Bidirectional Scheduling on a Path

Yann Disser, Max Klimm, and Elisabeth Lübbecke

Department of Mathematics, Technische Universität Berlin
{disser,klimm,eluebbecke}@math.tu-berlin.de

Abstract. We study the fundamental problem of scheduling bidirectional traffic across machines arranged on a path. The main feature of the problem is that jobs traveling in the same direction can be scheduled in quick succession on a machine, while jobs in the other direction have to wait for an additional transit time. We show that this tradeoff makes the problem significantly harder than the related flow shop problem, by showing that it is NP-hard even for jobs with identical processing and transit times. We give polynomial algorithms for a single machine and any constant number of machines. In contrast, we show the problem to be NP-hard on a single machine and with identical processing and transit times if some pairs of jobs in different directions are allowed to run on the machine concurrently. We generalize a PTAS of Afrati et al. [1] for one direction and a single machine to the bidirectional case on any constant number of machines.

1 Introduction

The scheduling of bidirectional traffic on a path is essential when operating single-track infrastructures such as single-track railway lines, canals, or telecommunication channels. Roughly speaking, the schedule governs when to move jobs from one node of the path to another along the edges of the path. The goal is to schedule all jobs such that the sum of their arrival times at their respective destinations is minimized. A central feature of real-world single-track infrastructures is that after one job enters a segment of the path, further jobs moving in the *same* direction can do so with relatively little headway, while traffic in the *opposite* direction has to wait until the whole segment is empty again.

In this paper, we introduce a novel scheduling model, which we term *bidirectional scheduling*, that models this tradeoff when scheduling bidirectional traffic. Our bidirectional scheduling model features a linear arrangement of machines similar to a flow shop model, but with the major difference that jobs may run on any consecutive subset of machines and in both directions. We distinguish between the *processing time* p_{ij} of a job j and its *transit time* τ_{ij} on machine i. While the former blocks the machine from being used by any other job (running in *either* direction), the latter only blocks the machine from being used by jobs running in *opposite* direction. For example, this allows us to model settings with bidirectional train traffic on a single-track railway line, where jobs correspond to trains, and the linear arrangement of machines corresponds to a linear arrangement of single-track lines connected by turnouts (cf. Lusby et al. [16, Section 2]).

The processing time of a job is the time needed for the train to move from a turnout entirely into the next railway line, while the transit time is the time to traverse the line.

We also study a generalization of the model to situations where some of the jobs are allowed to pass each other when traveling in different directions. This is a natural assumption, e.g., when scheduling the shipping traffic on a canal, where smaller ships are allowed to pass each other while larger ships are not (cf. Günther et al. [9]). In practice, the rules that decide which ships are allowed to pass each other are quite complex and depend on multiple parameters of the ships such as length, width, and draught (e.g., cf. [4]). We model these complex rules in the most general way by a bipartite compatibility graph for each machine, whose vertices correspond to the jobs and two jobs running in different directions are connected by an edge if they can run concurrently.

Our results and techniques. Table 1 gives a summary of our results. We first show that the bidirectional nature already makes our scheduling problem hard, even without processing times and with identical transit times (Section 3). Our reduction is from the NP-hard MaxCut problem, with multiple machines each containing a gadget for each vertex. The central trick of our construction is a way to swap the order of neighboring vertex gadgets from one machine to the next. With this tool, we can reach every vertex order on some machine, which allows to insert edge gadgets for each edge of the MaxCut-instance that make it desirable for two neighboring vertex gadgets to assign their vertices to different parts of the MaxCut partition.

We complement this hardness result by polynomial algorithms for identical jobs on constant numbers of machines (Section 4). We devise dynamic programs that even allow for a constant number of compatibility types, where each type is defined by a neighborhood in the compatibility graph. To this end, we observe that the jobs can only be in polynomially many configurations that can be ordered topologically. In contrast, we show that bidirectional scheduling with arbitrary compatibility graphs is hard already on a single machine (Section 5). We use a reduction from a variant of Sat with a sparse compatibility graph that only allows for specific combinations of jobs to run concurrently.

Finally, we turn to the general case with different processing times for each job and different transit times for each machine. This problem is known to be hard already in the one-directional case (and without transit times) [15]. Afrati et al. [1] gave a polynomial time approximation scheme (PTAS) for this setting. Generalizing the technique of [1], we are able to give a PTAS for the bidirectional case, and even extend this PTAS to constantly many machines (Section 6). To do this, we have to cope with issues arising from the interplay of processing and transit times for jobs in opposed directions as well as propagation effects between the different machines.

Related work. Scheduling problems are a fundamental class of optimization problems with a multitude of known hardness and approximation results (cf. Lawler et al. [14] for a survey). To the best of our knowledge, the bidirectional

Table 1. Overview of our results for bidirectional scheduling.
[1] only if $\tau_i/p \leq$ const, [2] even if $p = 0$, $\tau_i = 1$, [3] even if $\tau_i = p = 1$.

	Number m of machines		
compatibilities	$m = 1$	m const.	m arbitrary
Identical jobs ($p_{ij} = p$), $\tau_{ij} = \tau_i$			
none compatible	polynomial [Thm. 2]	polynomial[1] [Thm. 3]	NP-hard[2] [Thm. 1]
const. types			
arbitrary	NP-hard[3] [Thm. 7]		
Different jobs ($p_{ij} = p_j$), $\tau_{ij} = \tau_i$			
none compatible	NP-hard [15]/PTAS [Thm. 6]		NP-hard[2] [Thm. 1]
all compatible			

scheduling model that we propose and study in this paper has not been considered in the past nor is it contained as a special case in any other scheduling model. We give an overview of known results for related models.

For a single machine and jobs traveling from left to right, bidirectional scheduling reduces to the classical single machine scheduling problem, which Lenstra et al. [15] showed to be hard when minimizing total completion time. Afrati et al. [1] gave a polynomial-time approximation scheme (PTAS) with generalizations to multiple identical or unrelated machines. Chekuri and Khanna [5] further generalized the result to related machines. We give a different generalization for bidirectional scheduling on multiple machines. For unrelated machines Hoogeveen et al. [10] showed that the completion time cannot be approximated efficiently within arbitrary precision, unless P = NP. The case where jobs of both directions but without compatibilities are given has similarities to scheduling of two job families with a respective setup time. The general comments in Potts and Kovalyov [17] on dynamic programs for such kinds of problems apply in part to our technique for Theorem 2.

When all jobs need to be processed on all machines in the same order and all transit times are zero, bidirectional scheduling reduces to flow shop scheduling. Garey et al. [7] showed that it is NP-hard to minimize the sum of completion times in flow shop scheduling, even when there are only two machines and no release dates. They showed the same result for minimizing the makespan on three machines. Hoogeveen et al. [10] showed that there is no PTAS for flow shop scheduling without release dates, unless P = NP. In contrast, Brucker et al. [3] showed that flow shop problems with unit processing time can be solved efficiently, even when all jobs require a setup on the machines that can be performed by a single server only. Our PTAS for bidirectional scheduling is under the assumption that all processing times of a job coincide across machines.

Job shop scheduling is a generalization of flow shop scheduling that allows jobs to require processing by the machines in any (not necessarily linear) order (cf. Lawler et al. [14, Section 14] for a survey). Jansen et al. [11] gave a PTAS for a

constant number of machines and the minimization of the makespan, and Bansal et al. [2] gave improved algorithms for an unbounded number of machines. It is worth noting that job shop scheduling does not contain bidirectional scheduling as a special case, since it does not incorporate the distinction between processing and transit times for jobs passing a machine in different directions.

Scheduling bidirectional traffic is related to the so-called contraflow problem. Given a graph with travel times and capacities, the goal is to reverse some of the edges so that the total evacuation time for the graph is minimized. Kim and Shekhar [13] proposed a heuristic procedure that performs well on practical evacuation scenarios. Rebennack et al. [18] showed that an optimal set of arc reversals can be found efficiently for static flows, among other results.

2 Preliminaries

In the bidirectional scheduling problem, we are given a set $M = \{1, \dots, m\}$ of machines which we imagine to be ordered from left to right. Further, we are given two disjoint sets of J^{r} and J^{l} of *rightbound* and *leftbound* jobs, respectively, with $J = J^{\mathrm{r}} \cup J^{\mathrm{l}}$ and $n = |J|$. Each job is associated with a *release date* $r_j \in \mathbb{N}$, a *start machine* s_j and a *target machine* t_j, where $s_j \leq t_j$ for rightbound jobs and $s_j \geq t_j$ for leftbound jobs. A rightbound job j is processed in order by machines $s_j, s_j + 1, \dots, t_j - 1, t_j$, and a leftbound job is processed in order $s_j, s_j - 1, \dots, t_j + 1, t_j$. We denote the set of machines that job j needs to be processed on by M_j. Each job j is associated with a processing time $p_j \in \mathbb{N}$ and each machine i is associated with a transit time $\tau_i \in \mathbb{N}$. Note that we restrict ourselves to identical processing times for each job and identical transit times for each machine. We call $p_j + \tau_i$ the *running time* of job j on machine i.

A *schedule* is defined by fixing the start times S_{ij} for each job j on each machine $i \in M_j$. The *completion time* of job j on machine i is then defined as $C_{ij} = S_{ij} + p_j + \tau_i$. The overall completion time of job j is $C_j = C_{t_j j}$. A schedule is feasible if it has the following properties.

1. Release dates are respected, i.e., $r_j \leq S_{s_j j}$ for each $j \in J$.
2. Jobs travel towards their destination, i.e., $C_{ij} \leq S_{i+1,j}$ (resp. $C_{ij} \leq S_{i-1,j}$) for rightbound (resp. leftbound) jobs j and $i \in M_j \setminus \{t_j\}$.
3. Jobs j, j' traveling in the same direction are not processed on machine $i \in M_j \cap M_{j'}$ concurrently, i.e., $[S_{ij}, S_{ij} + p_j) \cap [S_{ij'}, S_{ij'} + p_{j'}) = \emptyset$.
4. Jobs j, j' traveling in different directions are neither processed nor in transit on machine $i \in M_j \cap M_{j'}$ concurrently, i.e., $[S_{ij}, C_{ij}) \cap [S_{ij'}, C_{ij'}) = \emptyset$.

We consider the objective of minimizing the *total completion time* $\sum C_j = \sum_{j \in J} C_j$. Other natural objectives are the minimization of the *makespan* $C_{\max} = \max\{C_j \mid j \in J\}$ or the *total waiting time* $\sum W_j = \sum_{j \in J} W_j$ where the individual waiting time of a job j is defined as $W_j = C_j - \sum_{i \in M_j} (p_j + \tau_i) - r_j$. Note that minimizing waiting time is equivalent to minimizing completion times.

We also consider a generalization of the model, where some of the jobs traveling in different directions are allowed to pass each other. Formally, for each machine i, we are given a bipartite *compatibility graph* $G_i = (J^{\mathrm{r}} \dot\cup J^{\mathrm{l}}, E_i)$ with

$E_i \subseteq J^{\mathrm{r}} \times J^{\mathrm{l}}$. Two jobs j, j' that are connected by an edge in G_i are allowed to run on machine i concurrently, i.e., condition 4 above need not be satisfied.

We denote the different variants of bidirectional scheduling considered in this paper by three-field notation $\alpha \,|\, \beta \,|\, \gamma$ as introduced by Graham et al. [8], where α defines the setup, β restrictions to the input, and $\gamma \in \{\sum C_j, \sum W_j, C_{\max}\}$ the objective. The input may be characterized by $\{B, Bm, B1\}$ for the general bidirectional scheduling, bidirectional scheduling on a constant number m of machines, or for bidirectional scheduling on a single machine, respectively. Further restrictions like $p_j = p$ or $\tau_j = \tau$ for β are straightforward.

All proofs omitted in the following sections can be found in the appendix.

3 Unbounded number of machines

In this section we show that the bidirectional scheduling problem is hard, even when all processing times are zero and all transit times coincide. In other words, we eliminate all interaction between jobs in the same direction and show that hardness is merely due to the decision when to switch between left- and right-bound operation of each machine. This is in contrast to one-directional (flow shop) scheduling with identical processing times, which is trivial. Formally, we show the following result.

Theorem 1. $B \,|\, p_j = 0, \tau_i = 1 \,|\, \sum C_j$ *is* NP*-hard.*

We reduce from the MaxCut problem which is contained in Karp's list of 21 NP-complete problems [12].

MaxCut
Input: An undirected graph $G = (V, E)$ and $k \in \mathbb{N}$.
Problem: Is there a partition $V = V_1 \dot\cup V_2$ with $|E \cap (V_1 \times V_2)| \geq k$?

For a given instance $\mathcal{I}$ of MaxCut we construct an instance of the bidirectional scheduling problem which can be scheduled without exceeding some specific waiting time if and only if $\mathcal{I}$ admits a solution. We give an intuitive overview of our construction and defer all details to Appendix A. Consult Figure 1 along with the following.

A cornerstone of our construction is the *vertex gadget* that occupies a fixed time interval on a single machine and can only be (sensibly) scheduled in two ways, which we interpret as the choice whether to put the corresponding vertex in the first or second part of the partition, respectively. We introduce multiple *vertex machines* that each have exactly one vertex gadget for each vertex in $\mathcal{I}$ and add further gadgets that ensure that the state of all vertex gadgets for the same vertex is the same across all machines. These gadgets allow us to synchronize vertex gadgets on consecutive vertex machines in two ways. We can either simply synchronize vertex gadgets that occupy the same time interval on the two vertex machines (*copy gadget*), or we can synchronize pairs of vertex gadgets occupying the same consecutive time intervals on the two vertex machines by linking the

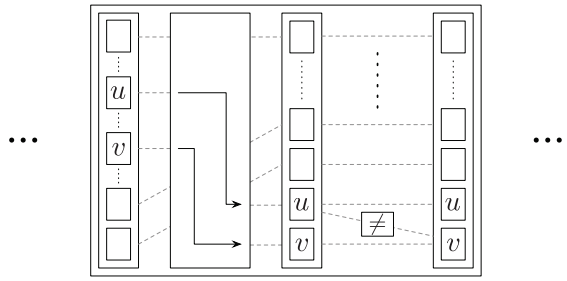

Fig. 1. Illustration of our hardness construction for a single edge $e = \{u, v\}$. First, a sequence of machines is used to change the order of vertex gadgets, such that the vertex gadgets corresponding to u and v occupy consecutive time intervals. Then, an edge gadget is added that incurs an increased waiting time if the vertex gadgets for u and v are in the same state.

first gadget on the first machine with the second one on the second machine and vice-versa, i.e., we can transpose the order of two consecutive gadgets from one vertex machine to the next (*transposition gadget*).

We construct an edge gadget for each edge in $\mathcal{I}$ that incurs a small waiting time if two vertex gadgets in consecutive time intervals and machines are in different states and a slightly higher waiting time if they are in the same state. By tuning the multiplicity of each job, we can ensure that only schedules make sense where vertex gadgets are scheduled consistently. Minimizing the waiting time then corresponds to maximizing the number of edge gadgets that link vertex gadgets in different states, i.e., maximizing the size of a cut.

In order to fully encode the given MaxCut instance $\mathcal{I}$, we need to introduce an edge gadget for each edge in $\mathcal{I}$. However, edge gadgets can only link vertex gadgets in consecutive time intervals. We can overcome this limitation by adding a sequence of vertex machines and transposing the order of two vertex gadgets from one machine to the next as described before. With a linear number of vertex machines we can reach an order where the two vertex gadgets we would like to connect with an edge gadget are adjacent. At that point, we can add the edge gadget, and then repeat the process for all other edges in $\mathcal{I}$ (cf. Figure 1).

We can reformulate Theorem 1 for nonzero processing times, simply by making the transit time large enough that the processing time does not matter.

Corollary 1. $B \,|\, p_j = 1, \tau_i = \tau \,|\, \sum C_j$ *is* NP*-hard.*

4 Constant number of machines

After establishing the hardness of bidirectional scheduling for an arbitrary number of machines and identical processing and transit times in the last section, in this section we turn to the case of a constant number of machines. We first show that the problem is easy for a single machine, and then expand our result to any fixed number of machines. Due to the identical processing times, the jobs

in each direction can simply be scheduled in the order of their release dates. The only decision left is when to switch between left- and rightbound operation of the machines. This decision is hard in the general case (Theorem 1), but we are able to formulate a dynamic program for any constant number of machines.

Our result generalizes to the case when some jobs of different directions are compatible (i.e., may pass each other), as long as the number of *compatibility types* is constant, where two jobs j_1, j_2 in the same direction are defined to have the same compatibility type if the set of jobs compatible with j_1 is equal to the set of jobs compatible with j_2 on each machine. Formally, j_1 and j_2 have the same compatibility type if $\{j : \{j_1, j\} \in E_i\} = \{j : \{j_2, j\} \in E_i\}$ for the compatibility graphs $G_i = (J^{\mathrm{l}} \dot\cup J^{\mathrm{r}}, E_i)$ of each machine i.

We partition J into κ subsets of jobs $J^1, \dots, J^\kappa$ where all jobs of J^c, $c \in 1, \dots, k$, have the same compatibility type c, and let $n_c = |J^c|$. Since the jobs of each subset only differ in their release dates, they can again be scheduled in the order of their release dates. This allows us to expand the dynamic program to encompass any constant number of compatibility types. We obtain the following result for a single machine.

Theorem 2. *$B1 \,|\, p_j = p, \kappa$ const.$\,|\, \sum C_j$ can be solved in polynomial time.*

Proof. We consider each subset J^c ordered non-increasingly by release dates and denote by J^c_i the i-th job of J^c in this order, i.e., the $(n_c - i)$-th job to be released. Each entry $T[i_1, t_1, \dots, i_\kappa, t_\kappa; c]$ of our dynamic programming table is designed to hold the minimum sum of completion times that can be achieved when scheduling only the $i_{c'}$ jobs of largest release date of each compatibility type c', such that $J^{c'}_{i_{c'}}$ is not scheduled before time $t_{c'}$ and $J^c_{i_c}$ is the first job that is scheduled. We start by setting $T[0, t_1, \dots, 0, t_\kappa; c] = 0$ and define the dependencies between table entries in the following.

Let $C(j, t) = \max\{t, r_j\} + p + \tau_1$ denote the smallest possible completion time of job j when scheduling it not before t. Depending on the types of jobs j_1, j_2 (and in particular of their directions), we can compute in constant time the earliest time $\theta(j_1, t_1, j_2, t_2)$ not before t_1 that job j_1 can be scheduled at, assuming that j_2 is scheduled earlier at time $\max\{t_2, r_{j_2}\}$. We let $\delta_{cc'} = 1$ if $c = c'$ and $\delta_{cc'} = 0$ otherwise, abbreviate $\theta_{c'} = \theta(J^{c'}_{i_{c'}}, t_{c'}, J^c_{i_c}, t_c)$, and get the following recursive formula for $i_c > 0$:

$$T[i_1, t_1, \dots, i_\kappa, t_\kappa; c] = \min_{c' : i_{c'} \neq 0} \{T[i_1 - \delta_{1c}, \theta_1, \dots, i_\kappa - \delta_{\kappa c}, \theta_\kappa; c'] + C(J^c_{i_c}, t_c)\}.$$

We can fill out our table in order of increasing sums $\sum i_c$ and finally obtain the desired minimum completion time as $\min_c T[n_1, 0, \dots, n_\kappa, 0; c]$. We can reconstruct the schedule from the dynamic programming table in straight-forward manner. It remains to argue that we only need to consider polynomially many times t_c. This is true, since all relevant times are contained in the set $\{r_j + k\tau + \ell p \mid j, k, \ell \leq n\}$ of cardinality $\mathcal{O}(n^3)$. □

We now consider a constant number of machines $m > 1$. The main complication in this setting is that decisions on one machine can influence decisions on

other machines, and, in general, every job can influence every other job in this way. In particular, we need to keep track of how many jobs of each type are in transit at each machine, and we can thus not easily adapt the dynamic program for a single machine. We propose a different dynamic program that relies on all transit times being bounded by a constant.

Theorem 3. *$Bm \,|\, p_j = 1, \tau_i$ const., κ const. $|\, \sum C_j$ can be solved in polynomial time.*

Proof. Again, we consider subsets of identical jobs. In addition to their conflict type c, we further distinguish jobs by their start and target machines s, t and form subsets $J^c_{s,t}$ correspondingly. The number of subsets is bounded by κm^2. Since all release times are integer and since $p_j = 1$, we only need to consider integer points in time. Hence, only $\tau_i + 1$ possible positions need to be considered for a job running on machine i, and no two jobs of the same direction can occupy the same position. The state of the system can be fully described by (i) the number of available jobs per machine and $J^c_{s,t}$, and (ii) for each position on each machine and each $J^c_{s,t}$, the fact whether a job of $J^c_{s,t}$ is occupying this position. The number of states is bounded by $\prod_{i=1}^{m} n^{\kappa m^2} \cdot \prod_{i=1}^{m} 2^{\kappa m^2 (\tau_i + 1)} = \text{poly}(n)$.

We define the successors of each state to be all states that can be reached in one time step where not all jobs wait, or by waiting for the next release date. This way, the state representation changes from one state to the next. The system always makes progress towards the final state where each job has arrived at its target. The state graph can thus not have a cycle, and we may consider states in a topological order. We formulate a dynamic program that computes for each state the smallest partial completion time to reach the state, where the partial completion time is defined as the sum of completion times of all completed jobs plus the current time for each uncompleted job. The dynamic program is well-defined as each value only depends on predecessor states. □

We conclude a complementary result to Theorem 1.

Corollary 2. *$Bm \,|\, p_j = 0, \tau_i = 1, \kappa$ const. $|\, \sum C_j$ can be solved in polynomial time.*

Proof. Since all release dates are integer, at each integer point in time no jobs are running on any machine. We can thus use a simpler version of the dynamic program we introduced in the proof of Theorem 3. □

5 Arbitrary compatibility graph

In the last section we demonstrated that the bidirectional scheduling problem can be solved efficiently for a constant number of machines, even when some jobs are compatible, as long as there are only a constant number of compatibility types among jobs. We now show that for arbitrary compatibility graphs the problem is hard already on a single machine with unit processing and transit times. For ease of exposition, we first consider the minimization of the makespan and then extend our result to minimum completion time.

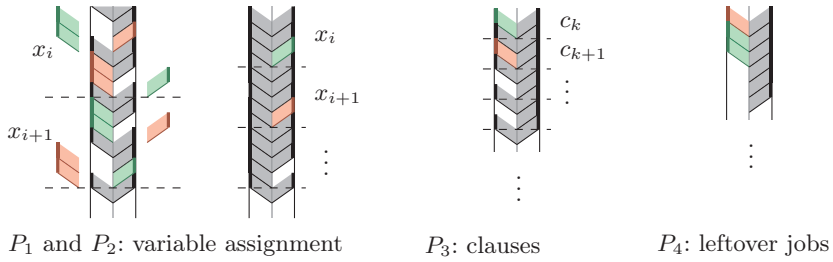

Fig. 2. Illustration (colored) of the four parts of our construction. Time is directed downwards, rightbound (leftbound) jobs are depicted on the left (right) of each figure.

Theorem 4. *$B1 \,|\, p_j = \tau_1 = 1, G_1 \,|\, \sum C_j$ is NP-hard.*

We give a reduction from an NP-hard variant of Sat (cf. [6]).

$(\leq 3, 3)$-Sat
Input: A formula with a set of clauses C of size three over a set of variables X, where each variable appears in at most three clauses.
Problem: Is there a truth assignment of X satisfying C?

For a given $(\leq 3, 3)$-Sat formula we construct a bidirectional scheduling instance that can be scheduled within some specific makespan T if and only if the given formula is satisfiable. Our construction is best explained by partitioning the time horizon $[0, T]$ into four parts (cf. Figure 2 along with the following).

We use a frame of blocking jobs that need to be scheduled at their release date. We can enforce this by making sure that at least one blocking job is released at (almost) each unit time step and that blocking jobs that are not supposed to run concurrently are incompatible. We release variable jobs that have to be scheduled into gaps between the blocking jobs. More precisely, in the first part of the construction we release 6 jobs within a separate time interval for each variable. Two of these jobs are leftbound and need to be scheduled within the first two parts of the construction, which implies that one of the two remaining pairs of rightbound jobs must be scheduled after the second part. If the first pair is delayed we interpret this as an assignment of *true* to the variable and otherwise as *false*.

The third part of the construction has a gap for each clause, with compatibilities ensuring that only variable jobs can be scheduled into the gap which satisfy the clause. Since each literal can only appear in at most two clauses, there are enough variable jobs to satisfy all clauses if the formula is satisfied. Finally, the last part has $2|X| - |C|$ gaps that fit any variable job. In order to schedule all variable jobs before the end of the last part, we thus need to schedule a variable job into each gap of a clause. This is possible if and only if the given $(\leq 3, 3)$-Sat formula is satisfiable. We can easily extend our result to completion or waiting times by adding many blocking jobs after the last part, such that violating the makespan also ruins the the total completion time.

6 Arbitrary processing times

In this section we consider the case of general processing times. Lenstra et al. [15] showed this problem to be hard already on a single machine if all jobs have the same direction. Afrati et al. [1] gave a polynomial time approximation scheme (PTAS), i.e., a polynomial $(1+\varepsilon)$-approximation algorithm for each $\varepsilon > 0$. Based on the same technique, we extend their result to the bidirectional case on a constant number of machines with complete or without conflicts on each machine. The main issue when trying to adopt the technique of [1] is to account for the different roles of processing and transit times for the interaction of jobs in the same and different directions. We start with a single machine.

Theorem 5. *$B1 \,|\, G_1 \in \{K_{n_r,n_l}, \emptyset\} \,|\, \sum C_j$ admits a PTAS.*

The first part of the proof in [1] is to restrict to processing times and release dates of the form $(1+\varepsilon)^x$ for some $x \in \mathbb{N}$ and $r_j \geq \varepsilon(p_j + \tau_1)$. Allowing fractional processing and release times we can show that any instance can be adapted to have these properties, without making the resulting schedule worse by a factor of more than $(1+\varepsilon)$. We may thus partition the time horizon into intervals $I_x = [(1+\varepsilon)^x, (1+\varepsilon)^{x+1}]$, such that every job is released at the beginning of an interval. Since jobs are not released too early, we may conclude that the maximum number of intervals s covered by the running time of a single job is constant. This allows us to group intervals together in blocks $B_t = \{I_{ts}, I_{ts+1}, \ldots, I_{(t+1)s-1}\}$ of s intervals each, such that every job scheduled to start in block B_t will terminate before the end of the next block B_{t+1}.

To use the fact that each block only interacts with the next block in our dynamic program, we need to specify an interface for this interaction. For that purpose we introduce the notion of a *frontier*. A block *respects an incoming frontier* $F = (f_l, f_r)$ if no leftbound (rightbound) job scheduled to start in the block starts earlier than f_l (f_r). Similarly, a block *respects an outgoing frontier* $F = (f_l, f_r)$ if no leftbound or rightbound job scheduled to start in the block would interfere with a leftbound (rightbound) job starting at time f_l (f_r). The symmetrical structure of the compatibility graph (K_{n_r,n_l} or $\emptyset$) allows us to use this simple interface. We introduce a dynamic programming table with entries $T[t, F, U]$ that are designed to hold the minimum total completion time of scheduling all jobs in $U \subseteq J$ to start in block B_t or earlier, such that B_t respects the outgoing frontier F. We define $C(t, F_1, F_2, V)$ to be the minimum total completion time of scheduling all jobs in V to start in B_t with B_t respecting the incoming frontier F_1 and the outgoing frontier F_2 (and ∞ if this is impossible). We have the following recursive formula for the dynamic programming table:

$$T[t, F, U] = \min_{F', V \subseteq U} \{T[t-1, F', U \setminus V] + C(t, F', F, V)\}.$$

To turn this into an efficient dynamic program, we need to limit the dependencies of each entry and show that $C(\cdot)$ can be computed efficiently. The number of blocks to be considered can be polynomially bounded by $\log D$,

where $D = \max_j r_j + n \cdot (\max_j p_j + \tau_1)$ is an upper bound on the makespan. The following lemma shows that we only need to consider polynomially many other entries to compute $T[t, F, U]$ and we only need to evaluate $C(\cdot)$ for job sets of constant size, which we can do in polynomial time by simple enumeration.

Lemma 1. *There is a schedule with a sum of completion times within a factor of $(1+\varepsilon)$ of the optimum and with the following properties:*

1. *The number of jobs scheduled in each block is bounded by a constant.*
2. *Every two consecutive blocks respect one of constantly many frontiers.*

Proof (sketch). Partitioning the released jobs of each interval direction-wise by processing time into *small* and *large* jobs and bundling small jobs into packages of roughly the same size allows us to bound the number of released jobs per interval by a constant, similarly as in [1]. Furthermore, we establish that we may assume jobs to remain unscheduled only for constantly many blocks.

For the second property, we stretch all time intervals by a factor of $(1+\varepsilon)$, which gives enough room to move the starting time of all jobs to the next $1/\varepsilon$-fraction of the same interval I_x. Thus, we only need to consider s/ε possible frontier values per direction, or a total of $(s/\varepsilon)^2$ possible frontiers. □

To generalize our dynamic program to a constant number of machines, we split our jobs into parts, one for each machine the job needs to be processed on, with the additional constraint that no part may be scheduled before any part of the same job on earlier machines. We are able to generalize Lemma 1 to this setting, using that each part of a job runs in at most two blocks and partitioning jobs into small and large for each direction and combination of start and target machines. The interface between consecutive time blocks needs to be extended to a frontier on each machine. In addition, a part running in block B_t imposes a lower bound on the start time of the next part of the same job running in block B_{t+1}. Since the number of parts running in block B_t is bounded by a constant b, the interface still has constant size. We assume that jobs are ordered and write $\boldsymbol{F} = (F_1, \ldots, F_m)$, $\boldsymbol{\theta} = (\theta_1, \ldots, \theta_b)$. We can define our table with entries $T[t, \boldsymbol{F}, U, V, \boldsymbol{\theta}]$ containing the minimum sum of (partial) completion times of scheduling the parts in U to start in block B_t or earlier, such that: B_t respects the outgoing frontier F_i on machine i, the parts in $V \subseteq U$ are scheduled to start in block B_t, and the l-th part in V stops running by time θ_l. Similarly, $C(t, \boldsymbol{F}', \boldsymbol{F}, V, \boldsymbol{\theta}', \boldsymbol{\theta})$ is the minimum sum of completion times for scheduling the parts in V in block B_t, respecting frontiers $\boldsymbol{F}', \boldsymbol{F}$ on the machines, such that the l-th part in V does not start running before time θ'_l and stops running by time θ_l (if possible, and ∞ otherwise). The recursive formula restricted to subsets that respect the order in which parts need to be processed becomes

$$T[t, \boldsymbol{F}, U, V, \boldsymbol{\theta}] = \min_{\substack{\boldsymbol{F}', V' \subseteq U \setminus V, \boldsymbol{\theta}' \\ |V'| \text{ is consistent}}} \{T[t-1, \boldsymbol{F}', U \setminus V, V', \boldsymbol{\theta}'] + C(t, \boldsymbol{F}', \boldsymbol{F}, V, \boldsymbol{\theta}', \boldsymbol{\theta})\}.$$

We obtain the following result.

Theorem 6. *$Bm \,|\, G_i \in \{K_{n_r, n_l}, \emptyset\} \,|\, \sum C_j$ admits a PTAS.*

References

1. Afrati, F., Bampis, E., Chekuri, C., Karger, D., Kenyon, C., Khanna, S., Milis, I., Queyranne, M., Skutella, M., Stein, C., Sviridenko, M.: Approximation schemes for minimizing average weighted completion time with release dates. In: Proc. 40th Symposium on Foundations of Computer Science (FOCS). pp. 32–43 (1999)
2. Bansal, N., Kimbrel, T., Sviridenko, M.: Job shop scheduling with unit processing times. Math. Oper. Res. 31, 381–389 (2006)
3. Brucker, P., Knust, S., Wang, G.: Complexity results for flow-shop problems with a single server. European J. Oper. Res. 165, 398–407 (2005)
4. Bundesamt für Seeschifffahrt und Hydrographie (BSH): German Traffic Regulations for Navigable Maritime Waterways. Hamburg and Rostock, Germany (2013)
5. Chekuri, C., Khanna, S.: A PTAS for minimizing weighted completion time on uniformly related machines. In: Proc. 28th Colloquium on Automata, Languages and Programming (ICALP), pp. 848–861 (2001)
6. Garey, M., Johnson, D.: Computers and intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman & Co., New York (1979)
7. Garey, M., Johnson, D., Sethi, R.: The complexity of flowshop and jobshop scheduling. Math. Oper. Res. 1(2), 117–129 (1976)
8. Graham, R.L., Lawler, E.L., Lenstra, J.K., Rinnooy Kan, A.H.G.: Optimization and approximation in deterministic sequencing and scheduling: A survey. Ann. Discrete Math. 5, 287–326 (1979)
9. Günther, E., Lübbecke, M.E., Möhring, R.H.: Challenges in scheduling when planning the ship traffic on the kiel canal. In: Proc. 10th Workshop on Models and Algorithms for Planning and Scheduling Problems (MAPSP) (2011)
10. Hoogeveen, H., Schuurman, P., Woeginger, G.J.: Non-approximability results for scheduling problems with minsum criteria. In: Proc. 6th International Conference on Integer Programming and Combinatorial Optimization (IPCO), pp. 353–366 (1998)
11. Jansen, K., Solis-Oba, R., Sviridenko, M.: Makespan minimization in job shops: A linear time approximation scheme. SIAM J. Discrete Math. 16(2), 288–300 (2003)
12. Karp, R.: Reducibility among combinatorial problems. In: Miller, R., Thatcher, J., Bohlinger, J. (eds.) Complexity of Computer Computations, pp. 85–103. The IBM Research Symposia Series (1972)
13. Kim, S., Shekhar, S.: Contraflow network reconfiguration for evacuation planning: A summary of results. In: Proc. 13th Annual ACM International Workshop on Geographic Information Systems (ACMGIS). pp. 250–259 (2005)
14. Lawler, E., Lenstra, J., Rinnooy Kan, A., Shmoys, D.: Sequencing and scheduling: Algorithms and complexity. In: Handbooks in Operations Research and Management Science, vol. 4, pp. 445–522 (1993)
15. Lenstra, J.K., Rinnooy Kan, A.H.G., Brucker, P.: Complexity of machine scheduling problems. Ann. Discrete Math. 1, 343–362 (1977)
16. Lusby, R.M., Larsen, J., Ehrgott, M., Ryan, D.: Railway track allocation: models and methods. OR Spectrum 33(4), 843–883 (2011)
17. Potts, C.N., Kovalyov, M.Y.: Scheduling with batching: A review. European J. Oper. Res. 120(2), 228 – 249 (2000)
18. Rebennack, S., Arulselvan, A., Elefteriadou, L., Pardalos, P.: Complexity analysis for maximum flow problems with arc reversals. J. Comb. Optim. 19(2), 200–216 (2010)

A Proofs of Section 3 – Unbounded number of machines

In this section, we give a detailed proof of the hardness of the bidirectional scheduling problem for a constant number of machines and identical processing and transit times. We describe our reduction from MaxCut. Let an instance $\mathcal{I} = (G_\mathcal{I}, k)$ of MaxCut be given, with $G = (V_\mathcal{I}, E_\mathcal{I})$, $|V_\mathcal{I}| = n_\mathcal{I}$, and $|E_\mathcal{I}| = m_\mathcal{I}$. We introduce a set of jobs on polynomially many machines that can be scheduled with a total waiting time of W if and only if $\mathcal{I}$ admits a solution. Our construction is comprised of various gadgets which we describe in the following. We make use of suitably large parameters $x \gg y \gg z \gg 1$ that we will specify later. For example, x is chosen in such a way that if ever x jobs are located at the same machine, these jobs need to be processed immediately in order to achieve a waiting time of W. Note that because jobs take no time in being processed (i.e., $p_j = 0$), we can schedule any number of jobs sharing direction simultaneously on a single machine. Also, since $\tau = 1$, it makes no sense for a machine to stay idle if jobs are available. This allows us to restrict our analysis to schedules that are *sensible* in the sense that for each machine and at every time step all jobs in one direction available at the machine get scheduled. On the other hand, the non-zero transit time induces a cost of switching the direction of jobs that are processed at a machine.

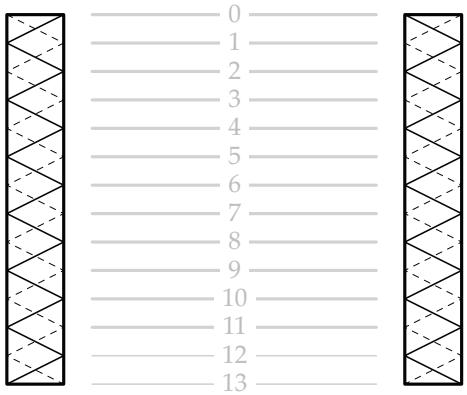

Fig. 3. Illustration of the vertex gadget in the leftbound (left) and the rightbound (right) state.

vertex gadget. Each of the machines $1, 10, 19, 28, \ldots$ hosts one vertex gadget for each of the vertices in $V_\mathcal{I}$ (cf. Figure 3 with the following). Each vertex gadget g_t on machine $9\ell+1$ occupies a distinct time interval $[13t, 13(t+1))$, $t < n_\mathcal{I}$, on the machine and is associated with one of the vertices $v \in V_\mathcal{I}$. The gadget comes with $24y$ *vertex jobs* that only need to be processed at machine $9\ell + 1$,

half of them being leftbound, half being rightbound. Exactly y jobs of each direction are released at times $13t, 13t+1, \ldots, 13t+11$. We say that g_t is scheduled *consistently* if either all leftbound vertex jobs are processed immediately when they are released and all rightbound jobs wait for one time unit, or vice-versa. We say the gadget is in the *leftbound* (*rightbound*) *state* and interpret this as vertex v being part of set V_1 (V_2) of the partition of $V_{\mathcal{I}} = V_1 \cup V_2$ we are implicitly constructing. A schedule is *consistent* if all vertex gadgets are scheduled consistently. The following lemma allows us to distinguish consistent schedules.

Lemma 2. *The vertex jobs of a single vertex gadget can be scheduled consistently with a waiting time of* $12y$*, while every inconsistent schedule has waiting time at least* $13y$*.*

Proof. Since $p = 0$, we can schedule all available jobs with the same direction simultaneously. It follows that both consistent schedules are valid, and, since in both exactly half of the vertex jobs wait for one unit of time, the total waiting time of such a schedule is $12y$. Any inconsistent (sensible) schedule would have to send jobs in the same direction in two consecutive unit time intervals, which means that in addition to the minimum waiting time of $12y$, at least y jobs have to wait an extra unit of time. □

synchronizing vertex gadgets. Since every vertex $v \in V_{\mathcal{I}}$ is represented by multiple vertex gadgets on different machines, we need a way to ensure that all vertex gadgets for v are in agreement regarding which part of the partition v is assigned to. We introduce two different gadgets that handle synchronization. The *copy gadget* synchronizes the vertex gadgets g_t occupying the same time interval on machines $9\ell + 1$ and $9\ell + 10$, while the *transposition gadget* synchronizes gadgets g_t, g_{t+1} on machine $9\ell+1$ with gadgets g_{t+1}, g_t on machine $9\ell+10$. Using a combination of copy and transposition gadgets, we can transition between any two orders of vertex gadgets on distant machines.

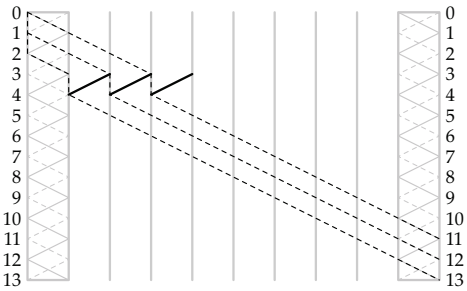

Fig. 4. Illustration of the copy gadget between two vertex gadgets. The dashed lines depict all sensible trajectories of the synchronizing jobs, assuming that the vertex gadgets are in the same state.

We first specify the copy gadget that synchronizes the vertex gadgets g_t on two machines $9\ell + 1$ and $9\ell + 10$ (cf. Figure 4 with the following). The gadget consists of $2z$ rightbound *synchronization jobs*, half of which are released at time $13t$ and half at time $13t + 1$. The jobs need to be processed on all machines $9\ell+1, \ldots, 9\ell+10$ in this order. In addition, we introduce $3x$ *blocking jobs* that are used to enforce that specific time intervals on a machine are reserved for leftbound/rightbound operation. Essentially, releasing x blocking jobs at time t on a single machine prevents any jobs to be processed in opposite direction during the time interval $[t, t + 1)$ (and even earlier). In this manner, we block the interval starting at time $13t + 3$ on machines $9\ell + 2, 9\ell + 3, 9\ell + 4$.

Lemma 3. *In any consistent schedule, the synchronization jobs of a single copy gadget can be scheduled with a waiting time of* $3z$ *if the two corresponding vertex gadgets are in the same state, otherwise their waiting time is at least* $5z$*.*

Proof. Since $x \gg z$, we need to schedule all blocking jobs as soon as they are released. If both vertex gadgets g_t linked by the copy gadget are in the rightbound state, the synchronization jobs released at time $13t$ only have to wait for one time unit at machine $9\ell+4$, while the other jobs have to wait at machines $9\ell+1$ and $9\ell + 2$. Similarly, if the vertex gadgets are in the leftbound state, the first half of the jobs have to wait at machines $9\ell + 1$ and $9\ell + 3$, while the other half only has to wait at machine $9\ell + 3$. The waiting time in either case is $3z$. If the vertex gadgets are in opposite states, all jobs have to additionally wait at machine $9\ell + 10$, which results in a total waiting time of at least $5z$. □

We now describe the transposition gadget that synchronizes the vertex gadgets g_t, g_{t+1} on machine $9\ell+1$ with the vertex gadgets g_{t+1}, g_t on machine $9\ell+10$ (cf. Figure 5 with the following). The challenge here is that jobs synchronizing the different pairs of vertex gadgets need to pass each other without interfering. We achieve this by making sure that the jobs never meet while being in transit at the same machine. The gadget consists of $4z$ synchronization jobs, half being rightbound and half being leftbound. Half of each are released at times $13t + 6$ and $13t + 7$, and all need to be processed at machines $9\ell + 1, \ldots, 9\ell + 10$ (in different directions). In addition, we introduce $12x$ blocking jobs to block the intervals starting at the following times: at times $13t+9$, $13t+10$ for rightbound jobs and at times $13t+14$, $13t+15$ for leftbound jobs on machine $9\ell+2$, at times $13t + 9$ for rightbound and at $13t + 15$ for leftbound on machine $9\ell + 3$, and the corresponding (symmetrical) intervals in opposite direction on machines $9\ell + 8$ and $9\ell + 9$ (cf. Figure 5).

Lemma 4. *In any consistent schedule, the synchronization jobs of a single transposition gadget can be scheduled with a waiting time of* $10z$ *if each of the two pairs of corresponding vertex gadgets are in the same state, otherwise their waiting time is at least* $12z$*.*

Proof. Since $x \gg z$, we need to schedule all blocking jobs as soon as they are released. It is easy to verify that all synchronization jobs wait at exactly 2 machines due to blocking jobs. In addition, half of the jobs wait for one unit of time

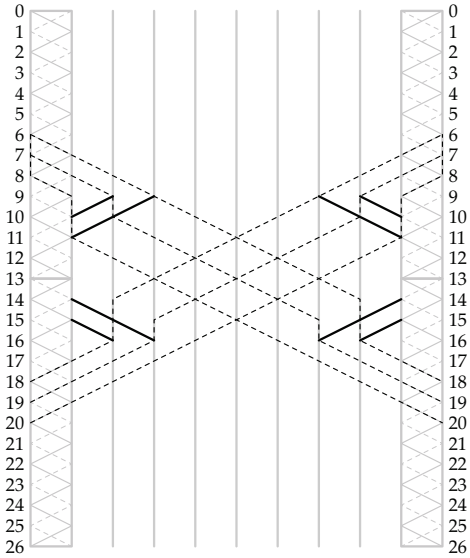

Fig. 5. Illustration of the transposition gadget. The dashed lines depict all sensible trajectories of the synchronizing jobs, assuming that the vertex gadgets are pairwise in the same states. Note that jobs in different directions never meet while in transit through the same machine.

at the machine where they are released – for a total of $10z$ time units. If the pair of vertex gadgets is in opposite states, all connecting synchronization jobs need to wait at least one additional unit of time at their last machine. Observe that synchronization jobs in opposite directions are never in transit on the same machine at the same time. □

edge gadget. The purpose of an edge gadget between vertex gadget g_t on machine $9\ell + 1$ and g_{t+1} on machine $9\ell + 10$ is to produce a small additional waiting time if the two vertex gadgets are in the same state (cf. Figure 6 with the following). We will introduce edge gadgets between vertex gadgets representing two vertices u, v that share an edge in G. This way, every edge that connects vertices in different parts of the partition is beneficial for the resulting waiting time. The edge gadget itself consists of 2 rightbound *edge jobs*, one being released at time $13t + 7$ and the other at time $13t + 8$. Both jobs need to be processed on machines $9\ell + 1, \ldots, 9\ell + 10$. We add $3x$ blocking jobs to block the unit time interval starting at time $13t + 15$ on machines $9\ell + 7, 9\ell + 8, 9\ell + 9$.

Lemma 5. *In any consistent schedule, the edge jobs of a single edge gadget can be scheduled with a waiting time of* 3 *if the two connected vertex gadgets are in opposite states, otherwise their waiting time is at least* 5*.*

Proof. One job always has to wait for a time unit at the first machine. Both jobs have to wait for the blocking jobs (since $x \gg 1$). If the vertex gadgets are

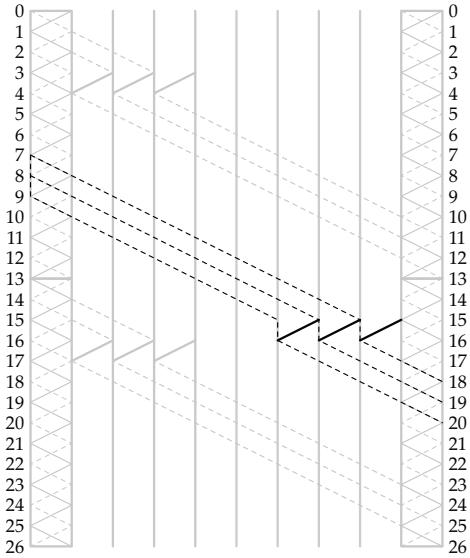

Fig. 6. Illustration of the edge gadget. The dashed lines depict all sensible trajectories of the synchronizing jobs, assuming that the vertex gadgets are in opposite states. Note that edge jobs do not interact with synchronization jobs of copy gadgets for both vertices.

in the same state, both jobs have to wait an additional unit of time at the last machine. □

construction. We are now ready to combine our gadgets and explain the final construction.

Theorem 1. $B \,|\, p_j = 0, \tau_i = 1 \,|\, \sum C_j$ *is* NP*-hard.*

Proof. We start by introducing a vertex gadget g_t on machine 1 for each vertex $v_t \in V_{\mathcal{I}}$ of the given MaxCut-instance. For each edge $\{u, v\}$ we extend the construction by appending more machines as follows. We add a sequence of blocks of 9 machines, the last of which contains again a vertex gadget for each vertex. In between we add copy and transposition gadgets in such a way that on the last machine i the vertex gadgets g_0 and g_1 represent the vertices u and v. We can achieve this by adding less than $n_{\mathcal{I}}$ machines. We add an additional block of 9 machines, and add copy gadgets for each of the variables. Finally, we add an edge gadget connecting vertex gadget g_0 on machine i with g_1 on the last machine. Observe that the edge jobs do not interfere with any of the synchronization jobs for the copy gadgets for the first two vertices (cf. Figure 6). We repeat the process once for each edge. The total number of machines is $\mathcal{O}(n_{\mathcal{I}} m_{\mathcal{I}})$, and the total number of jobs is $\mathcal{O}(n_{\mathcal{I}}^2 m_{\mathcal{I}}(x+y+z))$. The number of vertex gadgets is $n_v < n_{\mathcal{I}}^2 m_{\mathcal{I}}$, and the number of transposition and copy gadgets is $n_t < n_c < n_v$.

We claim that if the MaxCut instance admits a solution $\mathcal{S}$, we can schedule all jobs with waiting time at most $W = 12n_v y + 3n_c z + 10n_t z + 5m_{\mathcal{I}} - 2k$. We do this by scheduling all vertex gadgets consistently in the state corresponding to the part of the partition the corresponding vertex belongs to in $\mathcal{S}$. Lemmas 2 through 4 guarantee that we can schedule everything but the edge jobs without incurring a waiting time greater than $12n_v y + 3n_c z + 10n_t z$. Finally, since at least k edges in the MaxCut solution are between vertices in different sets of the partition, and the vertex gadgets are set accordingly, by Lemma 5, we obtain an additional waiting time of at most $5m_{\mathcal{I}} - 2k$ as claimed.

It remains to establish that the waiting time exceeds W in case the MaxCut instance does not admit a solution. We set $x = W + 1$, such that all blocking jobs have to be scheduled as soon as they are released. By Lemma 2, scheduling at least one vertex gadget inconsistently produces a total waiting time of at least $12n_v y + y$. We now set $y = 18n_{\mathcal{I}}^2 m_{\mathcal{I}} z > 3n_c z + 10n_t z + 5m_{\mathcal{I}}$ for the vertex jobs, such that a single inconsistent vertex gadget results in a waiting time greater than W. Hence, each vertex gadget needs to be scheduled consistently. By Lemmas 3 and 4, we have that if not all vertex gadgets corresponding to the same vertex are in the same state, the waiting time for vertex and synchronization jobs is at least $12n_v y + 3n_c z + 2n_t z + z$. We set $z = 5m_{\mathcal{I}}$, which allows us to conclude that all vertex gadgets are in agreement regarding the partition of the vertices. Finally, Lemma 5 enforces that there are at least k edge gadgets between vertices in different states. This however is impossible as our MaxCut instance does not admit a solution. □

Corollary 1. $B \,|\, p_j = 1, \tau_i = \tau \,|\, \sum C_j$ *is* NP*-hard.*

Proof. The same construction works for $p = 1$ and $\tau = n \cdot m$, since then even waiting at each machine for all other jobs to process is preferable to waiting once for the transit of another job. □

B Proofs of Section 5 – Arbitrary compatibility graph

In this section we give a detailed hardness proof for bidirectional scheduling on a single machine where jobs can be compatible. Our proof holds even for unit processing and transit times. We first consider the makespan objective and extend the proof in a second step to waiting time and total completion time.

B.1 Makespan Minimization

Theorem 7. *$B1 \,|\, p_j = \tau_1 = 1, G_1 \,|\, C_{\max}$ is strongly NP-hard.*

In the following, we construct a bidirectional scheduling instance for a given $(\leq 3, 3)$-SAT instance. The constructed instance yields a demanded makespan $C_{\max}$ if and only if the given $(\leq 3, 3)$-SAT formula is satisfiable. We partition the time horizon into 5 parts $P_1, \ldots, P_5$ with start time $A_1 = 0, A_2 = 6|X|, A_3 = 10|X|, A_4 = 10|X|+2|C|$, and $A_5 = 12|X|+|C|$. The demanded makespan $C_{\max} = A_5 + 1$ will enforce that all jobs start before the end of the fourth part.

The rough idea is as follows: In the first four parts we release a tight frame of *blocking jobs* B and *dummy jobs* H that have to start running immediately at their release date in any schedule that achieves $C_{\max}$. We use these jobs to create gaps for *variable jobs* that represent the variable assignments. By defining the compatibilities for the blocking jobs we are able to control which of these assignment jobs can be scheduled into each gap. In the first part of our construction, we release all variable jobs, which come in two *types*: one type representing a *true* assignment to the corresponding variable and the other type representing a *false* assignment. Our construction will enforce the following properties in each of its parts:

Lemma 6. *In every feasible schedule with makespan $C_{\max}$, all jobs released before S_3 are scheduled in parts P_1 and P_2, except for two rightbound variable jobs of same type for each variable.*

Lemma 7. *In every feasible schedule with makespan $C_{\max}$, the only jobs released before A_3 and scheduled in P_3 are rightbound variable jobs, where each corresponds to a variable assignment satisfying a different clause.*

Lemma 8. *In every feasible schedule with makespan $C_{\max}$, the only jobs released before A_4 and scheduled in P_4 are rightbound variable jobs, and there are not more than $2|X| - |C|$ of them.*

In the following we explicitly define the released jobs of each part achieving the above properties. Each part is accompanied by a figure illustrating when jobs are released, the respective compatibility graph and an example of a schedule. In all figures, time is directed downwards, and all rightbound jobs are depicted to the left and all leftbound jobs to the right of the machine. Since compatible jobs can run concurrently, the schedules of the leftbound and the rightbound jobs are drawn separately.

We start by specifying the jobs released in P_4.

jobs of P_4. In the last part, $2|X| - |C|$ leftbound blocking jobs $B_4 = \{b_i \mid i = 0, \ldots, 2|X| - |C| - 1\}$ are released at $A_4 + i$ for each $b_j \in B_4$ in part P_4 leaving space for leftover rightbound variable jobs not scheduled until the beginning of this part. Each of the blocking jobs is only compatible with all rightbound variable jobs.

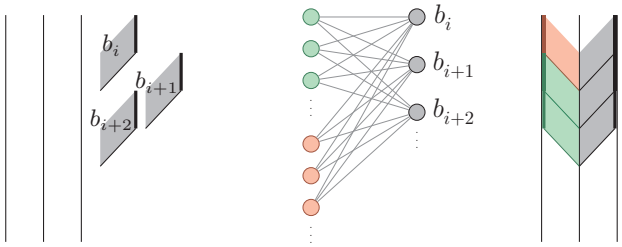

Fig. 7. Part P_4 with blocking jobs reserving space for all remaining rightbound variable jobs.

Proof (Proof of Lemma 8). First, observe that with the required makespan of $A_5 + 1 = A_4 + 2|X| - |C| + 1$ each blocking job of B_4 must be scheduled directly at its release date. Consequently, there is no room to delay the start of any leftbound job released before P_4 to this part. Due to the compatibilities, the rightbound blocking and dummy jobs released before P_4 are also forced to run before the start of P_4. Therefore, there are exactly $2|X| - |C|$ open slots within P_4 reserved for rightbound variable jobs. □

jobs of P_3. The third part (Figure 8) is responsible for the assignment of satisfying literals to each clause. It consists of a set of jobs B_3 containing one leftbound blocking job b_k per clause c_k released at $A_3 + 2k$, which is compatible with each rightbound variable job that represents a variable assignment satisfying this clause. The gaps between the blocking jobs are filled with dummy jobs H_3 containing one rightbound job h_k^{r} and one leftbound job h_k^{l} with release date $A_3 + 2k + 1$ per clause $c_k \in C$. Each leftbound dummy job is compatible with all rightbound variable jobs, furthermore each rightbound dummy job h is compatible with the three leftbound jobs released in $[r_h - 1, r_h + 1]$.

Proof (Proof of Lemma 7). By Lemma 8 all jobs released within P_3 must start before the end of P_3. Hence, each leftbound dummy and blocking job is forced to start at its release date. Therefore, due to the compatibilities, each rightbound dummy job must be scheduled directly when released. The only remaining $|C|$ free slots can be filled with rightbound variable jobs – exactly one free slot per clause c_k reserved for a variable job representing an assignment that satisfies c_k. □

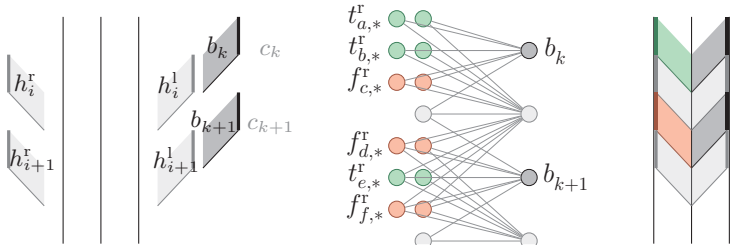

Fig. 8. Part P_3 for $c_k = (x_a \vee x_b \vee \bar{x}_c)$ and $c_{k+1} = (\bar{x}_d \vee x_e \vee \bar{x}_f)$. Note that each variable job can be adjacent with more than one clause job (although this does not occur in the example).

We continue by introducing the released jobs within P_1 and P_2 to prove Lemma 6 afterwards.

jobs of P_1. In the first part, we release different kinds of jobs per variable (cf. Figure 9). There are rightbound variable jobs $T^{\mathrm{r}} = \{t_{i,1}^{\mathrm{r}}, t_{i,2}^{\mathrm{r}} \mid x_i \in X\}$ representing a true assignment as well as rightbound variable jobs $F^{\mathrm{r}} = \{f_{i,1}^{\mathrm{r}}, f_{i,2}^{\mathrm{r}} \mid x_i \in X\}$ representing a false assignment. These jobs are complemented by leftbound variable jobs $F^{\mathrm{l}} = \{f_i^{\mathrm{l}} \mid x_i \in X\}$ and $T^{\mathrm{l}} = \{t_i^{\mathrm{l}} \mid x_i \in X\}$ and further leftbound jobs $Q = \{q_i^{\mathrm{t}}, q_i^{\mathrm{f}} \mid x_i \in X\}$ called *indefinite*, both with the purpose to enforce a consistent assignment. To do so, we implement a certain structure by leftbound jobs $B_1 = \{b_i^{\mathrm{t}}, b_i^{\mathrm{f}} \mid x_i \in X\}$ for blocking, and some further dummy jobs $H_1 = \{h_i^{\mathrm{rt}}, h_i^{\mathrm{lt}}, h_i^{\mathrm{rf}}, h_i^{\mathrm{lf}} \mid x_i \in X\}$ for filling, for each variable and each value one leftbound and one rightbound job. We release the rightbound true jobs $t_{i,1}^{\mathrm{r}}$ at $6i$ and $t_{i,2}^{\mathrm{r}}$ at $6i+1$, the rightbound false jobs $f_{i,1}^{\mathrm{r}}$ at $6i+3$ and $f_{i,2}^{\mathrm{r}}$ at $6i+4$. Each indefinite job q_i^{t} together with the leftbound f_i^{l} is released at $6i+1$, each q_i^{f} together with t_i^{l} at $= 6i+4$. Furthermore we release the blocking jobs b_i^{t} at $6i$ and b_i^{f} at $6i+3$ as well as the dummy jobs $h_i^{\mathrm{rt}}, h_i^{\mathrm{lt}}$ at $6i+2$ and $h_i^{\mathrm{rf}}, h_i^{\mathrm{lf}}$ at $6i+5$. The compatibility graph G_1 is defined such that each blocking job b_i^{t} is compatible with the corresponding $t_{i,1}^{\mathrm{r}}$ and $t_{i,2}^{\mathrm{r}}$, and each b_i^{f} with $f_{i,1}^{\mathrm{r}}$ and $f_{i,2}^{\mathrm{r}}$, respectively. The first indefinite job q_i^{t} is compatible with the corresponding rightbound true jobs t_i^{r} and t_i^{r} as well as the second q_i^{f} with f_i^{r} and f_i^{r}, respectively. Finally, we define each dummy job $h \in H_1$ to be compatible with the opposed jobs released in $[r_h - 1, r_h + 1]$. None of the remaining pairs of jobs are compatible.

jobs of P_2. In the second part (Figure 10), there is room for exactly one indefinite job and one leftbound variable job per variable. This is realized by a set of rightbound blocking jobs $B_2 = \{b_{i,1}, b_{i,2} \mid x_i \in X\}$ where each $b_{i,1}$ released at $A_2 + 4i$ is compatible with the corresponding two indefinite jobs q_i^{t} and q_i^{f}. Each $b_{i,2}$ released at $A_2 + 4i + 2$ is compatible with the corresponding two leftbound variable jobs f_i^{l} and t_i^{l}. The gaps between two subsequent released blocking jobs are closed in both directions by dummy jobs $H_2 = \{h_{i,1}^{\mathrm{r}}, h_{i,2}^{\mathrm{r}}, h_{i,1}^{\mathrm{l}}, h_{i,2}^{\mathrm{l}} \mid x_i \in$

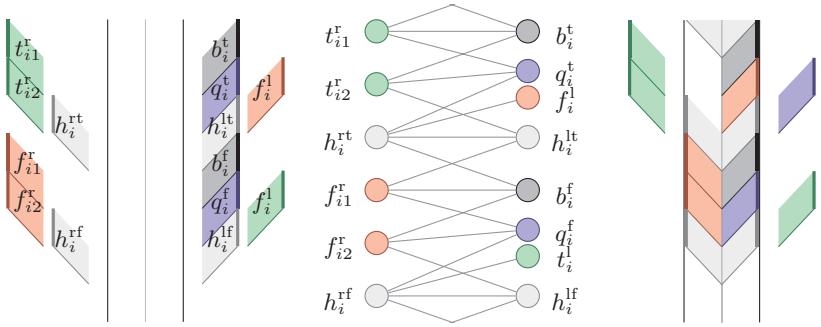

Fig. 9. Released jobs per variable x_i in P_1, the corresponding compatibilities given by G_1 and a scheduled example for a true variable assignment.

$X\}$ released at $A_2 + 4i + 1$ and $A_2 + 4i + 3$. Each dummy job is compatible with all jobs of $Q, T^{\mathrm{l}}, F^{\mathrm{l}}$, or B_2 and the corresponding opposed dummy job released concurrently.

Proof (Proof of Lemma 6). By Lemmas 8 and 7 each rightbound dummy and blocking job of H_2 and B_2 must be scheduled before the end of P_2 and hence, directly at its release. By the given compatibilities this is also true for the leftbound dummy jobs of H_2. Therefore, there are exactly two open slots per variable x_i, one reserved for the two corresponding indefinite jobs $q^{\mathrm{t}}_i, q^{\mathrm{f}}$ and one for the two corresponding leftbound variable jobs $f^{\mathrm{l}}_i, t^{\mathrm{l}}_i$. Since no further space is left, for both pairs exactly one can be scheduled within P_2. The remaining one must be completed already by the end of P_1.

Also, for the first part, we can conclude that no blocking and no dummy job released in P_1 can start after the end of P_1. Consider now one variable x_i and assume that no job corresponding to x_i can start within part P_1 after $6i + 5$. This assumption holds obviously for x_n. Then, h^{rf}_i and h^{lf}_i, the latest released jobs corresponding to x_i, must both start at their release.

If the leftbound job t^{l}_i is scheduled within part P_1 it must be scheduled at its release and hence $f^{\mathrm{r}}_{i,1}$ and $f^{\mathrm{r}}_{i,2}$ must be postponed to the next parts. In this case, also the second blocking job b^{f}_i as well as the first two dummy jobs h^{rt}_i and h^{lt}_i are forced to start at their release, consequently also b^{t}_i. In this case it is not possible anymore to schedule q^{f}_i within part P_1. For this reason, the counter part q^{t}_i must be scheduled at its release time and the leftbound f^{l}_i must be postponed. With this, there is exactly one free slot for $t^{\mathrm{r}}_{i,2}$ and one for $t^{\mathrm{r}}_{i,1}$.

If, on the other hand, the leftbound job t^{l}_i is scheduled after part P_1, we have to schedule f^{l}_i within part P_1. Due to the conflicts with h^{rf}_i, the start time of f^{l}_i and the blocking and dummy jobs in between must in particular be scheduled at their release. For that reason q^{t}_i must be postponed and q^{f}_i must be scheduled at its release. Hence, also the rightbound true jobs t^{r}_i and t^{r}_i must be postponed and there are exactly two slots for the two false jobs.

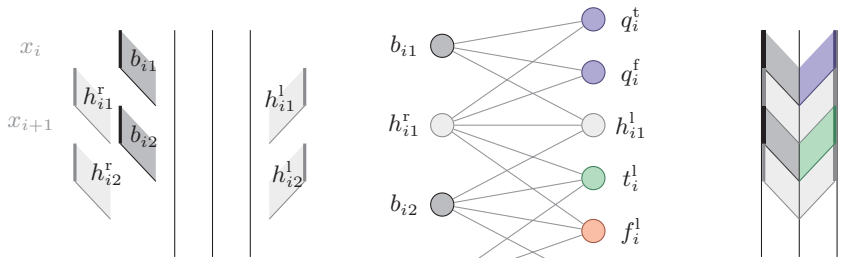

Fig. 10. Part P_2 creates a structure of blocking and dummy jobs with respective compatibilities that create space for exactly one indefinite job per variable x_i.

In both cases, the scheduled leftbound jobs ensure that no earlier released variable job can start after $6(i-1)+5$. Hence, it can be concluded by induction that, for each variable, either all corresponding false jobs or all corresponding true jobs must be scheduled after part P_1. And since, by Lemmas 8 and 7, at least $2n$ rightbound variable jobs must be scheduled within P_1 the free spots ensure that exactly the two counter parts are scheduled within P_1. □

We can conclude the following claim and hence, Theorem 7.

Claim. There is a satisfying assignment for the given $(\leq 3, 3) - \textsc{Sat}$ instance if and only if there is a feasible schedule for the constructed scheduling instance with makespan $C_{\max} = A_5 + 1$.

Proof (Proof of Theorem 7). If there is a schedule with makespan $C_{\max}$ we can apply Lemmas 8 to 6. Within the resulting schedule we can therefore be sure that $|C|$ rightbound variable jobs are scheduled within the clause part. Since by Lemma 6 the assignment of each variable is well defined we get by Lemma 7 a satisfying truth assignment for the clauses.

If on the other hand a satisfying truth assignment is given, the described schedule with demanded makespan can be created in straight-forward manner, by postponing the assignment jobs corresponding to the truth assignment and scheduling all other jobs within the part they are released in (or in part P_2 in the case of leftbound variable jobs or indefinite jobs). □

B.2 Minimization of Total Completion Time

Theorem 4. *$B1 \,|\, p_j = \tau_1 = 1, G_1 \,|\, \sum C_j$ is NP-hard.*

We give an analogous reduction as for Theorem 7. Note, that solutions optimal for the total completion time and those optimal for the total waiting time are equivalent. Hence, it is sufficient to prove the hardness for the latter. The goal is to enforce the same structure as for makespan minimization when minimizing the total waiting time. To do so, we start by calculating an upper bound of the resulting waiting time.

We can trivially bound the total waiting time of a schedule that achieves makespan $C_{\max}$ by $W = |J| \cdot C_{\max} = |J| \cdot (A_5 + 1)$, where J is the set of all jobs in our construction. With this polynomial bound we can extend the construction of a scheduling instance for a given $(\leq 3, 3)$-SAT instance by part P_5 with $W+1$ further leftbound blocking jobs $B_5 = \{b_i \mid i = 0, \ldots, W\}$ with release date $A_5 + i$ for each $b_i^5 \in B_5$ that are not compatible to any of the previous jobs.

Claim. There is a satisfying truth assignment for the given $(\leq 3, 3)$-SAT instance if and only if there is a feasible schedule for the constructed scheduling instance with total waiting time of at most W.

Proof (Proof of Corollary 4). Assume first that there is a satisfying assignment for the $(\leq 3, 3)$-SAT instance. In this case, there is a schedule where no job released in the first four parts starts processing after A_5 and hence the resulting total waiting time does not exceed W.

Assume on the other hand, that there is a solution for the constructed scheduling instance whose objective does not exceed W. For such a solution, either all jobs released in the first four parts start before A_5 or their is at least one starting later. In the first case, we get, by Lemmas 8 to 6, a schedule together with a satisfying truth assignment with waiting time bounded by W.

In the second case each postponed job j with starting time S'_j increases the already existing waiting time by at least an amount of $(S'_j - A_5) + W + 1 - (S'_j - A_5) = W + 1$. Hence, the first case applies.

□

C Proofs of Section 6 – Arbitrary processing times

In this Section we restate the Lemmas with detailed proofs that are necessary to show the existence of a PTAS if the processing times of the jobs are not restricted to be equal.

C.1 Single Machine

We consider first the case of a single machine, more precisely $B1 \,|\, p_j, \tau_1, G_1 \in \{K_{n_r,n_l}, \emptyset\} \,|\, \sum C_j$. Following the proof scheme of [1], we introduce several lemmas that allow us to make assumptions at "$\mathcal{O}(1+\varepsilon)$-loss", meaning that we can modify any input instance and optimum schedule to adhere to these assumptions, such that the resulting schedule is within a factor polynomial in $(1+\varepsilon)$ of the optimum schedule for the original instance. To not complicate matters unnecessarily, in the following we allow fractional release dates and processing times.

Lemma 9. *With $\mathcal{O}(1+\varepsilon)$-loss we can assume that $r_j, p_j \in \{(1+\varepsilon)^x \mid x \in \mathbb{N}\} \cup \{0\}$, $r_j \geq \varepsilon(p_j + \tau_1)$, and $r_j \geq 1$ for each $j \in J$.*

Proof. Increasing any value $v \in \mathbb{R}$ to the smallest power of $(1+\varepsilon)$ not smaller than v yields a value with $(1+\varepsilon)^x = (1+\varepsilon)(1+\varepsilon)^{x-1} < (1+\varepsilon)v$.

By shifting the completion times by a factor of $(1+\varepsilon)$, we obtain increased starting times S'_j for each job j:

$$S'_j = (1+\varepsilon)C_j - (p_j + \tau_1) \geq (1+\varepsilon)S_j + \varepsilon p_j + \varepsilon\tau_1 \geq \varepsilon(p_j + \tau_1).$$

Hence, by losing not more than a $(1+\varepsilon)$-factor we may assume that all jobs have release dates of at least an ε fraction of their running time.

Multiplying furthermore all start times of a schedule by $(1+\varepsilon)$ gives a feasible schedule even when rounded up all nonzero processing times and release dates to the next powers of $(1+\varepsilon)$. The total completion time does not increase by more than a factor of $(1+\varepsilon)$.

All times can be scaled if necessary such that the earliest release date is at least one (since jobs with $r_j = p_j = \tau_1 = 0$ can be ignored). □

We define $R_x = (1+\varepsilon)^x$ and consider time intervals $I_x = [R_x, R_{x+1}]$ of length εR_x.

Lemma 10. *Each job runs for at most $\sigma := \lceil \log_{1+\varepsilon} \frac{1+\varepsilon}{\varepsilon} \rceil$ intervals.*

Proof. Consider some job j and assume that j starts in I_x in some schedule. By Lemma 9 we get

$$|I_x| = \varepsilon R_x \geq \varepsilon r_j \geq \varepsilon^2(p_j + \tau_1).$$

Thus, the running time of j is bounded by $|I_x|/\varepsilon^2$. The constant upper bound of $1/\varepsilon^2$ for the number of used intervals can still be improved since the length of

the next σ succeeding intervals with increasing size is sufficient to cover a length of $|I_x|/\varepsilon^2$. Using the fact that $\sum_{k=0}^{n} z^k = \frac{1-z^{n+1}}{1-z}$ we get

$$\begin{aligned}\sum_{i=0}^{\sigma} |I_{x+i}| &= \sum_{i=0}^{\sigma} (R_{x+i+1} - R_{x+i}) = |I_x| \sum_{i=0}^{\sigma} (1+\varepsilon)^i \\ &= |I_x| \frac{1-(1+\varepsilon)^{\sigma+1}}{1-(1+\varepsilon)} \\ &\geq |I_x| \frac{1-\frac{\varepsilon+1}{\varepsilon}}{-\varepsilon} = |I_x| \frac{\varepsilon+1-\varepsilon}{\varepsilon^2} = \frac{|I_x|}{\varepsilon^2}\end{aligned}$$

□

Lemma 11. $\sum_{x<y} \varepsilon^2 |I_x| \leq \varepsilon |I_y|$

Proof. To prove the claim we again use that $\sum_{k=0}^{n} z^k = \frac{1-z^{n+1}}{1-z}$:

$$\begin{aligned}\varepsilon^3 \sum_{x<y} (1+\varepsilon)^x &= \varepsilon^3 \frac{1-(1+\varepsilon)^y}{1-(1+\varepsilon)} \\ &= \varepsilon^2((1+\varepsilon)^y - 1) \\ &\leq \varepsilon |I_y|\end{aligned}$$

□

To analyze the set of jobs released within each interval we partition them as follows. A job j available in I_x is called *small in* I_x if $p_j \leq \varepsilon^2 |I_x|$ and *large* otherwise. With this, we partition the released jobs of I_x for each direction $\mathrm{d} \in \{\mathrm{r}, \mathrm{l}\}$ into the subsets $S_x^{\mathrm{d}} = \{j \in J^{\mathrm{d}} \mid r_j = R_x \text{ and } j \text{ is small in } I_x\}$ and $L_x^{\mathrm{d}} = \{j \in J^{\mathrm{d}} \mid r_j = R_x \text{ and } j \text{ is large in } I_x\}$. Fortunately, the arrangement of jobs of each S_x^{d} does not influence the remaining jobs too much such that we can assume a fixed order for each of these sets:

Lemma 12. *With $\mathcal{O}(1+\varepsilon)$-loss we can restrict to schedules such that*

1. *the processing of no small job contains a release date,*
2. *jobs contained in the same $S_x^{\mathrm{d}}, x \geq 0$ are scheduled in SPT order, i.e., $S_{j_1} \leq S_{j_2}$ for any pair of jobs $j_1, j_2 \in S_x^{\mathrm{d}}$ with $p_{j_1} < p_{j_2}$, and*
3. *the jobs of S_x^{d} in SPT order are joined to unsplittable packages with size of at most $\varepsilon^2 |I_x|$ for all packages and at least $\varepsilon^2 |I_x|/2$ for all but the last packages.*

Proof. To prove claim 1. we consider some schedule and apply a time shift to the intervals, i.e., each R_x is multiplied by $(1+\varepsilon)$ and the start times of each interval I_x are shifted by an amount of $(1+\varepsilon)R_x$. By this, the schedule remains feasible, no further crossing of a processing over a release date is produced and the objective is increased by at most a factor of $(1+\varepsilon)$. If there was a release date R_x contained in the processing interval of a small job of I_{x-1} it is moved behind the processing since the length of I_{x-1} is increased by $\varepsilon |I_{x-1}|$ which is larger than the processing time of this job.

For a proof of claims 2. and 3. consider a schedule where no processing of a small job contains a release date. To achieve the demanded properties, apply the following procedure for each direction $\mathrm{d} \in \{\mathrm{r}, \mathrm{l}\}$. First, we apply two time stretches, i.e. each interval $|I_y|$ is increased by an amount of $(2\varepsilon + \varepsilon^2)|I_y|$. For each x consider the set S_x^{d} of small jobs released at R_x of the considered direction. Order these jobs in SPT order and iteratively join them to packages with maximum amount of processing time not greater than $\varepsilon^2|I_x|$. Then, each but the last package has in sum a processing time of at least $\varepsilon^2|I_x|/2$. Now, remove all jobs of S_x^{d} from the schedule and fill the resulting gaps again by the joined packages as follows: fill all but the last gap of an interval with the packages as long as it fits, shift all following jobs of the schedule in this interval left to eliminate eventual idle time, fill the last gap of this interval with packages until the amount of original processing time by jobs of S_x^{d} in this interval is covered (if enough jobs are still unscheduled), and shift following jobs of the schedule in this interval right if necessary. This is possible since all jobs of S_x^{d} have the same compatibilities and equal transit time. In the resulting schedule, the amount of processing time by jobs of S_x^{d} in this interval is increased by at most a value of $\varepsilon^2|I_x|$. Since this amount is not decreased as long as jobs are available the existing gaps are enough to schedule all jobs of S_x^{d}. By the described rearrangement, the sum of completion times of jobs in S_x^{d} does not increase. Note, that the movement of the other jobs within an interval does not increase their completion times by more than a factor of $(1 + \varepsilon)$. It remains to prove that the initial double time shift is sufficient to create enough room for each S_x^{d}. This follows from Lemma 11 since we got room for $\varepsilon^2|I_y|$ and $\sum_{x<y} \varepsilon^2|I_x| \leq \varepsilon|I_y|$ within each interval I_y. □

Since the order in which packages of each S_x^{d} are scheduled is now fixed we can consider each package simply as one small job. Nevertheless, the original jobs must be used for the evaluation of the completion times. Besides the scheduling restrictions for small jobs we can also bound how much is released at the beginning of each interval. To do so, we denote the sum of processing times of a subset $S \subseteq J$ of jobs as $p(S)$.

Lemma 13. *With $\mathcal{O}(1+\varepsilon)$-loss we can assume for each interval $I_x, x \geq 0$ and each $\mathrm{d} \in \{\mathrm{r}, \mathrm{l}\}$:*

1. *$p(S_x^{\mathrm{d}}) \leq (1 + \varepsilon^2)|I_x|$*
2. *the number of possible processing times in L_x^{d} is bounded by $4 \log_{(1+\varepsilon)} \frac{1}{\varepsilon}$, and*
3. *the number of jobs per processing time in L_x^{d} is bounded by $\frac{1}{\varepsilon^2}$.*

Proof. Consider some scheduling instance, some $\mathrm{d} \in \{\mathrm{r}, \mathrm{l}\}$ and some $x \geq 0$. By Lemma 12 we can assume at $\mathcal{O}(1 + \varepsilon)$-loss that the jobs of each S_x^{d} can be scheduled in SPT order. Let S' be the smallest SPT-subset, such that $p(S') \geq |I_x|$. Hence, by the SPT assumption we can be sure that all jobs of $S_x^{\mathrm{d}} \setminus S'$ cannot be scheduled within I_x and thus, we can move their release dates to R_{x+1}. We have $p(S') \leq (1 + \varepsilon^2)|I_x|$ since all jobs of S' are small in I_x.

The processing time of the jobs in L_x^{d} are, by definition, at least $\varepsilon^3(1+\varepsilon)^x$. On the other hand, by Lemma 9, the processing times are at most $\frac{1}{\varepsilon}(1+\varepsilon)^x$. Let x_j be such that $p_j = (1+\varepsilon)^{x_j}$. We get

$$\begin{aligned} & \varepsilon^3 \leq \tfrac{(1+\varepsilon)^{x_j}}{(1+\varepsilon)^x} \leq \frac{1}{\varepsilon} \\ \implies \quad & \log_{(1+\varepsilon)} \varepsilon^3 \leq x_j - x \leq \log_{(1+\varepsilon)} \frac{1}{\varepsilon} \end{aligned}$$

The difference of these bounds is $4 \log_{(1+\varepsilon)} \frac{1}{\varepsilon}$ which gives a constant number of possible integer values for x_j and, hence, a constant number of possible processing times for each job in L_x^{d}. Finally, since each large job in I_x has a processing time of at least $\varepsilon^2|I_x|$, we can schedule at most $1/\varepsilon^2$ jobs per direction within I_x, and the remaining jobs need to start after R_{x+1}. □

Lemma 14. *With $\mathcal{O}(1+\varepsilon)$-loss we can assume, that each job is finished within a constant number of intervals after its release.*

Proof. We stretch all time intervals by a factor of $(1+\varepsilon)$ to create some buffer for jobs that are scheduled too late, but keep the starting times of all jobs at the same offset relative to the start of the interval where the job started before. Consider the set of jobs J_x released at time R_x. By Lemma 9 the running time of each such job is at most R_x/ε. To first schedule all jobs of one direction and afterwards all jobs of the other direction, with an additional transit period afterwards, we need time equal to

$$\begin{aligned} \sum_{\mathrm{d}\in\{\mathrm{r,l}\}} \left[p(S_x^{\mathrm{d}}) + p(L_x^{\mathrm{d}}) + \tau_1\right] + \tau_1 &\leq 2\Bigg[(1+\varepsilon^2)\varepsilon(1+\varepsilon)^x \\ &\qquad + \frac{1}{\varepsilon^2} \cdot \frac{1}{\varepsilon}(1+\varepsilon)^x \cdot 4\log_{(1+\varepsilon)} \frac{1}{\varepsilon}\Bigg] \\ &= \varepsilon^2(1+\varepsilon)^x \cdot 2\left[\frac{(1+\varepsilon^2)}{\varepsilon} + \frac{4}{\varepsilon^5}\log_{(1+\varepsilon)} \frac{1}{\varepsilon}\right] \\ &\leq \varepsilon^2(1+\varepsilon)^x(1+\varepsilon)^{\sigma'} = \varepsilon|I_{x+\sigma'}|, \end{aligned}$$

where σ' is the smallest possible integer such that $2\left[\frac{(1+\varepsilon^2)}{\varepsilon} + \frac{4}{\varepsilon^5}\log_{(1+\varepsilon)} \frac{1}{\varepsilon}\right] \leq (1+\varepsilon)^{\sigma'}$. Note, that σ' is constant. The amount by which our time stretch increased the size of $I_{x+\sigma'}$ would already be sufficient to host all jobs in J_x.

However, some or all of the extra space in $I_{x+\sigma'}$ may potentially be occupied by another job j that previously occupied intervals $I_{x+\sigma'}$ and $I_{x+\sigma'+1}$. Let $k \leq \sigma' + \sigma$ be such that, in the stretched schedule, job j completes within interval I_{x+k}. Since j runs in interval $I_{x+\sigma'}$, and all later jobs start in later intervals, our time stretch ensures that no job is being processed for time $\varepsilon|I_{x+\sigma'}|$ after j stops being processed. By definition of σ', this time is sufficient to wait for the transit of j and then schedule all remaining jobs of J_x. This way, all jobs of J_x are scheduled before the end of the interval $I_{x+\sigma'+\sigma}$. □

We can now limit the interface of our dynamic program by showing Lemma 1 of Section 6.

Lemma 1. *There is a schedule with a sum of completion times within a factor of* $(1+\varepsilon)$ *of the optimum and with the following properties:*

1. *The number of jobs scheduled in each block is bounded by a constant.*
2. *Every two consecutive blocks respect one of constantly many frontiers.*

Proof. By Lemma 12 we may assume that small jobs in S^{d}_x have processing time at least $\varepsilon^2|I_x|/2$. By Lemma 13, the total processing time of these jobs is at most $(1+\varepsilon^2)|I_x|$, and hence the number of jobs in S^{d}_x is bounded by a constant. The same is true for large jobs, by Lemma 13. Finally, together with Lemma 14, this implies that the number of jobs running during each interval is bounded by a constant.

For the second property, we stretch all time intervals by a factor of $(1+\varepsilon)$, which gives enough room to move the starting time of all jobs to the next $1/\varepsilon$-fraction of the same interval I_x. Thus, we only need to consider s/ε possible frontier values per direction, or a total of $(s/\varepsilon)^2$ possible frontiers. □

C.2 Multiple Machines

We now consider the problem $Bm \,|\, G_i \in \{K_{n_{\mathrm{r}},n_{\mathrm{l}}}, \emptyset\} \,|\, \sum C_j$ with a constant number m of machines and give detailed proofs for the required extensions to the single machine case where the argumentation is more complex. We need to generalize or reformulate all of the above lemmas. We denote by $S^{\mathrm{d}}_{x,s,t}$ the set of all small jobs with respective source machine s and target machine t and direction d that are released at time R_x and we denote the corresponding large jobs by $L^{\mathrm{d}}_{x,s,t}$.

The proofs of Lemmas 9, 10, and 13 can be adapted with small modifications. Lemmas 12 and 14 become significantly more involved. We obtain the following lemmas.

Lemma 15. *With* $\mathcal{O}(1+\varepsilon)$*-loss we can restrict to schedules such that for each direction* $\mathrm{d} \in \{\mathrm{r},\mathrm{l}\}$*, each source and target pair* $s,t = 1,\dots,m$*, and each* $x \geq 0$*:*

- *the jobs contained in* $S^{\mathrm{d}}_{x,s,t}$ *are scheduled on each machine* $i = 1,\dots,m$ *in SPT order, i.e.,* $S_{i,j_1} \leq S_{i,j_2}$ *for any pair of jobs* $j_1, j_2 \in S^{\mathrm{d}}_{x,s,t}$ *with* $p_{j_1} < p_{j_2}$*, and*
- *the jobs of* $S^{\mathrm{d}}_{x,s,t}$ *in SPT order are joined in the complete schedule to unsplittable packages with size of at most* $\varepsilon^2|I_x|$ *for all packages and at least* $\varepsilon^2|I_x|/2$ *for all but the last packages.*

Proof. The proof works in principle as the proof of Lemma 12. During the rearrangement procedure we have to ensure that each interval on the target machine is filled with at least the same volume of small jobs as before, as long as jobs are unscheduled. For this, the jobs in the demanded order must have arrived at the respective machine in time. To ensure this property we have to deal with the following two difficulties. A convoy of very small jobs can be replaced by one

(larger) small job. This job can only continue its processing on the next machine after its completion on the previous machine, while the first very small job could already start on the next machine before the last very small job is completed on the previous machine. The other difficulty arises from the fact that some jobs scheduled within interval I_x arrive during interval $I_{x'}$ at the next machine, and the remaining jobs arrive one interval later. Since all but the last gaps within interval I_x have been decreased a bit within the rearrangement and only the last gap covers the lost volume there might be not enough volume available for the next machine in $I_{x'}$.

To deal with the second difficulty, we employ the following procedure for each direction $\mathrm{d} \in \{\mathrm{r}, \mathrm{l}\}$. First, we apply m^2 time stretches. Now, consider $S^{\mathrm{d}}_{x,s,t}$ for each source target pair $s, t = 1, \ldots, m$ compatible with d and each $x \geq 0$. If $s = t$, we can simply apply the same procedure as for Lemma 12. Otherwise, proceed as follows. Define $p(i, \tilde{x})$ to be the amount of processing time from $S^{\mathrm{d}}_{x,s,t}$ scheduled within $I_{\tilde{x}}$ on machine i. For each reasonable combination of i_1 and a succeeding i_2 and $x_1 \leq x_2$ define $p(i_1, x_1, i_2, x_2)$ to be the amount of processing time of jobs in $S^{\mathrm{d}}_{x,s,t}$ scheduled on machine i_1 in interval I_{x_1} and on machine i_2 on interval I_{x_2}. On the other hand, let $u(i_1, x_1, i_2, x_2)$ be the latest point for the end of processing of a small job within interval I_{x_1} on machine i_1, such that it still can be completely processed within interval I_{x_2} on machine i_2 if possible. Since the interval sizes are increasing with time, there is at most one interval I_{x_2} on machine i_2 that yields an upper bound below R_{x_1+1}.

Remove all jobs of $S^{\mathrm{d}}_{x,s,t}$ from the schedule. To refill the gaps consider each machine $i_1 = s, \ldots, t-1$ and on machine i_1 each interval I_{x_1} that contains gaps with increasing x_1. For each succeeding machine i_2 consider an interval I_{x_2}. We now apply the gap filling procedure from the proof of Lemma 12 but decrease and increase by at most one small job appropriately such that before each $u(i_1, x_1, i_2, x_2)$ a volume of at least $p(i_1, x_1, i_2, x_2)$ including the overage of the former intervals is ensured. More formally, let the *overage* $o(i_1, x_1, i_2, x_2)$ be defined as the difference of the scheduled volume before $u(i_1, x_1, i_2, x_2)$ plus $o(i_1, x_1 - 1, i_2, x_2)$ and the demanded $p(i_1, x_1, i_2, x_2)$ for $x_1 > x$ and zero otherwise. The refill now ensures that each $o(i_1, x_1, i_2, x_2) \geq 0$. For $i = s, \ldots, t$ we furthermore ensure the analog for $p(i, \tilde{x})$, $o(i, \tilde{x})$, and the end of $I_{\tilde{x}}$. For the moment, allow a small job to start already p_j time units earlier than the completion on the previous machine.

We now prove by induction over $i = s, \ldots, t$ that the required processing volume within each interval $I_{\tilde{x}}, \tilde{x} \geq x$ on machine i is available. To be more precise we claim: for each machine $i = s, \ldots, t$ and each $\tilde{x} \geq x$, enough jobs of $S^{\mathrm{d}}_{x,s,t}$ are available to ensure a volume of at least $p(i, \tilde{x}, i_2, x_2) - o(i, \tilde{x} - 1, i_2, x_2)$ until $u(i, \tilde{x}, i_2, x_2)$ for each succeeding machine i_2 until t and $x_2 \geq \tilde{x}$, and a volume of at least $p(i, \tilde{x}) - o(i, \tilde{x} - 1)$ within the complete interval, for as long as jobs are unscheduled. The claim is true for $i = s$ since all jobs of $S^{\mathrm{d}}_{x,s,t}$ are available by R_x. For some $i \in \{s, \ldots, t\}$ let i_1 be the corresponding preceding machine. Consider a $\tilde{x} \geq x$ with $p(i, \tilde{x}) > 0$. All the demanded volume must have been scheduled earlier on i_1. Hence, $p(i, \tilde{x}) =$

$\sum_{x_1 \leq \tilde{x}} p(i_1, x_1, i, \tilde{x})$. By the induction hypothesis, this amount on the previous machine is scheduled within each interval in time (in particular within the former gaps). If $\tilde{x}$ is the first one considered on i, enough jobs with the required processing volume are available. Otherwise, there might be one small job missing that is scheduled in an earlier interval. This is covered by the overage. Consider now a succeeding i_2 and an x_2 with $p(i, \tilde{x}, i_2, x_2) > 0$. If $u(i, \tilde{x}, i_2, x_2) = r_{\tilde{x}+1}$ the claim ensues from $p(i, \tilde{x}, i_2, x_2) \leq \sum_{x_1 \leq \tilde{x}} p(i_1, x_1, i, \tilde{x})$. Otherwise, note that $u(i, \tilde{x}, i_2, x_2) - \tau_{i_1} \in I_{x_1}$ is equal to $u(i_1, x_1, i_2, x_2)$. Then, we additionally have to use that $p(i, \tilde{x}, i_2, x_2) \leq \sum_{x_1' \leq x_1} p(i_1, x_1, i_2, x_2)$ is scheduled to be processed before $u(i_1, x_1, i_2, x_2)$. If one small missing job is scheduled again too early, we use the overage. To conclude, all jobs arrive in time on their target machine and the completion time is increased by a factor of at most $\mathcal{O}(1+\varepsilon)$.

The described rearrangement procedure for the jobs of $S^{\mathrm{d}}_{x,s,t}$ still only needs an extra time of $\varepsilon^2|I_x|$ on each machine for $x \geq 0$ and $s, t = 1, \ldots, m$ compatible with d. By Lemma 11 we again get that m^2 time stretches are sufficient to cover this amount. To see this, consider some machine i and some interval y and bound the needed amount as follows:

$$\begin{aligned}\sum_{i_1 \prec_{\mathrm{d}} i} \sum_{i \preceq_{\mathrm{d}} i_2} \sum_{x \leq y} \varepsilon^2 |I_x| &\leq \sum_{i_1 \prec_{\mathrm{d}} i} \sum_{i \preceq_{\mathrm{d}} i_2} (\varepsilon + \varepsilon^2)|I_y| \\ &\leq \sum_{i'=1}^{m} i'(\varepsilon + \varepsilon^2)|I_y| \leq m^2 \varepsilon |I_y|\end{aligned}$$

Finally, we have to resolve the first difficulty described initially. For this, we readjust those small jobs that have been scheduled a bit before the actual completion time on their previous machine. Since the respective error propagates from machine to machine we have to create an extra time window of $m\varepsilon^2|I_x|$ for each $S^{\mathrm{d}}_{x,s,t}$. We can do this by applying another $2m^3$ time stretches. □

Lemma 16. *At $\mathcal{O}(1+\varepsilon)$-loss we can assume, that each job is finished within a constant number of intervals after its release.*

Proof. We again start by applying one time stretch. This creates for each x extra space within interval $I_{x+\sigma'}$ (where σ' needs to be determined) for those jobs J_x released at time R_x at some machine that are scheduled too late and must be adapted. Unfortunately, at time $R_{x+\sigma'+1}$ there might be jobs with distinct processing times running on different machines. Hence, we cannot use one strip of extra space over all machines to reschedule all remaining jobs. For this reason, we employ the following strategy. For each direction $\mathrm{d} \in \{\mathrm{r}, \mathrm{l}\}$ iterate over the machines in the corresponding order and schedule the available jobs of J^{d}_x on each machine that have to be processed there. On the first machine, insert these jobs after the last job running at time $R_{x+\sigma'+1}$ is finished, which is within $I_{x+\sigma'+\sigma}$ at the very latest. With the created space, all inserted jobs arrive at the next machine before the end of this interval. Hence, we can schedule the open jobs on the next machine after the last job containing $R_{x+\sigma'+\sigma+1}$ in its running time

has arrived at the end of the machine, which is before the end of $I_{x+\sigma'+2\sigma}$. The extra space created by the time stretch was kept until this moment and the jobs can finish running on the machine before the end of $I_{x+\sigma'+2\sigma}$. Continuing analogously to the last machine ensures that the jobs arrive by interval $I_{x+\sigma'+m\sigma}$ at their destination.

Hence, we have to define σ' to be large enough such that extra space for both directions is available. On each machine, we have to provide space for jobs released at time R_x for the current and all earlier machines. Hence, we need the following space:

$$\sum_{\mathrm{d}\in\{\mathrm{r},\mathrm{l}\}} \sum_{s=1}^{m} \sum_{t=1}^{m} \left[p(S^{\mathrm{d}}_{x,s,t}) + p(L^{\mathrm{d}}_{x,s,t})\right] + 2\max_i \tau_i$$

$$\begin{aligned}
&\leq \quad 2m^2 \Bigg[(1+\varepsilon^2)\varepsilon(1+\varepsilon)^x \\
&\qquad\quad + \frac{1}{\varepsilon^2}\cdot\frac{1}{\varepsilon}(1+\varepsilon)^x \cdot 4\log_{(1+\varepsilon)}\frac{1}{\varepsilon}\Bigg] \\
&= \quad \varepsilon^2(1+\varepsilon)^x \cdot 2m^2\left[\frac{(1+\varepsilon^2)}{\varepsilon} + \frac{4}{\varepsilon^5}\log_{(1+\varepsilon)}\frac{1}{\varepsilon}\right] \\
&\leq \quad \varepsilon^2(1+\varepsilon)^x(1+\varepsilon)^{\sigma'} = \varepsilon|I_{x+\sigma'}|,
\end{aligned}$$

where we define σ' to be the smallest possible integer such that

$$2m^2\left[\frac{(1+\varepsilon^2)}{\varepsilon} + \frac{4}{\varepsilon^5}\log_{(1+\varepsilon)}\frac{1}{\varepsilon}\right] \leq (1+\varepsilon)^{\sigma'}.$$

This space also covers the needed space for the transit of the last fixed job running before the newly inserted jobs. □