RAIRO-Oper. Res. 60 (2026) 831–844
https://doi.org/10.1051/ro/2026027

RAIRO Operations Research
www.rairo-ro.org

OPTIMIZING PARALLEL BATCH SCHEDULING ON UNIFORM MACHINES: A FOCUS ON EQUAL JOB DURATIONS WITH VARIED RELEASE DATES AND SIZES

Shuguang Li*, Yuming Zhang, Zijian Liang and Armand Jean Noel Irangabiye

Abstract. In this paper, we explore a parallel batch scheduling problem, focusing on scenarios where jobs, equal in duration, differ in release dates and sizes, and are processed on uniform machines with varied batch capacities. The objective function to be minimized is makespan, *i.e.*, the maximum completion time of all the jobs. We present two exact algorithms tailored for a scenario characterized by jobs whose sizes are sequentially divisible. Addressing the general context where this divisibility does not hold, we introduce a 2-approximation algorithm which is considered the best achievable in some sense, since improving the approximation ratio superior to 2 is improbable without resolving the P *versus* NP problem.

Mathematics Subject Classification. 90B35, 68Q25.

Received January 11, 2024. Accepted March 2, 2026.

1. Introduction

In the current competitive market, manufacturing firms are increasingly challenged to lower production costs and expedite deliveries to their clients. One strategy to achieve these goals is to implement batch processing in certain manufacturing operations. This approach necessitates not only the usual decisions of assigning and sequencing jobs but also the additional complexity of grouping them. Fowler and Monch [7] have provided an extensive overview of scheduling in the context of parallel batch processing. In this system, a machine has the capability to process multiple jobs jointly within a single batch, with the batch's duration determined by the longest job in it. As highlighted in [7], parallel batch processing plays a critical role in various industries, including semiconductor production, aircraft and automobile manufacturing, and healthcare, among others.

We study a parallel batch scheduling problem on uniform machines with distinct batch capacities where jobs have equal durations but varied release dates and sizes. There is a collection of jobs represented as $\mathcal{J} = \{j|j = 1, 2, \ldots, n\}$ and a collection of uniform parallel batch machines represented as $\mathcal{M} = \{M_i|i = 1, 2, \ldots, m\}$. Each job j is characterized by a *release date* $r_j \geq 0$ (before which the job cannot be processed), a *duration* $p_j = p$ and a *size* $s_j > 0$ (referring to the resource requirement or space that the job occupies within a batch on a machine). Every machine M_i operates at a *speed* $v_i \geq 1$ and has a *batch capacity* K_i. Machine M_i processes v_i units of work per time unit, meaning that a job j assigned to M_i will need p_j/v_i time units to complete, and

Keywords. Scheduling, parallel batch, uniform machines, release dates, job sizes.

School of Computer Science and Technology, Shandong Technology and Business University, Yantai 264005, P.R. China.
*Corresponding author: sgliytu@hotmail.com

© The authors. Published by EDP Sciences, ROADEF, SMAI 2026

This is an Open Access article distributed under the terms of the Creative Commons Attribution License (https://creativecommons.org/licenses/by/4.0), which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

FIGURE 1. The schedule obtained in Example 1.

p_j/v_i is its *processing time* on M_i. These machines are ordered in the non-decreasing order of their capacities. For any job, its size cannot be larger than the largest batch capacity. However, it might occur that a job's size may exceed a machine's capacity so that this job cannot be processed on this machine. Machine M_i is capable of processing several jobs in one batch, provided their cumulative size does not exceed K_i. Importantly, no job can be divided in terms of size, implying that a job can only belong to one batch entirely. In the context of parallel batch scheduling, a batch's *duration* is equal to the longest job's duration in it. Since all the jobs have durations p, each batch's duration is also equal to p. All jobs within a batch start and complete simultaneously. The objective is to develop a schedule minimizing the overall completion time, termed as *makespan.* The makespan is $C_{\max} = \max_j C_j$, where C_j signifies the job j's completion time in the given schedule. For simplicity, let β^* denote "$s_j, p_j = p, p - batch$". Then the problem can be represented using the tri-field notation [2, 10] as $Q|r_j, \beta^*, K_i|C_{\max}$.

Example 1. Consider the following instance of the problem $Q|r_j, \beta^*, K_i|C_{\max}$. There are four jobs J_1, J_2, J_3, J_4 with equal duration $p = 2$, where $r_1 = 0$, $s_1 = 2$, $r_2 = 1$, $s_2 = 3$, $r_3 = 2$, $s_3 = 1$, $r_4 = 0$, $s_4 = 2$. There are also two machines M_1 and M_2, where $v_1 = 1$ and $K_1 = 3$ (slow, small batch capacity), $v_2 = 2$ and $K_2 = 4$ (fast, large batch capacity). We construct a feasible schedule as follows.

Machine M_1 processes jobs J_1 and J_3 in a batch (total size $= 3 \leq 3$). This batch starts at time $t = 2$ (due to J_3's release date). Machine M_2 processes two batches: $\{J_4\}$ at time $t = 0$, and $\{J_2\}$ at time $t = 1$. The resulting schedule is shown in Figure 1. Its makespan is $C_{\max} = 4$, determined by M_1.

Problem $Q|r_j, \beta^*, K_i|C_{\max}$ is classified as strongly NP-hard. This classification is evident from the fact that its special case $1|\beta^*, B|C_{\max}$ (single machine and all jobs have release dates zero) is equivalent to the bin-packing problem [4] and the latter is acknowledged as strongly NP-hard. Furthermore, it has been established that unless $P =$ NP the bin-packing problem cannot attain an approximation ratio better than 3/2 [8]. Dosa *et al.* [6] demonstrated that unless $P =$ NP, $P|\beta^*, B|C_{\max}$ (identical parallel batch machines with all $K_i = B$ and all jobs have release dates zero) cannot attain an approximation ratio better than 2. Consequently, it implies that unless $P =$ NP our problem $Q|r_j, \beta^*, K_i|C_{\max}$ cannot attain an approximation ratio better than 2, and this holds true even under the assumption of zero release dates for all jobs.

Wang and Leung [19] introduced a 2-approximation algorithm for $P|\beta^*, K_i|C_{\max}$, and they went further to develop an algorithm that achieves an asymptotic approximation ratio of 3/2. Ozturk *et al.* [18] presented a 2-approximation algorithm for $P|r_j, \beta^*, B|C_{\max}$. Inspired by a specific aspect of the bin-packing problem [5], they also devised an exact algorithm for $P|r_j, \beta^*, \textit{strongly divisible}, B|C_{\max}$, where "*strongly divisible*" means that the job sizes and batch capacity follow a particular divisibility pattern (to be explained in the next section). Xin *et al.* [20] studied the parallel batch scheduling of jobs with equal durations but varied sizes (with all jobs

having release dates zero) on uniform machines with distinct capacities, focusing on minimizing makespan. They introduced two exact algorithms under a divisibility constraint (to be defined in the next section). Furthermore, they proposed a 2-approximation algorithm to solve the general problem.

Parallel batch scheduling of jobs with equal durations belongs to a specific subclass of parallel batch known as *lot processing*, which is a prevalent mode of production in contemporary manufacturing factories. In these lot scheduling scenarios, numerous jobs of varied sizes can be grouped and processed together as a single lot. The cumulative size of all jobs within any given lot must not exceed the machine's lot capacity. Importantly, the processing duration for each lot remains constant and does not vary based on the quantity of jobs being processed in that lot.

Lot processing can be exemplified by the burn-in phase of integrated circuit (IC) final testing in semiconductor manufacturing. In this scenario, an oven at any given moment may handle various ICs, originating from distinct customer orders that arrive dynamically. The processing duration for each lot is standardized and dictated by the specific parameter needs of the IC products. These customer orders vary significantly in terms of product quantity and delivery time constraints. Here, the quantity of products in an order can be analogous to the "job size" in scheduling terms. It is crucial for efficiency that the finished products are swiftly moved to the next stage of the production process.

In the field of lot scheduling, significant advancements have been made by various researchers. Hou *et al.* [11] demonstrated the efficacy of the Smallest Size First (SSF) rule for achieving optimal solutions to minimize the total completion time in the context of single machine lot scheduling involving splitable jobs, where the splitting of jobs between consecutive lots is permitted. Yang *et al.* [21] extended this to non-splitable jobs, revealing its unary NP-hardness and proposing binary integer programming and heuristic solutions. Zhang *et al.* [22] highlighted the effectiveness of the Weighted Smallest Size First (WSSF) rule when applied to address the total weighted completion time minimization problem involving splitable jobs scheduled on a single machine. Geng and Liu [9] studied bicriteria lot scheduling involving splitable jobs and encompassing two non-disjoint agents. In this context, jobs were considered eligible for size splitting and processing in consecutive lots, and jobs from distinct agents were allowed to be processed together in a common batch. They demonstrated the unary NP-hard complexity of the problem when aiming to minimize the total weighted completion time for one agent's jobs, while simultaneously imposing a constraint that limits the maximum cost of the other agent's jobs not to exceed a specified threshold. Additionally, they developed an $O(n^4)$-time Pareto optimal algorithm for minimizing the total completion time for one agent while simultaneously controlling the maximum cost for the other. Mor *et al.* [16] provided insights into the single machine problem of minimizing the number of tardy jobs, both splitable and non-splitable, with the former being addressed through an exact algorithm and the latter through a heuristic method. Moreover, they also proved that for the case of splitable jobs, minimizing the total weighted number of tardy jobs is NP-hard, and for the case of non-splitable jobs, minimizing the total number of tardy jobs is strongly NP-hard. Furthering these studies, Mor [15] addressed variable lot processing times, presenting algorithms for minimizing the following objectives: total completion time, makespan, and the linear combination of the makespan and the total completion time. Chen *et al.* [3] expanded upon Zhang *et al.*'s [22] research by introducing variability in processing times, capacities, and sizes of jobs in lot scheduling. In this more complex scenario, they affirmed the efficacy of the WSPT (Weighted Shortest Processing Time first) rule, proving its optimality even under these generalized conditions. Nurit *et al.* [17] examined two job splitting approaches on identical parallel machines: consecutive and parallel splitting. Their aim is to minimize both total completion time and makespan. For the consecutive splitting model, they designed a dynamic programming algorithm capable of solving both objectives in pseudo-polynomial time. For the parallel splitting model, they showcased that both objectives can be addressed efficiently in polynomial time.

As far as our knowledge extends, $Q|r_j, \beta^*, K_i|C_{\max}$ remains unexplored in existing research. We broaden the insights gained from [18–20], generalizing them to uniform parallel batch machines while incorporating varied release dates and distinct machine capacities. The presence of non-zero release dates destroys the "availability-at-time-zero" property that underpins several key simplifications in [20]. This forces the development of new feasibility tests, new structures, and a different approach to constructing and certifying schedules. For the

special case of sequentially divisible job sizes, we provide two exact algorithms (AssignmentA and AssignmentA1) that are explicitly designed to handle release dates and uniform machine speeds. The procedures (AssignmentA and AssignmentA1) incorporate release-date feasibility checks at every step, and their correctness proofs rely on a new alignment argument that respects both release dates and divisibility. For the general problem, while Xin *et al.* [20] gave a 2-approximation for the case without release dates, extending that result to the setting $Q|r_j, \beta^*, K_i|C_{\max}$ is not straightforward. We design a new feasibility-checking procedure (AssignmentB) that works directly with release dates and uniform machine speeds, and prove that its failure implies a lower bound larger than the target makespan. We obtain a 2-approximation algorithm for $Q|r_j, \beta^*, K_i|C_{\max}$, which is considered the best achievable in some sense. This is because improving the approximation ratio superior to 2 is improbable without resolving the P *versus* NP problem.

Before moving forward, let us establish several key notations for clarity. Let $K_0 = 0$. Let $\mathcal{J}_i = \{j \in \mathcal{J} | K_{i-1} < s_j \le K_i\}$,$i = 1, 2, \ldots, m$. If $\mathcal{J}_i$ contains no job then let $\mathcal{J}_i = \emptyset$. Clearly, $\mathcal{J} = \bigcup_{i=1}^{m} \mathcal{J}_i$.

The organization of the paper is presented as follows. Section 2 discusses a special case of $Q|r_j, \beta^*, K_i|C_{\max}$ where job sizes are sequentially divisible, denoted as $Q|r_j, \beta^*, \text{divisible}, K_i|C_{\max}$. Two exact algorithms are presented for this case. In Section 3, we propose a 2-approximation algorithm for $Q|r_j, \beta^*, K_i|C_{\max}$. Finally, in Section 4, we wrap up the paper, reflecting on our findings and proposing future direction in this field.

2. Two algorithms for $Q|r_j, \beta^*, \text{divisible}, K_i|C_{\max}$

In this section, our attention is on a special case where $\mathcal{J}$ possesses *divisible sizes*. That is, for any two jobs j and j' in $\mathcal{J}$, when $s_j \le s_{j'}$, s_j is an exact divisor of $s_{j'}$. In a different phrasing, we express that the job sizes are *sequentially divisible*. When considering a batch capacity B, if $\mathcal{J}$ has divisible sizes, then we call $(\mathcal{J}, B)$ *weakly divisible*. Furthermore, we call $(\mathcal{J}, B)$ *strongly divisible* if, additionally, the size of the biggest job is an exact divisor of B [5]. The problem concerning weakly divisible sizes is denoted as $Q|r_j, \beta^*, \text{divisible}, K_i|C_{\max}$.

Lemma 2.1 ([5])**.** *When $\mathcal{J}$ possesses divisible sizes, any subset of $\mathcal{J}$ consisting of jobs whose sizes are individually not larger than s_j (the size of job j) and collectively amount to at least s_j will definitely include a subset whose size is precisely s_j.*

Let OPT represent the optimal schedule's makespan for $Q|r_j, \beta^*, \text{divisible}, K_i|C_{\max}$. Define $\Lambda = \{r_j + kp/v_i | 1 \le j \le n, 1 \le k \le n, \text{ and } 1 \le i \le m\}$. The subsequent lemma, which is derived from [13], is applicable for $Q|r_j, \beta^*, \text{divisible}, K_i|C_{\max}$.

Lemma 2.2. *As defined above, Λ encompasses all potential candidates for the value of OPT.*

By Lemma 2.2, on each machine there are at most n^2 possible values for OPT. A method provided by Li and Li [14] enables these values to be sorted in $O(n^2)$ time. Given that $|\Lambda| \le mn^2$, we evaluate no more than mn^2 potential values for OPT. We utilize the divide-and-conquer method outlined in [1] to arrange these values in the increasing order. This is achieved in $O(mn^2 \log m)$ time by merging m ordered lists, each of a length of $O(n^2)$.

We sort the jobs in $\mathcal{J}_i$ $(i = 1, 2, \ldots, m)$ in the non-increasing order based on their sizes, resulting in the ordered list denoted as J_i. Let $J = J_m || J_{m-1} || \cdots || J_1$.

We determine the value of OPT in $O(\log(mn))$ iterations by employing the binary search. During each iteration, a target value T is selected and we strive to create a schedule whose makespan is no more than T, by AssignmentA procedure given below, or claim that such a schedule does not exist. For any specified T, AssignmentA runs in $O(mn \log m + mn^2)$ time. Consequently, the algorithm for $Q|r_j, \beta^*, \text{divisible}, K_i|C_{\max}$ runs in $O(mn \log m \cdot \log(mn) + mn^2 \log(mn))$ time.

In AssignmentA described below, a job is called *feasible* for a batch, and vice versa, if its release date and size are no more than the start time and capacity of the batch, respectively. An *empty batch* on a machine is a batch that has been created on the machine but not yet been assigned any jobs. Each empty batch has a start

time (determined by the machine's speed and target makespan T), a capacity (equal to the machine's batch capacity), and a duration (equal to p, since all jobs have equal duration p).

The variable U represents the set of unscheduled jobs. At the start of AssignmentA, U is initialized as J (as defined earlier). During execution, jobs are removed from U once they are successfully assigned to a batch. If U becomes empty ($U = \emptyset$), AssignmentA succeeds, indicating that all jobs have been scheduled; otherwise, if there is any job left in U after processing all possible batches, AssignmentA fails to find a feasible schedule.

AssignmentA(T)

Step 1. Let $U = J$. For $i = 1, 2, \ldots, m$, assign $\min\{n, \lfloor T \cdot v_i/p \rfloor\}$ empty batches to machine M_i. All these batches have durations p and capacities K_i. Let M_i process these batches in succession (without any gaps of idle time between them), ensuring the final batch is completed precisely at time T.

Step 2. Arrange all the empty batches on the m machines by the increasing order based on their start times, with ties being broken by prioritizing the machine with the highest index. Mark them in order as $X_1, X_2, \ldots, X_z$.

Step 3. For $h = 1, 2, \ldots, z$ (this ordering is used crucially), do:
If $U \neq \emptyset$, then proceed to scan U from the first job to the last one. For each scanned job, if it is feasible for batch X_h and there is enough space in X_h to accommodate the job, then place the job in X_h and delete it from U.

Step 4. If $U \neq \emptyset$, then the procedure fails to generate a feasible schedule. If $U = \emptyset$, then a feasible schedule is generated. (Clean up the schedule by deleting all the empty batches in it.)

Example 2. We give an example to illustrate how the procedure works. There are two uniform machines M_1 and M_2, where M_1 has speed $v_1 = 1$ and capacity $K_1 = 4$, while M_2 has speed $v_2 = 2$ and capacity $K_2 = 8$. There are four jobs J_1, J_2, J_3, J_4 with equal duration $p = 2$ and sequentially divisible sizes, where $r_1 = 0$, $s_1 = 1$, $r_2 = 0$, $s_2 = 2$, $r_3 = 1$, $s_3 = 4$, $r_4 = 2$, $s_4 = 8$. Let the target makespan $T = 6$.

In Step 1, AssignmentA initializes $U = [J_4, J_3, J_2, J_1]$ (jobs sorted by size in non-increasing order). It creates empty batches on each machine as follows.

Machine M_1 gets $\lfloor 6 \times 1/2 \rfloor = 3$ batches: B_1: starts at 6–2/1 = 4, capacity 4; B_2: starts at 6–4/1 = 2, capacity 4; B_3: starts at 6–6/1 = 0, capacity 4.

Machine M_2 gets $\lfloor 6 \times 2/2 \rfloor = 6$ batches: B_4: starts at 5, capacity 8; B_5: starts at 4, capacity 8; B_6: starts at 3, capacity 8; B_7: starts at 2, capacity 8; B_8: starts at 1, capacity 8; B_9: starts at 0, capacity 8.

In Step 2, all empty batches are marked as $X_1 = B_9$, $X_2 = B_3$, $X_3 = B_8$, $X_4 = B_7$, $X_5 = B_2$, $X_6 = B_6$, $X_7 = B_5$, $X_8 = B_1$, $X_9 = B_4$, by the increasing order based on their start times, with ties being broken by prioritizing the machine with the highest index.

In Step 3, we iterate over the empty batches $X_1, X_2, \ldots, X_9$ in order, and for each batch we scan the remaining jobs in U (in the fixed order by non-increasing size) and assign every job that is feasible for that batch and fits into the remaining capacity.

Initially, $U = \{J_4, J_3, J_2, J_1\}$. For batch X_1, since it starts at time 0 while $r_4 = 2$ and $r_3 = 1$, X_1 cannot accommodate jobs J_4 and J_3. The procedure assigns J_2 and J_1 to X_1 and X_1 now has remaining capacity 5. Update $U = \{J_4, J_3\}$. For batch X_2, no job is assigned. For batch X_3, J_3 is assigned to it and its remaining capacity becomes 4. Update $U = \{J_4\}$. For batch X_4, J_4 is assigned to it and its remaining capacity becomes 0. Update $U = \emptyset$.

In Step 4, since $U = \emptyset$, the following feasible schedule is generated, which is depicted in Figure 2. Machine M_2 processes all the four jobs. It processes J_1 and J_2 in one batch which starts at time 0 and completes at time 1. It then processes J_3 in the second batch which starts at time 1 and completes at time 2. Finally, it processes J_4 in the third batch which starts at time 2 and completes at time 3. After removing empty batches, the makespan is 3 which is no more than $T = 6$.

Lemma 2.3. *If* OPT $\leq T$, *AssignmentA will create a feasible schedule in* $O(mn \log m + mn^2)$ *time for* $Q|r_j, \beta^*, divisible, K_i|C_{\max}$, *ensuring that the makespan is no more than* T.

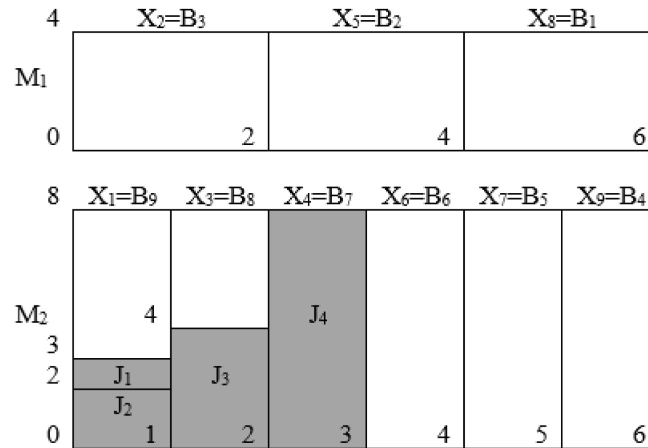

FIGURE 2. The schedule obtained in Example 2.

Proof. To establish the claim, we must demonstrate that if $\mathrm{OPT} \leq T$, then $U = \emptyset$ will hold upon the completion of AssignmentA.

Consider an optimal schedule Σ^*. In this schedule, if any batch on machine M_i does not start at time $T - kp/v_i$ for an integer $k > 0$, then we delay the batch so that it starts at time $T - k'p/v_i$, where the integer $k' > 0$ is chosen such that $T - k'p/v_i$ is at least as much as the original start time of the batch. This adjustment is repeated for each batch on every machine in Σ^*, in the decreasing order of the start times of the batches. Given that $\mathrm{OPT} \leq T$, this modification will result in a feasible schedule, which we denote as $\bar{\Sigma}^*$. This schedule will have a makespan of T and will be structured so that on each machine, batches are processed consecutively (with no idle time in between), with the final batch completing precisely at time T.

Consider Σ as the schedule produced by AssignmentA following Step 3. Number all the batches in Σ in the increasing order based on their start times, designating them as $X_1, X_2, \ldots, X_z$. It is possible that some batches within $X_1, X_2, \ldots, X_z$ are empty. Accordingly, identify the corresponding batches in $\bar{\Sigma}^*$ as $\bar{X}_1^*, \bar{X}_2^*, \ldots, \bar{X}_z^*$. Similarly, there may be empty batches among $\bar{X}_1^*, \bar{X}_2^*, \ldots, \bar{X}_z^*$.

We aim to adjust $\bar{\Sigma}^*$ so that the batches $\bar{X}_1^*, \bar{X}_2^*, \ldots, \bar{X}_z^*$ are transformed into $X_1, X_2, \ldots, X_z$ one after the other. This process, once completed, will convert $\bar{\Sigma}^*$ into Σ. Consequently, this implies that $U = \emptyset$ upon the completion of AssignmentA.

To simplify our explanation, let us assume temporarily that batch X_1 contains jobs $1, 2 \ldots, x$ such that $s_1 \geq s_2 \geq \cdots \geq s_x$. Suppose that $\bar{X}_1^*$ includes jobs $1, 2 \ldots, a$, but excludes job $a+1$, where $a+1 \leq x$. Define $\bar{X}^*$ as the batch in $\bar{\Sigma}^*$ that contains job $a+1$. Let A be the set of jobs present in $\bar{X}_1^*$ but absent in X_1. According to Step 3 of AssignmentA, each job in A has size no larger than s_{a+1}. Based on Lemma 2.1, for the jobs in A, there are two possibilities: the cumulative size of these jobs is less than s_{a+1}, or there exists a specific subset within these jobs whose size collectively equals s_{a+1}. In each of these cases, we can swap the relevant jobs in $\bar{X}_1^*$ with job $a+1$ in $\bar{X}^*$, ensuring that job $a+1$ is included in the (modified) $\bar{X}_1^*$. This modification procedure is iteratively applied until batch X_1 appears in (modified) $\bar{\Sigma}^*$.

Continue the aforementioned process, modifying batches $\bar{X}_1^*, \bar{X}_2^*, \ldots, \bar{X}_z^*$ in sequence, until $\bar{\Sigma}^*$ is transformed into Σ. Therefore, AssignmentA is guaranteed to generate a feasible schedule with makespan no more than T.

Analyzing the time complexity of AssignmentA is straightforward. Step 1 can be executed in $O(mn)$ time, considering there are only $O(mn)$ empty batches. Sorting these batches in the non-decreasing order of their start times can be done in $O(mn \log m)$ time. This is accomplished by merging m ordered lists, each of length $O(n)$, using the divide-and-conquer method mentioned in [1]. Consequently, Step 2 can be completed in $O(mn \log m)$

time. As for Step 3, it requires $O(n)$ time per batch to fill, leading to an $O(mn^2)$ overall time requirement for this step. Thus, AssignmentA runs in $O(mn \log m + mn^2)$ time. $\square$

Based on the preceding discussion and analysis, we can establish the following theorem:

Theorem 2.4. *There exists an exact algorithm, which combines binary search with AssignmentA, for solving* $Q|r_j, \beta^*, divisible, K_i|C_{\max}$ *that runs in* $O(mn \log m \cdot \log(mn) + mn^2 \log(mn))$ *time.*

In implementing AssignmentA, our current method emphasizes a batch-focused strategy, where batches are filled greedily in an order that prioritizes their earliest start times. However, an alternative approach centers on the jobs themselves. In this job-focused method, jobs are assigned based on their sizes in a descending order, and each job is assigned to the most suitable batch available – one that not only has the capacity to fit the job but also is the earliest to start among all qualifying batches.

Example 3. We use the same instance as presented in Example 2 to demonstrate the application of the job-focused method.

Steps 1 and 2 remain identical to those in Example 2. In Step 3, for J_4, since $r_4 = 2$ and $s_4 = 8$, the procedure scans the batches until X_4. Therefore, J_4 is assigned to X_4 and X_4 now has remaining capacity 0. Update $U = [J_3, J_2, J_1]$. For J_3, since $r_3 = 1$ and $s_3 = 4$, the procedure scans the batches until X_3. Therefore, J_3 is assigned to X_3 and X_3 now has remaining capacity 4. Update $U = [J_2, J_1]$. For J_2, since $r_2 = 0$ and $s_2 = 2$, the procedure assigns J_2 to X_1 and X_1 now has remaining capacity 6. Update $U = [J_1]$. For J_1, since $r_1 = 0$ and $s_1 = 1$, the procedure assigns J_1 to X_1 and X_1 now has remaining capacity 5. Update $U = \emptyset$. Finally, we get the same schedule (depicted in Fig. 2) as the batch-focused iteration described in Example 2.

The distinction between batch-focused and job-focused methods originates in Step 3 of Assignment A, where the order of the nested loops is simply reversed. While such a reversal is generally not harmless for greedy assignment rules, these two methods are guaranteed to coincide in the specific setting of weakly divisible job sizes. This equivalence can be justified using an argument similar to the one employed in the proof of Lemma 2.3.

Next, we describe an improved implementation of the job-focused method, called AssignmentA1. This slight modification achieves an improved time complexity of $O(mn \log m + n \log(mn))$.

AssignmentA1(T)

The procedure is the same as AssignmentA, except for

Step 3. Construct a binary tree to store the batches as described in the following proof of Lemma 2.5. Scan U from the first job to the last. For every scanned job, from among the feasible batches having enough space to accommodate the job, select the one which starts earliest. Fill the job into the chosen batch. Then, remove it from U. If it turns out that the job cannot be accommodated in any batch, the procedure is deemed to have failed and thus terminates.

Lemma 2.5. *If* OPT $\leq T$*, AssignmentA1 will create a feasible schedule in* $O(mn \log m + n \log(mn))$ *time for* $Q|r_j, \beta^*, divisible, K_i|C_{\max}$*, ensuring that the makespan is no more than* T*.*

Proof. The schedule generated by AssignmentA1 is the same as the one generated by the job-focused version of AssignmentA. The correctness of AssignmentA1 follows.

AssignmentA1 allows a faster implementation than AssignmentA. For this goal, we will adapt the binary tree introduced in [12]. In Step 2, the binary tree has $\sum_{i=1}^{m} \min\{n, \lfloor T \cdot v_i/p \rfloor\} = O(mn)$ leaves associated with the empty batches $X_1, X_2, \ldots, X_z$, arranged from left to right in ascending order based on their start times. The tree has depth $O(\log(mn))$.

The initial binary tree is constructed as follows:

First, insert a new node as the root, with its left son set as X_1. Since the root is the lowest node of degree one, we include X_2 in the tree as the right son of the root. At this stage, because the tree contains no node of

degree one, we insert another new node as the tree root, making the old root its left son and setting its right son as X_3.

Next, observe that X_3 is a leaf but not at the same depth as the other leaves (namely X_1 and X_2). To address this, we insert a new node as the right son of the current root, and attach X_3 as the left son of this new node.

We proceed in this manner, incorporating batches $X_4, X_5, \ldots, X_z$ one after another into the binary tree, while ensuring that all batches remain as leaves and are all placed at the same depth. During the construction, for any node of the binary tree, its left son is attached first; if an additional son is attached later, it becomes the right son.

Each node is labelled with two values f and g (called the *f-value* and the *g-value* respectively). These labels are assigned in the following manner: For a leaf, f and g store respectively the start time and the unused capacity of the corresponding batch. For a non-leaf with two sons whose labels are f_1, g_1 and f_2, g_2 respectively, $f = \max\{f_1, f_2\}$ and $g = \max\{g_1, g_2\}$. Therefore, for every internal node in the tree, f and g respectively store the maximum start time and the maximum unused capacity across all the batches associated with the leaves descending from that node. It is worth to note that the f-values of the nodes in the binary tree keep unchanged throughout the procedure.

We scan U in Step 3. Whenever a job is scanned, the binary tree is utilized to either allocate it to a batch, or conclude that such an allocation is impossible, leading to the termination of the process.

Precisely, for each job, the procedure examines the f-value of the tree root. This evaluation leads to two distinct cases:

Case 1. The f-value of the root is equal to or greater than the job's release date.

In this case, there exists at least one batch that starts at or after the job's release date. Subsequently, the g-value of the tree root is examined, leading to two possible subcases:

Subcase 1.1. The g-value of the root is equal to or greater than the job's size.

To find the batch which starts earliest among all the feasible batches having enough space to accommodate the job, we need to traverse the binary tree twice.

During the first traversal, starting at the tree root r, the path progresses towards a leaf u_1 by consistently choosing the leftmost node where the f-value is not less than the job's release date. Let X_g represent the batch associated with the destination leaf u_1. Evidently, only batches $X_g, X_{g+1}, \ldots, X_z$ start at or after the job's release date.

The path leading from r to u_1 is referred to as a *restricting path*. According to the restricting path, all the nodes of the tree can be partitioned into three parts: the *left part* consisting of the nodes placed on the left of the restricting path, the *middle part* consisting of the nodes on the restricting path, and the *right part* consisting of the nodes placed on the right of the restricting path. More precisely, if a node is not in the middle part and is the left son of a node in the middle part, then it is in the left part, and all its descendants are in the left part. If a node is not in the middle part and is the right son of a node in the middle part, then it is in the right part, and all its descendants are in the right part.

Imagine that the nodes in the left part have been removed. The desired effect is to set implicitly the *temporary* g-values of the nodes in the left part to be zero. All the temporary g-values of the nodes in the right part are the same as their g-values. To explicitly compute the temporary g-values of the nodes on the restricting path, proceed by retracing the restricting path from back to the root r. During this reverse traversal, assign each internal node a temporary g-value, which is the larger of the temporary g-values of its (up to) two sons. (We do not need to store the temporary g-values of the nodes in the left part. The way for computing the temporary g-values of the nodes on the restricted path can be described equivalently as follows. Mark the nodes on the restricting path as $u_1, u_2, \ldots, u_l = r$, back up the restricting path. Let the temporary g-value of u_1 be equal to its g-value. For $a = 1, 2, \ldots, l-1$, if u_a is the left son of u_{a+1}, then the temporary g-value of u_{a+1} is equal to the larger of the temporary g-values of its two sons; Otherwise, the temporary g-value of u_{a+1} is equal to the temporary g-value of u_a.)

Should the temporary g-value of the root be smaller than the job's size, it indicates that the job cannot be accommodated in any batch. This is because the job's size exceeds the largest available unused capacity across all feasible batches. In such a case, the procedure must terminate.

In the second traversal, the journey begins at the tree root r and proceeds to a leaf w by consistently selecting the leftmost node whose temporary g-value is at least as large as the job's size. The batch associated with this destination leaf w is denoted by X_h. Clearly, X_h is the first batch in $X_g, X_{g+1}, \ldots, X_z$ which has enough space to accommodate the job. The job should be placed into X_h and subsequently removed from U.

Imagine that the nodes in the left part come back. Use the g-values of the nodes of the tree. (We need only to do so for the nodes on the restricting path.) The path from r to w is an *update path*. To update the g-values of the nodes on this path, follow these steps: First, adjust the g-value of w to its current value minus the job's size. Then, move back up the update path, at each internal node updating its g-value by the larger of the g-values of its two sons.

Subcase 1.2. The g-value of the root is less than the job's size.

Since the job's size exceeds the largest available unused capacity across all batches, it cannot be allocated to any batch. Consequently, the procedure has to terminate.

Case 2. The f-value of the root is less than the job's release date.

In this case, the job cannot be assigned to any batch because its release date is larger than the largest start time across all the batches. Therefore, the procedure has to terminate.

The time complexity of AssignmentA1 is straightforward to determine. As proved in Lemma 2.3, completing Step 1 takes $O(mn)$ time. Sorting the batches in the increasing order by their start times in Step 2 is achievable in $O(mn \log m)$ time. Building the binary tree in Step 3 is a $O(mn)$-time operation, given that the tree has only $O(mn)$ leaves and a depth of $O(\log(mn))$, necessitating merely $O(mn)$ comparisons. During Step 3, each scanned job when we scan U can be allocated to a batch (Subcase 1.1) or identified as unassignable, leading to termination of the procedure (Subcase 1.1, Subcase 1.2, and Case 2) in $O(\log(mn))$ time. Consequently, Step 3 can be executed in $O(n \log(mn))$ time. It follows that AssignmentA1 can be implemented in $O(mn \log m + n \log(mn))$ time.

□

Example 4. We use the same instance as presented in Example 2 to illustrate how AssignmentA1 works. The initial binary tree is shown in Figure 3, with the (f, g)-value of each node already displayed within the node. The empty batches $X_1, X_2, \ldots, X_z$ associated with the leaves of the tree are arranged from left to right in ascending order based on their start times (ties broken in favor of the machine with the highest index).

In Step 3, for J_4, since $r_4 = 2$, while the f-value of the root is 5, there is at least one batch that starts no earlier than J_4's release date. Since g-value of the root is 8 and $s_4 = 8$, we proceed to find the restricting path $v_1v_2v_4v_8X_4$, and then compute the temporary g-values of the nodes in the tree, which are shown in Figure 4.

Since the temporary g-value of the root is 8 and $s_4 = 8$, we proceed to find the update path $v_1v_2v_4v_8X_4$. Place the job J_4 into X_4, and then remove it from U. The binary tree after placing J_4 into X_4 is shown in Figure 5.

For J_3, since $r_3 = 1$, while the f-value of the root is 5, there is at least one batch that starts no earlier than J_3's release date. Since g-value of the root is 8 and $s_3 = 4$, we proceed to find the restricting path $v_1v_2v_4v_8X_3$, and then compute the temporary g-values of the nodes in the tree, which are shown in Figure 6.

Since the temporary g-value of the root is 8 and $s_3 = 4$, we proceed to find the update path $v_1v_2v_4v_8X_3$. Place the job J_3 into X_3, and then remove it from U. The binary tree after placing J_3 into X_3 is shown in Figure 7.

The procedure continues and assigns J_2 to X_1, J_1 to X_1. Finally, we get the same schedule as the one generated in Example 2. The final binary tree is depicted in Figure 8.

We get the following theorem.

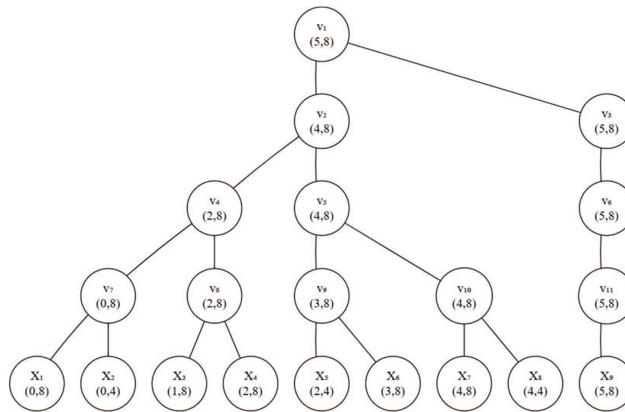

Figure 3. The initial binary tree.

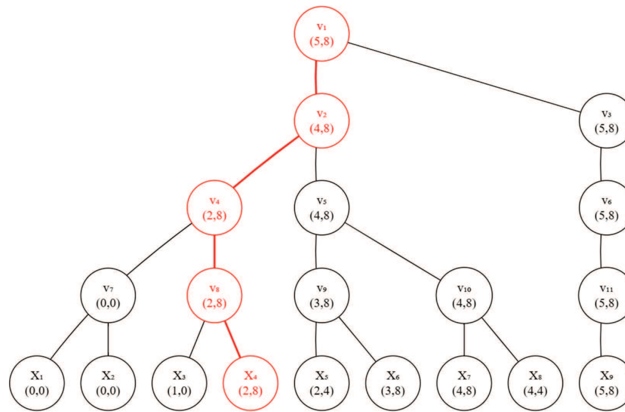

Figure 4. The temporary g-values of the nodes for J_4.

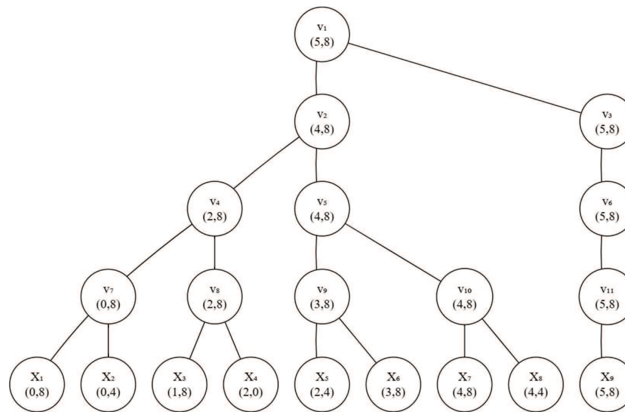

Figure 5. The binary tree after placing J_4 into X_4.

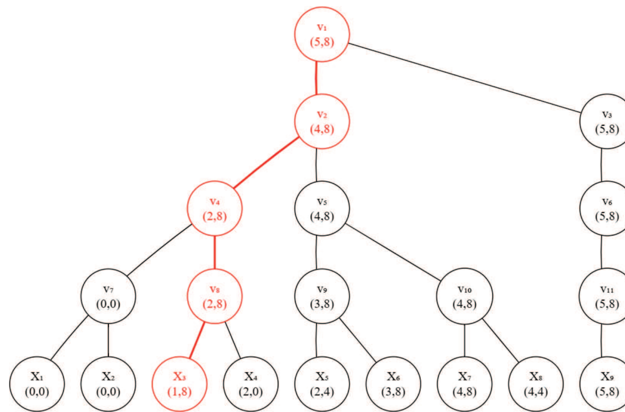

FIGURE 6. The temporary g-values of the nodes for J_3.

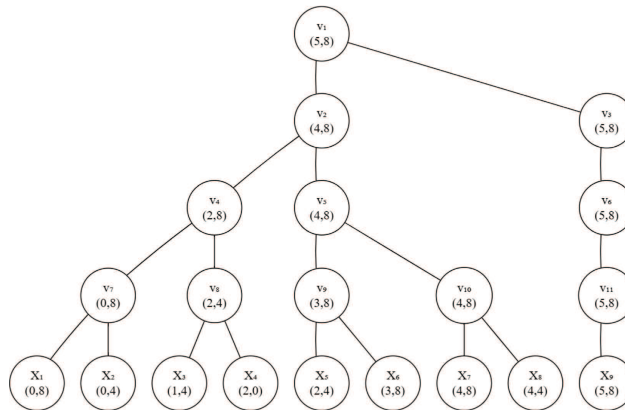

FIGURE 7. The binary tree after placing J_3 into X_3.

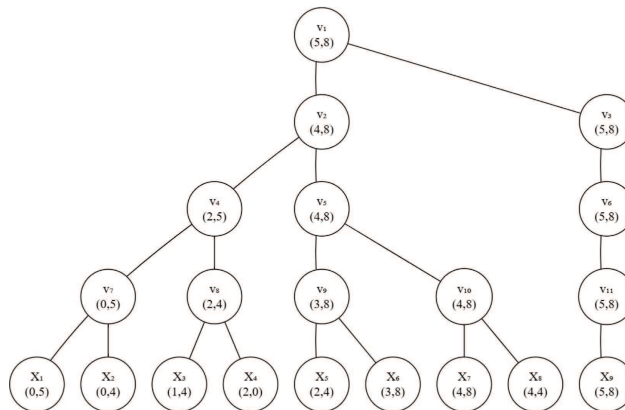

FIGURE 8. The final binary tree.

Theorem 2.6. *There exists an exact algorithm, which combines binary search with AssignmentA1, for solving $Q|r_j,\beta^*,divisible,K_i|C_{\max}$ that runs in $O(mn\log m\cdot\log(mn)+n\log^2(mn))$ time.*

3. An algorithm for $Q|r_j,\beta^*,K_i|C_{\max}$

In this section, we study the general case which removes the assumption of job sizes being sequentially divisible, *i.e.*, $Q|r_j,\beta^*,K_i|C_{\max}$.

Let us denote by OPT the optimal makespan for $Q|r_j,\beta^*,K_i|C_{\max}$. Note that Lemma 2.2 remains applicable to $Q|r_j,\beta^*,K_i|C_{\max}$ as well. Consequently, we limit our consideration to no more than mn^2 potential values for OPT. These values can be efficiently arranged in the increasing order in $O(mn^2\log m)$ time, using the divide-and-conquer method outlined in [1]. Then, a binary search is employed to determine the value of OPT, which can be accomplished in $O(\log(mn))$ iterations.

Initially, we arrange all the jobs in $\mathcal{J}_i$ $(i=1,2,\ldots,m)$ in the non-increasing order based on their **release dates**. Denote this ordered list as J_i. It should be stressed that we use J_i here with a different meaning from that in the previous section.

Within the binary search process, we employ the AssignmentB procedure as follows: For each chosen value of T, AssignmentB seeks a schedule with makespan no more than T, while allowing for batches containing one job in excess of their designated capacity. Such batches are called *one-job-overfull batches.*

AssignmentB(T)

Step 1. Let $Q_0\leftarrow\emptyset$.

Step 2. For $i=1,2,\ldots,m$, perform the following actions:

- (i) Merge Q_{i-1} into J_i to get Q_i. Set $t=T$.
- (ii) Let $t=T-p/v_i$. If $t\geq 0$, let $D_i(t)$ denote the ordered set of the jobs in Q_i with release dates no more than t. The jobs in $D_i(t)$ are arranged in the non-increasing order based on their release dates. If $D_i(t)\neq\phi$, then let M_i start to process a new batch of duration p and capacity K_i at time t. Continuously pick the first job from $D_i(t)$ and include it in the new batch until the cumulative size of the jobs in this batch exceeds K_i, or until there are no remaining jobs in $D_i(t)$ to pick. Afterward, remove the newly selected jobs from Q_i.
- (iii) Repeat Step 2(ii) until $t<0$ or until there is no job in $D_i(t)$.

Lemma 3.1. *If* OPT $\leq T$*, then AssignmentB procedure will create a schedule in $O((m+n)\log n)$ time for $Q|r_j,\beta^*,K_i|C_{\max}$ such that the schedule may contain one-job-overfull batches while ensuring that the makespan does not exceed T.*

Proof. We prove the correctness of AssignmentB by contradiction. Suppose that OPT $\leq T$ but AssignmentB fails to produce a schedule – even allowing one-job-overfull batches – with makespan at most T. Then, at the end of AssignmentB, there remains at least one unassigned job $j\in Q_m$. Assume, without loss of generality, that job j can be processed only on machines $M_i,M_{i+1},\ldots,M_m$. According to the AssignmentB rules, each of these machines M_l $(l=i,i+1,\ldots,m)$ must have exactly $\lfloor(T-r_j)v_l/p\rfloor$ one-job-overfull batches that contain only jobs released no earlier than r_j. If this were not the case, job j could still be assigned to one of these machines and finished by time T, contradicting the assumption that it remains unassigned.

Let V be the total size of all jobs with release dates $\geq r_j$ that are assigned by AssignmentB to machines $M_i,M_{i+1},\ldots,M_m$ plus the size s_j of job j. From the above observation, we have: $V>\sum_{l=i}^{m}\lfloor(T-r_j)v_l/p\rfloor\cdot K_l+s_j$.

AssignmentB assigns jobs greedily to machines with smaller indices (*i.e.*, smaller capacities), favoring jobs with larger release dates and packing each batch as fully as possible – even allowing it to become one-job-overfull. Consequently, V is a lower bound on the total size of jobs released no earlier than r_j that must be processed on machines $M_i,M_{i+1},\ldots,M_m$ in any feasible schedule.

The inequality above implies that in any schedule with makespan at most T, the total capacity available on these machines for processing such jobs is insufficient to accommodate a total job size of V. Hence, no schedule with makespan $\leq T$ can exist, contradicting the assumption that OPT $\leq T$. Therefore, AssignmentB must succeed in producing a schedule with makespan at most T, possibly containing one-job-overfull batches.

Step 2(i) can be completed in $O(n)$ time. Moving on to Step 2(ii), a binary search is conducted within Q_i to identify the job with the largest release date among all the jobs in Q_i with release dates no later than t. Consequently, $D_i(t)$ can be determined in $O(\log n)$ time. If $D_i(t) = \emptyset$, we proceed directly to the next machine. Since each batch includes at least one job, even though there are mn possible batches, we need to examine at most $m + n$ batches. Step 2 for all iterations can be done in $O((m + n) \log n)$ time. In summary, AssignmentB can be completed in $O((m + n) \log n)$ time. □

We get:

Theorem 3.2. *There exists a 2-approximation algorithm for $Q|r_j, \beta^*, K_i|C_{\max}$ that can be executed in $O(mn^2 \log m)$ time.*

Proof. The algorithm employs a binary search framework to estimate the optimal makespan for $Q|r_j, \beta^*, K_i|C_{\max}$. At each iteration of the binary search, the AssignmentB procedure is invoked to generate (when successful) a schedule that permits one-job-overfull batches (*i.e.*, batches where the total job size may exceed the machine capacity by at most one job). From the set of candidate schedules produced during the binary search, the algorithm selects the one with the smallest makespan. By Lemma 3.1, this schedule has makespan no more than the optimal value. To convert this schedule into a strictly feasible one (where no batch violates capacity constraints), for each one-job-overfull batch, we open a new batch for the largest (in size) job in it and process the new batch immediately after it on the same machine. Clearly, the obtained schedule is a feasible schedule and has makespan no more than twice of the optimal value. □

It is worth noting that a critical bottleneck operation in the algorithm is the sorting procedure carried out just once at the outset of the process.

4. Conclusions

In this paper, we studied a parallel batch scheduling problem aiming at minimizing the makespan. We focused on scenarios where jobs, equal in duration, differ in release dates and sizes, and are processed on uniform machines with varied batch capacities. Our contributions include two exact algorithms designed to address this problem under a divisibility constraint, along with a 2-approximation algorithm applicable to the general case which eliminates the divisibility constraint. Looking ahead, a potential avenue for future research involves exploring some other objective functions for scheduling problems on uniform parallel batch machines, considering the presence of varied release dates and job sizes.

Funding

This work is supported by Natural Science Foundation of Shandong Province China (No. ZR2020MA030).

References

[1] A.V. Aho, J.E. Hopcroft and J.D. Ullman, The Design and Analysis of Computer Algorithms. Addison-Wesley, Boston (1974).

[2] P. Brucker, Scheduling Algorithms, 5th edition. Springer, Berlin (2007).

[3] Y. Chen, Y. Cheng and G. Zhang, Single machine lot scheduling with non-uniform lot capacities and processing times. *J. Comb. Optim.* **43** (2022) 1359–1367.

[4] E. Coffman, M.R. Garey and D.B. Johnson, Approximation algorithms for bin packing: a survey. Approximation Algorithms for NP–Hard Problems. PWS, Boston (1996).

[5] E.G. Coffman Jr, M.R. Garey and D.S. Johnson, Bin packing with divisible item sizes. *J. Complex.* **3** (1987) 406–428.

[6] G. Dosa, Z. Tan, Z. Tuza, Y. Yan and C. Sik Lányi, Improved bounds for batch scheduling with nonidentical job sizes. *Nav. Res. Logist.* **61** (2014) 351–358.

[7] J.W. Fowler and L. Mönch, A survey of scheduling with parallel batch (p-batch) processing. *Eur. J. Oper. Res.* **298** (2022) 1–24.

[8] M.R. Garey and D.S. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness. W.H. Freeman & Co. (1979).

[9] Z. Geng and J. Liu, Single machine batch scheduling with two non-disjoint agents and splitable jobs. *J. Comb. Optim.* **40** (2020) 774–795.

[10] R.L. Graham, E.L. Lawler, J.K. Lenstra and A.H.G. Rinnooy Kan, Optimization and approximation in deterministic sequencing and scheduling: a survey, in Vol. 5 of *Annals of Discrete Mathematics.* Elsevier, B.C. Canada (1979) 287–326.

[11] Y.-T. Hou, D.-L. Yang and W.-H. Kuo, Lot scheduling on a single machine. *Inf. Process. Lett.* **114** (2014) 718–722.

[12] D.S. Johnson, *Near-optimal bin packing algorithms.* Ph.D. thesis, Massachusetts Institute of Technology (1973).

[13] K. Lee, J.Y.-T. Leung and M.L. Pinedo, Scheduling jobs with equal processing times subject to machine eligibility constraints. *J. Sched.* **14** (2011) 27–38.

[14] C.-L. Li and Q. Li, Scheduling jobs with release dates, equal processing times, and inclusive processing set restrictions. *J. Oper. Res. Soc.* **66** (2015) 516–523.

[15] B. Mor, Single-machine lot scheduling with variable lot processing times. *Eng. Optim.* **53** (2021) 321–334.

[16] B. Mor, G. Mosheiov and D. Shapira, Lot scheduling on a single machine to minimize the (weighted) number of tardy orders. *Inf. Process. Lett.* **164** (2020) 106009.

[17] B. Nurit, M. Baruch, S. Yitzhak and S. Dana, Lot scheduling involving completion time problems on identical parallel machines. *Oper. Res.* **23** (2023) 12.

[18] O. Ozturk, M.-L. Espinouse, M.D. Mascolo and A. Gouin, Makespan minimisation on parallel batch processing machines with non-identical job sizes and release dates. *Int. J. Prod. Res.* **50** (2012) 6022–6035.

[19] J.-Q. Wang and J.Y.-T. Leung, Scheduling jobs with equal-processing-time on parallel machines with non-identical capacities to minimize makespan. *Int. J. Prod. Econ.* **156** (2014) 325–331.

[20] X. Xin, M.I. Khan and S. Li, Scheduling equal-length jobs with arbitrary sizes on uniform parallel batch machines. *Open Math.* **21** (2023) 20220562.

[21] D.-L. Yang, Y.-T. Hou and W.-H. Kuo, A note on a single-machine lot scheduling problem with indivisible orders. *Comput. Oper. Res.* **79** (2017) 34–38.

[22] E. Zhang, M. Liu, F. Zheng and Y. Xu, Single machine lot scheduling to minimize the total weighted (discounted) completion time. *Inf. Process. Lett.* **142** (2019) 46–51.

Please help to maintain this journal in open access!

This journal is currently published in open access under the Subscribe to Open model (S2O). We are thankful to our subscribers and supporters for making it possible to publish this journal in open access in the current year, free of charge for authors and readers.

Check with your library that it subscribes to the journal, or consider making a personal donation to the S2O programme by contacting subscribers@edpsciences.org.

More information, including a list of supporters and financial transparency reports, is available at https://edpsciences.org/en/subscribe-to-open-s2o.