi \leq s^\infty$, and similarly, $f^\infty \leq (\lambda + 1)2^\chi$, hence the uniqueness of λ, ζ and the result. □

Corollary 3. *All tasks in the same category share the same power level and longitude. Thus, in the following, we will refer to the power level χ and longitude λ as two attributes of a category ζ.*



Task	t	p	s^∞	f^∞	λ	χ	ζ
A	6	1	0	6	1	2	4
B	2	2	0	2	1	0	1
C	2.5	1	0	2.5	1	1	2
D	3	3	0	3	1	1	2
E	2.8	1	2	4.8	1	2	4
F	0.6	1	3	3.6	7	-1	3.5
G	0.8	3	3	3.8	7	-1	3.5
H	1.2	2	4.8	6	5	0	5
I	0.6	2	3.6	4.2	1	2	4
J	0.8	3	6	6.8	13	-1	6.5
K	1.4	3	4.2	5.6	5	0	5

Figure 4: Top-left: An example task graph consisting of 11 tasks; Right: The various attributes of each task in the task graph; Bottom-left: Graphical representation of the tasks' criticalities and categories, which can also be viewed as an ASAP schedule of the tasks with an unbounded number of processors.

Figure 4 illustrates all these definitions with an example. In the top left is a graph of tasks to be scheduled. For each task, its length t, processor allocation p and various attributes (i.e., criticality $[s^\infty, f^\infty]$, longitude λ, power level χ and category ζ) are given in the table on the right. The bottom left represents the tasks' criticalities, which can be viewed as an ASAP schedule with an unbounded number of processors. Categories are represented as vertical lines, and the color of each line (or category) represents its power level: by increasing order, yellow for $\chi = -1$, green for $\chi = 0$, blue for $\chi = 1$ and red for $\chi = 2$. Correspondingly, the color of each task also indicates its power level, and the category of the task is given by the vertical line of the highest power level that separates the task into two parts.

5.1.2 Category Length and L-matrix

We will define one last attribute for a category ζ, namely, its **length** $L_\zeta(\mathcal{C})$, which corresponds to an upper bound on the length of any task belonging to that category given any instance with critical path length $\mathcal{C}$. We point out that $L_\zeta(\mathcal{C})$ depends only on ζ and $\mathcal{C}$, not on a specific instance. To ease notation, we will denote it simply as L_ζ in the rest of the paper. Furthermore, this attribute will only be used in the analysis to derive bounds on our algorithm's makespan, while the algorithm does not explicitly use it.

Figure 5 illustrates the categories and their corresponding lengths, using the example presented in Figure 4. Based on Lemma 4, any task belonging to category $\zeta = \lambda 2^\chi$ has $s^\infty \in [(\lambda-1)2^\chi, \lambda 2^\chi)$ and $f^\infty \in (\lambda 2^\chi, (\lambda+1)2^\chi]$, therefore its length may not exceed $2^{\chi+1}$. For example, in category $\zeta = 5$ with power level $\chi = 0$ and $\lambda = 5$, any task (e.g., H and K) must have $s^\infty \in [4, 5)$ and $f^\infty \in (5, 6]$. Indeed, if $f^\infty > 6$, the task would be in a higher category (i.e., with higher χ), and if $f^\infty \leq 5$, the task would be in a lower category. Similarly, if $s^\infty < 4$, the task would be in a higher category (in this case, at the very top with tasks A, E, I), and if $s^\infty \geq 5$, the task would be in a lower category. Therefore, for all categories that do not reach $\mathcal{C}$ (i.e., the rightmost line in Figure 5), $2^{\chi+1}$ is a clear and tight upper bound. Further, this bound can be refined for categories that reach $\mathcal{C}$. For example, in the category to which task J belongs, 0.8 is a clear upper bound since $6 \leq s^\infty < f^\infty \leq \mathcal{C} = 6.8$. This represents a better bound than 1, obtained by considering its category alone (i.e., with power level $\chi = -1$, thus $2^{\chi+1} = 1$). The following definition and lemma state this formally:

Definition 5. *Given any instance with critical-path length $\mathcal{C}$ and a category ζ with power level χ and longitude λ, we define the* ***length*** *of the category as:*

$$L_\zeta = \min(2^{\chi+1}, \mathcal{C} - (\lambda - 1)2^\chi) \tag{6}$$

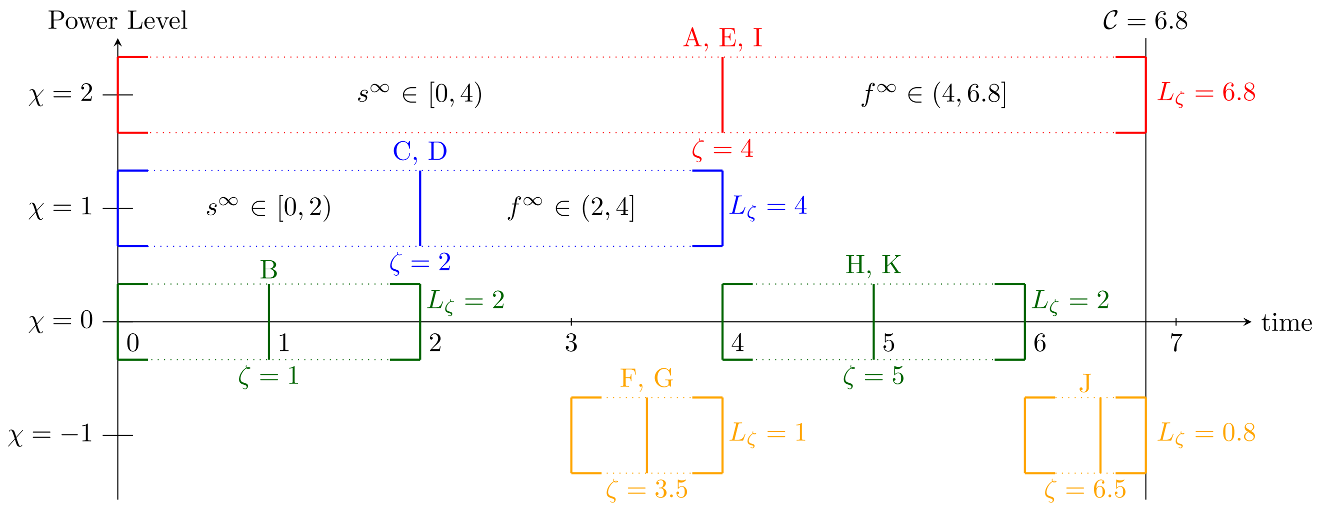


Figure 5: Graphical representation of the categories and their lengths for the example of Figure 4.

$$\begin{pmatrix} 6.8 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \cdots \\ 4 & 2.8 & 0 & 0 & 0 & 0 & 0 & 0 & \cdots \\ 2 & 2 & 2 & 0 & 0 & 0 & 0 & 0 & \cdots \\ 1 & 1 & 1 & 1 & 1 & 1 & 0.8 & 0 & \cdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots \end{pmatrix}$$

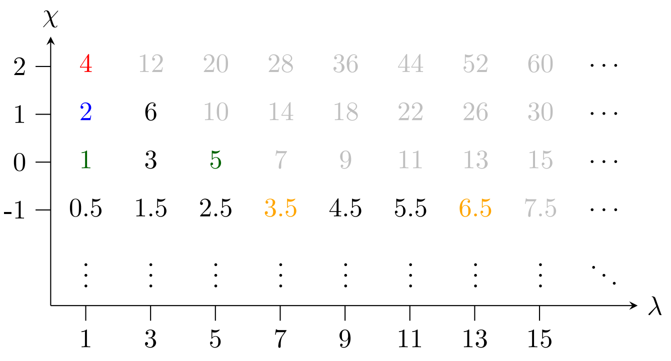


Figure 6: For any instance with $\mathcal{C} = 6.8$, left: L-matrix $\mathcal{L}(\mathcal{C})$; right: corresponding category values. The colors of the numbers correspond to the ones in the example of Figures 4 and 5. The numbers in black correspond to possible length and category values with $\mathcal{C} = 6.8$ but are not present in the example, while the numbers in gray correspond to impossible length and category values with $\mathcal{C} = 6.8$.

Lemma 5. *For any task $\mathcal{T}$ in an instance with critical path length $\mathcal{C}$, suppose its execution time is t, and it belongs to category ζ. Then, we have:*

$$t \leq L_\zeta \tag{7}$$

Proof. Based on Lemma 4, we have $s^\infty \geq (\lambda - 1)2^\chi$ and $f^\infty \leq (\lambda + 1)2^\chi$. We can derive $t = f^\infty - s^\infty \leq (\lambda + 1)2^\chi - (\lambda - 1)2^\chi = 2^{\chi+1}$. Therefore, if $2^{\chi+1} \leq \mathcal{C} - (\lambda - 1)2^\chi$, we have the result. If not, we use $\forall \mathcal{T}_j, f_j^\infty \leq \mathcal{C}$ (otherwise, an ASAP schedule with an unbounded number of processors would have a makespan strictly larger than the critical path length). Thus, we obtain $t = f^\infty - s^\infty \leq \mathcal{C} - (\lambda - 1)2^\chi \leq 2^{\chi+1}$, which concludes the proof. □

The category length is an excellent tool for analysis since our algorithm will split an instance into batches of tasks of identical categories, as shown in Figure 5, and process them by increasing category value ζ. Furthermore, for tasks belonging to the same category, there are no precedence constraints between them since an ASAP schedule with an unbounded number of processors would be able to process them concurrently during the interval corresponding to their category. This is the motivation for structuring categories in such a manner because it is known that efficient strategies exist for processing independent tasks [**?**]. The potential time loss depends only on the length of the longest task, which is bounded by L_ζ. Each time the power level increases by 1, the number of categories is roughly reduced by 2 (as shown in Figure 3), and the length of categories is roughly doubled (as shown in Figure 5). Therefore, most categories will only include small tasks and will be processed efficiently.

To further aid analysis, we introduce one last construction, the **length matrix** (or **L-matrix**) $\mathcal{L}(\mathcal{C})$, for any instance with critical path length $\mathcal{C}$. We point out again that $\mathcal{L}(\mathcal{C})$ depends only on $\mathcal{C}$ but not on a specific instance. Thus, we will denote it simply as $\mathcal{L}$ to ease notation. This (infinite) matrix contains the different possible values of

L_ζ, and each element $\ell_{i,j} \in \mathcal{L}$ also corresponds to a category. Figure 6 (left) shows the L-matrix for the example of Figures 4 and 5. Each row of the matrix corresponds to a power level χ, and each column corresponds to a longitude value λ. The top-left element of the matrix ($\ell_{1,1}$) corresponds to the category that spreads across $[0, \mathcal{C}]$, where the length of each task is at most $\mathcal{C}$. In our example, this corresponds to category $\zeta = 4$, which includes tasks A, E, and I (see Figure 5). More generally, the correspondence between elements of the L-matrix and the category values can be viewed in Figure 6. Because $\mathcal{C} = 6.8$ in this example, the 0's in the L-matrix correspond to impossible categories with $\zeta \geq \mathcal{C}$, marked in gray. The L-matrix is formally defined below.

Hongyang: Should the $X \in \mathbb{N}$ below be $X \in \mathbb{Z}$ instead? Possibly you had $X \in \mathbb{Z}$ initially and I changed it to $X \in \mathbb{N}$ by mistake at some point. It seems that $X \in \mathbb{Z}$ makes more sense here, since some instances can have critical-path length less than 1. Note this appeared in multiple places below as well.

Lucas: Definitely $\mathbb{Z}$ everywhere, well spotted

Hongyang: Okay, I changed it, but possibly missed some places, so please check again.

Definition 6. *For any instance with critical-path length $\mathcal{C} \in (2^X, 2^{X+1}]$ for some $X \in \mathbb{Z}$, we define $\mathcal{L} = (\ell_{i,j})^{\mathbb{N}\times\mathbb{N}}$ as its **length matrix** (or **L-matrix**) such that:*

$$\ell_{i,j} = \begin{cases} L_{\zeta(X+1-i,2j-1)} & \text{if } \zeta(X+1-i, 2j-1) < \mathcal{C}, \text{ where } \zeta(\chi, \lambda) = \lambda 2^\chi \\ 0 & \text{otherwise} \end{cases}$$

Using the above definition, we can compute the values of $\ell_{i,j}$'s by considering three different cases: $L_\zeta = 2^{\chi+1}$, $L_\zeta = \mathcal{C} - (\lambda - 1)2^\chi$ (See Definition 5), and $\zeta > \mathcal{C}$ (i.e., no tasks belong to it and therefore $\ell_{i,j} = 0$). This is shown in the following lemma.

Lemma 6. *For any instance with critical-path length $\mathcal{C} \in (2^X, 2^{X+1}]$ for some $X \in \mathbb{Z}$, the elements of its L-matrix $\mathcal{L} = (\ell_{i,j})^{\mathbb{N}\times\mathbb{N}}$ can be expressed as:*

$$\ell_{i,j} = \begin{cases} 2^{X+2-i} & \text{if } j2^{X+2-i} \leq \mathcal{C} \\ \mathcal{C} - (j-1)2^{X+2-i} & \text{if } (2j-1)2^{X+1-i} < \mathcal{C} < j2^{X+2-i} \\ 0 & \text{otherwise} \end{cases} \tag{8}$$

Proof. We can verify the correctness of the three cases as follows:

- If $j2^{X+2-i} \leq \mathcal{C}$, then $\zeta(X+1-i, 2j-1) = (2j-1)2^{X+1-i} < \mathcal{C}$, and by Definition 6, $\ell_{i,j} = L_{\zeta(X+1-i,2j-1)}$. Using Definition 5, $L_{\zeta(X+1-i,2j-1)} = \min\left(2^{X+2-j}, \mathcal{C} - (2j-2)2^{X+1-j}\right)$. By the assumption $\mathcal{C} \geq (2j)2^{X+1-i}$, the minimum is the first term: $\ell_{i,j} = 2^{X+2-j}$.
- If $(2j-1)2^{X+1-i} < \mathcal{C} < j2^{X+2-i}$, then $\zeta(X+1-i, 2j-1) = (2j-1)2^{X+1-i} < \mathcal{C}$, and by Definition 6, $\ell_{i,j} = L_{\zeta(X+1-i,2j-1)}$. Using Definition 5, $L_{\zeta(X+1-i,2j-1)} = \min\left(2^{X+2-j}, \mathcal{C} - (2j-2)2^{X+1-j}\right)$. By the assumption $\mathcal{C} < (2j)2^{X+1-i}$, the minimum is the second term: $\ell_{i,j} = \mathcal{C} - (j-1)2^{X+2-j}$.
- Otherwise, $\mathcal{C} \leq (2j-1)2^{X+1-i}$, then $\zeta(X+1-i, 2j-1) = (2j-1)2^{X+1-i} \geq \mathcal{C}$, and by Definition 6, $\ell_{i,j} = 0$. □

5.2 Algorithm

The algorithm presented here computes each task's category and processes non-empty categories by increasing category value ζ, using a greedy scheduling routine to process independent tasks efficiently. The principles of the algorithm are summarized below, and its detailed analysis is given in the next subsection.

- When processing tasks in a category, the algorithm will only discover tasks in strictly larger categories. Therefore, all tasks in a category are independent and become known when processing that category.
- It is relatively efficient to greedily process tasks in a category (i.e., a batch of independent tasks). The time we might lose depends on the length of the longest task in the batch, which is upper bounded by the length of the category.

Based on the above principles, the algorithm uses category to split the graph into batches of independent tasks and process them one after another. This limits the number of categories with long tasks, which may only be in the upper rows of the L-matrix and whose numbers diminish extremely quickly. Our algorithm makes use of the following two subroutines:

- ComputeCat($\mathcal{T}$), presented in Algorithm 1, computes the category ζ of a ready task $\mathcal{T}$. It first computes the earliest start time s^∞ and earliest finish time f^∞ of the task, and then based on Definition 4 computes the task's unique category value ζ.
- ScheduleIndep($\mathcal{B}$), presented in Algorithm 2, greedily schedules a batch $\mathcal{B}$ of independent tasks in any arbitrary order. The order of the tasks will not affect the performance bound, as shown in the analysis (Section 5.3). Specifically, whenever a task completes[1], the algorithm considers each remaining task in $\mathcal{B}$ and executes it if there are sufficient processors. It is worth noting that the subroutine ends only when all the tasks in $\mathcal{B}$ are completed and not simply scheduled (Line 17). The algorithm also collects any newly discovered tasks during this batch and stores them in a list $\mathcal{R}$, which will be returned and processed in subsequent batches.

Algorithm 1 ComputeCat($\mathcal{T}$)

```
1: if $\mathcal{P}(\mathcal{T}) = \emptyset$ then        ▷ $\mathcal{P}(\mathcal{T})$ contains the list of predecessors of task $\mathcal{T}$
2:     $s^\infty \leftarrow 0$
3: else
4:     $s^\infty \leftarrow \max_{\mathcal{T}_j \in \mathcal{P}(\mathcal{T})} f_j^\infty$        ▷ start time in the ASAP schedule
5: end if
6: $f^\infty \leftarrow s^\infty + t$        ▷ finish time in the ASAP schedule
7: find $\chi$ and $\lambda$ from $s^\infty$ and $f^\infty$ based on Definition 4
8: $\zeta \leftarrow \lambda 2^\chi$
9: return $\zeta$
```

Algorithm 2 ScheduleIndep($\mathcal{B}$)

```
1: organize tasks in $\mathcal{B}$ in any arbitrary order
2: $P_{avail} \leftarrow P$        ▷ number of available processors
3: $\mathcal{R} \leftarrow \emptyset$        ▷ $\mathcal{R}$ will store newly discovered tasks
4: while $\mathcal{B}$ is not empty and whenever a task $\mathcal{T}_i$ completes do
5:     $P_{avail} \leftarrow P_{avail} + p_i$
6:     for each discovered task $\mathcal{T}_j$ do
7:         add $\mathcal{T}_j$ to $\mathcal{R}$
8:     end for
9:     for each task $\mathcal{T}_k \in \mathcal{B}$ do
10:        if $P_{avail} \geq p_k$ then
11:            execute $\mathcal{T}_i$ now
12:            $P_{avail} \leftarrow P_{avail} - p_k$
13:            remove $\mathcal{T}_k$ from $\mathcal{B}$
14:        end if
15:    end for
16: end while
17: wait until all tasks in $\mathcal{B}$ complete
18: return $\mathcal{R}$
```

Finally, our algorithm Batch is presented in Algorithm 3. It uses a dictionary $\mathcal{A}$, which is a collection of lists, to store tasks, and each list contains tasks that belong to a particular category. The algorithm processes tasks in batches of increasing category values. Let $\mathcal{B}_\zeta$ denote a batch of tasks with category ζ, and the batch is scheduled using subroutine ScheduleIndep($\mathcal{B}_\zeta$). While processing a batch, any newly discovered task $\mathcal{T}$ that becomes ready will be collected, and its category computed using subroutine ComputeCat($\mathcal{T}$). The task will then be added to the corresponding list in $\mathcal{A}$. Our analysis in Section 5.3 will show that all tasks in a category will be ready when their category starts to be processed.

Figure 7 illustrates how the algorithm works using the example of Figure 4 on $P = 4$ processors. At first, only tasks A, B, C, and D are ready, and they are added to the corresponding lists of $\mathcal{A}$ with their respective category

[1]

Lucas: Footnote looks confusing to me

a virtual task $\mathcal{T}$ with $p = 0$ and $t = 0$ can be considered to complete at the beginning of the batch.

Algorithm 3 Batch

```
 1: $\mathcal{A} \leftarrow \emptyset$          ▷ $\mathcal{A}$ is a collection of lists (dictionary), each list stores tasks of a particular category
 2: $\mathcal{R} \leftarrow \emptyset$          ▷ $\mathcal{R}$ will store the set of ready tasks
 3: for each ready task $\mathcal{T}_i$ do
 4:     add $\mathcal{T}_i$ to $\mathcal{R}$
 5: end for
 6: repeat
 7:     for each task $\mathcal{T}_i \in \mathcal{R}$ do
 8:         $\zeta_i \leftarrow$ ComputeCat($\mathcal{T}_i$)
 9:         add $\mathcal{T}_i$ to $\mathcal{A}[\zeta_i]$          ▷ $\mathcal{A}[\zeta_i]$ contains a list of tasks of category $\zeta_i$
10:     end for
11:     Find $\zeta_{\min}$, the smallest category of all tasks in $\mathcal{A}$
12:     $\mathcal{R} \leftarrow$ ScheduleIndep($\mathcal{A}[\zeta_{\min}]$)
13:     delete $\mathcal{A}[\zeta_{\min}]$
14: until all tasks are completed
```

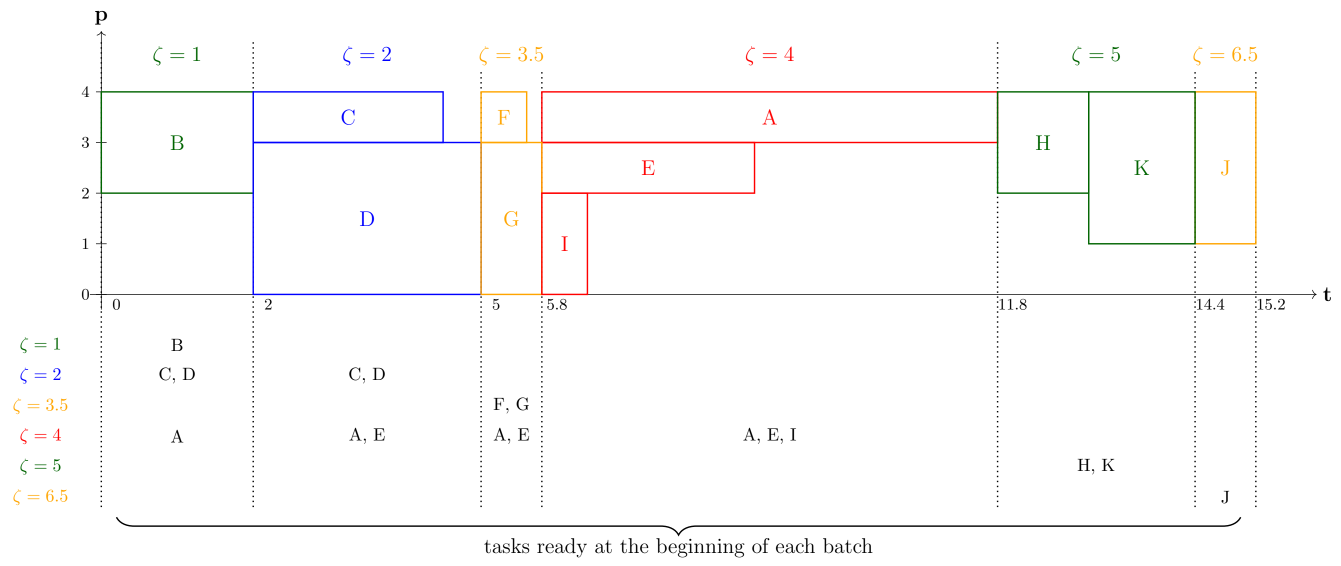


Figure 7: Illustration of the Batch algorithm for the example of Figure 4 on $P = 4$ processors.

values. The smallest category, $\zeta = 1$, contains only task B. Thus, the subroutine ScheduleIndep will be called with $\mathcal{B}_1 = \{\text{B}\}$. When this batch is completed, task $\mathcal{R} = \{\text{E}\}$ is discovered, and it is placed in the list of task A that shares the same category. Now, the algorithm processes the smallest category, $\zeta = 2$, that contains $\mathcal{B}_2 = \{\text{C, D}\}$, discovering two new tasks $\mathcal{R} = \{\text{F, G}\}$. The algorithm keeps going, processing the third batch $\mathcal{B}_{3.5} = \{\text{F, G}\}$ with category $\zeta = 3.5$, then the fourth batch $\mathcal{B}_4 = \{\text{A, E, I}\}$ with category $\zeta = 4$, then the fifth batch $\mathcal{B}_5 = \{\text{H, K}\}$ with category $\zeta = 5$, and finally, the last batch $\mathcal{B}_{6.5} = \{\text{J}\}$ with category $\zeta = 6.5$, after which all tasks are completed.

Lucas: I like the notation $\mathcal{B}_{6.5}$. Should we use $\mathcal{B}_\zeta$ instead of $\mathcal{A}[\zeta_i]$ in the algorithm and text?

Hongyang: Yes, please make the change during clean-up.

5.3 Analysis

The general steps of the analysis follow the principles of the algorithm stated previously. First, we show that tasks in a category are independent and all known when we start processing the category.

Lemma 7. *If there is a dependency between task $\mathcal{T}_i$ and task $\mathcal{T}_j$, then $\zeta_i < \zeta_j$.*

Proof. If $\mathcal{T}_j$ must be processed after the completion of task $\mathcal{T}_i$, then we must have $f_i^\infty \leq s_j^\infty$. Using Lemma 4, we derive $\zeta_i < f_i^\infty \leq s_j^\infty < \zeta_j$, hence the result. □

Corollary 4. *When* Batch *calls* ScheduleIndep($\mathcal{B}_\zeta$)*, all the tasks of this category have already been discovered (and are independent).*

Proof. By induction, when Batch calls ScheduleIndep($\mathcal{B}_\zeta$), all tasks of strictly smaller categories are completed because of Algorithm 2. Indeed, the algorithm does not return until the batch is entirely finished. As we always call the subroutine with the tasks of the smallest category, and because tasks may only release tasks with category strictly larger according to Lemma 7, the induction is immediate. Therefore, when Batch calls ScheduleIndep($\mathcal{B}_\zeta$), any task $\mathcal{T}$ of category ζ must be ready since all of its parents have category strictly smaller as stated in Lemma 7 and thus must be completed. ☐

Lemma 8. *For any batch $\mathcal{B}_\zeta$ of tasks with category ζ, the execution time of the batch scheduled by* ScheduleIndep($\mathcal{B}_\zeta$) *satisfies:*

$$T(\mathcal{B}_\zeta) \leq 2\frac{\mathcal{A}(\mathcal{B}_\zeta)}{P} + L_\zeta$$

where $\mathcal{A}(\mathcal{B}_\zeta) = \sum_{\mathcal{T}_i \in \mathcal{B}_\zeta} t_i p_i$ is the area of all tasks in batch $\mathcal{B}_\zeta$.

Proof. From Corollary 4, when batch $\mathcal{B}_\zeta$ is scheduled, all tasks of category ζ are ready. In the schedule for $\mathcal{B}_\zeta$, let t' be the first moment when less than $\frac{P}{2}$ processors are used. All tasks not yet started by t' must require at least $\frac{P}{2}$ processors. Otherwise, one of them would have been launched at t' (by the for-loop in lines 9-15), since there would be enough processors to process it. Therefore, all tasks requiring less than $\frac{P}{2}$ processors have already started at this moment.

Let t'' be the moment when all tasks requiring less than $\frac{P}{2}$ processors are completed. The inequality $t'' - t' \leq L_\zeta$ holds, since all of these tasks have already started at t', and L_ζ serves as an upper bound on the length of any task in category ζ (note that we may have $t' > t''$). Because all remaining tasks require more than $\frac{P}{2}$ processors after t'', at least $\frac{P}{2}$ processors are used from t'' to the end of the schedule.

Therefore, at least $\frac{P}{2}$ processors are used except potentially in $[t', t'']$, which represents a duration of at most L_ζ. Let $T_{\geq P/2}$ be the set of times where more than $\frac{P}{2}$ processors are used, and $T_{<P/2}$ the rest of the schedule. Since the total area processed in $T_{\geq P/2}$ is at least $\frac{P}{2}|T_{\geq P/2}|$, and at most the total area $\mathcal{A}(\mathcal{B}_\zeta)$ of the tasks in $\mathcal{B}_\zeta$, we get $|T_{\geq P/2}| \leq 2\frac{\mathcal{A}(\mathcal{B}_\zeta)}{P}$. Furthermore, $T_{<P/2}$ is included in $[t', t'']$, thus $|T_{<P/2}| \leq t_2 - t_1 \leq L_\zeta$. From $T(\mathcal{B}_\zeta) = |T_{<P/2}| + |T_{\geq P/2}|$, we achieve the result. ☐

Lucas: To check

Hongyang: Checked with only minor edits. In particular, I changed t_1 and t_2 to t' and t'', since t_i denotes the length of task i.

Lemma 9. *For any instance $\mathcal{I}$, the makespan of the schedule produced by* Batch *satisfies:*

$$T_{\text{Batch}}(\mathcal{I}) \leq 2\frac{\mathcal{A}(\mathcal{I})}{P} + \sum_\zeta L_\zeta \tag{9}$$

where $\sum_\zeta L_\zeta$ represents the sum of lengths from all non-empty categories (i.e., containing at least one task).

Proof. Because we process each category one after another without idle time between categories, the makespan of the schedule given by Batch is exactly the sum of the execution times of all calls to ScheduleIndep, one for each category. Using Lemma 8, we obtain $T_{\text{Batch}}(\mathcal{I}) = \sum_\zeta T(\mathcal{B}_\zeta) \leq \sum_\zeta 2\frac{\mathcal{A}(\mathcal{B}_\zeta)}{P} + \sum_\zeta L_\zeta$. We get the result since each task is processed in exactly one category, thus $\sum_\zeta \mathcal{A}(L_\zeta) = \mathcal{A}(\mathcal{I})$. ☐

5.3.1 Unbounded Task Execution Times

We may finally show the competitiveness of the Batch algorithm. The following theorem gives the result when tasks have unbounded execution times. The following subsection considers tasks with bounded execution times.

Theorem 3. Batch *satisfies* $\forall n > 0, R_{\text{Opt}}(\text{Batch}_n) \leq R_{\text{LB}}(\text{Batch}_n) \leq 3 + \log(n)$. *In other words,* Batch *is* $(3 + \log(n))$*-competitive.*

Proof. Using Lemma 9, we simply have to show $\sum_\zeta L_\zeta \leq (\log(n) + 1)\mathcal{C}(\mathcal{I})$. We point out that, by construction, for each non-empty category ζ, its length L_ζ must be present in the L-matrix $\mathcal{L}$. Clearly, $\sum_\zeta L_\zeta$ is maximized if there is one task per category (to maximize the number of terms in the sum), and the categories correspond to the n largest values in $\mathcal{L}$ (to maximize the sum).

$$\begin{pmatrix} 6.8 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \cdots \\ 4 & 2.8 & 0 & 0 & 0 & 0 & 0 & 0 & \cdots \\ 2 & 2 & 2 & 0 & 0 & 0 & 0 & 0 & \cdots \\ 1 & 1 & 1 & 1 & 1 & 1 & 0.8 & 0 & \cdots \\ 0.5 & 0.5 & 0.5 & 0.5 & 0.5 & 0.5 & 0.5 & 0.5 & \cdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots \end{pmatrix} \qquad \begin{array}{c} \begin{pmatrix} 2.3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \cdots \\ 2.3 & 2.3 & 0 & 0 & 0 & 0 & 0 & 0 & \cdots \\ 2 & 2 & 2 & 0 & 0 & 0 & 0 & 0 & \cdots \\ 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & \cdots \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \cdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots \end{pmatrix} \end{array} \begin{array}{l} \updownarrow \text{R} \\ \\ \updownarrow \text{U} \\ \\ \uparrow \text{I} \\ \vdots \end{array}$$

Figure 8: For any instance with $\mathcal{C} = 6.8$, left: L-matrix $\mathcal{L}(\mathcal{C})$ without constraints on the task execution times; right: L-matrix $\mathcal{L}^*(\mathcal{C})$ with bounds $m = 0.9$ and $M = 2.3$ on the task execution times.

Suppose the critical-path length of the instance $\mathcal{I}$ satisfies $2^X < \mathcal{C}(\mathcal{I}) \leq 2^{X+1}$ for some $X \in \mathbb{N}$. The proof consists of the following three steps, which can also be verified visually in Figure 8 (left) for any instance with $\mathcal{C} = 6.8$.

1. *The n largest values in $\mathcal{L}$ can be picked from the positive values one row after another, from left to right.*
   Indeed, each row's values clearly decrease from left to right: using Lemma 6, all values are equal except potentially the last positive one $v_{i,j}$, if it corresponds to the second case with $\mathcal{C}(\mathcal{I}) < j2^{X+2-i}$. This condition shows $v_{i,j} = \mathcal{C}(\mathcal{I}) - (j-1)2^{X+2-i} < 2^{X+2-i}$. Therefore, we just need to show that the last positive value of the row $v_{i,j}$ is larger than the first value of the next row $v_{i+1,1}$. If $v_{i,j}$ corresponds to the first case in Equation (8), we clearly have $v_{i,j} = 2^{X+2-i} > 2^{X+1-i} = v_{i+1,1}$. If $v_{i,j}$ corresponds to the second case in Equation (8), $v_{i,j} = \mathcal{C}(\mathcal{I}) - (2j-2)2^{X+1-i} > 2^{X+1-i} = v_{i+1,1}$ since $(2j-1)2^{X+1-i} < \mathcal{C}(\mathcal{I})$ in this case.

2. *Each row in $\mathcal{L}$ has a sum at most $\mathcal{C}(\mathcal{I})$, the first row, $i = 1$, has a single positive value, and each row $i \geq 2$ has at least 2^{i-2} positive values.*
   The first row consists of just one positive value $v_{1,1} = \mathcal{C}(\mathcal{I})$, and all the other values are 0, since $\mathcal{C}(\mathcal{I}) \leq 2^{X+1}$. For each row $i \geq 2$, we can rewrite $\mathcal{C}(\mathcal{I}) = k2^{X+2-i} + r$ for some positive integer k, where $r < 2^{X+2-i}$. Since $\mathcal{C}(\mathcal{I}) > 2^X$, we must have $k \geq 2^{i-2}$. Therefore, $v_{i,1} = \cdots = v_{i,k} = 2^{X+2-i}$, and $v_{i,k+1}$ is either $\mathcal{C}(\mathcal{I}) - k2^{X+2-i} = r$ (second case in Equation (8)) or 0, and all other $v_{i,j}$'s are 0. This shows $\sum_{j=1}^{\infty} v_{i,j} \leq k2^{X+2-i} + r = \mathcal{C}(\mathcal{I})$.

3. *The sum of any set of n values in $\mathcal{L}$ is at most $(\log(n) + 1)\mathcal{C}(\mathcal{I})$.*
   Let $n = 2^k + r$, where k and r are both integers with $0 \leq r < 2^k$. From Claim 1 above, we know that the sum is maximized by picking values from the rows one after another. By Claim 2, fully completing $k+1$ rows would require choosing at least $1 + \sum_{i=2}^{k+1} 2^{i-2} = 2^k$ values, and we may additionally choose r values from row $k+2$. Thus, we obtain $\sum_{\zeta} L_{\zeta} \leq (k+1)\mathcal{C}(\mathcal{I}) + r2^{X-k} < (k+1+\frac{r}{2^k})\mathcal{C}(\mathcal{I})$. To conclude, we just need to show $k + \frac{r}{2^k} \leq \log(2^k + r) = \log(n)$ for $r \in [0, 2^k)$. We have exact equalities for $r = 0$ and $r = 2^k$. In between, because $f(r) = k + \frac{r}{2^k}$ is affine and $g(r) = \log(2^k + r)$ is concave, we must have $f(r) \leq g(r)$.

From Claim 3 above and using Lemma 9 and Equation (1), we obtain the result:

$$\begin{aligned} T_{\text{BATCH}}(\mathcal{I}) &\leq 2\frac{\mathcal{A}(\mathcal{I})}{P} + (\log(n) + 1)\mathcal{C}(\mathcal{I}) \\ &\leq (3 + \log(n))\text{LB}(\mathcal{I}) \end{aligned}$$

□

5.3.2 Bounded Task Execution Times

We now show the result when the execution times of the tasks are bounded in a range $[m, M]$, i.e. $\forall i, m \leq t_i \leq M$. When m and M are constants, we show that BATCH has a constant competitive ratio. Indeed, all categories such that $L_{\zeta} < m$ are now empty because no tasks may fit in these categories. Figure 8 (right) shows the L-matrix for any instance with $\mathcal{C} = 6.8$ and $m = 0.9$. Compared to Figure 8 (left) that does not place any bounds on the task execution times, all positive values less than 0.9 in the L-matrix now become 0 (shown in red), and all rows turned to 0 are labeled as I (for Impossible). Furthermore, the values in the top rows will also be reduced since the task lengths are upper-bounded by M. Figure 8 (right) shows the reduced values (in blue) with $M = 2.3$, and the rows whose positive values are reduced are labeled as R (for Reduced). The remaining rows whose positive values satisfy $m \le L_\zeta \le M$ are labeled as U (for Unchanged). These rows capture the most significant weight of the resulting matrix, which we denote as $\mathcal{L}^*(\mathcal{C})$ or simply $\mathcal{L}^*$, and the number of such rows will be shown to be around $\log(\frac{M}{m})$.

More formally, given L_ζ, m and M, we define L^*_ζ as follows:

$$L^*_\zeta = \begin{cases} \min(M, L_\zeta) & \text{if } L_\zeta \ge m \\ 0 & \text{otherwise} \end{cases}$$

Here, L^*_ζ is an upper bound on the length of the longest task of category ζ, and based on Lemma 9, we have:

$$T_{\text{Batch}}(\mathcal{I}) \le 2\frac{\mathcal{A}(\mathcal{I})}{P} + \sum_\zeta L^*_\zeta \tag{10}$$

Theorem 4. *For any instance $\mathcal{I}$ such that $\forall i, m \le t_i \le M$, we have $\frac{T_{\text{Batch}}(\mathcal{I})}{\text{LB}(\mathcal{I})} \le 6 + \log\left(\frac{M}{m}\right)$.*

Lucas: Do you like the $R_{\text{OPT}}(\text{Batch}_n)/R_{\text{LB}}(\text{Batch}_n)$ notation? I was trying to be rigorous in Section 2/3 but we can remove it completely and just claim all results in the form of the Theorem 4?

Hongyang: I like the Theorem 4 style, as it is the familiar notation :) Why don't we change Theorem 3 then?

Lucas: Sure, actually k used to be the power level notation which is probably while I took j and l

Hongyang: Okay, I changed j to k below, except in Claim 3 where j is used for something different. Please check I didn't mess up.

Proof. Let $\mathcal{I}$ be an instance with $m = \min_i(t_i)$ and $M = \max_i(t_i)$. We define $k \in \mathbb{Z}$ such that $2^k < m \le 2^{k+1}$ and $l \in \mathbb{Z}$ such that $2^l < M \le 2^{l+1}$. Suppose the critical-path length of the instance satisfies $2^X < \mathcal{C}(\mathcal{I}) \le 2^{X+1}$ for some $X \in \mathbb{Z}$. We have $k \le l \le X$. For any integer χ, we further define Z_χ to be the set of categories whose power levels are exactly χ, and these categories correspond to one particular row in $\mathcal{L}^*$. The proof consists of the following four steps.

1. *There are no tasks in categories whose power levels are strictly less than k.*
   According to Definition 5 and Lemma 5, any task in a category ζ whose power level χ is strictly less than k must have an execution time t that is at most $L_\zeta \le 2^k < m$. Hence, its existence would contradict the assumption that $t_i \ge m, \forall i$. In Figure 8 (right), with $m = 0.9$, this corresponds to having no task in the fifth row and below in $\mathcal{L}^*$, labeled as I (Impossible).

2. *For any set Z_χ of categories with power level $\chi \in [k, l-1]$, the sum of their lengths is at most $\mathcal{C}(\mathcal{I})$. There are $l - k$ such sets of categories, and we have $l - k \le \log\left(\frac{M}{m}\right) + 1$.*
   There are $l-k$ rows in $\mathcal{L}^*$ that correspond to power levels in the range $[k, l-1]$. From $m \le 2^{k+1}$ and $2^l < M$, we get $\frac{M}{m} > \frac{2^l}{2^{k+1}}$ and $l - k - 1 \le \log\left(\frac{M}{m}\right)$. We have already shown in the proof of Theorem 3 that the sum of values in each row, which corresponds to the sum of lengths from all categories of the respective power level, is at most $\mathcal{C}(\mathcal{I})$. Therefore, using $L^*_\zeta \le L_\zeta$, we get $\sum_{\zeta \in Z_\chi} L^*_\zeta \le \sum_{\zeta \in Z_\chi} L_\zeta \le \mathcal{C}(\mathcal{I})$. In Figure 8 (right), with $m = 0.9$ and $M = 2.3$, we have $k = -1$ and $l = 1$, corresponding to rows 3 and 4, whose values are mostly unchanged except for the last value of row 4. These rows are labeled as U (Unchanged).

3. *For all categories with power level at least l, the sum of their lengths is at most $3\mathcal{C}(\mathcal{I})$.*
   The first row of $\mathcal{L}^*$ has only one positive value, corresponding to the category with power level X, and the second row has at most 2 positive values. Generally, row i has at most 2^{i-1} positive values. Otherwise, we would have $l_{i,2^{i-1}+1} > 0$, which according to Equation (8) means $(2j-1)2^{X+1-i} < \mathcal{C}(\mathcal{I})$ with $j = 2^{i-1}+1$. This implies $2^{X+1} + 2^{X+1-i} < \mathcal{C}(\mathcal{I})$, which contradicts $\mathcal{C}(\mathcal{I}) \le 2^{X+1}$. The categories with power level at least $l+1$ correspond to the $X - l$ first rows of $\mathcal{L}^*$. The number of such categories is at most $\sum_{i=1}^{X-l} 2^{i-1} \le 2^{X-l}$, and the longest task in these categories has length at most M. From $2^X < \mathcal{C}(\mathcal{I})$ and $M \le 2^{l+1}$, we derive $M2^{X-l} < 2\mathcal{C}(\mathcal{I})$, which means $\sum_{\chi=l+1}^{X} \sum_{\zeta \in Z_\chi} L^*_\zeta < 2\mathcal{C}(\mathcal{I})$. Finally, adding in the categories with power level l, which satisfies $\sum_{\zeta \in Z_l} L^*_\zeta \le \sum_{\zeta \in Z_l} L_\zeta \le \mathcal{C}(\mathcal{I})$, we get the sum of lengths from all categories with power level at least l to be at most $3\mathcal{C}(\mathcal{I})$, i.e., $\sum_{\chi=l}^{X} \sum_{\zeta \in Z_\chi} L^*_\zeta < 3\mathcal{C}(\mathcal{I})$. In Figure 8 (right), this corresponds to the first two rows, labeled as R (Reduced).

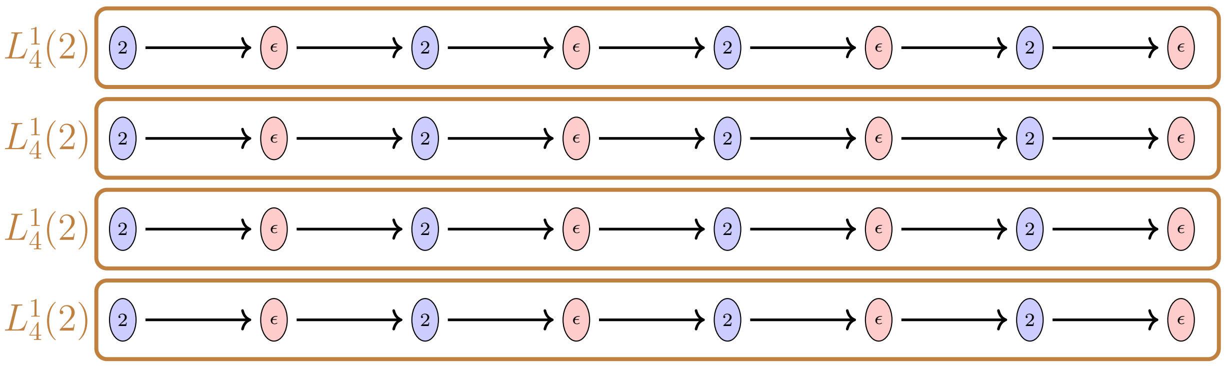


Figure 9: $Y_4^1(2)$, minimum time of processing is $8 + 16\epsilon$

4. *The sum of lengths from all non-empty categories satisfies* $\sum_\zeta L_\zeta^* \leq \left(\log\left(\frac{M}{m}\right) + 4\right)\mathcal{C}(\mathcal{I})$.
   This can be shown by simply summing the contributions from the three types of categories considered above.
   - Those from the I (Impossible) rows contribute 0 to the sum (Claim 1);
   - Those from the U (Unchanged) rows contribute $\left(\log\left(\frac{M}{m}\right) + 1\right)\mathcal{C}(\mathcal{I})$ to the sum (Claim 2);
   - Those from the R (Reduced) rows contribute $3\mathcal{C}(\mathcal{I})$ to the sum (Claim 3).

From Claim 4 above and using Equations (10) and (1), we obtain the result:

$$\begin{aligned} T_{\text{BATCH}}(\mathcal{I}) &\leq 2\frac{\mathcal{A}(\mathcal{I})}{P} + \left(4 + \log\left(\frac{M}{m}\right)\right)\mathcal{C}(\mathcal{I}) \\ &\leq \left(6 + \log\left(\frac{M}{m}\right)\right)\text{LB}(\mathcal{I}) \end{aligned}$$

□

5.4 Lower bound on best competitive ratio

This section shows lower bounds on the best possible competitive ratio achievable.

In Section 4, we have seen that the graph $X_P(K)$, consisting of P distinct linear chains from $L_P^0(K)$ to $L_P^{P-1}(K)$ takes a very long time to process compared to the lower bounds. However, a graph consisting of identical linear chains $L_P^i(K)$ can be processed very efficiently.

Definition 7. *For $i \in [0, P-1]$, let $Y_P^i(K)$ be a graph of tasks that contains P identical linear chains $L_P^i(K)$*

$Y_4^1(2)$ is illustrated in Figure 9. In this section, we will build an adversary instance using $X_P(K)$ and $Y_P^i(K)$, forcing any algorithm to process graphs in the shape of $X_P(K)$ sequentially, resulting in very poor processor utilization, while a clairvoyant scheduler would be able to process graphs in the shape of $Y_P^i(K)$, with near-optimal processor allocation.

Lemma 10. *For all i, K, P, the optimal time of processing of a graph $Y_P^i(K)$ is*

$$T_{\text{OPT}}(Y_P^i(K)) = K^{P-1} + PK^{P-i-1}\epsilon \tag{11}$$

Proof. We consider an algorithm that processes the first blue task of all linear chains in parallel (requesting all P processors), then processes the first red task of all linear chains sequentially (requesting, again, all P processors), and repeats these two steps alternatively until completion of all tasks. By Definition 1, the first step takes time K^i, while the second takes time $P\epsilon$. As there are K^{P-i-1} tasks of each color in each line, this algorithm has a time of processing of $T_{\text{ALG}}(Y_P^i(K)) = K^{P-i-1}(K^i + P\epsilon) = K^{P-1} + PK^{P-i-1}\epsilon$. This schedule is optimal since all processors are always used, hence the result. □

In the following, we fix an arbitrary algorithm ALG, and show lower bounds in its competitive ratio.

Definition 8. *For any K and P, and given an algorithm ALG, we build the graph $Z_P^{\text{ALG}}(K)$ as follows:*

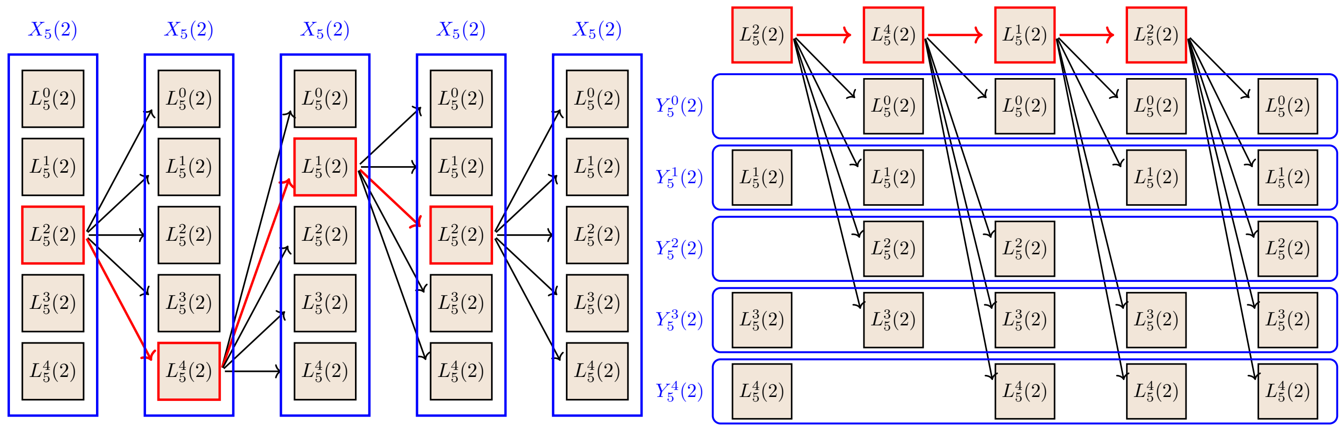


Figure 10: $Z_5^{\text{ALG}}(2)$. Left, ALG will have to process layers one by one. Right, better clairvoyant schedule, where red chains are processed first, then the $Y_5^i(2)$ can be processed efficiently.

- *The structure consists of P layers consisting of identical $X_P(K)$ graphs*
- *A new graph $X_P(K)$ is revealed only after the last one is fully completed by* ALG *(the dependencies between graphs only come out of the last task processed).*

$Z_5^{\text{ALG}}(2)$ is illustrated in Figure 10, assuming the last task completed by the algorithm in the first layer is in $L_5^2(2)$, the second is in $L_5^4(2)$, the third is in $L_5^1(2)$, and the fourth is in $L_5^2(2)$.

Lemma 11. *For any* ALG*, the processing time of $Z_P^{\text{ALG}}(K)$ satisfies*

$$T_{\text{ALG}}(Z_P^{\text{ALG}}(K)) \geq P^2K^{P-1} - P(P-1)K^{P-2} \tag{12}$$

Proof. The result directly comes from Lemma 2 and Definition 8, as T_{ALG} is the time required to process P identical copies of $X_P(K)$, each of them taking time larger than $T_{\text{OPT}}(X_P(K))$, and a copy may not start before the completion of the precedent one □

Lemma 12. *An optimal clairvoyant scheduler can process $Z_P^{\text{ALG}}(K)$ faster than*

$$T_{\text{OPT}}(Z_P^{\text{ALG}}(K)) < 2PK^{P-1} + 2P^2K^P\epsilon \tag{13}$$

Proof. A clairvoyant scheduler could process $Z_P^{\text{ALG}}(K)$ in two steps: first, process all chains $L_P^i(K)$ that have successors after the last task of each chain. After the first step, all the remaining $L_P^i(K)$ are independent. It is then possible to regroup them by identical i to form subgraphs included in $Y_P^i(K)$ and process them optimally one after the other. This is illustrated by Figure 10, right with $Z_5^{\text{ALG}}(2)$.

Clearly, processing any of the chain $L_P^i(K)$ takes time smaller than $\mathcal{C}(X_P(K)) \leq \text{LB}(X_P(K))$, and processing any of the resulting subgraph of $Y_P^i(K)$ takes time smaller than $T_{\text{OPT}}(Y_P^i(K))$. Using Equation 2 and Equation 11, we obtain:

$$\begin{aligned} T_{\text{OPT}}(Z_P^{\text{ALG}}(K)) &\leq (P-1)\text{LB}(X_P(K)) + PT_{\text{OPT}}(Y_P^i(K)) \\ &< PK^{P-1} + P^2K^P\epsilon + PK^{P-1} + P^2K^P\epsilon \\ &= 2PK^{P-1} + 2P^2K^P\epsilon \end{aligned}$$

□

Theorem 5. *In **online clairvoyant settings**, $\forall P > 0, \forall \text{ALG}, R_{\text{OPT}}(\text{ALG}^P) \geq \frac{P}{2}$. In other words, for any $\mu > 0$, no algorithm (polynomial or not) may be $\left(\frac{P}{2} - \mu\right)$-competitive for any P*

Proof. We fix $P > 0$, an algorithm ALG, and $\mu > 0$ arbitrarily small. Choosing $K > \frac{P-1}{\mu}$ and $0 < \epsilon < \frac{\mu}{P^2K}$, we derive from Lemma 11 and Lemma 12:

$$\begin{aligned}
R_{\text{OPT}}(\text{ALG}^P) &\geq \frac{T_{alg}(Z_P^{\text{ALG}}(K))}{T_{\text{OPT}}(Z_P^{\text{ALG}}(K))} \\
&> \frac{P^2K^{P-1} - P(P-1)K^{P-2}}{2PK^{P-1} + 2P^2K^P\epsilon} \\
&= \frac{P - \frac{P-1}{K}}{2 + 2PK\epsilon} \\
&> \frac{P-\mu}{2\left(1+\frac{\mu}{P}\right)} = \frac{P\left(1+\frac{\mu}{P}\right) - \mu - \mu}{2\left(1+\frac{\mu}{P}\right)} \\
&> \frac{P}{2} - \mu
\end{aligned}$$

We got $\forall \mu > 0, R_{\text{OPT}}(\text{ALG}^P) > \frac{P}{2} - \mu$, therefore $R_{\text{OPT}}(\text{ALG}^P) \geq \frac{P}{2}$ and the result.

□

Theorem 6. *In* ***online clairvoyant settings****,* $\forall N > 0, \forall \text{ALG}, \exists n \geq N, R_{\text{OPT}}(\text{ALG}_n) > \frac{\log(n) - \log(\log(n))}{4}$*. In other words, for any* $\mu > 0$*, no algorithm (polynomial or not) may be* $\frac{\log(n)-\log(\log(n))}{4}$*-competitive for all n large enough.*

Proof. We notice the total number of tasks in $Z_P^{\text{ALG}}(K)$ is, by Definition 2 and Definition 8, $P\left(\sum_{i=0}^{P-1} 2K^{P-i-1}\right) = P\frac{2K^P-2}{K-1}$. With $K = 2$, we find $n = 2P(2^P - 1)$, thus $2^P \leq n < 2P2^P$ and $P \leq \log(n) < P + \log(2P)$, therefore $\log(n) - \log(\log(2n)) < P \leq \log(n)$.

Let $N > 0$, and an algorithm ALG. We choose $K = 2$ and $P \geq \max(2, \log(N))$ such that $Z_P^{\text{ALG}}(K)$ has $n \geq \max(N, 12)$ tasks. Then, using Lemma 11 and Lemma 12 with $0 < \epsilon < \frac{0.4}{2P(\log(n)-\log(\log(n)))}$, we derive:

$$\begin{aligned}
R_{\text{OPT}}(\text{ALG}_n) &\geq \frac{T_{alg}(Z_P^{\text{ALG}}(K))}{T_{\text{OPT}}(Z_P^{\text{ALG}}(K))} \\
&> \frac{P^2K^{P-1} - P(P-1)K^{P-2}}{2PK^{P-1} + 2P^2K^P\epsilon} = \frac{P - \frac{P-1}{K}}{2 + 2PK\epsilon} = \frac{P+1}{4(1+2P\epsilon)} \\
&> \frac{\log(n) - \log(\log(2n)) + 1}{4(1+2P\epsilon)} = \frac{\log(n) - \log(\log(n)+1) + 1}{4(1+2P\epsilon)} \\
&> \frac{\log(n) - \log(\log(n)) + 0.4}{4(1+2P\epsilon)} \qquad \text{because } \log(x+1) \leq \log(x) + 0.6 \text{ for } x \geq 2 \text{ applied with } n \geq 12 \\
&= \frac{(\log(n) - \log(\log(n)))\left(1 + \frac{0.4}{\log(n)-\log(\log(n))}\right)}{4(1+2P\epsilon)} \\
&> \frac{\log(n) - \log(\log(n))}{4} \qquad \text{because } \frac{0.4}{2P(\log(n) - \log(\log(n)))} > \epsilon
\end{aligned}$$

□

5.5 Alternate versions of the algorithm for practical use

In practice, the algorithm BATCH described in Section 5.2 is not efficient, because of all the idle time between the categories. Although it is close to optimal in terms of competitive ratio as shown in Section 5.3 and Section 5.4, it is not likely to have any practical interest. To fix this, first, let's define a certain category of algorithms:

Definition 9. *We say an algorithm is* ***greedy*** *if it follows the following rule : At any given time, if there are enough processors idle to process at least one task that is ready, then some task must be scheduled (we don't wait)*

A classical definition of **greedy** algorithms is given in Algorithm 4. In practice, using those kind of algorithms is very reasonable. Intuitively, because we shouldn't leave processors idle when we can process tasks, especially in online settings when we never have real reason to do so except imagining what could be the worst possible tasks that may come next, which is unlikely to happen. More precisely, it is reasonable because of the following result

Algorithm 4 GreedyList (Scheduling Strategy)

```
$L = \emptyset$
$p_{avail} \leftarrow P$
repeat
    if at time 0 or a running task $\mathcal{T}_k$ completes execution then
        Add all new available tasks to $L$
        Organize all jobs in $L$ according to some priority rule
        $p_{avail} \leftarrow p_{avail} + p_k$
        for $j = 1, \dots, |L|$ do
            $\mathcal{T}_j \leftarrow L(j)$
            if $p_{avail} \geq p_j$ then
                execute job $\mathcal{T}_j$ at the current time
                $p_{avail} \leftarrow p_{avail} - p_j$
                remove $\mathcal{T}_j$ from $L$
            end if
        end for
    end if
until All tasks are completed
```

Theorem 7. *If for all tasks $\mathcal{T}_i$, $p_i \leq \frac{P}{2}$, then any **greedy** algorithm is 3-competitive.*

Proof. Known, will add citation. ☐

This assumption is very reasonable in most practical scenarios, therefore we should use this greedy settings. However, a good algorithm in the worst case may not be **greedy** because of the following negative result:

Theorem 8. *Any **greedy** algorithm that never terminates tasks has a competitive ratio of at least $n/3$*

Proof. We consider a graph of $n = 3P$ tasks, P tasks $(a_1, \dots, a_P)$ of length ϵ with $p(a_i) = 1$, P tasks $(b_1, \dots, b_P)$ of length ϵ with $p(b_i) = P$, and P tasks $(c_1, \dots, c_P)$ of length 1 with $p(c_i) = 1$. The precedence constraints are between a_i and b_i for all $i > 0$, and between b_i and a_{i+1} for all $i < P$. The optimal schedule is processing all the tasks a_i and b_i alternatively, which takes time $2P\epsilon$, then all tasks c_i in parallel which takes time 1. However, if the algorithm is greedy, it will first launch tasks a_1 and c_1 in parallel, and may not launch b_1 before c_1 ends. Similarly, whenever a task b_i is completed for $i < P$, a greedy heuristic will launch a_{i+1} and c_{i+1} and task b_{i+1} will not start before the end of c_{i+1}. Therefore the total execution time T verify;

$$\begin{aligned}
\frac{T}{T_{\text{Opt}}} &= \frac{(1+\epsilon)P}{1+2P\epsilon} \\
&= \frac{(1+2P\epsilon)P - (2P^2 - P)\epsilon}{1+2P\epsilon} \\
&> P - 2P^2\epsilon \\
&= \frac{n}{3} - 2P^2\epsilon
\end{aligned}$$

Hence the result since we can choose ϵ arbitrarily small. ☐

With that in mind, the objective of the section is to design several versions of an algorithm, playing with the trade-off between efficiency and safeness in the worst case. In general, our algorithm is defined in Algorithm 5: it is the same as the greedy algorithms with a twist: when a certain condition is met, the scheduler will interrupt tasks, put them back to the list, and start new tasks instead. For simplicity, we assume that it takes no time to interrupt tasks and schedule others. Note that the lower bound provided in Theorem 6 still holds in this setting, therefore it is still impossible to be always within a factor $\frac{log(n)-log(log(n))}{4}$ even when allowing task termination.

The only difference between our heuristics will be the condition InterruptionCondition where we would terminate tasks, and which tasks to terminate if this condition is reached. The trade-off is as follow, if the condition is very strict, with a lot of tasks termination, the schedule will follow pretty closely the ordering specified by the categories and this would be very safe in term of worst case scenario. However, interrupting many tasks will lose efficiency in most cases. On the other hand, if we do little to no interruptions, none of the results obtained in this paper will hold. In the following we give few examples of possible conditions, assuming the list of ready tasks L is not empty (otherwise there is no reason to terminate tasks):

Algorithm 5 GreedyListCondition

```
1: $L = \emptyset$
2: $p_{avail} \leftarrow P$
3: repeat
4:     if at time 0 or a running task $\mathcal{T}_i$ completes execution then
5:         Add all new ready tasks to $L$
6:         Organize all jobs in $L$ by increasing $\zeta$, breaking ties with any arbitrary priority rule
7:         $p_{avail} \leftarrow p_{avail} + p_i$
8:         repeat
9:             for $j = 1, \dots, |L|$ do
10:                $\mathcal{T}_j \leftarrow L(j)$
11:                if $p_{avail} \geq p_j$ then
12:                    execute job $\mathcal{T}_j$ at the current time
13:                    $p_{avail} \leftarrow p_{avail} - p_j$
14:                    remove $\mathcal{T}_j$ from $L$
15:                end if
16:            end for
17:            if InterruptionCondition then
18:                Terminate some of the tasks and add them to $L$.
19:            end if
20:        until not InterruptionCondition
21:    end if
22: until All tasks are completed
```

5.5.1 Condition 1: Safest

We define the first condition as follows:

- Given ζ_{min} the smallest category in the tasks in L (ready and waiting to be scheduled), if some tasks are running with a category strictly larger than ζ_{min}
- And, if terminating those tasks would free enough processors to launch one of the tasks of category ζ_{min}
- Then, we terminate enough task to launch one or more tasks of category ζ_{min}

Advantage: although we are a little more flexible than Batch and allow backfilling of some tasks with higher category, we still give highest priority to the tasks of smallest category by terminating all other tasks that could delay the processing of this category. Therefore all the results in Section 5.3 hold.

Drawback: there will be a lot of task termination, losing time in most cases.

Lemma 13. *With this condition, the algorithm is* $3 + log(n)$*-competitive.*

5.5.2 Condition 2: Good trade-off in theory

We define the second condition as follow, here we define as ζ^P_{min} the smallest category of tasks currently being processed, and ζ^L_{min} as the smallest category of tasks in L, then $\zeta_{min} = \min(\zeta^P_{min}, \zeta^L_{min})$.

- If there exists tasks of category ζ_{min} in L
- And, if less than $\frac{P}{4}$ processors are being utilized processing tasks of category ζ_{min}
- And, if terminating some tasks currently running of category strictly larger than ζ_{min} would free enough processors to launch one of the tasks of category ζ_{min} in L
- Then, we terminate the least possible amount of tasks to launch either all tasks of category ζ_{min}, or enough such that at least $\frac{P}{4}$ processors become utilized processing tasks of category ζ_{min} (for instance, the choice of which task to kill may aim at minimizing the work lost, although finding the exact set minimizing work lost is NP-complete).

Advantage: we have both theoretical results in the worst case, and if all tasks require a small number of processors (i.e. $\forall i, p_i \leq \frac{P}{4}$, see Lemma 14 and Lemma 15).

Drawback: probably still too much task termination, in practice it could be worse than a greedy algorithm.

Lemma 14. *With this condition, the algorithm is* $5 + log(n)$*-competitive.*

Proof. In short, we had for any instance $\mathcal{I}$, $T_{\text{Batch}}(\mathcal{I}) \leq 2\frac{\mathcal{A}(\mathcal{I})}{+}(\log(n)+1)\mathcal{C}(\mathcal{I})$ because either we were using $\frac{P}{2}$ processors or all tasks in the category that we are currently processing that requires less than $\frac{P}{2}$ have started, which may not continue for a time longer than L_ζ, the longest task of the category. The first area term comes from the fact that when we use more than $\frac{P}{2}$ processors, we may not be twice as more efficient, and the other cases represent at most $(log(n)+1)\mathcal{C}(\mathcal{I}) \geq \sum_\zeta L_\zeta$ time. With this version of the heuristic, this fact is still true but with $\frac{P}{4}$ instead of $\frac{P}{2}$, so when we use at least $\frac{P}{4}$ processors, we may not be 4 times more efficient. Hence, we get $T_{\text{Batch}}(\mathcal{I}) \leq 4\frac{\mathcal{A}(\mathcal{I})}{P} + (\log(n)+1)\mathcal{C}(\mathcal{I}) \leq (5 + log(n))T_{\text{Opt}}$ □

Lemma 15. *With this condition, and if* $\forall i, p_i \leq \frac{P}{4}$ *the algorithm is* 5*-competitive.*

Proof. Not easy or short, will add if we keep this section □

5.5.3 Condition 3: Avoiding disasters

Again, ζ_{min} is the smallest category within the tasks ready in L

- If less than $\frac{P}{2}$ processors are currently used
- And, if all tasks currently running are in categories strictly larger than ζ_{min}
- Then, terminate enough task to launch at least one task of Category ζ_{min}

The motivation for this condition is the following: if it is true, we don't have a good processor utilization, and we process only tasks with higher categories. If the tasks we are processing now don't have successors, it's a disaster since the tasks were not at all critical, and this situation could happen again and again since we don't work on tasks near the start of the graph. The fact that this can happen repeatedly is the reason why the execution time of a greedy algorithm may explode in the worse case. Unfortunately, this condition is not enough to claim any competitive ratio in the worse case, as it is still possible to design instances that breaks it. However, those instances are very ridiculous and this condition should be enough to be mostly protected from bad scenarios.

Additionally, note that in practice the termination condition should never happen, because it means that all tasks in L request more than $\frac{P}{2}$ processors (otherwise there would be enough processors to launch them). So in real settings where no task may fill the whole system, this would behave as any greedy algorithm, with a likely decent priority rule. Indeed, under those settings, one of the worse situation is to end up with only the longest path at the end, resulting in a very poor processor utilization during the processing of this path. With our ordering by category, we try to process the graph by order of closeness to the beginning of the graph, therefore we should always make progress for all paths, including the longest one. In online settings, it's not possible to process the longest path faster than the others, so trying to balance all paths seems reasonable.

Advantage: never terminates task in reasonable settings.

Drawback: no theoretical garanties in general.

Lemma 16. *With this condition, and if* $\forall i, p_i \leq \frac{P}{2}$*, the algorithm is* 3*-competitive.*

Proof. In this case, the scheduler never terminates tasks so it is greedy. □

In short, probably a good condition in practice, but no more theoretical results in the worse case.

5.5.4 Condition 4: Never terminate tasks

Of course, we should consider never interrupting tasks as a possible variation, but there is not much to say here.

Lemma 17. *With this condition, the algorithm is no better than* $\frac{n}{3}$*-competitive*

Lemma 18. *With this condition, and if* $\forall i, p_i \leq \frac{P}{2}$*, the algorithm is* 3*-competitive*

Proof. Both of these lemmas hold because in this case we are greedy with no termination. □

6 Online semi-clairvoyant scheduling (completion time unknown)

Lucas: This section is currently inconsistent with the notations introduced in Section 5.1 and therefore shouldn't be read yet.

In this setting recall that we assume a new task $\mathcal{T}_i$ becomes known to the scheduler whenever all its predecessors have been completed. Additionally, the exact length of the task is unknown but here we assume we are provided an estimator t_i^* of the length t_i, and a margin of error e_i such that $t_i \in \left[\frac{t_i^*}{e_i}, e_i t_i^*\right]$, and we define $E = \max_i e_i$.

6.1 Extension to Batchest in semi-clairvoyant settings

Here, we may not compute the category of a task when it arrives, as we only know s^∞. To be able to provide a competitive ratio, we want our algorithm to validate the following two properties, so that the general ideas in the analysis of previous sections may be adapted:

- If a task is allocated a category $\zeta = \lambda 2^\chi$, then we may not let it run for more than $2^{\chi+1}$ unit of time. If it has not been completed by then, we must terminate it, and allocate it to another (larger) new category
- When we start processing a category, all the tasks within this category must be ready (and independent)

There is a simple way to achieve this although it requires task termination: for all tasks $\mathcal{T}_i$, we are given their criticality s_i^∞, their length estimate t_i^*, and their error margin $e_i \leq E$. We first compute their smallest possible category, if the tasks had length $\frac{t_i^*}{e_i}$. We then apply Batch, but whenever a task that is running in a category would become to long for the category to be well chosen, we terminate it. This maximum possible time of execution is $2^{\chi_i} + \zeta_i - s_i^\infty$, since if $t > 2^{\chi_i} + \lambda 2^{\chi_i} - s_i^\infty$, then $f_i^\infty = s_i^\infty + t_i > (\lambda_i + 1)2^{\chi_i}$; and the category is no longer acceptable (Lemma 4). When this situation happens, we assign it to a larger category for a new execution, where it will be able to run for at least twice as long, since the next category will have a higher power level χ. More specifically, if a task of category $\zeta = \lambda 2^\chi$ and label $[a|b]$ doesn't terminate in time, we interrupt it and allocate it to a new category $\zeta' = b > \zeta$, from which we can easily derive λ' by dividing by 2 until we get an odd number, and $\chi' > \chi$. Finally, in this next category, we will try to complete this task during at most $t = 2^{\chi'} + \zeta' - s^\infty$, before interrupting it again and try on yet another category. On the tree of categories, it corresponds to moving to the closest node to the right among the nodes in a higher row. The competitive ratio will be impacted because we have multiple processing of all task, increasing both the area and the number of tasks, but we will see in Section 6.2 the increase is competitive ratio is limited for reasonable error margin E.

Note that this algorithm is not very resilient, since although it is fine if the task happens to be longer than $t_i^* e_i$, if it is shorter than expected and terminates too soon, it might release tasks from smaller categories (more specifically situated in the left branch of our current category) which themselves could release tasks from our current category which would break the algorithm principle. For instance if we are at time 0 and are given a task whose minimum completion time is supposed t be 1.1, we will try it in the category 1. If its actual completion time is 0.1, maybe it will unlock multiple tasks of categories strictly smaller than 1, which themselves will unlock new tasks of category 1, that will have to be processed again. Therefore, although the higher bound is not important the algorithm must be provided with a real lower bound.

Lucas: Because of this, alternatively we can ask for a lower bound l_i and use $E = \max(t_i/l_i)$ in the analysis, but maybe the actual version is more consistent with the literature.

Lucas: Of course in practice there are ways to make the algorithm resilient. An easy one is to simply "pretend" the task is of length $\frac{t_i^*}{e_i}$ if it happens to be shorter than expected when computing the categories of its successors, to avoid loops. However it breaks the analysis. We can probably remove notes and previous paragraph when we submit

Based on the previous paragraph, in Algorithm 6 and 7, we present an updated version of Batch in semi-clairvoyant settings, and highlight in blue the differences with the previous versions. Note that we still use ComputeCat, but each time with a different time estimate.

6.2 Analysis

In this section, we will show the competitive ratio of Batchest as well as another tighter result if the estimates of the tasks are bounded, i.e. $\exists m, M, \forall i, m \leq \frac{t_i^*}{e_i} \leq t_i^* e_i < M$. The results are the following, with $E = \max_i e_i$:

- $\forall \mathcal{I}, \frac{T_{\text{Batchest}}(\mathcal{I})}{T_{\text{Opt}}(\mathcal{I})} \leq 8 + \log(n) + \log(\log(E) + 1)$ (compared to $3 + \log(n)$ in clairvoyant settings)

Algorithm 6 *scheduleIndepSemiClairvoyant*(L, χ)

```
1: Sort L by increasing $p_i$, breaking ties with any arbitrary priority rule
2: $M \leftarrow \emptyset$    (List of tasks currently processing)
3: $R \leftarrow \emptyset$    (List of tasks that we will discover in this batch)
4: $F \leftarrow \emptyset$    (List of tasks that will fail completion in this batch)
5: while $L$ or $M$ is not empty do
6:     if $p_{\text{avail}} \geq p(L[0])$ then
7:         Schedule $L[0]$
8:         $M \leftarrow L[0]$
9:         $p_{avail} \leftarrow p_{avail} - p(L[0])$
10:        delete $L[0]$
11:    end if
12:    if a task $\mathcal{T}$ is discovered then
13:        $R \leftarrow \mathcal{T}$
14:    end if
15:    if a task $\mathcal{T}$ is finished then
16:        remove $\mathcal{T}$ from $M$
17:        Update $f^\infty$
18:        $p_{avail} \leftarrow p_{avail} + p(\mathcal{T})$
19:    end if
20:    if a task $\mathcal{T}_i$ is running for $t_i$ units of time then
21:        remove $\mathcal{T}_i$ from $M$
22:        $t_i \leftarrow \zeta + 2^{\chi+1} - f_i^\infty$    (to make sure it will be put to the next category)
23:        $F \leftarrow M$
24:    end if
25: end while
26: return $R, F$
```

Algorithm 7 BATCHEST (semi-clairvoyant)

```
1: $A \leftarrow \emptyset$
2: for all $\mathcal{T}_i$ ready do
3:     $R \leftarrow \mathcal{T}_i$
4: end for
5: repeat
6:     for all $\mathcal{T}_i$ in R do
7:         $t_i \leftarrow t_i^*/e_i$
8:         $\zeta_i \leftarrow$ COMPUTECAT$(\mathcal{T}_i)$
9:         $t_i \leftarrow \zeta_i + 2^{\chi_i} - s_i^\infty$    (We try until the maximal possible time so that the task fit in the category)
10:        $A(\zeta_i) \leftarrow \mathcal{T}_i$    ($A(\zeta_i)$ represents the list of tasks of category $\zeta_i$)
11:    end for
12:    for all $\mathcal{T}_i$ in F do
13:        $Compute\_data(\mathcal{T}_i)$
14:        $A(\zeta_i) \leftarrow \mathcal{T}_i$    ($A(\zeta_i)$ represents the list of tasks of category $\zeta_i$)
15:    end for
16:    Find $\zeta_{min}$, smallest category of tasks available
17:    $R, F = scheduleIndepSemiClairvoyant(A(\zeta_{min}))$
18:    delete $A(\zeta_{min})$
19: until all tasks are scheduled
```

- $\forall \mathcal{I}, \frac{T_{\text{Batchest}}(\mathcal{I})}{T_{\text{Opt}}(\mathcal{I})} \leq 10 + \log\left(\frac{M}{m}\right)$ (compared to $6 + \log\left(\frac{M}{m}\right)$ in clairvoyant settings)

One way to show these results is to get into the details of the analysis again and adapt our previous proofs. However, this is not how we tackle this. Instead, given an instance $\mathcal{I}$, we will show that the schedule of Batchest on this instance will be identical to the schedule of Batch on a similar instance $\mathcal{I}'$ in clairvoyant settings, where $\mathcal{I}'$ has more tasks $n' > n$ and a larger area $\mathcal{A}(\mathcal{I}') > \mathcal{A}(\mathcal{I})$, accounting for the interruptions and re-executions of Batchest. We will then simply plug in the ratios obtained in Section 5.3 and replace the area and number of tasks by our initial parameters.

Before presenting the transformation, we need one last definition:

Definition 10. *Given a category $\zeta = \lambda 2^{\chi}$ we define its **serial extension** $\zeta^{\infty} = \{\zeta_1, \zeta_2, \cdots,\}$, its corresponding power level serial extension $\chi^{\infty} = \{\chi_1, \chi_2, \cdots,\}$ and longitude extension $\lambda^{\infty} = \{\lambda_1, \lambda_2, \cdots\}$ as follows:*

- $\zeta_1 = \zeta$, $\lambda_1 = \lambda$, $\chi_1 = \chi$
- $\forall i > 0, \zeta_{i+1} = (\lambda_i + 1)2^{\chi_i}$
- $\forall i > 0, \chi_{i+1} = \max\{\chi \in \mathbb{N}, \frac{\zeta_{i+1}}{2^{\chi}} \in \mathbb{N}\}$
- $\lambda_{i+1} = \frac{\zeta_{i+1}}{2^{\chi_{i+1}}}$

*Furthermore, given an instance $\mathcal{I}$ and a task $\mathcal{T}$ whose lower bound on time of execution is $\frac{t^*}{e}$ with category ζ (based on this lower bound) and criticality s^{∞}. Then if its real time of execution is t, we define as its **category tries** $\mathcal{T}^{\zeta} = \{\zeta \in \zeta^{\infty}, \zeta < s^{\infty} + t\}$. This set corresponds to all the different execution of this task by* Batchest*, one per category in $\mathcal{T}^{\zeta}$.*

For example, if a task $\mathcal{T}$ has a criticality $s^{\infty} = 10$ and a minimum completion time of $\frac{t^*}{e}$ which corresponds to a category $\zeta = 10.5$, then $\zeta^{\infty} = \{10.5, 11, 12, 16, 32, \cdots\}$, $\chi^{\infty} = \{-1, 0, 2, 4, 5, \cdots\}$, $\lambda^{\infty} = \{23, 11, 3, 1, 1, \cdots\}$. Assuming its actual time of execution is $t = 10$, then $\mathcal{T}^{\zeta} = \{10.5, 11, 12, 16\}$, and we know this task will have to be tried 4 times. The first try will be in category 10.5 of label [10|11] and the task will be interrupted after 1 units of time, the second try in [10|12] will be interrupted after 2 units of time, the third try in [8|16] will be interrupted after 6 units of time (and not 8 since we have $s^{\infty} = 10$), and the last try in [0|32] will be executed until completion.

We move on to the transformation using $\mathcal{T}^{\zeta}$; given an instance $\mathcal{I}$, we define $\mathcal{I}'$ as follows:

- For all tasks $\mathcal{T}$ in $\mathcal{I}$ of length t, processor allocation p, criticality s^{∞}, containing $j \geq 1$ category tries $\mathcal{T}^{\zeta} = \{\zeta_1, \zeta_2, \cdots, \zeta_j\}$ of corresponding power levels $\{\chi_1, \chi_2, \cdots, \chi_j\}$, we add j tasks $\{\mathcal{T}_1, \mathcal{T}_2, \cdots, \mathcal{T}_j\}$ to $\mathcal{I}'$. For $i < j$, $\mathcal{T}_i$ has length $2^{\chi_i} + \zeta_i - s^{\infty}$, and $\mathcal{T}_j$ has length t. All of them have processor allocation p.
- Given a precedence constraint between two tasks $\mathcal{U}$ and $\mathcal{V}$ in $\mathcal{I}$, containing u and v category tries respectively, we add a precedence constraint, for all $j \leq v$, between $\mathcal{U}_u$ and $\mathcal{V}_j$ in $\mathcal{I}'$

Remark 2. *We are currently doing a performance analysis, so regardless of the setting, we can use all the data from the task, including the criticalities, categories, etc, to design any other instance and compute bounds on execution time.*

An example on this transformation is given in Figure 11, and the corresponding data and categories of the tasks are given in Table 2

Remark 3. *Both* ScheduleIndep *and scheduleIndepSemiClairvoyant arrange tasks of a certain categories by increasing processors before greedily scheduling them. We didn't define a precise way to break ties, and in the following analysis we will need them to be predictable. Therefore we assume in the following that ties are broken using a deterministic priority value linked to tasks, that we transfer from a task $\mathcal{T}$ in $\mathcal{I}$ to its copies in $\mathcal{I}'$.*

Lemma 19. *The resulting schedule of $\mathcal{I}$ using* Batchest *in semi-clairvoyant settings is exactly the same as the schedule of $\mathcal{I}'$ using* Batch *in clairvoyant settings. More precisely the failed attempts of a task $\mathcal{T}$ with j category tries in $\mathcal{I}$ (semi-clairvoyant settings) will have the exact same start date and termination date as the processing of tasks $\mathcal{T}_1, \cdots, \mathcal{T}_{j-1}$ in $\mathcal{I}'$ (clairvoyant settings). Additionally, the successful execution of $\mathcal{T}$ in $\mathcal{I}$ has exactly the same start date and completion date as the processing of task $\mathcal{T}_j$ in $\mathcal{I}'$.*

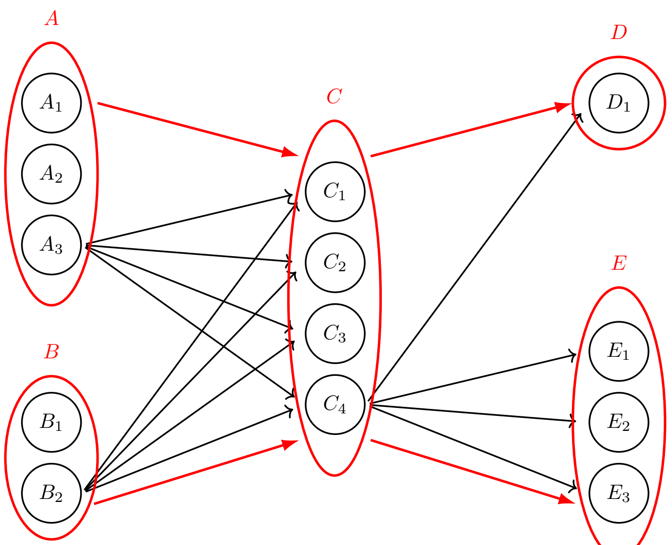


Figure 11: In red, representation of $\mathcal{I}$, in black, representation of $\mathcal{I}'$

Task in $\mathcal{I}$	t_{min}	t	s^∞	f^∞	Task in $\mathcal{I}'$	t	ζ	Label of ζ
A	3	9	0	9	A_1	4	2	[0|4]
					A_2	8	4	[0|8]
					A_3	9	8	[0|16]
B	5	10	0	10	B_1	8	4	[0|8]
					B_2	10	8	[0|16]
C	1	10	10	20	C_1	1	10.5	[10|11]
					C_2	2	11	[10|12]
					C_3	6	12	[8|16]
					C_4	10	16	[0|32]
D	5	10	20	30	D_1	10	24	[16|32]
E	3	15	20	35	E_1	4	22	[20|24]
					E_2	12	24	[16|32]
					E_3	15	32	[0|64]

Table 2: Tasks data

Proof. By construction, each tries of each task of the instance $\mathcal{I}$ using BATCHEST has the same length as it's corresponding task in $\mathcal{I}'$. They also share the same criticality, since we added for each task $\mathcal{T}$ a precedence constraints of its longest copy in $\mathcal{I}'$ that matches the length $\mathcal{T}$, therefore all their data (category, power level, etc.) is identical since they derive from criticality and length.

Additionally, when a task fails, it will be retried in a category always strictly superior, and identically all tasks $\mathcal{T}_1, \mathcal{T}_2, \cdots, \mathcal{T}_j$ have increasing category. Based on this fact, we proceed to a quick induction to show the following property: "the $i-th$ call to SCHEDULEINDEP and *scheduleIndepSemiClairvoyant* contains the same number of tasks that share the same parameters (processor allocation, length, tie breaker), thus will take the same decision".

Initially, the number of task in the smallest category is identical since $\mathcal{T} \in \mathcal{I}$ is in the smaller category if and only if $\mathcal{T}_1 \in \mathcal{I}'$ is, and we have already seen that their parameters are identical by construction. The only difference between SCHEDULEINDEP and *scheduleIndepSemiClairvoyant* is the way it handles data when a task terminates or fail, so the fact that the input is identical implies that the decisions are. Finally, to show the heredity, we simply notice that it is still true that all the tasks of a category are ready when the category starts (since re-execution always are allocated to another strictly bigger category), so again, SCHEDULEINDEP and *scheduleIndepSemiClairvoyant* will be called with the same number of tasks, the same parameters, and will execute the same schedule.

Lucas: Not too satisfied with this proof, honestly is clear and obvious to me, just because of the way $\mathcal{I}'$ is constructed, but it's always hard to give clear arguments when it's not just equations. Is it convincing?

□

Lemma 20. *The total area of the instance $\mathcal{I}'$ is at most three times the total area of the instance $\mathcal{I}$, $\mathcal{A}(\mathcal{I}') < 3\mathcal{A}(\mathcal{I})$*

Proof. To prove this result, we will show that for any task $\mathcal{T}$ in $\mathcal{I}$ of length t that has $j = |\mathcal{T}^{\zeta}|$ category tries, corresponding to tasks $\{\mathcal{T}_1, \cdots, \mathcal{T}_j\}$ of length $t_1, \cdots, t_j$ verify $\sum_{i=1}^{j} t_i < 3t$.

To show that, we start by computing, for $i < j - 1$,

$$t_{i+1} - 2t_i = 2^{\chi_{i+1}} + \zeta_{i+1} - s^{\infty} - 2\left(2^{\chi_i} + \zeta_i - s^{\infty}\right) \geq \zeta_{i+1} - 2\zeta_i + s^{\infty}$$

By definition $(\lambda_i - 1)2^{\chi_i} \leq s^{\infty} < \lambda_i 2^{\chi_i} = \zeta_i < (\lambda_i + 1)2^{\chi_i} = \zeta_{i+1}$. We thus obtain

$$t_{i+1} - 2t_i \geq (\lambda_i + 1)2^{\chi_i} - 2\lambda_i 2^{\chi_i} + (\lambda_i - 1)2^{\chi_i} = 0$$

Because $\forall i < j-1, 2t_i \leq t_{i+1}$, a direct induction shows, $\sum_{i=1}^{j-1} t_i < 2t_{j-1}$. Finally using $t_{j-1} < t_j = t$,we get $\sum_{i=1}^{j} t_i < 3t$

Using this, we can derive

$$\begin{aligned}\mathcal{A}(\mathcal{I}') &= \sum_{\mathcal{T}_i' \in \mathcal{I}'} t_i p_i = \sum_{\mathcal{T} \in \mathcal{I}} \sum_{i=1}^{|\mathcal{T}^{\zeta}|} t_i' p \\ &\leq \sum_{\mathcal{T} \in \mathcal{I}} 3tp \\ &< 3\mathcal{A}(\mathcal{I})\end{aligned}$$

□

Lemma 21. *The longest path of the instance $\mathcal{I}'$ has the same length as the longest path of the instance $\mathcal{I}$, $\mathcal{C}(\mathcal{I}) = \mathcal{C}(\mathcal{I}')$.*

Proof. By construction, all tasks $\mathcal{T}_i$ in $\mathcal{I}'$ are copies of a task $\mathcal{T}$ in $\mathcal{I}$ with same criticality s^{∞} and shorter length $t_i \leq t$. Therefore $\mathcal{C}(\mathcal{I}') \leq \mathcal{C}(\mathcal{I})$. Let $\mathcal{T}$ be the task with highest end criticality (s^{∞},f^{∞}) of $\mathcal{I}$, i.e. $f^{\infty} = \mathcal{C}(\mathcal{I})$, and of length t. If $\mathcal{T}$ has j category tries, then by construction $\mathcal{T}_j$ in $\mathcal{I}'$ has criticality s^{∞} and length t therefore $f_j^{\infty} = f^{\infty} = \mathcal{C}(\mathcal{I})$. This shows $\mathcal{C}(\mathcal{I}') \geq \mathcal{C}(\mathcal{I})$ and the result. □

Lemma 22. *The number of task of $\mathcal{I}'$ is at most a factor $2\log(E)+2$ larger than the one of $\mathcal{I}$, $n' \leq (2\log(E)+2)n$.*

Proof. To show this result, we simply need to prove that the number of tries of any tasks $\mathcal{T}$ in $\mathcal{I}$, is at most $2\log(E)+2$, as this corresponds to its number of copies in $\mathcal{I}'$. We take a task $\mathcal{T}$ with j category tries, length t estimator t^* and error e. If $j \leq 2$, we are good. Otherwise, we saw in the proof of Lemma 20 that $\forall i < j-1, t_i \leq 2t_{i+1}$, therefore $t_{j-1} \geq 2^{j-2}t_1$. By construction, $t_1 \geq \frac{t^*}{e} \geq \frac{t^*}{E}$ and $t \leq t^* e \leq t^* E$, which proves $E^2 t_1 \geq t^* E \geq t > t_{j-1} \geq 2^{j-2} t_1$, hence $j \leq 2\log(E) + 2$ and the result. □

Theorem 9. Batchest *is* $8+\log(n)+\log(\log(E)+1)$*-competitive (compared to* $3+\log(n)$ *in clairvoyant settings)*

Proof. We have to show that $\forall \mathcal{I}$, $T_{\text{Batchest}}(\mathcal{I}) \leq (8+\log(n)+\log(\log(E)+1)\,\mathcal{C}(\mathcal{I})$. Let $\mathcal{I}$ be an instance of n tasks, we build the instance $\mathcal{I}'$ with n' according to our transformation and derive the result as follows:

$$\begin{aligned}
T_{\text{Batchest}}(\mathcal{I}) &= T_{\text{Batch}}(\mathcal{I}') && \text{(Lemma 19)}\\
&\leq 2\frac{\mathcal{A}(\mathcal{I}')}{P} + (\log(n')+1)\mathcal{C}(\mathcal{I}') && \text{(Equation ??)}\\
&\leq 6\frac{\mathcal{A}(\mathcal{I})}{P} + (\log(n')+1)\mathcal{C}(\mathcal{I}) && \text{(Lemma 20, Lemma 21)}\\
&\leq 6\frac{\mathcal{A}(\mathcal{I})}{P} + (\log(2\log(E)+2)+1)\mathcal{C}(\mathcal{I}) && \text{(Lemma 22)}\\
&\leq (8+\log(\log(E)+1))\text{Lb}(\mathcal{I})
\end{aligned}$$

□

Remark 4. *Even if our only information is that tasks length are between a second and 30 years, corresponding to providing an estimator of around 8h30 for all tasks with* $E \approx 30758$*, then* $\log(\log(E)+1) < 4$

Theorem 10. *Let's define an arbitrary instance* $\mathcal{I}$ *and assume* $\forall \mathcal{T}, m \leq \frac{t^*}{e} \leq t \leq t^* e < M$*. Then* $\frac{T_{\text{Batchest}}(\mathcal{I})}{T_{\text{Opt}}(\mathcal{I})} \leq 10+\log\left(\frac{M}{m}\right)$ *(compared to* $6+\log\left(\frac{M}{m}\right)$ *in clairvoyant settings)*

Proof. Again we define an instance $\mathcal{I}$ and its transformation $\mathcal{I}'$, This time we can use Theorem 4 because $\forall \mathcal{T} \in \mathcal{I}, m \leq \frac{t^*}{err} \leq t \leq M$, therefore $\forall \mathcal{T}_i \in \mathcal{I}', m \leq t_i \leq M$.

$$\begin{aligned}
T_{\text{Batchest}}(\mathcal{I}) &= T_{\text{Batch}}(\mathcal{I}') && \text{(Lemma 19)}\\
&\leq 2\frac{\mathcal{A}(\mathcal{I}')}{P} + \left(4+\log\left(\frac{M}{m}\right)\right)\mathcal{C}(\mathcal{I}') && \text{(Equation ??)}\\
&\leq 6\frac{\mathcal{A}(\mathcal{I})}{P} + \left(4+\log\left(\frac{M}{m}\right)\right)\mathcal{C}(\mathcal{I}) && \text{(Lemma 20, Lemma 21)}\\
&\leq \left(10+\log\left(\frac{M}{m}\right)\right)\text{Lb}(\mathcal{I})
\end{aligned}$$

□

7 Virtualization works for any setting

Lucas: Not sure where to put this, but I think it is an interesting perspective

We allow **Virtualization**, i.e. if p processors are requested, we can use any $q < p$ processors and the execution time will be increased by a factor p/q. Assuming the speedup function is not superlinear:

Theorem 11. *There exists a* $\frac{1+\sqrt{5}}{2} \approx 2.618$ *competitive algorithm, that doesn't require any information on the graph or the length of the task (in other words, it is valid in all settings even with* $E=\infty$*).*

Proof. Known result, will add a reference □

Theorem 12. *For any* ϵ*, no algorithm can be* $\frac{1+\sqrt{5}}{2} - \epsilon$ *competitive in semi-clairvoyant settings when* $E=\infty$*.*

Proof. Known result, will add a reference □

In short, these results highlight the importance of allowing flexibility for processing a graph of tasks, as in this case a small constant competitive ratio is obtainable, which is impossible for rigid tasks.

8 Conclusion and Future work

Lucas: Conclusion is AI generated, temporary and for internal use only

This paper presented a comprehensive analysis of rigid scheduling problems in both offline and online settings. We introduced novel algorithms Batch and Batchest for clairvoyant and non-clairvoyant online scheduling respectively, and proved their competitive ratios to be near-optimal. **Our results demonstrate that for rigid scheduling, it is possible to achieve performance close to the theoretical lower bound even with limited knowledge of the task graph.** We also showed that rigid scheduling instances can be inherently difficult to schedule efficiently, even when the request is high and a large number of tasks are waiting to be scheduled.

The key theoretical contributions include:

- Proving lower bounds on optimal completion times in offline settings compared to the maximum between the longest path and the area of the instance
- Developing $(3 + \log(n))$-competitive and $(8 + \log(n) + \log(\log(E) + 1))$-competitive algorithms for online clairvoyant and non-clairvoyant settings respectively
- Establishing lower bounds on the best possible competitive ratios achievable by any algorithm

Additionally, we explored practical variants of our algorithms that balance worst-case guarantees with average-case performance. The analysis provides insights into the inherent challenges of rigid scheduling and the limits of scheduling efficiency in various settings.

While this paper provides a strong theoretical foundation, there are several promising directions for future work:

1. **Simulations and empirical evaluation:** Implementing the proposed algorithms in simulation environments to assess their practical performance across a wide range of synthetic and real-world workloads. This would help validate the theoretical results and identify any gaps between theory and practice.
2. **Integration with HPC systems:** Adapting and implementing the algorithms in actual high-performance computing environments to evaluate their effectiveness in real-world scenarios. This would involve addressing system-specific constraints and optimizing for practical considerations.
3. **Extension to heterogeneous systems:** Expanding the model and algorithms to account for heterogeneous computing resources, which are increasingly common in modern HPC environments.
4. **Dynamic processor allocation:** Investigating models that allow for some flexibility in processor allocation, bridging the gap between fully rigid and fully malleable tasks.
5. **Incorporation of energy efficiency:** Extending the optimization objectives to include energy consumption alongside makespan, which is a critical concern in large-scale computing systems.
6. **Online learning approaches:** Developing adaptive versions of the algorithms that can learn from historical workloads to improve future scheduling decisions.
7. **Fault tolerance:** Extending the model to account for potential processor failures and developing robust scheduling strategies.
8. **Multi-objective optimization:** Considering additional objectives beyond makespan, such as fairness, priority, and deadline constraints.

These future directions would further enhance the practical applicability of the theoretical results presented in this paper and contribute to more efficient resource utilization in high-performance computing environments.

Lucas: That's is a lot of ideas for sure, perhaps the plan is to keep 1 and 4