A New Algorithm for Online scheduling of Rigid Task Graphs with Near-Optimal Competitive Ratio

Lucas Perotin
Vanderbilt University
lucas.perotin@vanderbilt.edu

Hongyang Sun
University of Kansas
hongyang.sun@ku.edu

Padma Raghavan
Vanderbilt University
padma.raghavan@vanderbilt.edu

Abstract

This paper addresses the challenges of online scheduling within high-performance computing (HPC) systems, focusing on rigid parallel tasks with precedence constraints organized as a directed acyclic graph (DAG). We introduce an online algorithm, called Batch, which efficiently schedules tasks to minimize the overall completion time, or the makespan. We show that Batch achieves a competitive ratio of $3+\log(n)$, with n being the number of tasks. Although Batch only discovers tasks on the fly when they are ready, it almost matches the best offline algorithm, which has an approximation ratio of $2+\log(n+1)$. We further show that Batch achieves a constant competitive ratio when task lengths are bounded. Finally, our analysis indicates the algorithm's near-optimal performance in worst-case scenarios, showing that no online algorithm can have a competitive ratio lower than $\Theta(\log(n))$ in this context.

1 Introduction

Efficient resource allocation and task management are fundamental challenges in modern computational systems, particularly in high-performance computing (HPC) systems. Designing scheduling algorithms for these systems becomes increasingly critical as they grow in scale and complexity. This paper delves into a scheduling problem known as the online scheduling of rigid tasks with precedence constraints.

In rigid task scheduling, we consider a set of n tasks to be executed on a platform consisting of P identical processors. A rigid task is characterized by its execution time (or length) t and processor requirement p. The tasks are organized in a directed acyclic graph (DAG), where edges represent precedence constraints. If a task i needs the execution result of another task j before it can be launched, it is represented by a precedence constraint from j to i in the graph. The objective is to assign start times to tasks to minimize the overall completion time, or the makespan, while respecting the platform's capacity (i.e., the total number of processors P) as well as the precedence constraints among the tasks.

In the offline version of the scheduling problem, the graph and each task's parameters (i.e., t and p) are known before the execution starts. The problem has been widely studied, particularly for the special case where all tasks are sequential (hence, require a single processor). Since the problem is $\mathcal{NP}$-complete [16], and the goal is to design good approximation algorithms. On the contrary, in the online version of the problem, tasks are released on the fly, and their existence is unknown to the scheduler until they are ready to be executed (i.e., all of their predecessors are completed). In this case, to evaluate the quality of a heuristic, a standard way is to derive its competitive ratio, which represents the performance of a scheduling algorithm against an optimal offline scheduler that knows in advance all the tasks and their dependencies in the graph. More precisely, the competitive ratio is established against all possible strategies devised by an adversary trying to force the online algorithm to take *bad* decisions. In this work, we will consider the competitive ratio with respect to the size of the instance: the number of tasks n. An algorithm is said to be $f(n)$-competitive if the makespan of any instance with n tasks is always within a factor $f(n)$ from the optimal offline scheduler.

Online task scheduling has been widely studied in different contexts (see Section 2 for a sample of results). However, to the best of our knowledge, the online scheduling of rigid task graphs has not yet been tackled. In this context, we aim to design a heuristic that achieves a decent competitive ratio. At first, designing such heuristics seems hopeless, as illustrated in Figure 1. Here, we consider a simple DAG with P repetitions of three tasks: A, B, and C. For each repetition, B must be processed after A, and the next A and C are released after the completion of B. Moreover, A and B have length ϵ, and C has length 1. Finally, A and C each use a single processor, while B requires all P processors. In the online setting, the scheduler is only aware of the tasks that are ready to be executed.

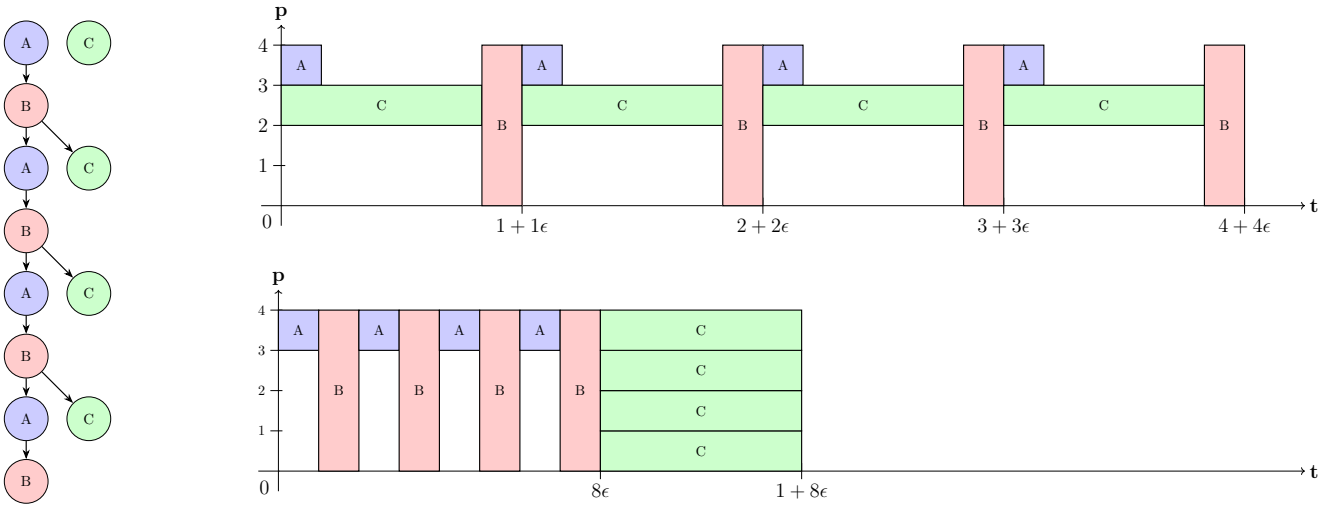


Figure 1: An introductory example with $P = 4$ processors. Left: DAG; Top right: schedule for any ASAP heuristic; Bottom right: an optimal schedule.

Therefore, in the beginning, only the first A and C are known, and the first B is discovered after A completes its execution. This simple example shows that if we process tasks as soon as possible (ASAP), we will always have to wait until each long task C is completed before launching B and unlocking the next repetition for a total makespan of $P + P\epsilon$, utilizing only around 1 processor on average. An optimal schedule will process the A's and B's first, resulting in a makespan of $1 + 2P\epsilon$, which is almost a factor of $P = \frac{n}{3}$ lower, and it uses all the processors most of the time. The only way to avoid such scenarios would be to wait on purpose, not launch tasks ASAP, and delay the execution of C. Since we study competitive ratios in the worst case against an adversary, it is counter-intuitive to wait, as an adversary would only release new tasks after completing their precedentor tasks, thus delaying the execution of any task does not seems to be a good idea. Intuitively, the only way to achieve a decent result would be to relax the online scheduling setting, for example, by allowing task termination.

However, in this paper, we design a novel algorithm, called Batch, and show that it can achieve near-optimal competitive ratio under this stringent online setting. The following summarizes our main results:

- Batch is $(3 + \log(n))$-competitive for the online scheduling of rigid task graphs.[1]
- Batch has a constant competitive ratio if tasks have bounded lengths. Specifically, the ratio is $6 + \log\left(\frac{M}{m}\right)$, where M is the length of the longest task, and m is the length of the shortest task.
- No algorithm can be $\frac{\log(n)-\log(\log(n))-1}{4}$-competitive for $n \geq N$, where N is arbitrarily large. This lower bound shows that Batch achieves the near-optimal asymptotic competitive ratio of $\Theta(\log(n))$.
- No algorithm can be $\left(\frac{P}{2} - \mu\right)$-competitive for any $\mu > 0$. The instance used in this lower bound has a huge number of tasks, with $P < \log(n)$, hence the result is compatible with the competitive ratio of Batch.

Further, the performance of Batch almost matches the best known result [1] for the offline setting, which has an approximation ratio of $2 + \log(n + 1)$, and 3 if all tasks have equal length (we obtain 6 in this case based on the second result above). Note that the result in [1] was derived for the strip packing problem, where a set of rectangles must be placed without overlap inside a strip of limited width (typically normalized to 1), and the objective is to minimize the height of the strip needed to cover all rectangles. Strip packing represents a very similar problem to rigid task scheduling, with the width and height of a rectangle corresponding to the processor requirement and execution time of a task, respectively. However, there are a couple of subtle differences between the two problems: (1) strip packing requires an allocation of continuous space for each rectangle, while rigid task scheduling can allow free choice of processors for each task; (2) the width of each rectangle in strip packing can be a fractional number in $(0, 1]$, while the processor requirement of a task must be an integer in $[1, P]$. Despite those differences, our algorithm can be slightly modified for the online strip packing problem with precedence constraints, and the analysis and competitive ratio hold for both problems. Our lower bounds also apply to the strip packing context, as the instances

[1] For simplicity, we use $\log(x)$ to denote $\log_2(x)$, and this holds for the entirety of the paper.

use only tasks with 1 or P processors, and therefore, can be adapted to strip packing by using rectangles of widths $\frac{1}{P}$ or 1.

Finally, we point out that our results are not incremental. As seen in the introductory example, to achieve such results, we have to strategically delay some tasks, which can only be done based on their relative positions in the DAG with respect to other tasks available. To that end, our algorithm is designed based on new attributes of the tasks within a DAG, such as criticality and category, as well as a new technique to analyze the performance of the algorithm, introduced as the L-matrix. Outside of this work, these new tools may lead to improved results in other scheduling contexts, such as the online scheduling of moldable task graphs, where the scheduler can choose the number of processors for each task. Indeed, a recent work derived the best competitive ratios for some specific speedup models among any ASAP schedule with local processor allocation decisions [27]. Therefore, those results could only be improved by considering the position of each task within the DAG, which is especially challenging under the online setting. Our work could help build new solutions for this specific problem, or other related problems that involve graphs being discovered online.

The rest of this paper is organized as follows. Section 2 surveys the related work. Section 3 formally states the problem and gives a few key definitions and notations. Section 4 first defines the new attributes that lead to the conception of Batch before introducing the algorithm. Section 5 derives the competitive ratio of Batch. Section 6 shows the asymptotic optimality of Batch with lower bounds. Finally, Section 7 concludes the paper and discusses future directions.

2 Related Work

In this section, we review some related work on scheduling DAGs of rigid tasks on identical parallel processors to minimize the makespan. Despite being a fundamental scheduling problem with significant importance in HPC, this problem has received limited attention in the scheduling literature. One possible reason is the combination of task rigidness and precedence constraint, which renders the problem difficult for achieving good theoretical bounds. Considering the online setting further complicates the problem. However, relaxing either constraint (i.e., rigidness or precedence) makes the problem more tractable with more optimistic bounds. Thus, we also review some related results for the two relaxed problems.

2.1 Scheduling DAGs of Rigid Tasks

In the seminar work [17], Graham studied the *list scheduling* algorithm, one of the most well-known and widely adopted algorithms in parallel scheduling. The original list scheduling algorithm is designed for sequential tasks (i.e., requiring only one processor), and it works as follows: all the tasks are initially organized in a list with arbitrary priority order, and whenever a processor becomes idle, it takes the first task in the list whose predecessors have been completed and executes the task. Graham showed that list scheduling achieves $(2 - \frac{1}{P})$-approximation for DAGs of sequential tasks, where P is the total number of processors. Li [24] extended list scheduling for DAGs of rigid parallel tasks, and in this case, a task is executed only when there are sufficient processors to process it. List scheduling has a trivial approximation ratio of P (in fact, any scheduling algorithm that does not idle all processors when there are ready tasks is a P-approximation). Li showed that P is also a lower bound on list scheduling's worst-case approximation ratio for many priority orderings. However, if all tasks require at most qP processors for $q \in (0, 1]$, list scheduling has an approximation ratio of $\frac{(2-q)P}{(1-q)P+1}$, which when q becomes sufficiently small converges to Graham's original result [17]. Since list scheduling can be naturally applied to the online setting, in which case only ready tasks are inserted into the list, the approximation ratios above can also be interpreted as competitive ratios for the respective online problems.

Augustine et al. [1] considered the related problem of strip packing with precedence constraints. They designed a divide-and-conquer algorithm that achieves an approximation ratio of $2 + \log(n + 1)$, where n is the total number of rectangles. They also considered a special case where all rectangles have uniform height and gave a 3-approximation algorithm for this case. Scheduling DAGs of rigid tasks has also been studied by Demirci et al. [11] in the context of power-aware scheduling, where each task is sequential but at the same time also requires a certain amount of power and there is a total power budget in the system. Here, two types of resources are considered — processor resource and power resource. From the processor resource's perspective, a DAG of sequential tasks needs to be scheduled, but from the power resource's perspective, it is a DAG of rigid parallel tasks. In [11], a two-step algorithm is presented: the first step applies Graham's list scheduling algorithm while considering the processor resource only to obtain an initial schedule; the second step adopts the divide-and-conquer algorithm of Augustine et al. [1] to modify the initial schedule in order to satisfy the power resource constraint. It is shown that this algorithm achieves an approximation

ratio of $2+2\log(n+1)$. In a follow-up paper, Demirci et al. [12] improved this ratio to $2+\log(n)$ while additionally allowing tasks to choose among a discrete set of configurations with power-performance tradeoffs.

While all the algorithms above are offline (except for list scheduling considered in [24], which gave an algorithm-specific lower bound matching the trivial upper bound of P), we present an online algorithm in this paper with a competitive ratio of $\Theta(\log n)$, asymptotically matching the approximation ratios in [1, 11, 12] for the offline problem. We also prove general lower bounds (in terms of both P and n) on the best competitive ratio achievable by any online algorithm. To the best of our knowledge, our results represent the first set of bounds for scheduling DAGs of rigid tasks in the online setting.

2.2 Scheduling DAGs of Moldable Tasks

A moldable task is a parallel task whose processor allocation can be varied prior to execution (but cannot be changed once the task starts running). Moldable tasks represent a more flexible task model compared to rigid tasks, and scheduling DAGs of moldable tasks can yield better performance bounds (often constant ratios). The exact ratio depends on the speedup model, which specifies the execution time of a task as a function of its processor allocation, and whether the problem is online or offline.

In the offline setting, Belkhale et al. [4] considered the *monotonic* speedup model, where the execution time of a task is a non-increasing function and the area (processor allocation times execution time) is a non-decreasing function of the number of allocated processors. They presented a 2.618-approximation algorithm while assuming the availability of an optimal processor allocation. Lepère et al. [23] proposed a 5.236-approximation algorithm without making this assumption. They also showed that optimal processor allocation can be obtained in pseudo-polynomial time for some special graphs (e.g., series-parallel graphs, trees), thus achieving 2.618-approximation for these graphs. Jansen and Zhang [21] later improved the approximation ratio to around 4.73. When further assuming that the area of a task is a concave function of the processor allocation, Jansen and Zhang [20] proposed a 3.29-approximation algorithm. Finally, Chen and Chu [7] improved both results, providing an algorithm that achieves an approximation ratio of 3.42 in general, and 2.96 when further assuming concave speedup functions.

The online setting is harder, and achieving constant performance ratios often requires assuming more specific speedup models. Feldmann et al. [13] considered the *roofline* model, where the execution time of a task decreases linearly with increased processor allocation until a maximum degree of parallelism, after which the execution time stays constant. They designed a 2.618-competitive online algorithm for this model. In fact, their algorithm works even in the *non-clairvoyant* setting, where the execution time of a task is also unknown to the scheduler until the task completes execution. Benoit et al. [5] considered several other variants of the linear speedup models with additional costs. They proved constant competitive ratios (between 3 and 6) for these models. Specifically, the models they considered include the *communication* model, where a communication cost is incurred that scales linearly with the number of allocated processors, the *Amdahl*'s model where a constant cost is incurred representing the sequence fraction of the parallel application, and a general combination of these models. Perotin and Sun [27] later provided improved results for all of these models and showed matching lower bounds. They also proved that any deterministic online algorithm that allocates processors based only on the local properties of the tasks cannot achieve $o(\log D)$-competitiveness, where D is the number of tasks in the longest path of the DAG.

2.3 Scheduling Independent Rigid Tasks

Relaxing the precedence constraint also makes the problem easier to tackle. In this case, scheduling a set of independent rigid tasks can yield constant performance ratios as well, in both offline and online settings.

In the offline setting, Coffman et al. [8] showed that the *Next-Fit Decreasing Height (NFDH)* algorithm is a 3-approximation, and the *First-Fit Decreasing-Height (FFDH)* algorithm is a 2.7-approximation. Both algorithms group tasks in subsets (called shelves) and schedule shelves of tasks one after another. The survey by Lodi et al. [25] provided more results and lower bounds on shelf-based scheduling algorithms and the best possible approximation ratios. Baker et al. [3] considered list-based scheduling, and they showed that the *Bottom-up Left-justified (BL)* heuristic that orders tasks in decreasing processor requirement is a 3-approximation. Turek et al. [28] showed that ordering tasks in decreasing execution time is also a 3-approximation. All the algorithms above also guarantee contiguous processor allocation for each task, thus the results also apply to strip packing. Without the contiguous processor constraint, several works [28, 15, 14] showed that greedy list-scheduling achieves a 2-approximation. Finally, Jansen [19] presented a $(3/2+\epsilon)$-approximation algorithm for any fixed $\epsilon > 0$, which is the best possible result given the lower bound of $3/2$ on the approximation ratio [22].

Different online settings of the problem have also been studied. In one online setting, tasks have different release times, and information about a task is unknown until it is released. Naroska and Schwiegelshohn [26] proved that

the greedy list-scheduling algorithm has a competitive ratio of 2 in this case. The same result was also shown independently by Johannes [22]. Chen and Vestjens [6] showed a 1.3473 lower bound on the competitive ratio of any deterministic online algorithm for this setting, even for sequential tasks. In another online setting, tasks are all available at time 0 but are presented one by one to the scheduler, which must irrevocably schedule each task before the next one is revealed. Johannes [22] showed that greedy list-scheduling has a competitive ratio of P in the worst case, and presented a 12-competitive algorithm. Baker and Schwarz [2] extended the two shelf-based algorithms presented in [8] and showed that *Next-Fit* is 7.46-competitive and *First-Fit* is 6.99-competitive. The surveys by Csirik and Woeginger [9, 10] described more results and lower bounds that use shelf-based algorithms under this setting. To date, the best known competitive ratio is 6.6623, obtained by Hurink and Paulus [18] and independently by Ye et al. [29].

3 Problem Statement

In this section, we precisely define our scheduling problem, as well as the objective function that evaluates the quality of the algorithm presented in Section 4. Table 1 provides a list of notations and vocabulary used throughout the paper.

Table 1: List of notations and vocabulary

Hongyang: I think it's still better to list some trivial notations used in the paper, so I added them.

Lucas: OK, was it intentional to not put n?

Notation	Name	Definition
$\mathcal{I}$	Instance	Section 3.1
$\mathcal{T}$	Task	Section 3.1
P	Total number of processors	Section 3.1
t	Execution time of a task	Section 3.1
p	Processor allocation of a task	Section 3.1
$\mathcal{P}(\mathcal{T})$	Set of predecessors of task $\mathcal{T}$	Section 3.1
$\text{LB}(\mathcal{I})$	Lower bound of instance $\mathcal{I}$	Section 3.2
$\mathcal{C}(\mathcal{I})$	Critical path length of $\mathcal{I}$	Section 3.2
$\mathcal{A}(\mathcal{I})$	Area of $\mathcal{I}$	Section 3.2
$T_{\text{ALG}}(\mathcal{I})$	Makespan of ALG for $\mathcal{I}$	Section 3.2
$T_{\text{OPT}}(\mathcal{I})$	Optimal makespan for $\mathcal{I}$	Section 3.2
s^{∞}	Criticality (start time)	Definition 1
f^{∞}	Criticality (end time)	Definition 1
χ	Power level	Definition 2
λ	Longitude	Definition 2
ζ	Category	Definition 2
L_{ζ}	Length of a category ζ	Definition 3
$\mathcal{L}$	Length matrix (or L-matrix)	Definition 5

3.1 Online Scheduling Model for Rigid DAGs

In our scheduling model, we consider an instance $\mathcal{I}$ that is defined with the following components:

- A fixed DAG of n tasks need to be scheduled on a platform consisting of P identical processors. The tasks are denoted as $\mathcal{T}_i$ for $i \in [1, n]$.
- 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 edge exists from task $\mathcal{T}_i$ to task $\mathcal{T}_j$ if and only if $\mathcal{T}_j$ cannot start executing until task $\mathcal{T}_i$ is completed. In this case, we say $\mathcal{T}_i$ is a predecessor of $\mathcal{T}_j$ and $\mathcal{T}_j$ is a successor of $\mathcal{T}_i$. For each task $\mathcal{T}_i$, we use $\mathcal{P}(\mathcal{T}_i)$ to denote the set of predecessors of $\mathcal{T}_i$.

We consider the **online** scheduling setting: before a task is ready for processing, the scheduler is unaware of its existence. In other words, a task $\mathcal{T}_i$ is discovered only when all of its predecessors are completed. When that happens, its execution time t_i and processor allocation p_i become known. We further assume that the DAG structure is also discovered on the fly by the scheduler, which means that the set of predecessors of a task becomes known upon its release but not the set of successors. The goal is to design an online scheduling algorithm that assigns to each task $\mathcal{T}_i$ a start time $s_i \geq 0$ such that at most P processors may be used at all times. More precisely, for any time $x \in \mathbb{R}$, if we denote as $\mathcal{J}(x) \subseteq \mathcal{I}$ the subset of tasks running at time x, i.e., for which $s_i < x < s_i + t_i$, then we must have $\sum_{\mathcal{T}_i \in \mathcal{J}(x)} p_i \leq P$. The objective function to be minimized is the overall completion time of the DAG, also known as the *makespan*, defined as $T(\mathcal{I}) = \max_{\mathcal{T}_i \in \mathcal{I}}(s_i + t_i)$.

Hongyang: I added the successor sentence above in blue. For your introductory example, if we know the successors of a task upon its release, some smart algorithms might be able to use a policy to wait on long task C, thus contradicting what you said in the intro.

Lucas: Sure, I thought that was clear from context (task become known when all their predecessors are completed) but it's never bad to be precise

3.2 Makespan Lower Bound and Worst-Case Ratios

Given an instance $\mathcal{I}$, a well-known lower bound [17] 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, and therefore, $\frac{\mathcal{A}(\mathcal{I})}{P}$ corresponds to the total execution time if we could always utilize all P processors.
- $\mathcal{C}(\mathcal{I})$ is the **critical-path length** of the instance, 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 as $T_{\textsc{Alg}}(\mathcal{I})$ the makespan of the schedule under Alg and as $T_{\textsc{Opt}}(\mathcal{I})$ the optimal makespan of any schedule. In this work, we aim at minimizing $\frac{T_{\textsc{Alg}}(\mathcal{I})}{T_{\textsc{Opt}}(\mathcal{I})}$ for any possible instance $\mathcal{I}$. However, as we will show in Section 6, no algorithm may be always within a constant factor from the optimal for any instance, i.e., $\forall \textsc{Alg}, \sup_{\mathcal{I}}\left(\frac{T_{\textsc{Alg}}(\mathcal{I})}{T_{\textsc{Opt}}(\mathcal{I})}\right) = \infty$. Therefore, we will study the worst-case ratio for instances consisting of n tasks for analyzing an algorithm's performance. We say that an algorithm Alg is $f(n)$-competitive if $\sup_{\mathcal{I} \in \{\mathcal{I}_n\}}\left(\frac{T_{\textsc{Alg}}(\mathcal{I})}{T_{\textsc{Opt}}(\mathcal{I})}\right) \leq f(n)$, where $\{\mathcal{I}_n\}$ denotes the set of all possible instances with n tasks. Note that if Alg is not deterministic, we assume $\sup_{\mathcal{I} \in \{\mathcal{I}_n\}}\left(\frac{T_{\textsc{Alg}}(\mathcal{I})}{T_{\textsc{Opt}}(\mathcal{I})}\right)$ consider the worst possible makespan for $T_{\textsc{Alg}}(\mathcal{I})$ instead of its expected value. Because computing $T_{\textsc{Opt}}(\mathcal{I})$ is a generally NP-complete problem [**?**], we will instead aim to minimize $\frac{T_{\textsc{Alg}}(\mathcal{I})}{\textsc{Lb}(\mathcal{I})}$ when analyzing an algorithm, which gives a guarantee on the competitive ratio since $\textsc{Lb}(\mathcal{I}) \leq T_{\textsc{Opt}}(\mathcal{I})\ \forall \mathcal{I}$. This is a common technique widely adopted by the scheduling literature. For proving the lower bounds in Section 6, however, we will use $\frac{T_{\textsc{Alg}}(\mathcal{I})}{T_{\textsc{Opt}}(\mathcal{I})}$.

Hongyang: Do we consider randomized algorithms or lower bound in this paper? If not, I suggest removing the blue sentence above.

Lucas: Not really, except that no algorithm may be **always** $o(\log(n))$ competitive (the lower bounds works for non-determinisitic algorithms as well). To be fair, we can omit this, in which case I will change the lower bound section accordingly.

4 Definitions and Algorithm

Before presenting the algorithm, we will start with a few definitions. They introduce new attributes of a task within its instance $\mathcal{I}$. All the definitions and attributes are illustrated with an example throughout the section.

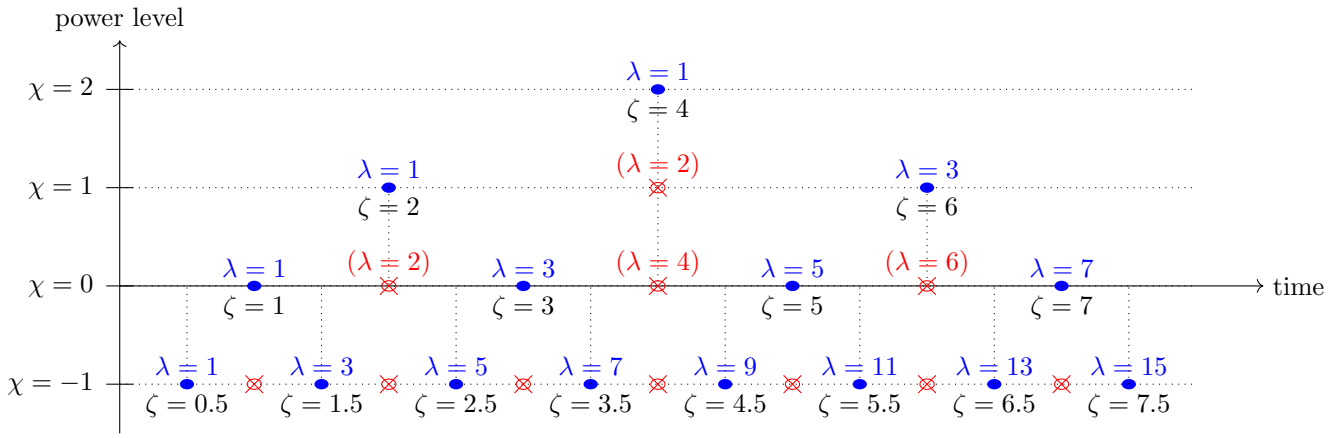


Figure 2: Graphical representation of some possible values of category ζ, power level χ and longitude λ.

4.1 Definitions

4.1.1 Criticality and Category

Definition 1. *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 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 executed in an ASAP schedule. As $\mathcal{C}(\mathcal{I})$ corresponds to the longest path in the graph, or the minimum completion time with unlimited number of processors of an ASAP schedule, we have $\mathcal{C}(\mathcal{I}) = \max_j f_j^\infty$.*

Lemma 1. *Given a task $\mathcal{T}$ and its set of predecessors $\mathcal{P}(\mathcal{T})$, we have:*

$$s^\infty = \begin{cases} \max_{\mathcal{T}_j \in \mathcal{P}(\mathcal{T})} f_j^\infty, & \text{if } \mathcal{P}(\mathcal{T}) \neq \emptyset \\ 0, & \text{otherwise} \end{cases} \tag{2}$$

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

$$\chi = \max\{\chi \in \mathbb{Z} : \exists \lambda \in \mathbb{N}, s^\infty < \lambda 2^\chi < f^\infty\} \tag{3}$$

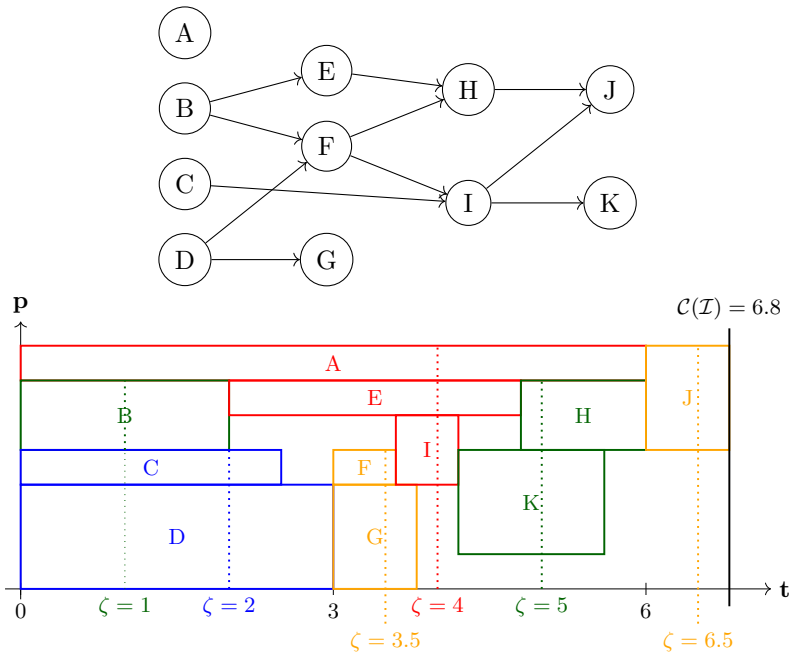
*where λ is the associated **longitude**. Given the power level and longitude of the task, we further define $\zeta = \lambda 2^\chi$ as the task's **category**. Lemma 2 shows that all these values are unique and well-defined for any given task.*

Figure 2 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 2. *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



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 3: 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.

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 2, 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 2, 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 1. *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.*

Figure 3 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.

4.1.2 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 4 illustrates the categories and their corresponding lengths, using the example presented in Figure 3. Based on Lemma 2, 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

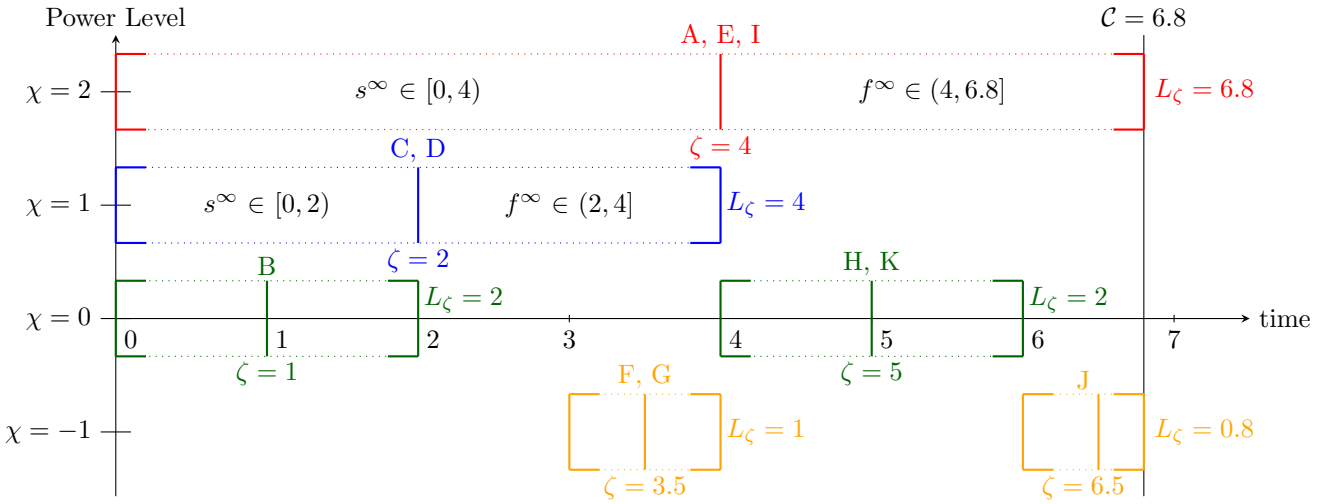


Figure 4: Graphical representation of the categories and their lengths for the example of Figure 3.

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 4), $2^{\chi+1}$ 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 3. *Given any instance with critical-path length $\mathcal{C}$ and a category ζ with power level χ and longitude λ, if $\zeta < \mathcal{C}$, we define the **length** of the category as:*

$$L_\zeta = \begin{cases} \min(2^{\chi+1}, \mathcal{C} - (\lambda-1)2^\chi) & \text{if } \mathcal{C} > \lambda 2^\chi \\ 0 & \text{otherwise} \end{cases} \tag{4}$$

Hongyang: Seems that you added the condition in blue above. Isn't it captured by the otherwise?

Definition 4. *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:*

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

Lucas: Before there were no otherwise, nor the blue condition. I added the blue condition to avoid negative L_ζ. I added the otherwise to replace the old definition of L-matrix above, since I found it more straightforward to simply define it with the L_ζ. I forgot about the blue condition at that time. Do you want me to check if I made other changes in 4.1 and 5 since the creation of the file and put them in blue in case I messed up something else? There are very few of them.

Lemma 3. *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:*

$$t \leq L_\zeta \tag{5}$$

Proof. If a task has category $\zeta = \lambda 2^\chi$, it must verify $\lambda 2^\chi < s^\infty \leq \mathcal{C}$ and we're on the first case. Based on Lemma 2, 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. □

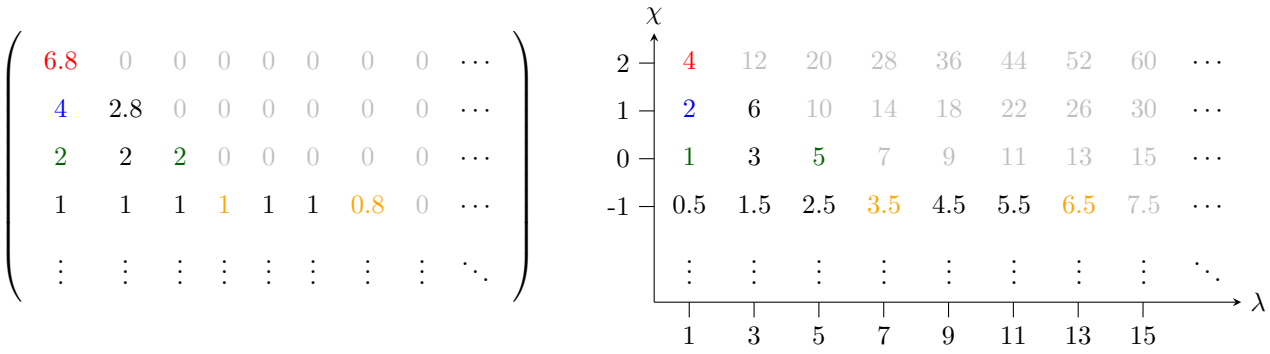


Figure 5: For any instance with $\mathcal{C} = 6.8$, left: L-matrix $\mathcal{L}(\mathcal{C})$; right: corresponding category values. The colors of the numbers correspond to the ones in the example of Figures 3 and 4. The numbers in black correspond to possible length and category values with $\mathcal{C} = 6.8$ but are not present in the example. In contrast, the numbers in gray correspond to impossible length and category values with $\mathcal{C} = 6.8$.

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 4, 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 [19]. 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 2), and the length of categories is roughly doubled (as shown in Figure 4). 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 5 (left) shows the L-matrix for the example of Figures 3 and 4. 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 4). More generally, the correspondence between elements of the L-matrix and the category values can be viewed in Figure 5. 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.

Definition 5. *For any instance with critical-path length $\mathcal{C}$, we define $\mathcal{L} = (\ell_{i,j})^{\mathbb{N}\times\mathbb{N}}$ as its **length matrix** (or **L-matrix**) such that:*

$$\forall i, j, \ell_{i,j} = L_{\zeta(X+1-i,2j-1)} \text{ where } \zeta(\chi, \lambda) = \lambda 2^\chi$$

Lucas: I'm okay with the way Definition 3 and 4 are currently. But is it more logical to drop the $\mathcal{C}$ in $L_\zeta(\mathcal{C})$ and $\mathcal{L}(\mathcal{C})$ after the definitions? We have to introduce the $\mathcal{C}$ in the definition above since the L_ζ depend on it, but it doesn't appear at all in the definition, which may be odd? You can remove comment if you think it's fine.

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 3), and $\zeta \geq \mathcal{C}$ (i.e., no tasks belong to it and therefore $\ell_{i,j} = 0$). This is shown in the following lemma.

Lemma 4. *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:*

$$\ell_{i,j} = \begin{cases} 2^{X+2-i} & \text{if } j2^{X+2-i} \leq \mathcal{C} \\ \mathcal{C} - (j-1)2^{X+2-i} & \text{if } (2j-1)2^{X+1-i} < \mathcal{C} < j2^{X+2-i} \\ 0 & \text{otherwise} \end{cases} \tag{6}$$

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

- If $j2^{X+2-i} \leq \mathcal{C}$, then $\zeta(X+1-i, 2j-1) = (2j-1)2^{X+1-i} < \mathcal{C}$. Using Definition 3, $L_{\zeta(X+1-i,2j-1)} = \min\left(2^{X+2-j}, \mathcal{C} - (2j-2)2^{X+1-j}\right)$. 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 $\zeta(X+1-i, 2j-1) = (2j-1)2^{X+1-i} < \mathcal{C}$. Using Definition 3, $L_{\zeta(X+1-i,2j-1)} = \min\left(2^{X+2-j}, \mathcal{C} - (2j-2)2^{X+1-j}\right)$. 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 3, $\ell_{i,j} = 0$. □

4.2 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 section.

- 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 are discovered before starting 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 these principles, the algorithm uses categories to split the graph into batches of independent tasks and process them one after another. This limits the number of categories with long tasks, as they correspond to the upper rows of the L-matrix that only contains a few non-zero elements. 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 2 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). Specifically, at the beginning of the batch and whenever a task completes[2], 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 P(T) = ∅ then                          ▷ P(T) contains the list of predecessors of task T
2:     s^∞ ← 0
3: else
4:     s^∞ ← max_{T_j ∈ P(T)} f_j^∞          ▷ start time in the ASAP schedule
5: end if
6: f^∞ ← s^∞ + t                             ▷ finish time in the ASAP schedule
7: find χ and λ from s^∞ and f^∞ based on Definition 2
8: ζ ← λ2^χ
9: return ζ
```

Finally, our algorithm Batch is presented in Algorithm 3. It uses a collection of lists to store tasks, and each list $\mathcal{B}_\zeta$ contains tasks that belong to a particular category ζ. The algorithm processes tasks in batches of increasing category values, by finding the batch with tasks of minimal categories, $\mathcal{B}_{\zeta_{\min}}$, and schedule it using subroutine ScheduleIndep($\mathcal{B}_{\zeta_{\min}}$). 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

[2]In the pseudo-code provided, a virtual task $\mathcal{T}$ with $p = 0$ and $t = 0$ can be considered to complete at the beginning of the batch.

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

organize tasks in $\mathcal{B}$ in any arbitrary order
$P_{avail} \leftarrow P$ ▷ number of available processors
$\mathcal{R} \leftarrow \emptyset$ ▷ $\mathcal{R}$ will store newly discovered tasks
while $\mathcal{B}$ is not empty and whenever a task $\mathcal{T}_i$ completes **do**
    $P_{avail} \leftarrow P_{avail} + p_i$
    for each discovered task $\mathcal{T}_j$ **do**
        add $\mathcal{T}_j$ to $\mathcal{R}$
    end for
    for each task $\mathcal{T}_k \in \mathcal{B}$ **do**
        if $P_{avail} \geq p_k$ **then**
            execute $\mathcal{T}_i$ now
            $P_{avail} \leftarrow P_{avail} - p_k$
            remove $\mathcal{T}_k$ from $\mathcal{B}$
        end if
    end for
end while
wait until all tasks in $\mathcal{B}$ complete
return $\mathcal{R}$

Algorithm 3 Batch

$\mathcal{R} \leftarrow \emptyset$ ▷ $\mathcal{R}$ will store the set of ready tasks
for each ready task $\mathcal{T}_i$ **do**
    add $\mathcal{T}_i$ to $\mathcal{R}$
end for
repeat
    for each task $\mathcal{T}_i \in \mathcal{R}$ **do**
        $\zeta \leftarrow$ ComputeCat($\mathcal{T}_i$)
        Add $\mathcal{T}_i$ to $\mathcal{B}_\zeta$ ▷ $\mathcal{B}_\zeta$ contains a list of tasks of category ζ
    end for
    Find $\mathcal{B}_{\zeta_{\min}}$, containing the tasks of smallest category
    $\mathcal{R} \leftarrow$ ScheduleIndep($\mathcal{B}_{\zeta_{\min}}$)
    delete $\mathcal{B}_{\zeta_{\min}}$
until all tasks are completed

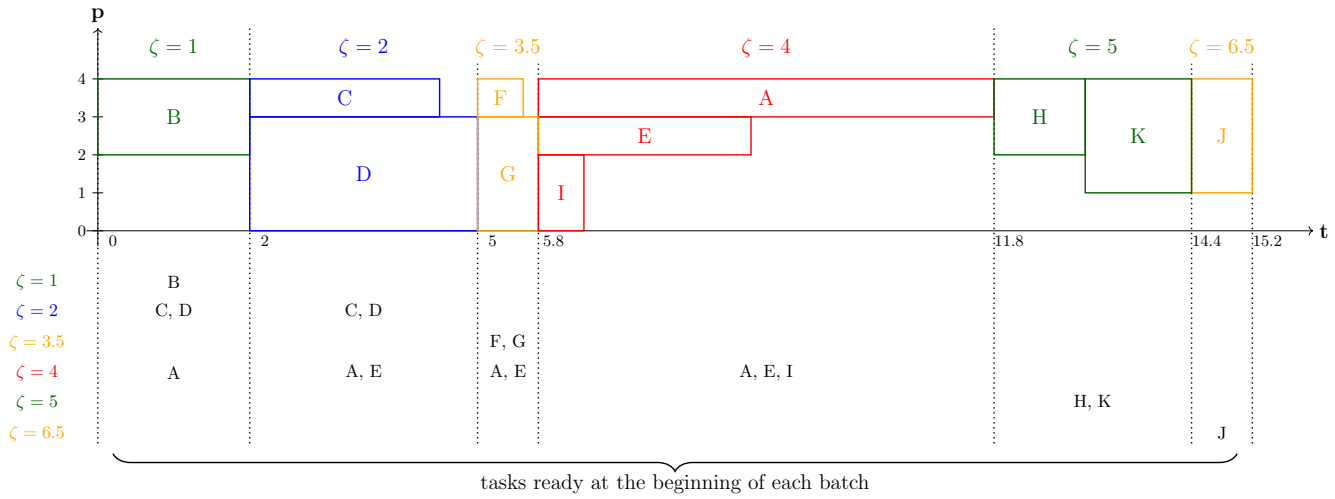


Figure 6: Illustration of the BATCH algorithm for the example of Figure 3 on $P = 4$ processors.

corresponding list $\mathcal{B}_\zeta$ of identical category[3]. Our analysis in Section 5 shows that all tasks in a category are ready when their category starts to be processed.

Figure 6 illustrates how the algorithm works using the example of Figure 3 on $P = 4$ processors. At first, only tasks A, B, C, and D are ready and added to the corresponding lists of $\mathcal{A}$ with their respective category 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.

5 Analysis

The analysis's general steps follow the algorithm's principles stated previously. First, we show that tasks in a category are independent and that all are discovered before starting to process the category.

Lemma 5. *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 completing task $\mathcal{T}_i$, then we must have $f_i^\infty \leq s_j^\infty$. Using Lemma 2, we derive $\zeta_i < f_i^\infty \leq s_j^\infty < \zeta_j$, hence the result. □

Corollary 2. *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 5, 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 5 and thus must be completed. □

Lemma 6. *For any batch $\mathcal{B}_\zeta$ of tasks of 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 $\mathcal{B}_\zeta$.

[3]If no tasks of identical category had been discovered so far, a new batch will be created.

Proof. From Corollary 2, 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 t'.

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 $t' > t''$). Since 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 $|T_{<P/2}| \leq t''-t' \leq L_\zeta$. From $T(\mathcal{B}_\zeta) = |T_{<P/2}| + |T_{\geq P/2}|$, we achieve the result. □

Remark 1. *The criticality and category of tasks can be extended naturally for the online strip packing problem with precedence constraint. Additionally,* Batch *can be adapted to guarantee continuous space allocation for strip packing by replacing the subroutine* ScheduleIndep($\mathcal{B}$) *with an algorithm that solves a strip packing instance without precedence constraints, such as the NFDH algorithm presented in [8]. Using NFDH, it is known that the resulting height is at most twice the total area plus the maximum height of the rectangles, hence the analysis can be applied analogously.*

Lemma 7. *For any instance $\mathcal{I}$, the makespan of the schedule produced by* Batch *satisfies:*

$$T_{\text{Batch}}(\mathcal{I}) \leq 2\frac{\mathcal{A}(\mathcal{I})}{P} + \sum_\zeta L_\zeta \tag{7}$$

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 6, 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 $\sum_\zeta \mathcal{A}(L_\zeta) = \mathcal{A}(\mathcal{I})$. □

5.1 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 1. Batch *is* $(3+\log(n))$*-competitive. In other words for any instance $\mathcal{I}$ with n tasks,* $\frac{T_{\text{Batch}}(\mathcal{I})}{T_{\text{Opt}}(\mathcal{I})} \leq 3+\log(n)$

Proof. Using Lemma 7, 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).

Suppose the critical-path length of the instance $\mathcal{I}$ satisfies $2^X < \mathcal{C}(\mathcal{I}) \leq 2^{X+1}$ for some $X \in \mathbb{Z}$. The proof consists of the following three steps, which can again be verified visually in Figure 7 (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 decrease from left to right: using Lemma 4, all values are equal except potentially the last positive one $\ell_{i,j}$, if it corresponds to the second case with $\mathcal{C}(\mathcal{I}) < j2^{X+2-i}$. This condition shows $\ell_{i,j} = \mathcal{C}(\mathcal{I}) - (j-1)2^{X+2-i} < 2^{X+2-i}$. Therefore, we just need to show that the last positive value of the row $\ell_{i,j}$ is larger than the first value of the next row $\ell_{i+1,1}$. If $\ell_{i,j}$ corresponds to the first case in Equation (6), we clearly have $\ell_{i,j} = 2^{X+2-i} > 2^{X+1-i} = \ell_{i+1,1}$. If $\ell_{i,j}$ corresponds to the second case in Equation (6), $\ell_{i,j} = \mathcal{C}(\mathcal{I}) - (2j-2)2^{X+1-i} > 2^{X+1-i} = \ell_{i+1,1}$ 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 $\ell_{1,1} = \mathcal{C}(\mathcal{I})$, 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

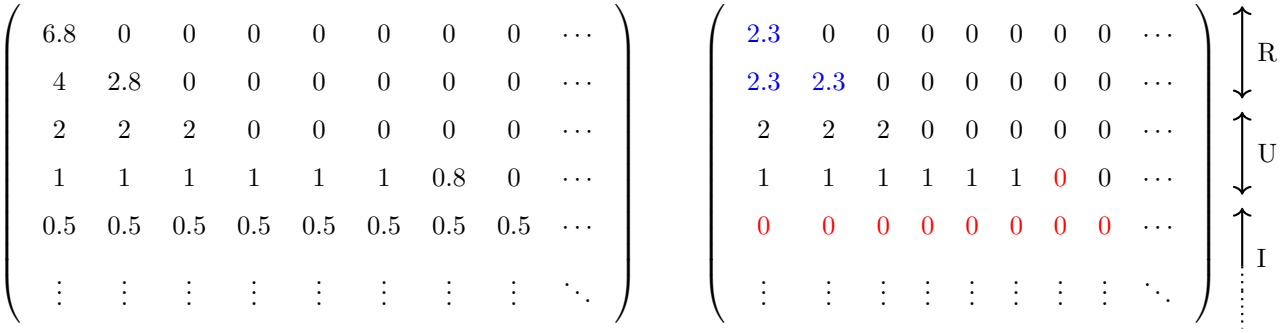


Figure 7: For any instance with $\mathcal{C} = 6.8$, left: L-matrix $\mathcal{L}(\mathcal{C})$ without constraints on the task execution times; right: L-matrix $\mathcal{L}^*(\mathcal{C})$ with bounds $m = 0.9$ and $M = 2.3$ on the task execution times.

$\mathcal{C}(\mathcal{I}) > 2^X$, we must have $k \geq 2^{i-2}$. Therefore, $\ell_{i,1} = \cdots = \ell_{i,k} = 2^{X+2-i}$, and $\ell_{i,k+1}$ is either $\mathcal{C}(\mathcal{I}) - k2^{X+2-i} = r$ (second case in Equation (6)) or 0, and all other $\ell_{i,j}$'s are 0. This shows $\sum_{j=1}^{\infty} \ell_{i,j} \leq k2^{X+2-i} + r = \mathcal{C}(\mathcal{I})$.

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 7 and Equation (1), we obtain the result:

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

□

5.2 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 7 (right) shows the L-matrix for any instance with $\mathcal{C} = 6.8$ and $m = 0.9$. Compared to Figure 7 (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 7 (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 $m \leq L_\zeta \leq M$ 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:

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

Here, L^*_ζ is an upper bound on the length of the longest task of category ζ, and based on Lemma 7, we have:

$$T_{\textsc{Batch}}(\mathcal{I}) \leq 2\frac{\mathcal{A}(\mathcal{I})}{P} + \sum_{\zeta} L^*_\zeta \tag{8}$$

Theorem 2. *For any instance $\mathcal{I}$ such that $\forall i, m \leq t_i \leq M$, we have* $\frac{T_{\textsc{Batch}}(\mathcal{I})}{\textsc{Lb}(\mathcal{I})} \leq 6 + \log\left(\frac{M}{m}\right)$.

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 $2^k < m \leq 2^{k+1}$ and $h \in \mathbb{Z}$ such that $2^h < M \leq 2^{h+1}$. Suppose the critical-path length of the instance satisfies $2^X < \mathcal{C}(\mathcal{I}) \leq 2^{X+1}$ for some $X \in \mathbb{Z}$. We have $k \leq h \leq X$. 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 3 and Lemma 3, 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 $L_\zeta \leq 2^k < m$. Hence, its existence would contradict the $t_i \geq m, \forall i$ assumption. In Figure 7 (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 $\chi \in [k, h-1]$, the sum of their lengths is at most $\mathcal{C}(\mathcal{I})$. There are $h-k$ such sets of categories, and we have $h - k \leq \log\left(\frac{M}{m}\right) + 1$.*
   There are $h-k$ rows in $\mathcal{L}^*$ that correspond to power levels in the range $[k, h-1]$. From $m \leq 2^{k+1}$ and $2^h < M$, we get $\frac{M}{m} > \frac{2^h}{2^{k+1}}$ and $h - k - 1 \leq \log\left(\frac{M}{m}\right)$. We have already shown in the proof of Theorem 1 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 $L^*_\zeta \leq L_\zeta$, we get $\sum_{\zeta \in Z_\chi} L^*_\zeta \leq \sum_{\zeta \in Z_\chi} L_\zeta \leq \mathcal{C}(\mathcal{I})$. In Figure 7 (right), with $m = 0.9$ and $M = 2.3$, we have $k = -1$ and $h = 1$, corresponding to rows 3 and 4 labeled as U (Unchanged), whose values are mostly unchanged except for the last value of row 4.

3. *For all categories with power level at least h, 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 $\ell_{i,2^{i-1}+1} > 0$, which according to Equation (6) 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 $\mathcal{C}(\mathcal{I}) \leq 2^{X+1}$. The categories with a power level at least $h+1$ correspond to the $X - h$ first rows of $\mathcal{L}^*$. The number of such categories is at most $\sum_{i=1}^{X-h} 2^{i-1} \leq 2^{X-h}$, and the longest task in these categories has length at most M. From $2^X < \mathcal{C}(\mathcal{I})$ and $M \leq 2^{h+1}$, we derive $M2^{X-h} < 2\mathcal{C}(\mathcal{I})$, which means $\sum_{\chi=h+1}^{X} \sum_{\zeta \in Z_\chi} L^*_\zeta < 2\mathcal{C}(\mathcal{I})$. Finally, adding in the categories with power level h, which satisfies $\sum_{\zeta \in Z_h} L^*_\zeta \leq \sum_{\zeta \in Z_h} L_\zeta \leq \mathcal{C}(\mathcal{I})$, we get the sum of lengths from all categories with a power level at least h to be at most $3\mathcal{C}(\mathcal{I})$, i.e., $\sum_{\chi=h}^{X} \sum_{\zeta \in Z_\chi} L^*_\zeta < 3\mathcal{C}(\mathcal{I})$. In Figure 7 (right), this corresponds to the first two rows, labeled as R (Reduced).

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 (8) and (1), we obtain the result:

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

□

6 Lower Bound on Best Competitive Ratio

In the previous section, we have shown that without further assumption on the tasks' lengths, Batch is $(3+\log(n))$-competitive for all n. We will now show its near-optimal asymptotic competitive ratio, stating that no algorithm may be $\frac{\log(n)-\log(\log(n))}{4}$-competitive for all $n \geq N$ with N arbitrarily large. Hence, a competitive ratio of $\Theta(\log(n))$ for all n, achieved by Batch, is the best possible.

The following section holds for any given $\mu > 0$ and $P > 0$. We will also use $\epsilon > 0$ and K, an integer larger than 2. Both are arbitrary throughout the section: their values will be fixed with respect to μ and P to show the final results in Theorem 3 and Theorem 4.

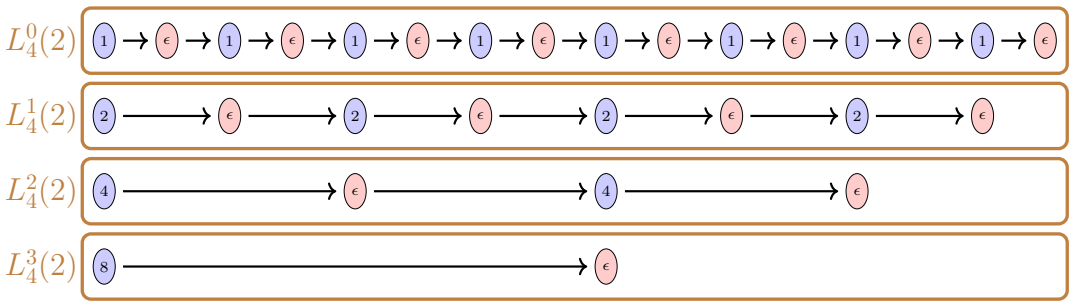


Figure 8: $X_4(2)$

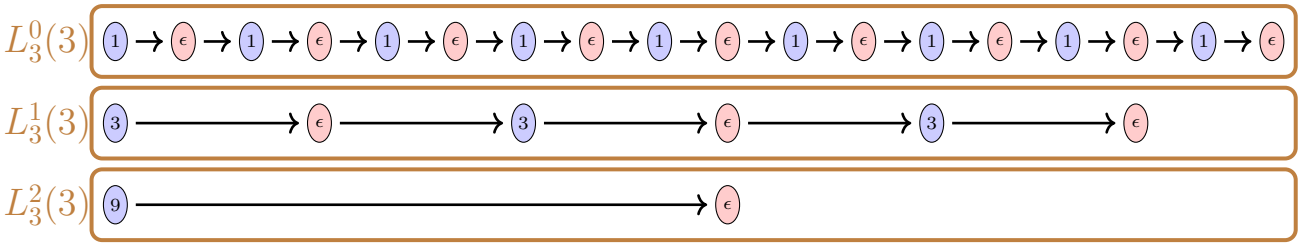


Figure 9: $X_3(3)$

Definition 6. *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 7. *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 8 and $X_3(3)$ is illustrated in Figure 9, where blue tasks require a single processor and red tasks require all P processors.

Lemma 8. *The optimal time of processing of $X_P(K)$ satisfies:*

$$T_{\text{OPT}}(X_P(K)) > PK^{P-1} - (P-1)K^{P-2} \tag{9}$$

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 8, 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.

We are given an optimal schedule in the following. 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 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 $T_{\text{OPT}}(X_P(K)) > \sum_{i=1}^{K^{P-1}} t_i$.

$t_1 \geq K^{P-1}$ 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 6). Similarly, $\forall j \leq P-1, \forall i \leq K^{P-j-1}, t_i \geq K^j$.

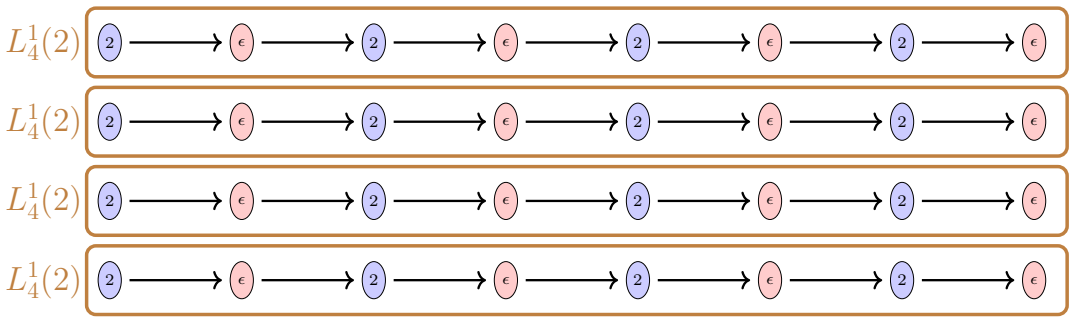


Figure 10: $Y_4^1(2)$

Therefore,

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

□

Remark 2. *$X_P(K)$ can be viewed as an extension of the strip-packing instance used in [1]. They used $K = 2$ and P arbitrarily large to show the existence of instances such that the optimal processing time was $\Theta(\log(n))$ times larger than the lower-bound $\text{LB}(X_P(K))$. This showed that finding an offline polynomial algorithm with a better approximation ratio than $\Theta(\log(n))$ would at least require better bounds for analysis. Not only have we shown that $\Theta(\log(n))$ is also achievable online, but we also show in this section that having a better competitive ratio is impossible in this context.*

The lower bound for this graph is around K^{P-1} since it corresponds to the exact lower bound without the tasks of length ϵ. The optimal execution time is roughly PK^{P-1}, which shows $X_P(K)$ may not be processed efficiently. However, a graph consisting of identical linear chains $L_P^i(K)$ can be processed using all P processors at all times.

Definition 8. *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 10. In this section, we 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. We also show that a clairvoyant scheduler could instead process sub-graphs in the shape of $Y_P^i(K)$, with near-optimal processor allocation.

Lemma 9. *For all i, K, P, the optimal time of processing of a graph $Y_P^i(K)$ is*

$$T_{\text{OPT}}(Y_P^i(K)) = K^{P-1} + PK^{P-i-1}\epsilon \tag{10}$$

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

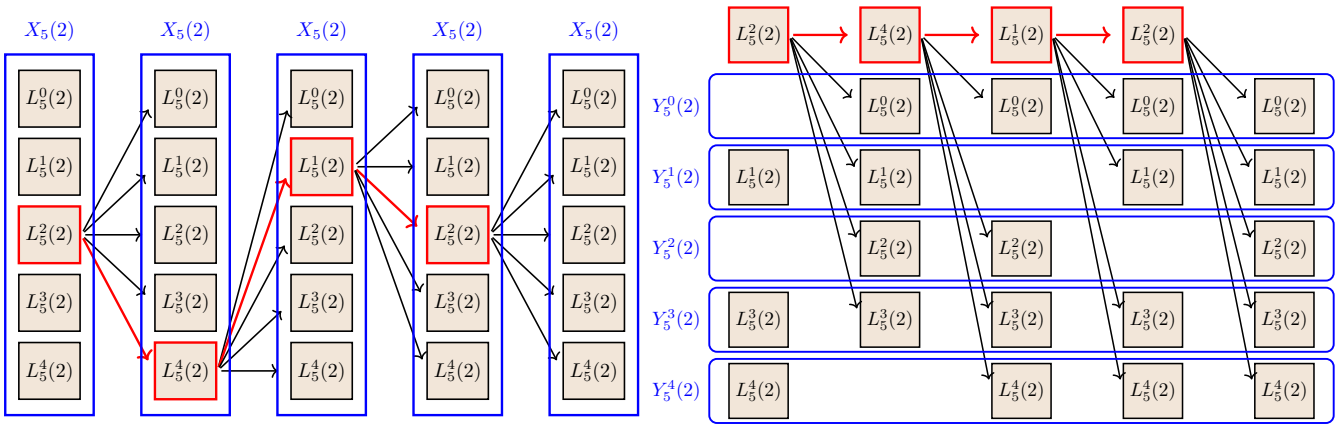


Figure 11: $Z_5^{\text{ALG}}(2)$. Left, ALG will have to process layers one by one. Right, better clairvoyant schedule, where red chains are processed first, then all the $Y_5^i(2)$ can be processed efficiently.

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

$Z_5^{\text{ALG}}(2)$ is illustrated in Figure 11, 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 10. *For any* ALG*, the processing time of $Z_P^{\text{ALG}}(K)$ satisfies:*

$$T_{\text{ALG}}(Z_P^{\text{ALG}}(K)) \geq P^2K^{P-1} - P(P-1)K^{P-2} \tag{11}$$

Proof. The result directly comes from Lemma 8 and Definition 9, as T_{ALG} is the time required to process P identical copies of $X_P(K)$, each taking time larger than $T_{\text{OPT}}(X_P(K))$, and a copy may not start before the completion of the precedent one. □

Lemma 11. *An optimal scheduler can process $Z_P^{\text{ALG}}(K)$ faster than*

$$T_{\text{OPT}}(Z_P^{\text{ALG}}(K)) < 2P(K^{P-1} + PK^P\epsilon) \tag{12}$$

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 11, right with $Z_5^{\text{ALG}}(2)$.

Any chain $L_P^i(K)$ is one of the chains in $Y_P^i(K)$ and can be processed in time at most $T_{\text{OPT}}(Y_P^i(K))$. Processing any of the resulting sub-graphs of $Y_P^i(K)$ can also be done in time at most $T_{\text{OPT}}(Y_P^i(K))$. There are $P-1$ chains $L_P^i(K)$ to process to unlock all the other chains, which can be arranged in at most P sub-graphs of $Y_P^i(K)$. Using Equation 10,

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

□

Our construction holds for any K, ϵ, and any algorithm ALG. Therefore, using $T_{\text{ALG}}(Z_P^{\text{ALG}}(K)) \geq P^2K^{P-1} - P(P-1)K^{P-2}$ (Lemma 10) and $T_{\text{OPT}}(Z_P^{\text{ALG}}(K)) \leq 2P(K^{P-1} + PK^P\epsilon)$ (Lemma 11), we choose good values for K and ϵ to achieve our main result regarding the lower bound on the best possible competitive ratio in the next two theorems.

Theorem 3. *With N arbitrarily large, no algorithm may be $\frac{\log(n)-\log(\log(n))-1}{4}$-competitive for all $n \geq N$.*

Proof. We notice the total number of tasks in $Z_P^{\text{ALG}}(K)$ is, by Definition 7 and Definition 9,

$$n = P\left(\sum_{i=0}^{P-1} 2K^{P-i-1}\right) = 2P\frac{K^P-1}{K-1}.$$

With $K=2$, we find $n = 2P(2^P-1)$, thus $2^P \leq n < 2P2^P$ and $P \leq \log(n) < P + \log(P) + 1$, therefore

$$\log(n) - \log(\log(n)) - 1 < P + \log(P) + 1 - \log(\log(n)) - 1 \leq P \leq \log(n)$$
$$\log(n) - \log(\log(n)) - 1 < P \leq \log(n).$$

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 10 and Lemma 11 with $0 < \epsilon < \frac{1}{2P(\log(n)-\log(\log(n))-1)}$, we derive:

$$\begin{aligned}
\frac{T_{\text{ALG}}(Z_P^{\text{ALG}}(K=2))}{T_{\text{OPT}}(Z_P^{\text{ALG}}(K=2))} &> \frac{P^2K^{P-1} - P(P-1)K^{P-2}}{2P(K^{P-1}+PK^P\epsilon)} = \frac{PK-(P-1)}{2(K+PK^2\epsilon)} = \frac{P+1}{2(2+4P\epsilon)} \\
&> \frac{\log(n)-\log(\log(n))-1+1}{4(1+2P\epsilon)} = \frac{(\log(n)-\log(\log(n))-1)\left(1+\frac{1}{\log(n)-\log(\log(n))-1}\right)}{4(1+2P\epsilon)} \\
&> \frac{\log(n)-\log(\log(n))-1}{4} \qquad \text{because } \frac{1}{2P(\log(n)-\log(\log(n))-1)} > \epsilon
\end{aligned}$$

□

Theorem 4. *For any algorithm* ALG *and $\mu > 0$, there exists an instance $\mathcal{I}$ for which we may have $\frac{T_{\text{ALG}}(\mathcal{I})}{T_{\text{OPT}}(\mathcal{I})} > \frac{P}{2} - \mu$*

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 10 and Lemma 11:

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

□

7 Conclusion and Future Work

In this paper, we have explored the challenging problem of online scheduling of rigid tasks with precedence constraints in order to minimize the makespan. We introduced the BATCH algorithm, which achieves a competitive ratio of $3 + \log(n)$ for unbounded task lengths and is always within a factor $6 + \log(\frac{M}{m})$ from the optimal, where M is the length of the longest task and m is the length of the shortest task. Notably, these results mirror those achieved by the best offline algorithms, demonstrating that BATCH is close in the worst case without requiring complete knowledge of the instance. Our analysis further reveals the near-optimal nature of the BATCH algorithm within the framework of competitive ratios, establishing that no online algorithm can achieve $o(\log(n))$-competitiveness in this setting.

We believe the results in this paper can be viewed as a cornerstone for several avenues of future work. First, assuming the execution time of each task is exactly known could be a strong assumption, and we are currently designing more general heuristics based on the concept of task categories to handle the uncertainty. Second, even though BATCH is near-optimal in the worst case, it is likely inefficient in practice. Indeed, not starting a new category until the previous one is fully completed is probably a slow approach for real-case scenarios. For practicability, we are working on different heuristics, again based on task categories, that could have both theoretical guarantees and

practical efficiency, whether the exact length of a task is known or estimated. These two directions, combined with the practical evaluations of such heuristics in real HPC systems, are worth further investigation.

More generally, the strategies employed in the development of the Batch algorithm, as well as the analytical framework introduced, are entirely novel. Therefore, it would be worth exploring these ideas in similar settings, such as the online scheduling of moldable task graphs, as it could help design efficient heuristics for the worst-case scenario compared to those using local decisions presented in [27]. Additionally, the tools might even be useful for offline problems. For instance, the current best algorithm for the offline scheduling of moldable task graphs uses a two-steps approach, with the first step allocating processors for all the tasks and the second step involving greedy scheduling without considering the position of each task within the graph [7]. Considering the criticality of the tasks, the algorithm and its theoretical analysis should potentially be refined.

References

[1] J. Augustine, S. Banerjee, and S. Irani. Strip packing with precedence constraints and strip packing with release times. *Theoretical Computer Science*, 410(38):3792–3803, 2009.

[2] B. Baker and J. Schwarz. Shelf algorithms for two-dimensional packing problems. *SIAM Journal on Computing*, 12(3):508–525, 1983.

[3] B. S. Baker, E. G. Coffman, and R. L. Rivest. Orthogonal packings in two dimensions. *SIAM Journal on Computing*, 9(4):846855, 1980.

[4] K. P. Belkhale, P. Banerjee, and W. S. Av. A scheduling algorithm for parallelizable dependent tasks. In *IPPS*, pages 500–506, 1991.

[5] A. Benoit, L. Perotin, Y. Robert, and H. Sun. Online scheduling of moldable task graphs under common speedup models. In *ICPP*, pages 51:1–51:11, 2022.

[6] B. Chen and A. P. Vestjens. Scheduling on identical machines: How good is LPT in an on-line setting. *Operations Research Letters*, 21(4):165–169, 1997.

[7] C.-Y. Chen and C.-P. Chu. A 3.42-approximation algorithm for scheduling malleable tasks under precedence constraints. *IEEE Transactions on Parallel and Distributed Systems*, 24(8):1479–1488, 2013.

[8] E. G. Coffman, M. R. Garey, D. S. Johnson, and R. E. Tarjan. Performance bounds for level-oriented two-dimensional packing algorithms. *SIAM J. Comput.*, 9(4):808826, 1980.

[9] J. Csirik and G. J. Woeginger. Shelf algorithms for on-line strip packing. *Information Processing Letters*, 63(4):171–175, 1997.

[10] J. Csirik and G. J. Woeginger. On-line packing and covering problems. In A. Fiat and G. J. Woeginger, editors, *Online Algorithms: The State of the Art*, chapter 7, pages 147–177. Springer, 1998.

[11] G. Demirci, H. Hoffmann, and D. H. K. Kim. Approximation algorithms for scheduling with resource and precedence constraints. In *STACS*, pages 25:1–25:14, 2018.

[12] G. Demirci, I. Marincic, and H. Hoffmann. A divide and conquer algorithm for DAG scheduling under power constraints. In *SC*, 2018.

[13] A. Feldmann, M.-Y. Kao, J. Sgall, and S.-H. Teng. Optimal on-line scheduling of parallel jobs with dependencies. *Journal of Combinatorial Optimization*, 1(4):393–411, 1998.

[14] A. Feldmann, J. Sgall, and S.-H. Teng. Dynamic scheduling on parallel machines. *Theoretical Computer Science*, 130(1):49–72, 1994.

[15] M. R. Garey and R. L. Graham. Bounds for multiprocessor scheduling with resource constraints. *SIAM J. Comput.*, 4(2):187–200, 1975.

[16] M. R. Garey and D. S. Johnson. *Computers and Intractability, a Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, 1979.

[17] R. L. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics*, 17(2):416–429, 1969.

[18] J. L. Hurink and J. J. Paulus. Online algorithm for parallel job scheduling and strip packing. In C. Kaklamanis and M. Skutella, editors, *Approximation and Online Algorithms*, pages 67–74. Springer, 2008.

[19] K. Jansen. A $(3/2+\epsilon)$ approximation algorithm for scheduling moldable and non-moldable parallel tasks. In *SPAA*, pages 224–235, 2012.

[20] K. Jansen and H. Zhang. Scheduling malleable tasks with precedence constraints. In *SPAA*, page 8695, 2005.

[21] K. Jansen and H. Zhang. An approximation algorithm for scheduling malleable tasks under general precedence constraints. *ACM Trans. Algorithms*, 2(3):416–434, 2006.

[22] B. Johannes. Scheduling parallel jobs to minimize the makespan. *J. of Scheduling*, 9(5):433–452, 2006.

[23] R. Lepère, D. Trystram, and G. J. Woeginger. Approximation algorithms for scheduling malleable tasks under precedence constraints. In *ESA*, pages 146–157, 2001.

[24] K. Li. Analysis of the list scheduling algorithm for precedence constrained parallel tasks. *Journal of Combinatorial Optimization*, 3(1):73–88, 1999.

[25] A. Lodi, S. Martello, and M. Monaci. Two-dimensional packing problems: A survey. *European Journal of Operational Research*, 141(2):241–252, 2002.

[26] E. Naroska and U. Schwiegelshohn. On an on-line scheduling problem for parallel jobs. *Inf. Process. Lett.*, 81(6):297–304, 2002.

[27] L. Perotin and H. Sun. Improved online scheduling of moldable task graphs under common speedup models. *ACM Trans. Parallel Comput.*, 11(1), 2024.

[28] J. Turek, J. L. Wolf, and P. S. Yu. Approximate algorithms scheduling parallelizable tasks. In *SPAA*, 1992.

[29] D. Ye, X. Han, and G. Zhang. A note on online strip packing. *Journal of Combinatorial Optimization*, 17(4):417–423, 2009.