Article

A Polynomial Algorithm for Sequencing Jobs with Release and Delivery Times on Uniform Machines

Nodari Vakhania [1,†,‡] and Frank Werner [2,†,‡,*]

[1] Centro de Investigacion en Ciences, UAEMor; nodari@uaem.mx
[2] Faculty of Mathemaics, Otto-von-Guericke University Magdeburg; frank.werner@ovgu.de
* Correspondence: frank.werner@ovgu.de; Tel.: +49-391-675-2025 (F.W.)

Abstract: The problem of sequencing n equal-length non-simultaneously released jobs with delivery times on m uniform machines to minimize the maximum job completion time is considered. To the best of our knowledge, the complexity status of this classical scheduling problem remained open up to the date. We establish its complexity status positively by showing that it can be solved in polynomial time. We adopt for the uniform machine environment the general algorithmic framework of the analysis of behavior alternatives developed earlier for the identical machine environment. The proposed algorithm has the time complexity $O(\gamma m^2 n \log n)$, where γ can be any of the magnitudes n or $q_{\max}$, the maximum job delivery time. In fact, n can be replaced by a smaller magnitude $\kappa < n$, which is the number of special types of jobs (it becomes known only upon the termination of the algorithm).

Keywords: scheduling; uniform machines; release time; delivery time; time complexity; algorithm

1. Introduction

In this paper, we consider a basic optimization problem of scheduling jobs with release and delivery times on uniform machines with the objective to minimize the makespan. More precisely, n jobs from the set $J = \{1, 2, ..., n\}$ are to be processed by m parallel *uniform machines* (or *processors*) from the set $M = \{1, 2, ..., m\}$. Job $j \in J$ is available from its *release time* r_j, it needs a continuous (integer) *processing time* p, which is the time that it needs on a *slowest* machine. We assume that the machines in the set M are ordered by their speeds, the fastest machines first, i.e., $s_1 \geq s_2 \geq \cdots \geq s_m$ are the corresponding machine speeds, s_i being the speed of machine i. Without loss of generality, we assume that $s_m = 1$, and the processing time of job j on machine i is an integer p/s_i. Job j has one more parameter, the *delivery time* q_j, an integer number which represents the amount of additional time units which are necessary for the *full* completion of job j *after* it completes on the machine. So notice that the delivery of job j consumes no machine time (the delivery is accomplished by an independent agent).

Now we define a *feasible schedule* S as a function that assigns to each job j a starting time t_j^S and a machine i from set M, such that $t_j^S \geq r_j$ and $t_j^S \geq t_k^S + p/s_i$, for any job k scheduled earlier on the same machine (the first inequality ensures that a job cannot be started before its release time, and the second reflects the restriction that each machine can handle only one job at any time). The *completion time* of job j in schedule S is $c_j^S = t_j^S + p/s_i$ and the *full completion time* of job j in schedule S is $C_j^S = c_j^S + q_j$ (the full completion time of job j takes into account the delivery time of that job, whereas the completion time of job j does not depend on its delivery time). The objective is to find an *optimal schedule*, i.e., a feasible schedule S that minimizes the maximal full job completion time

$$C_{max}(S) = \max_j C_j$$

commonly referred to as the *makespan*.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 8 January 2021 doi:10.20944/preprints202101.0142.v1

The studied multiprocessor optimization problem, described below, is commonly abbreviated as $Q|p_j = p, r_j, q_j|C_{\max}$ (its version with identical parallel machines is abbreviated as $P|p_j = p, r_j, q_j|C_{\max}$, the first field specifies the machine environment, the second one the job parameters, and the third one the objective function).

It is well-known that there is an equivalent (perhaps more traditional) formulation of the above described problem, in which instead of the delivery time q_j, every job j has its due-date d_j. The *lateness* of job j in the schedule S is $L_j^S = c_j^S - d_j$. Then the objective becomes to minimize the maximum job lateness $L_{\max}$, i.e., find a feasible schedule S_{OPT} in which the maximum job lateness is not more than in any other feasible schedule, i.e., S_{OPT} is an *optimal* schedule. The equivalence is easily established by associating with each job delivery time a corresponding due-date, and vise-versa, see e.g., Bratley et al. [1]). The version of the problem with due-dates with identical and uniform machine environments are commonly abbreviated as $P|r_j, d_j|L_{\max}$ and $Q|r_j, d_j|L_{\max}$, respectively.

For the problem considered, each machine from a group of parallel uniform machines is characterized by its own speed, independent from a particular job that can be assigned to it, unlike a machine from a group of unrelated machines whose speed is job-dependent. Because of the uniform speed characteristic, scheduling problems with uniform machines are essentially easier than scheduling problems with unrelated machines, whereas scheduling problems with identical machines are easier than those with uniform ones. Here we consider the setting with equal-length jobs the complexity status of which remained open. At the same time, the version of this problem with identical machines is known to be polynomially solvable. In this paper, we settle the complexity status of the problem $Q|p_j = p, r_j, q_j|C_{\max}$ by presenting a polynomial algorithm.

The algorithm that we describe here essentially relies on the algorithmic framework proposed earlier for the version of the problem with identical machines in [2]. The framework is based on the analysis of some nice structural properties of specially created schedules which are analyzed in terms of the so-called behavior alternatives. The framework resulted in an $O(q_{\max} mn \log n + O(m\kappa n))$ time algorithm, where $q_{\max}$ is the maximum job delivery time and $\kappa < n$ is a parameter which is known only after the termination of the algorithm. Each schedule is easily created by a well-known greedy algorithm commonly referred to as Largest Delivery Time heuristic (LDT-heuristic for short): iteratively, among all released jobs, it schedules one with the largest delivery time. The algorithm from [2] carries out the enumeration of LDT-schedules (ones created by the LDT-heuristic) - it is known that there is an optimal LDT-schedule. Based on the established properties, the set of LDT-schedules is reduced to a subset of polynomial size which yields a polynomial time overall performance. Although the LDT-heuristic applied to a problem instance with uniform machines does not provide the desirable properties, it can be modified to a similar method that takes into account the uniform speed characteristic. While scheduling identical machines, the minimum completion time of each next selected job is always achieved on the machine, which is the next to the machine to which the previously selected job is assigned. With uniform machines, this is not necessarily the case, for example, the next machine can be much slower than the current one. Hence, the time moment at which the job will complete on each of the machines needs to be additionally determined and then this job can be assigned to a machine on which the above minimum is reached. Here, this modified version is referred to as the LDTC-heuristic and (a schedule created by this heuristic as an LDTC-schedule). Instead of enumerating the LDT-schedules (as in [2]), the algorithm proposed here enumerates LDTC-schedules. Some properties for the identical machine environment which do not immediately hold for uniform machines are reformulated in terms of uniform machines, which allows to maintain the basic framework from [2] which, as suggested earlier, turned out to be sufficiently flexible.

Similarly as there exists an optimal LDT-schedule for the identical machine environment, there exists an optimal LDTC-schedule for the uniform machine environment. The complete enumeration of the LDTC-schedules is avoided by the generalization of nice

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 8 January 2021 doi:10.20944/preprints202101.0142.v1

properties of LDT-schedules to LDTC-schedules for uniform machines. These properties are obtained via the analysis of the behavior alternatives from [2] that are generalized for uniform machines. The proposed algorithm has the time complexity $O(\gamma m^2 n \log n)$, where γ can be any of the magnitudes n or $q_{\max}$, the maximum job delivery time. In fact, n can be replaced by a smaller magnitude κ, the number of special types of jobs; this is the same parameter κ as for the algorithm from [2] which becomes known only when the algorithm halts. The running time of the proposed algorithm is worse than that of the one from [2], basically, because of the cost of the LDTC-heuristic which is repeatedly used during the solution process.

The remainder of this paper is as follows. In Section 2, we give a brief literature review. Section 3 presents some necessary preliminaries. Then the basic algorithmic framework is given in Section 4. Section 5 discusses the performance analysis of the developed algorithm. Finally, Section 6 gives some concluding remarks.

2. Literature Review

If the job processing times are arbitrary, then the problem is known to be strongly NP-hard, even if there is only a single machine $1|r_j, d_j|L_{\max}$ [3], see e.g., McMahon & Florian [4] for an efficient enumerative algorithm for the problem $1|r_j, d_j|L_{\max}$. For this problem with so-called embedded jobs, where the data fulfill special conditions, Vakhania presented fast-polynomial algorithms in [5]. For the single machine case, Baptiste gave an $O(n^7)$ algorithm for the problem $1|r_j, p_j = p|\sum T_j$ [6] and also an algorithm of the same complexity for the problem $1|r_j, p_j = p|\sum w_j U_j$ [7] of minimizing the weighted number of late jobs. Chrobak et al. [8] have derived an algorithm of improved complexity $O(n^5)$ for the case of unit weights, i.e., for the problem $1|r_j, p_j = p|\sum U_j$. Later, Vakhania [9] gave an $O(n^2 \log n)$ algorithm for this problem. Note that for the problem $1|r_j, p_j, pmtn|\sum U_j$ with arbitrary processing times and allowed preemptions, Vakhania [10] derived an $O(n^3 \log n)$ algorithm.

One may consider a slight relaxation of problems $1|r_j, d_j|L_{\max}$, $P|r_j, d_j|L_{\max}$ and $Q|r_j, d_j|L_{\max}$ in which one looks for a schedule in which no job completes after its due-date. Such a feasibility setting with a single machine was considered by Garey et al. [11]. They have proposed an $O(n^2 \log n)$ algorithm which has further been improved to an $O(n \log n)$ one by using a very sophisticated data structure. The authors have relied on the concept of a so-called forbidden region, an artificial gap in a schedule in which it is forbidden to start any job. Later Simons and Warmuth [12] have constructed an $O(n^2 m)$ time algorithm for the feasibility setting with the identical machine environment also using the concept of forbidden regions. (It can be mentioned that the minimization version of the problem can be solved by applying an algorithm for the feasibility problem by repeatedly increasing the due-dates of all jobs until a feasible schedule with the modified due dates is found. Using binary search makes such a reduction procedure more efficient and reduces the reduction cost to $O(\log(np/m))$.)

Dessousky et al. [13] considered scheduling problems on uniform machines with simultaneously released jobs (i.e., with $r_j = 0$ for every job j) and with different objective criteria. They proposed fast polynomial-time algorithms for these problems, in particular, for the criterion $L_{\max}$. In fact, the LDTC-heuristic is an adaptation of an optimal solution method that the authors in [13] constructed for the criterion $L_{\max}$.

For a uniform machine environment with allowed preemptions ($pmtn$), the problem $Q|r_j, pmtn|C_{max}$ is polynomially solvable even for arbitrary processing times [14], while a polynomial algorithm for the problem $Q|r_j, p_j = p, pmtn|\sum C_j$ with minimizing total weighted completion time in the case of equal processing times has been given in [15]. The case of unrelated machines is very hard. A polynomial algorithm exists for the problem $R|r_j, pmtn|L_{max}$ with allowed preemptions and minimizing maximum lateness, even for the case of arbitrary processing times [16]. If preemptions are forbidden, Vakhania et al. [17] gave a polynomial algorithm for the case of minimizing the makespan when only two processing times p and $2p$ are possible (i.e., for the problem $R|p \in$

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 8 January 2021 doi:10.20944/preprints202101.0142.v1

$\{p, 2p\}|C_{max}$. Note that the case of two arbitrary processing times p and q is known to be NP-hard [18]. For the special case of identical parallel machines, there exist several works for the same setting as considered in this paper but for more complicated objective functions regarding the complexity status. In particular, the problems $P|r_j, p_j = p|\sum w_j C_j$ of minimizing the weighted sum of completion times [19] and $P|r_j, p_j = p|\sum T_j$ of minimizing total tardiness [20] can be polynomially solved by a reduction to a linear programming problem. In [21], Vakhania presented an $O(n^3 \log n)$ algorithm for the problem $P|r_j, p_j = p|\sum U_j$ of minimizing the number of late jobs. His blesscmore ('branch less, cut more') algorithm uses a solution tree, where the branching and cutting criteria are based on the analysis of behavior alternatives. Moreover, the problem $P|r_j, p_j = p|\sum f_j(C_j)$ can also be polynomially solved for the case that f_j is an arbitrary non-decreasing function such that the difference $f_i - f_j$ is monotonic for any indices i and j [22]. The authors also applied a linear programming approach. It can also be mentioned that a detailed survey on parallel machine scheduling problems with equal processing times has been given in [23].

3. Preliminaries

This section contains some useful properties, necessary terminology and concepts, some of which were introduced in [2] for identical machines.

LDTC-heuristic. We first describe the LDTC-heuristic, an adaptation of the LDT-heuristic for uniform machines. As earlier briefly noted, unlike an LDT-schedule, an LDTC-schedule is not defined by a mere permutation of the given n jobs since the machine to which the next selected job is assigned depends on the machine speed. Starting from the minimal job release time, the current scheduling time is iteratively set as the minimum release time among all yet unscheduled jobs. Iteratively, among all jobs released by the current scheduling time, the LDTC-heuristic determines one with the largest delivery time (a most *urgent* one) and schedules it on the machine on which the earliest possible completion time of this job is attained (ties can be broken by selecting the machine with the minimum index). Note that in an LDTC-schedule S, a machine will contain an idle-time interval (a *gap*) if and only if there is no unscheduled job released by the current scheduling time. The running time of the modified heuristic is the same as that of LDT-heuristic with an additional factor of m due to the machine selection at each iteration (which is not required for the uniform machine environment), which results in the time complexity $O(nm \log n)$.

Let σ be the LDTC-schedule obtained by the LDTC-heuristic for the initially given problem instance. As we will see in the following subsection, we may generate alternative LDTC-schedules by iteratively modifying the originally given problem instance. The next property of an LDTC-schedule easily follows from the definition of the LDTC-heuristic (and the equality of the job processing times).

Property 1. *If in an LDTC-schedule S, job j is scheduled after job i, i.e., the ordinal number of job j in S is larger than that of job i, then $c_j^S \geq c_i^S$.*

Next, we give another easily seen important property of an LDTC-schedule S on which the proposed method essentially relies. Let A be a set of, say k jobs, all of which being released by time moment t, and let π be a permutation of k jobs, all of which being also released by time t (recall that all these jobs have equal length). Let, further, $S(A)$ be a partial LDTC-schedule constructed for the jobs in the set A, and let $S(\pi)$ be a list schedule constructed for the permutation π.

Property 2. *The completion time of every machine in both schedules $S(A)$ and $S(\pi)$ is the same. Moreover, the ith scheduled job in the schedule $S(A)$ starts and completes at the same time as the ith scheduled job in the schedule $S(\pi)$.*

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 8 January 2021 doi:10.20944/preprints202101.0142.v1

The above property also holds for a group of identical machines and is helpful for the generalization of the earlier results for identical machines from [2] to uniform machines. Roughly, ignoring the job release times, the property states that two list schedules constructed for two different permutations with the same number of jobs have the same structure. Although the starting and completion times of the jobs scheduled in the same position are the same in both schedules, the full completion times will not necessarily be the same (this obviously depends on the delivery times of these jobs).

A block. An independent part in a schedule S is commonly referred to as a *block* in the scheduling literature. We define it as a largest fragment of schedule S such that for each two successively scheduled jobs i and j, job j starts no later than job i finishes (jobs i and j can be scheduled on the same or different machines). It follows that there is a single block that starts schedule S and finishes this schedule. If these blocks coincide, then there is a single block in the schedule S, otherwise, each next block is "separated" by the previous one with gaps on each of the machines. Here a zero length gap between jobs i and j will be distinguished in case job j is scheduled at time $r_j = t_i(S)$ on the same machine as job i (it immediately succeeds job i on that machine). A block B (with at least two elements) possesses the following property that will be used later. Suppose the ιth scheduled job j is removed from that block and the LDTC-heuristic is applied to the remaining jobs of the block. Then in the resultant (partial) schedule, the processing interval of the ιth scheduled job overlaps with the earlier processing interval of job j in block B.

Some additional definitions are required to specify how the proposed algorithm creates and evaluates the feasible schedules. At any stage of the execution of the algorithm, independently whether a new LDTC-schedule will be generated or not, depends on specific properties of the LDTC-schedules already generated by that stage. The definitions below are helpful for the determination of these properties.

An overflow job. In an LDTC-schedule S, let o be a job realizing the maximum full completion time of as job, i.e.,

$$\mathcal{C}_o(S) = C_{max}(S), \tag{1}$$

and let $B(S)$ be the *critical block* in the schedule S, i.e., the block containing the earliest scheduled job o satisfying equation (1). The *overflow job* $o(S)$ in the schedule S is the latest completed job on the machine in the block $B(S)$ satisfying condition (1), i.e., one with the maximum $c_o(S)$ (further ties are broken by selecting the job scheduled on the machine with the largest index).

A kernel. Next, we define an important component in an LDTC-schedule S defined as its segment containing the overflow job $o = o(S)$ such that the jobs scheduled before job o in the block $B(S)$ have a delivery time not smaller than q_o (we will write o instead $o(S)$ when this causes no confusion). This segment of the schedule S is called its *kernel* and is denoted by $K(S)$.

Intuitively, on the one hand, the kernel $K(S)$ is a critical part in a schedule S, and on the other hand, it is relatively easy to arrange the kernel jobs optimally. In fact, we will explore different LDTC-schedules identifying the kernel in each of these schedules. We will also relate this kernel to the kernels of the earlier generated LDTC-schedules. We need to introduce a few more definitions.

An emerging job. Suppose that a job j of kernel $K(S)$ is *pushed* by a non-kernel job i scheduled earlier on the same machine, that is, the LDTC-heuristic would schedule job j earlier if job i was forced to be scheduled after job j. Assume further that job e scheduled before job o in the block $B(S)$ is such that $q_e < q_o$. Then job e is called a *regular emerging job* in the schedule S, and the latest scheduled (regular) emerging job (the one closest to job o) is called the *delaying* emerging job.

The following optimality condition can be established already in the initial LDTC-schedule σ.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 8 January 2021 doi:10.20944/preprints202101.0142.v1

Lemma 1. *If the initial LDTC-schedule σ contains a kernel K such that no job of that kernel is pushed by a non-kernel (emerging) job, then this schedule is optimal.*

Proof. By the condition, if a job j from kernel K does not start at its release time, then it is pushed by another job i from this kernel with $q_i \geq q_j$ (the latter inequality follows from the definition of a kernel). The lemma is now easily established by an interchange argument and by the fact that the makespan of the schedule σ is realized by a job from the kernel K. □

3.1. Constructing alternative LDTC-schedules

Due to Lemma 1, from here on, it is assumed that the condition in this lemma is not satisfied, i.e., there exists an emerging job e in the schedule S (note that $e \in B(S)$ as otherwise job e may not push a job of the kernel $K(S)$). Since job e is pushing a job of kernel $K(S)$, the removal of this job may potentially decrease the starting and hence the full completion time of the overflow job $o(S)$. At the same time, note again that by the definition of a block, the omission of a job not from the block $B(S)$ may not affect the starting time of any job from the block $B(S)$. That is why we restrict here our attention to the jobs of the block $B(S)$. (Later we will introduce an alternative notion of a passive emerging job and then will use "emerging job" for either a regular or passive emerging job; until then we use "emerging job" for a "regular emerging job".)

Clearly, no emerging job can actually be removed as the resultant schedule would be infeasible. Instead, to restart the jobs in the kernel $K(S)$ earlier, an emerging job e is *applied* to this kernel, i.e., it is forced to be rescheduled after all jobs of the kernel $K(S)$ whereas any job, scheduled after the kernel $K(S)$ in the schedule S is maintained to be scheduled after that kernel. The application of job e is accomplished in two steps: first, the original release time of job e and that of the jobs, scheduled after kernel $K(S)$ in schedule S, is increased to the release time of any job in the kernel $K(S)$. Then the resultant schedule denoted by S_e (the so-called *complementary schedule* or a *C-schedule*) is obtained by the LDTC-heuristic which is merely applied to the modified problem instance). (Such a schedule generation technique for a single-machine setting was suggested by McMahon and Florian [4].)

By Lemma 1, the kernel $K(S)$, a fragment of the LDTC-schedule S considered as an independent LDTC-schedule is optimal if it possesses no emerging job. Otherwise, the jobs of the kernel $K(S)$ are pushed by the corresponding emerging jobs. Some of these emerging jobs can be scheduled after the kernel $K(S)$ in an optimal complete schedule S_{OPT}. Such a rescheduling is achieved by the creation of the corresponding C-schedules (as we will see in Lemma 2, it will suffice to consider only C-schedules, i.e., S_{OPT} is a C-schedule).

The application of an emerging job has two "opposite" effects. On the positive side, since the number of jobs scheduled before the kernel $K(S)$ in the schedule S_e is one less than that in the schedule S, the overflow job $o(S)$ in the schedule S_e will be completed earlier than it was completed in the schedule S; likewise, the completion time of the latest scheduled job of the kernel $K(S)$ in the schedule S_e will be less than the completion time of job $o(S)$ in the schedule S. Hence, the application of an emerging job gives a potential to improve schedule S. On the negative side, it creates a new gap within the former execution interval of job e or at a later time moment before kernel $K(S)$ (see Lemma 1 in [2] for a proof for the case of identical machines, the uniform machine case can be proved similarly). Such a gap may enforce a right-shift (delay) of the jobs included after job e in the schedule S_e. So roughly, the C-schedule S_e favors the kernel $K(S)$ but creates a potential conflict for later scheduled jobs.

4. The basic algorithmic framework

In this section, we give the basic skeleton of the proposed algorithm and prove its correctness. The schedule S_{OPT} is characterized by a proper processing order of the

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 8 January 2021 doi:10.20944/preprints202101.0142.v1

emerging jobs scheduled in between the kernels. Starting with the initial LDTC-schedule σ, an emerging job in the current LDTC-schedule is applied and a new C-schedule is created; in this schedule the kernel is again determined. The same operation is iteratively repeated as long as the established optimality conditions are not satisfied. As we will show later, it will suffice to enumerate all C-schedules to find an optimal solution to the problem.

We associate a complete feasible C-schedule with each node in a *solution tree* T, the initial LDTC-schedule being associated with the root. Aiming to avoid a brutal enumeration of all C-schedules, we carry out a deeper study of the structure of the problem and some additional useful properties of LDTC-schedules. In fact, our solution tree T consists of a single chain of C-schedules. We will refer to a node of the tree as a *stage* (since each node represents a particular stage in the algorithm with the corresponding LDTC-schedule). We let $T_h = (S^0, ..., S^h)$ be the sequence of C-schedules generated by stage h. Thus, S^0 is the initial LDTC-schedule, and the schedule S^h of stage $h > 0$, the immediate successor of schedule S^{h-1}, is obtained by one of the extension rules as described below.

In the schedule S^h, the overflow job $o(S^h)$, the delaying emerging job l and the kernel $K(S^h)$ are determined. Using the *normal extension rule*, we let $S^{h+1} := S^h_l$, where l is the delaying emerging job in the schedule S^h. Alternatively, the schedule S^{h+1} is constructed from the schedule S^h by the *emergency extension rule* as described in the following subsection.

4.1. Types of emerging jobs and the extended behavior alternatives

A marched emerging job. An emerging job may be in different possible states. It is useful to distinguish these states and treat them accordingly. Suppose that e is an emerging job in the schedule S^g and it is applied by stage h, $h > g$ (in a predecessor-schedule of schedule S^h). Then job e is called *marched* in the schedule S^h if $e \in B(S^h)$. Intuitively, the existence of a marched job in the schedule S^h indicates an "interference" of the kernel $K(S^h)$ with an earlier arranged part of the schedule preceding that kernel.

A stuck emerging job. Suppose that job e is marched in the schedule S^h and $E(S^h) = \emptyset$, where $B(S^h)$ is a non-primary block. Then job e is called *stuck* in the schedule S^h if either it is scheduled before job $o(S^h)$ or $e = o(S^h)$ (observe that any job stuck in the schedule S^h belongs to the kernel $K(S^h)$).

Block evolution in the solution tree T. Although $E(S^h) = \emptyset$, since $B(S)$ is a non-primary block, a "potential" regular emerging job might be "hidden" in some block preceding block $B(S^h)$, in the schedule S^h. In general, a block in the schedule S^h can be a part of a larger block from the schedule S^g for some $g < h$. Recall that the application of an emerging job e in a C-schedule S yields the raise of a new gap in the C-schedule S_e. As it can be straightforwardly seen, this may cause a separation or the *splitting* of the critical block $B(S)$ into two (or even more) new blocks. Likewise, since job e may push the following jobs in the schedule S_e, because of the forced right-shift of these jobs, two or more blocks may *merge* forming a bigger block consisting of the jobs from the former blocks.

We will refer to blocks from two different C-schedules as *congruent* if both of them are formed by the same set of jobs. A block in a C-schedule, which is congruent to a block from the initial LDTC-schedule, will be referred to as a *primary* block. Observe that a non-primary block may arise because of either block splitting or/and block merging and that all blocks in the initial LDTC-schedule are primary.

If the block B is arisen as a result of an application of an emerging job e, then this block is said to be a non-primary block *of* job e, and the latter job is said to be the *splitting* job of B. Note that, since the application of an emerging job not necessarily causes a block split, a non-primary block of job e may contain some other already applied emerging jobs (recall that before each application of an emerging job, the current release time of that

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 8 January 2021 doi:10.20944/preprints202101.0142.v1

job is increased accordingly; e.g., if job e is applied ι times, its release time is modified ι times).

We will refer to the blocks arisen after the splitting of one particular block as the *direct descendants* of this block (the latter block is the *direct predecessor* of the former ones). If $B \in S$ is the direct predecessor of the blocks $B_1, ..., B_k$, then $B \subseteq B_1 \cup ... \cup B_k$, but not vice versa (because of a possible block merging).

A *descendant* of a block is its direct descendant recurrently, any descendant of a descendant of a block is also a descendant of the latter block. If a block is a descendant of another block, then the latter is a *predecessor* of the former block. Two or more blocks are said to be *relative* if they have at least one common predecessor block.

A passive emerging job. A *passive emerging job* e in the schedule S^h is a ("hidden") regular emerging job from the schedule S^g, i.e., job e belongs to a block from schedule S^g, relative to block $B(S^h)$ (preceding this block), such that $q_e < q_{o(S^h)}$.

Extended behavior alternatives. Now we define two extended behavior alternatives which, together with the five basic behavior alternatives were introduced earlier in [2] (Section 2.3). Suppose that there exists no regular emerging job in the C-schedule S (i.e., the block $B(S)$ starts with the kernel $K(S)$), and there is no stuck job in this schedule. Then an *exhaustive instance of alternative (a)* (abbreviated EIA(a)) in the schedule S is said to occur. If now there exists a stuck job in the schedule S, then an *extended instance of alternative (b)* with this job in the schedule S (abbreviated EIA(b)) is said to occur (there may exist more than one job stuck in the schedule S^h).

The first above behavior alternative immediately yields an optimal solution, and the second one indicates that some rearrangement of the already applied emerging jobs might be required. The first and the second behavior alternatives, respectively, are treated in the following lemma and in the next subsection, respectively.

Lemma 2. *A C-schedule S^h is optimal if an EIA(a) in it occurs.*

Proof. By the condition, there exists no stuck job in the schedule S^h. This implies that none of the jobs of the kernel $K(S^h)$ can be scheduled at some earlier time moment without causing a forced delay of a more urgent job from this kernel, and the lemma can easily be proved by an interchange argument. □

4.2. Emergency extension rule

Throughout this subsection, we assume that there arises an EIA(b) in the schedule S^h and that there exists a passive emerging job in this schedule. Let l be the latest scheduled passive emerging job in S^h. By the definition of job l, there is a schedule S^g, a predecessor of schedule S^h in the solution tree T, and a job $e \in B(S^h)$, stuck in the schedule S^h, such that $e \in B(S^g)$ and $l \in B(S^g)$. Let e be the latest applied job stuck in the schedule S^h. Although jobs l and e belong to different blocks in the schedule S^h, the corresponding blocks can be merged by reverting the application(s) of job e. This can clearly be accomplished by restoring the corresponding earlier release time of job e (recall that the release time of an emerging job is increased each time it is applied). Once these blocks are merged, job l becomes a regular emerging job and hence, it can be applied.

Let r be the release time of an emerging job j before it is applied to a kernel K. Then job j is said to be *revised* (for K) if it is placed back before K, i.e., its release time is reassigned the value r and the LT-heuristic is applied.

In more detail, let B be a block relative to $B(S^h)$ in the schedule S^h containing job l. Not only B and $B(S^h)$ are different blocks, there might be a chain $B_1, ..., B_{k-1}$ of succeeding (relevant) blocks between the blocks B and $B(S^h)$ in the schedule S^h. Let $B_0 = B$ and $B_k = B(S^h)$. First, the blocks B_{k-1} and B_k are merged by reverting the application of the corresponding emerging job. Then the resulting block is similarly merged with block B_{k-2}, and so on. In general, to merge the block B' with its successive (relative) block B'', the corresponding release time of one of the currently applied jobs

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 8 January 2021 doi:10.20944/preprints202101.0142.v1

scheduled in block B'' (scheduled between the jobs of B' and B'' before its application) is restored, i.e., this job is *revised*.

Note that by the definition, the revision of the splitting job of blocks B' and B'' will merge these two blocks. Similarly, the revision of any other applied emerging job, scheduled in B'', will have the same effect. Among all such jobs with the largest delivery time, the latest scheduled one in block B'' will be referred to as the *active splitting* job for the blocks B' and B''.

The blocks B_{k-1} and B_k are merged by the revision of the active splitting job of these two blocks which is scheduled in block B_k. Similarly, the active splitting job of block B_{k-2} is revised to merge block B_{k-2} with the earlier obtained block, and so on, this process continues until all blocks from the chain $B_0, B_1, ..., B_{k-1}, B_k$ are merged.

We denote the resultant merged block by $\mathcal{B}(l)$ (this block, ending with the jobs from block $B(S^h)$ can, in general, be non-primary), and we will refer to the above described procedure as the *chain of revisions* for the passive emerging job l. Observe that, although the outcome of this procedure somewhat resembles the traditional backtracking, it is still different as it keeps untouched the "intermediate" applications that could have been earlier carried out between the reverted applications.

Let $Rev_l(S^h)$ be the C-schedule, obtained from schedule S^h by the chain of revisions for job l. Observe that job l changes its status from a passive to a regular emerging job in this schedule, and hence the emergency extension rule applies job l in the schedule $Rev_l(S^h)$ to kernel $K(Rev_l(S^h))$, setting $S^{h+1} := (Rev_l(S^h))_l$ (we will also use the shorter notation $S^h_{+l} = (Rev_l(S^h))_l$ for the resultant schedule).

4.3. The description of the algorithm and its correctness

We give the following Algorithm 1 and prove its correctness.

Algorithm 1:

Step 1: Set $h = 0$ and $S^h := \sigma$.

Step 2: If the condition of Lemma 1 holds, then return the schedule σ and stop.

Step 3: Set $h = h + 1$.

Step 4: { *stopping rule* } If in the schedule S^h, either (i) there exists neither a regular nor a passive emerging job or there occurs an EIA(a), then return a schedule from the tree T with the minimum makespan and stop;

Step 5: { *normal extension rule* }: if in the schedule S^h, there occurs no EIA(b), then $S^{h+1} := S^h_l$, where l is the regular delaying emerging job else
{ *emergency extension rule* } if in the schedule S^h there occurs an EIA(b), then $S^{h+1} := S^h_{+l}$, where l is the passive delaying emerging job.

Step 6: goto Step 3.

Now we can prove the following theorem.

Theorem 1. *For some stage h, the C-schedule S^h is an optimal schedule S_{OPT}.*

Proof. First, we show that an optimal schedule is a C-schedule. Suppose that the condition in Lemma 1 is not satisfied for the schedule $S^0 = \sigma$, and that the C-schedule S^h, $h > 0$, is not optimal. Then, due to Property 2, in any feasible schedule S with a better makespan, the number of jobs scheduled before the kernel $K(S^h)$ in block $B(S^h)$ must be one less than in the latter schedule (otherwise, a job from the kernel $K(S^h)$ will not have a smaller full completion time in the schedule S' than job $o(S^h)$ in the schedule S^h). Hence, some job $e \in B(S^h)$ included before the kernel $K(S^h)$ in the schedule S^h must be scheduled after that kernel in the schedule S. Moreover, e is to be either a regular or passive emerging job. Indeed, if e is not an emerging job, then $q_e \geq q_o$,

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 8 January 2021 doi:10.20944/preprints202101.0142.v1

$o = o(S^h)$ or/and $e \notin B(S^h)$. In the latter case, by the definition of a block, no feasible rearrangement involving jobs of a non-critical block may decrease the full completion time of the overflow job o. Otherwise, $t_e^{(S^h)_e} \geq t_o^{S^h}$ and due to inequality $q_e \geq q_o$ and Proposition 2, we obtain $|(S^h)_e| \geq |S^h|$, which implies that the makespan of any feasible schedule in which job e is scheduled after the kernel $K(S^h)$ cannot be less than that of schedule S^h. Then the correctness of the stopping rule clearly follows from Lemma 2.

It remains to show that the search in the space of the C-schedules is correctly organized. There are two extension rules. The normal extension rule is used at stage h if there exists a regular emerging job in the schedule S^h. In this case, the delaying emerging job l is applied, i.e., $S^{h+1} := S_l^h$. Consider an alternative feasible schedule S_e^h, where e is another emerging job (above we have shown that only emerging jobs need to be considered). It is easy to see that the left-shift of the kernel jobs in the schedule S_e^h cannot be more than that in the schedule S_l^h, and the forced right-shift for the jobs scheduled after job e in the schedule S_e^h cannot be less than that of the jobs scheduled after job l in the schedule S_l^h (recall that $p_e = p_l$). Hence, the schedule S_e^h is dominated by the schedule S_l^h unless job l gets stuck at a later stage $h' > h$. In the latter case, the emergency extension rule revises first job l. In the resultant C-schedule, the passive delaying job converts into a regular delaying emerging job. Then the emergency extension rule applies this (converted) regular regular delaying emerging job. We complete the proof by repeatedly applying the above reasoning for the normal extension rule. □

5. Performance analysis

It is not difficult to see that the direct application of Algorithm 1 of the previous section may yield the generation of some redundant C-schedules: the jobs from the same kernel K including the overflow job o may be forced to be right-shifted after they are already "arranged" (i.e., the corresponding emerging job(s) are already applied to that kernel) due to the arrangement accomplished for the kernel K' preceding kernel K. As a result (because of the application of an emerging job for the kernel K'), one or more redundant C-schedules in which a job from the kernel K repeatedly becomes an overflow job might be created. Such an unnecessary rearrangement of the portion of a C-schedule between the kernels K' and K is avoided by restricting the number of jobs that are allowed to be scheduled in that portion. This number becomes well-defined after the first disturbance of this portion caused by the application of an emerging job for the kernel K'. This issue was studied in detail for the case of identical machines in [2] (see Section 4.1). It can be readily verified that the basic estimations for the case of identical machines similarly hold for uniform machines. In particular, the number of the enumerated C-schedules remains the same for uniform machines. A complete time complexity analysis requires a number of additional concepts and definitions from [2] and would basically repeat the arguments for identical machines.

Recall that we use a different schedule generation mechanism for identical and uniform machines: the LDT-heuristic applied for the schedule generation in the identical machine environment is replaced by the LDTC-heuristic for the uniform machine environment. LDT-schedules possess a number of nice properties used in the algorithm from [2]. LDTC-schedules also possess such necessary useful properties (Properties 1 and 2) that allowed us to use the basic framework from [2]. While generating an LDT-schedule for identical machines, every next job is scheduled on the next available machine (the next to the last machine m being machine 1) and the starting and completion time of each next scheduled job is not smaller than that of all previously scheduled ones. In some sense, the generalization of these properties are Properties 1 and 2, which still assure that the structural pattern of the generated schedules is kept, and it does not depend on which particular jobs are being scheduled in a particular time interval (note that this would not be the case for an unrelated machine environment). This allowed us to use the basic framework from [2] for the uniform machine environment (for instance, intuitively, while restricting the number of the scheduled jobs between two successive kernels, no matter which particular jobs are being scheduled between these kernels).

Another "redundancy issue" occurs when a series of emerging jobs are successively applied to the same kernel K without reaching the desired result, i.e., the applied emerging jobs become new overflow jobs in the corresponding C-schedules (see Section 4.2 in [2]). In Lemma 7 from the latter reference, it is shown that this yields an additional factor of p in the running time of the algorithm. This result also holds for the uniform machine environment. The magnitude p remains valid for the uniform machine environment as the difference between the completion times of two successively scheduled jobs on the same machine cannot be more than p (recall that p is the processing time of any job on the slowest machine m). The desired result follows since the delivery time of each emerging job next applied to kernel K is strictly less than that of the previously applied one (see the proof of Lemma 7 in [2]).

The above results yield the same bound $O(\gamma m)$ on the number of the enumerated C-schedules as in [2]), where γ can be either n or $q_{\max}$ (see Lemma 8 from Section 4.3 and Theorem 2 from Section 6.1, in [2]). In fact, γ is the total number of the applied emerging jobs, a magnitude, that can be essentially smaller than n. This yields the overall cost $O(\gamma m^2 n \log n)$ due to the cost $O(mn \log n)$ of the LDTC-heuristic (instead of $O(n \log n)$ for the LDT-heuristic). A further refinement of the overall time complexity accomplished in [2] is not possible for the uniform machine environment. In the algorithm from [2], while generating every next C-schedule, instead of applying the LDT-heuristic to the whole set of jobs, it is only applied to the jobs from a small part of the current C-schedule, the so-called critical segment (a part containing its kernel), and the remaining jobs are scheduled in linear time just by right-shifting the jobs following the critical segment by the required amount of time units (conserving their current processing order). This is not possible for uniform machines as such an obtained schedule will not necessarily remain an LDTC-schedule, i.e., a linear time rescheduling will not provide the desired structure.

6. Concluding remarks

The complexity status of a long-standing open multiprocessor scheduling problem was established. Constructively, a polynomial-time solution was proposed. The employed here approach of the analysis of behavior alternatives from [2] turned out to be sufficiently general so that it was possible to extend it also to a uniform machine environment. At the same time, as we have mentioned earlier, this approach cannot be extended to the unrelated machine environment, mainly because the structural pattern of the generated schedules will depend on which particular jobs are scheduled in a particular time interval on each machine from a group of unrelated machines. However, the approach might be extensible for shop scheduling problems. It is a challenging question whether it can also be extended to the case when there are two allowable job processing times (this turned out to be possible for the single-machine environment see [24]) and for a much more general setting with mutually divisible job processing times (this turned out to be also possible for the single-machine environment, a maximal polynomially solvable special case of problem $1|r_j, d_j|L_{\max}$ dealt with recently in [25]) for identical and uniform machine environments.

Author Contributions: conceptualization, N.V.; investigation, N.V. and F.W.; writing–original draft preparation, N.V. and F.W.

Funding: The first author was supported by DAAD grant for Research Stays for University Academics and Scientists (2020) 57507438.

Conflicts of Interest: "The authors declare no conflict of interest."

References

1. Bratley, P.; Florian, M.; Robillard, P. On sequencing with earliest start times and due–dates with application to computing bounds for ($n/m/G/F_{max}$) problem. *Naval Res. Logist. Quart*, **1973**, *20*, 57–67.

2. Vakhania, N. A better algorithm for sequencing with release and delivery times on identical processors, *Journal of Algorithms* **2003]**, *48*, 273–293.
3. Garey, M.R.; Johnson, D.S. Computers and Intractability: A Guide to the Theory of NP–completeness. Freeman, San Francisco (1979).
4. McMahon, G; Florian, M. On scheduling with ready times and due dates to minimize maximum lateness, *Operations Research* **1975**, *23*, 475–482.
5. Vakhania, N. Fast solution of single-machine scheduling problem with embedded jobs, *Theoretical Computer Science* **2019**, *782*, 91–106.
6. Baptiste, P. Scheduling equal-length jobs on identical parallel machines, *Discrete Applied Mathematics* **2000**, *103*, 21–32.
7. Baptiste, P. Polynomial time algorithms for minimizing the weighted number of late jobs on a single machine with equal processing times. *Journal of Scheduling*, **1999**, *2* (6), 245–252.
8. Chrobak, M.; Dürr, C.; Jawor, W.; Kowalik, L.; Kurowski, M. A note on scheduling equal-length jobs to maximize throughput. *Journal of Scheduling*, **2006**, *9*, 71–73.
9. Vakhania, N. A study of single-machine scheduling problem to maximize throughput, *Journal of Scheduling* **2013**, *16*, No. 4, 395–403.
10. Vakhania, N. Scheduling jobs with release times premptively on a single machine to minimize the number of late jobs, *Operations Research Letters* **2009**, *37*, 405–410.
11. Garey, M.R.; Johnson, D.S.; Simons, B.B.; Tarjan, R.E.: Scheduling unit–time tasks with arbitrary release times and deadlines, *SIAM J. Comput.* **1981**, *10*, 256–269.
12. Simons, B.; Warmuth, M.: A.: fast algorithm for multiprocessor scheduling of unit-length jobs, *SIAM J. Comput.* **1989**, *18*, 690–710.
13. Dessouky, M.; Lageweg, B.J.; Lenstra, J.K.; van de Velde, S.L.: Scheduling identical jobs on uniform parallel machines, *Statistica Neerlandica* **1990**, *44*, 115–123.
14. Labetoulle, J.; Lawler, E.L.; Lenstra, J.K.; Rinnooy Kan, A.H.G.: Preemptive scheduling of uniform machines subject to release dates, in Pulleyblank, H.R. (ed.) Progress in Combinatorial Optimization, New York, Academic Press, **1984** 245–261.
15. Kravchenko, S.; Werner, F. Preemptive scheduling on uniform machines to minimize mean flow time, *Computers and Operations Research* **2009**, *36*, 2816–2821.
16. Lawler, E.L.; Labetoulle, J.: On preemptive scheduling of unrelated parallel processors by linear programming, *Journal of the Association of Computing Machinery* **1978**, *25* (4), 612–619.
17. Vakhania, N.; Hernandez, J.; Werner, F. Scheduling unrelated machines with two types of jobs. *Int. J. Prod. Res.* **2014**, *52* (13), 3793–3801.
18. Lenstra, J.K.; Shmoys, D.B.; Tardos, E. Approximation algorithms for scheduling unrelated machines, *Mathematical Programming*, 46, 259 - 271 (1990).
19. Brucker, P.; Kravchenko, S. Scheduling jobs with release times on parallel machines to minimize total tardiness, OSM Reihe P, **2005**, Heft 257, Universität Osnabrück, Fachbereich Mathematik/Informatik.
20. Brucker, P.; Kravchenko, S. Scheduling jobs with equal processing times and time windows on identical parallel machines, *Journal of Scheduling* **2008**, *11* (4), 229–237.
21. Vakhania, N. Branch less, cut more and minimize the number of late equal-length jobs on identical machines, *Theoretical Computer Science* **2012**, *465*, 49–60.
22. Kravchenko, S.; Werner, F. On a parallel machine scheduling problem with equal processing times, *Discrete Applied Mathematics*, **2009**, *157*, 848–852.
23. Kravchenko, S.; Werner, F.: Parallel machine problems with equal processing times: A survey, *Journal of Scheduling* **2011**, *14* (5), 435–444.
24. Vakhania, N. "Single-Machine Scheduling with Release Times and Tails". *Annals of Operations Research*, 129, p.253-271 2004.
25. Vakhania, N. Dynamic Restructuring Framework for Scheduling with Release Times and Due-Dates. Journal version: *Mathematics* 2019, 7(11), 1104; https://doi.org/10.3390/math7111104