Faster Approximation Algorithms for Scheduling with Fixed Jobs

Klaus Jansen[1] **Lars Prädel**[1] **Ulrich M. Schwarz**[1] **Ola Svensson**[2]

[1] Institut für Informatik, Christian-Albrechts-Universität zu Kiel, 24098 Kiel, Germany
Email: {kj,lap,ums}@informatik.uni-kiel.de

[2] KTH Royal Institute of Technology, SE-100 44 Stockholm, Sweden
Email: osven@kth.se

Abstract

We study the problem of scheduling jobs on identical parallel machines without preemption. In the considered setting, some of the jobs are already assigned machines and starting times, for example due to external constraints not explicitly modelled. The objective is to assign the rest of the jobs in order to minimize the makespan.

It is known that this problem cannot be approximated better than within a factor of 3/2 unless $\mathsf{P} = \mathsf{NP}$. An algorithm that achieves $3/2 + \epsilon$ for any $\epsilon > 0$ was presented by Diedrich and Jansen [DJ09], but its running time is doubly exponential in $1/\epsilon$. We present an improved algorithm with approximation ratio 3/2 and polynomial running time. We also give matching results for the related problem of scheduling with reservations. The new algorithm is both faster and conceptually simpler than the previously known algorithms.

1 Introduction

In parallel machine scheduling, an important issue is the scenario where either some jobs are already fixed in the system [SSW99a, SSW99b, DJ09] or intervals of non-availability of some machines must be taken into account [DJPT07, DJPT10, HLC05, Lee96, Leu04, LSL03, DJ09]. The first problem occurs when high-priority jobs are already scheduled in the system while the latter problem is due to regular maintenance of machines. Both models are relevant for turnaround scheduling [MMS] and distributed computing where machines are donated on a volunteer basis.

Formally, the problem can be defined as follows: an instance consists of m, the number of machines, considered part of the input and n jobs with non-negative processing times $p_1, \ldots, p_n \in \mathbb{N}$. The first k jobs are fixed via a list $(m_1, s_1), \ldots, (m_k, s_k)$ giving a machine index and starting time for the respective job. We assume that these fixed jobs do not overlap. A schedule is a non-preemptive assignment of the jobs to machines and starting times such that the first k jobs are assigned as encoded in the instance and that the jobs do not intersect.

For the problem with *fixed jobs*, the objective is to minimize the makespan of all jobs, including the fixed ones. In the setting with *non-availability*, the fixed jobs are not included when finding the makespan.

Without loss of generality, we may assume $m < n$: if $m \geq n$, there are at least $m - k \geq n - k$ machines without fixed jobs on them, which can execute the $n - k$ unfixed jobs optimally in a trivial way.

Both problems generalize the well-known problem $\mathrm{P}||C_{\max}$(scheduling jobs on parallel identical machines to minimize makespan) [HS88] and hence are strongly NP-hard.

Related work. Scheduling with fixed jobs was studied by Scharbrodt, Steger & Weisser [Sch00, SSW99a, SSW99b]. They mainly studied the problem for constant m; for this strongly NP-hard formulation (which consequently does not admit an FPTAS) they present a PTAS. They also found approximation algorithms for general m with ratios 3 [Sch00] and $2+\epsilon$ [SSW99a, SSW99b]; since the finishing time of the last fixed job is a lower bound for the optimal makespan $C^*_{\max}$, we can simply use a PTAS for the well-known problem $\mathrm{P}||C_{\max}$ [HS88] to schedule the remaining $n - k$ jobs after the fixed job which finishes last. Scharbrodt, Steger & Weisser [SSW99a, SSW99b] also proved that for scheduling with fixed jobs there is no approximation algorithm with ratio $3/2-\epsilon$, unless $\mathsf{P} = \mathsf{NP}$, for any $\epsilon \in (0, 1/2]$. In [DJ09], Diedrich and Jansen present a $3/2+\epsilon$-approximation for arbitrary $\epsilon > 0$ for both settings, however, it relies on large enumeration steps and involves up to $m^{1/\epsilon^{1/\epsilon^2}}$ calls to a subroutine to approximately solve a difficult maximization subproblem, the Multiple Subset Sum Problem (MSSP; see Section 2), with accuracy ϵ. We denote by $T_{MSSP}(n, \epsilon)$ the time complexity of this subroutine.

Results. We present improved algorithms for scheduling with fixed jobs and scheduling with non-availability. These algorithms on the one hand achieve exactly the bound of 3/2 and, on the other hand, are both faster and conceptually simpler than the previous algorithms in [DJ09]. Formally stated, our results are the following:

Theorem 1. *Scheduling with fixed jobs admits an approximation algorithm with ratio* 3/2 *and running time* $\mathcal{O}(n \log n + \log(n \max_{j=1,\ldots,n} p_j)(n + T_{MSSP}(n, 1/8)))$.

For scheduling with non-availability, the result is slightly weaker for technical reasons:

KJ and LAP supported in part by DFG project JA 612/14-1, "Entwicklung und Analyse von effizienten polynomiellen Approximationsschemata für Scheduling- und verwandte Optimierungsprobleme."
OS supported by ERC Advanced investigator grant 226203.

Theorem 2. *Scheduling with non-availability, even if at any time, there is only at most one unavailable machine, does not admit a polynomial time algorithm with a constant approximation ratio unless* $\mathsf{P} = \mathsf{NP}$.

Theorem 3. *Scheduling with non-availability, as long as a constant fraction* $\rho \geq 1/m$ *of machines is always available, admits a* $3/2$*-approximation with running time* $\mathcal{O}(n \log n + \log(n \max_{j=1,\dots,n} p_j)(n + T_{MSSP}(n, \rho/8)))$.

The remainder of the paper is structured as follows: in Sections 2–5, we describe the algorithm and prove its correctness for the case of fixed jobs, i.e. Theorem 1. In Section 6, we prove the lower bound of Theorem 2 and show the minor changes that are needed to use our algorithm for the case of non-availability.

2 Scheduling with Fixed Jobs and the Multiple Subset Sum Problem

Our approach, as well as the one presented in [DJ09], relies heavily on algorithms for the Multiple Subset Sum Problem. In its optimization variant, this problem is defined as follows:

Definition 4. *Given* n *items with sizes* $w_1, \dots, w_n$ *and* $m \leq n$ *target capacities* $C_1, \dots, C_m$*, possibly not all equal, we are asked to find a partition of the items into* $m+1$ *sets* $S_1, \dots, S_{m+1}$ *such that* $\sum_{j \in S_i} w_j \leq C_i$ *for all* $i \in \{1, \dots, m\}$ *and* $\sum_{i=1}^{m} \sum_{j \in S_i} w_j$ *is maximized. The set* S_{m+1} *collects items that remain unpacked.*

This problem in itself is strongly NP-hard, as shown by Caprara et al. [CKP00]. The problem that is more commonly considered is the Multiple Knapsack Problem, where items have profits that may be different from their size and the overall packed profit has to be maximized. Chekuri and Khanna were the first to present a PTAS for this problem [CK05]. The best currently known algorithm for both MKP and MSSP is an *efficient PTAS* due to Jansen [Jan09b] with a running time of $T_{MSSP}(n, \epsilon) = 2^{\mathcal{O}(\log(1/\epsilon) \cdot 1/\epsilon^5)} + \text{poly}(n)$ for n items and $m \leq n$ target capacities, which was subsequently improved to $T_{MSSP}(n, \epsilon) = 2^{\mathcal{O}(\log(1/\epsilon)^4 \cdot 1/\epsilon)} + \text{poly}(n)$ [Jan09a] and, if the *Modified Integer Roundup Conjecture* (MIRUP) of Scheithauer and Terno [ST95] holds, this even reduces to $T_{MSSP}(n, \epsilon) = 2^{\mathcal{O}(\log(1/\epsilon)^2 \cdot 1/\epsilon)} + \text{poly}(n)$ [Jan09a].

The connection of MSSP to our scheduling problem is the following: guessing a makespan T for the scheduling problem will induce, along with the prepositioned jobs, bins of different sizes into which the remaining jobs have to be placed, so solving MSSP exactly is equivalent to solving the decision version of the scheduling problem. Since MSSP is hard, we can only solve it approximately, but with arbitrary precision ϵ. The major problem now is that even though the total size of jobs not assigned by an MSSP algorithm (which will have to be scheduled after the guessed optimal makespan T) will only be a small fraction of the total length of all jobs, some of these rejected jobs may still be long, which results in a bad approximation ratio.

Much of the running time of the algorithm in [DJ09] is spent in solving network flow problems for a very large number of candidate solutions which have to be enumerated in order to avoid this problem by placing large jobs in advance. In contrast, our new algorithm uses a single post-processing step and does not need any enumerative steps, nor network flow solvers, beyond those in the MSSP subroutine.

The outline of our algorithm is given in Figure 1: we give a relaxed decision algorithm that generates a schedule of length at most $3/2 \cdot T$ provided there exists some schedule that packs all jobs in the interval $[0, T)$. This algorithm is combined with a binary search. The number of iterations of the binary search is polynomially bounded in the input length by the following easy insight:

Remark 5. *For the shortest possible makespan, OPT, we have*

$$\begin{aligned} \max_{j \in \{1,\dots,k\}} (s_j + p_j) &\leq OPT \\ &\leq \max_{j \in \{1,\dots,k\}} (s_j + p_j) + n \max_{j \in \{k+1,\dots,n\}} p_j \end{aligned} \quad (1)$$

i.e. the range to be searched over has length $\max_{j \in \{1,\dots,k\}}(s_j + p_j) + n \max_{j \in \{k+1,\dots,n\}} p_j - \max_{j \in \{1,\dots,k\}}(s_j + p_j) \leq n \max_{j=1,\dots,n} p_j$*, which is pseudopolynomial in the instance size.*

In particular, we have a polynomial number $\mathcal{O}(\log(n \max_{j=1,\dots,n} p_j))$ *of binary search steps.*

Proof. The lower bound follows from the fact that the "latest" fixed job counts towards the makespan; a schedule proving the upper bound is easily obtained in linear time by scheduling all unfixed jobs on a single machine. □

We note that by using Graham's List Scheduling algorithm [Gra69] after all the fixed jobs have finished, we can reduce the size of the search region to $\sum_{j \in \{k+1,\dots,n\}} p_j/m + \max_{j=1,\dots,n} p_j \leq (1 + n/m) \max_{j=1,\dots,n} p_j$, which would be preferable for a practical implementation.

Hence, we will concentrate on one iteration of the binary search in the following. In each iteration, we first apply a low-complexity check, described more closely in Section 3, that correctly rejects some infeasible guesses of T. We then pack almost all jobs into the schedule as described in Section 4. A novel postprocessing step ensures that the unpacked jobs have suitable properties to pack them later. Finally, we pack these jobs into an extra timeframe of length $T/2$ as described in Section 5.

3 Quickly discarding too-small T

For a given target makespan T we generate all intervals of availability of machines, in the following called *gaps*, within the planning horizon $[0, T)$ from the k encoded fixed jobs. Let $q(T) \in \mathbb{N}^*$ denote the number of gaps and let $G(T) := \{G_1, \dots, G_{q(T)}\}$ denote the set of gaps. For each $i \in \{1, \dots, q(T)\}$ we also use G_i to denote the size of gap G_i. Without loss of generality, we assume $G_1 \geq \dots \geq G_{q(t)}$. Note that $q(T) \leq k + m \leq 2n$ since at most k fixed jobs induce a gap "left" to them and there are at most m gaps whose "right" limit is not created by a fixed job but by the limit of the planning horizon. In total, $q(T)$ is polynomially bounded in the instance size. These gaps can easily be processed by sorting the fixed jobs on each machine by their execution times and assigning the gaps between them. This is done in time $\mathcal{O}(n \log n)$. Furthermore, we need that in each iteration of the while-loop all gaps are sorted. The gaps that are not limited by the planning horizon will not changed in the algorithm. Hence we can sort them by their sizes

1 Set $LB := \max_{j\in\{1,\dots,k\}}(s_j + p_j)$,
$UB := LB + \max_{j\in\{k+1,\dots,n\}} p_j + \frac{1}{m}\sum_{j=k+1}^{n} p_j$
2 Let σ_{best} a schedule of makespan at most UB.
3 Sort the jobs by non-increasing length.
4 Generate and sort the gaps $G(T)$ as described in Section 3.
5 **while** $UB - LB \geq 1$ **do**
6 Set $T := \lceil (UB + LB)/2 \rceil$.
7 Update the gaps and partition into large and small jobs $J_L(T) \dot\cup J_S(T)$ as described in Section 3. The large jobs and gaps are sorted by non-increasing sizes.
8 **for** $j = 1, \dots, m$ **do**
9 **if** *the jth largest job is large and bigger than the jth largest gap* **then**
10 reject T.
11 Run a 7/8-approximation for MSSP on $G(T)$ and the jobs, as described in Section 4.
12 **if** *more than $mT/8$ is unpacked* **then**
13 reject T.
14 **if** *T was rejected* **then**
15 $LB := T$
16 **else**
17 set σ_{best} to the generated packing and $UB := T$.
18 Modify the packing to include all items from $J_L(T)$ as described in Section 4.
19 Partition the remaining jobs into two groups and use Next-fit to schedule the remaining jobs in the interval $[UB, 3/2\,UB]$ as described in Section 5.

Figure 1: Outline of the approximation algorithm for scheduling with fixed jobs.

in line 4 before the while-loop is processed. The sizes of the other gaps are modified in each loop by the same amount. Hence we can sort them also before the while-loop is processed and update only the lengths of the gaps in line 7. Afterward we do a merge step of the two sets of gaps to have a sorting of all gaps by their sizes. In total we need $\mathcal{O}(n \log n)$ to compute and sort the two sets of gaps, before the while-loop in line 5 is processed. In each iteration we need time $\mathcal{O}(n)$ to update the lengths of the gaps and merge them. We define

$$J_L(T) := \{j \in \{k+1, \dots, n\} \mid p_j \in (T/2, T]\},$$
$$J_S(T) := \{j \in \{k+1, \dots, n\} \mid p_j \in (0, T/2]\}$$

to partition the set of non-fixed jobs into *large* and *small* jobs. If any unplaced job is longer than T, we can obviously immediately reject the guessed makespan of T as too small. We can partition the jobs and sort the large jobs in each iteration in linear time, by taking the sorting of all jobs before the while-loop is processed.

The procedure in step 8 of the algorithm quickly checks a necessary condition for feasible solutions: bear in mind that a large job has length $p_j > T/2$, so there are at most m large jobs, one per machine, and no gap, being of size at most T by definition, can be large enough to accommodate two large jobs at once.

Lemma 6. *If a guessed makespan T is rejected in step 10 of the algorithm, then no feasible schedule of length T exists.*

Proof. Assume j minimal such that the jth largest job is large and does not fit into the jth biggest gap. For convenience, denote the lengths of the j largest jobs $p_1 \geq \ldots \geq p_j$ and the size of the gaps $G_1 \geq \ldots \geq G_j$, adding "dummy gaps" of size 0 if needed.

By definition, we have $p_1 \geq \ldots \geq p_j > G_j$, hence a feasible schedule of length T would have to assign these j jobs to at most $j-1$ gaps $G_1, \dots, G_{j-1}$. This is a contradiction, since

$$p_1 \geq \ldots \geq p_j > T/2 \geq G_1/2 \geq \ldots \geq G_{j-1}/2\,,$$

so no two large jobs fit into one single gap. □

4 Packing almost all jobs

In this section, we will show that if $T \geq \text{OPT}$, we can generate a schedule of length T that assigns "almost all" jobs. To ensure an approximation guarantee of $3/2 \cdot T$ it is critical that all jobs are scheduled within the time window $[0, T]$. We proceed in two steps. First, we show:

Lemma 7. *If a feasible schedule of length T exists, then step 11 generates a packing such that the total length of unpacked jobs is bounded by $mT/8$.*

To do this, we create an instance of MSSP as follows: each gap G_i corresponds to a knapsack of capacity G_i and each job of length p_i, $i = k+1 \dots n$, corresponds to an item of size p_i. We run the EPTAS of [Jan09b, Jan09a] on this instance with accuracy 1/8.

Observe that if (and only if) our current guessed makespan T is at least the optimal makespan OPT, it is possible to pack all items into the gaps, so the EPTAS will leave items of total area at most $(\sum_{i=k+1}^{n} p_i)/8 \leq mT/8$ unpacked. Here, we use that mT is a natural upper bound on $\sum_{i=1}^{n} p_i$ if $T \geq \text{OPT}$.

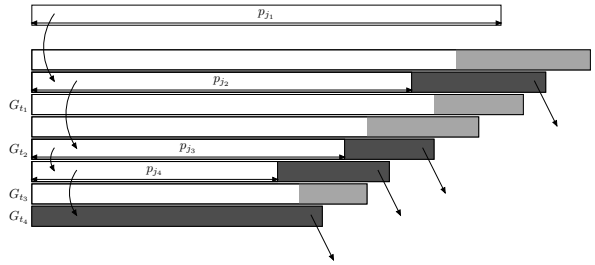

Figure 2: Choice of gaps $G_{j_1}, \ldots$ in the proof of Lemma 8. Shaded areas indicate possible small jobs; the darker areas are actually unpacked.

Hence, if more than total length $mT/8$ is not packed, we can also immediately reject our guessed T.

At this stage, the unpacked jobs may still include up to $\lfloor (mT/8)/(T/2) \rfloor = \lfloor m/4 \rfloor$ large jobs. Obviously, if large jobs, which have length $> T/2$, are not packed in the gaps in the period $[0, T)$, we cannot hope to find an overall schedule of length at most $\frac{3}{2}T$. Hence, we modify the packing to include all large jobs, at the cost of increasing the total area of unpacked jobs by a constant factor, using the following construction:

Lemma 8. *Given a packing of some jobs into the gaps such that jobs of total length δ are unpacked, amongst them a large job j_1 of length $p_{j_1} > T/2$, we can either find in polynomial time a modified packing such that the total length of unpacked jobs is at most $\delta + p_{j_1}$ and the additional large job j_1 is packed as well as all previously packed large jobs or else prove that no packing of all jobs into the gaps exists at all.*

Proof. Let t_1 be the largest index such that $G_{t_1} \geq p_{j_1}$. (Recall that $T \geq G_1 \geq \cdots \geq G_{q(T)}$.) Clearly, p_{j_1} can only possibly be scheduled in one of the gaps $G_1, \ldots, G_{t_1}$, so if each one of these already contains a job at least as large as p_{j_1}, no packing can exist at all. This condition is already tested for in step 8 of the algorithm, before the EPTAS is called. Hence, we select one gap G_{j_1} among the gaps $G_1, \ldots, G_{t_1}$ that contains a large job of minimal size. For this purpose, a gap without large job contains a 'dummy large job' of size 0. Denote this job j_2, of size $p_{j_2} < p_{j_1}$. We temporarily unpack j_2, and permanently unpack (evict) all small jobs that might have been in its gap as well, which have total length $\ell_1 \leq G_{j_1} - p_{j_2} \leq T - p_{j_2}$. (See also Figure 2 for this construction.)

If $p_{j_2} = 0$, we have now scheduled one more large job. Otherwise, we need to re-schedule j_2. As for j_1, let $t_2 \geq t_1$ be the largest index such that $G_{t_2} \geq p_{j_2}$. Furthermore, we already know that gaps $G_1, \ldots, G_{t_1}$ all carry large jobs at least as large as j_2, since j_2 was chosen to be of minimal size amongst the large jobs there. Hence, we can restrict our attention to the gaps $G_{t_1+1}, \ldots, G_{t_2}$. Again, if all these gaps already contain jobs at least as large as j_2, no feasible packing exists for this choice of T at all. Otherwise, we select a gap G_{j_2} with a large job j_3 of minimal size p_{j_3} (possibly 0) and iterate as above, discarding small jobs of total size $\ell_2 \leq G_{j_2} - p_{j_3} \leq G_{t_1+1} - p_{j_3} \leq p_{j_1} - p_{j_3}$.

After some number $r \leq m$ of iterations, we have $p_{j_{r+1}} = 0$, i.e. we did not need to unpack another large job, and the number of packed large jobs has increased by one. (Otherwise, step 8 would have rejected T already because there are more large jobs than large gaps.) Finally, the total size of small jobs that were unpacked can be bounded by

$$\begin{aligned}\sum_{i=1}^{r} \ell_i &\leq (T - p_{j_2}) + (p_{j_1} - p_{j_3}) + \ldots \\ &\quad + (p_{j_{r-1}} - p_{j_{r+1}}) \\ &= T + \sum_{i=1}^{r-1} p_{j_i} - \sum_{i=2}^{r+1} p_{j_i} \\ &= T + p_{j_1} - p_{j_r} \leq 2p_{j_1} \,.\end{aligned}$$

The final inequality holds since we know that j_r is a large job, so $T - p_{j_r} < T/2 < p_{j_1}$. Since we have now additionally packed p_{j_1}, the net loss incurred is bounded by p_{j_1}. □

With a slight modification we can schedule all large unpacked jobs of total size $mT/8$ in time $\mathcal{O}(n \log n)$. Note, that a naive approach would require n^2 steps to assign all large jobs.

Lemma 9. *Given a packing of some jobs into the gaps such that jobs of total length $mT/8$ are unpacked, we obtain a packing that includes all large jobs and has unpacked jobs with total size at most $2mT/8 = mT/4$.*

The running time of the procedure is bounded by $\mathcal{O}(n \log n)$.

Proof. We schedule the unpacked large jobs with the algorithm given in Figure 3. Here we assume that the large unpacked jobs are initially sorted by non-increasing lengths and the gaps by non-increasing sizes. Here we take use of heaps, one for the jobs, denoted by JH, and one for the gaps, denoted by GH. The heap JH is sorted by non-increasing processing lengths of the unscheduled jobs, i.e. the root of this heap is a large job of maximal processing time. The heap GH is initially empty and we add the gaps one after another. This heap is sorted by non-decreasing lengths of a large job inside the gap (note that there is at most one large job in each gap). The gap that contains a large job of minimal size, possibly a 'dummy large job' of length 0, is the root of the heap. Using heaps allows us to extract the root (and rebuild the heap) and insert an element in logarithmic time in the number of elements inside the heap.

```
1  Build the heap JH of all unpacked jobs, sorted
   by non-increasing lengths.
2  Let GH be an empty heap of the gaps.
3  Let t = 1.
4  while JH is not empty do
5      extract root j_h of JH, i.e. the unscheduled
       job of maximal processing time.
6      while G_t ≥ p_{j_h} do
7          add G_t to GH sorted by non-decreasing
           sizes of large jobs containing in the gaps.
8          t = t + 1.
9      extract root G_h of GH, i.e. the gap that
       contains a job j_{G_h} of minimal size, or one
       gap with a dummy large job.
10     if j_{G_h} has processing time larger 0 then
11         add j_{G_h} to JH.
12     unpack all jobs in G_h.
13     schedule j_h on G_h.
```

Figure 3: Outline of the algorithm for scheduling the unpacked large jobs.

For the analysis we use the same sequences of jobs as in Lemma 8. It does not matter for the analysis whether we schedule one sequence after another, or we schedule the jobs sorted by their lengths. Note, that after we scheduled a job into a gap the gap will never be considered again, since afterwards we schedule only jobs of smaller sizes. We use the analysis of Lemma 8 for all initially large unscheduled jobs of total size $mT/8$. Here consider the sequence of jobs generated by our algorithm to schedule the unpacked large jobs. Using the analysis in Lemma 8, each large job of length p_{j_1} is inserted by removing small jobs of total length $2p_{j_1}$. Therefore we have removed (or unpacked) small jobs of total size at most $2mT/8 = mT/4$ after applying the algorithm.

The running time of the algorithm given in Figure 3 is as follows. The algorithm re-schedules at most n jobs, hence the while loop in line 4 has at most $z \leq n$ iterations. In each iteration i we extract one job of the heap JH, which will not re-scheduled again. To extract and delete one job and rebuild the heap we need time at most $\mathcal{O}(\log\lfloor m/4\rfloor)$, since there are at most $\lfloor m/4\rfloor$ many jobs in the heap. We possibly add one job to the heap JH, which needs the same amount of time as extracting one.

While we are in the ith iteration of the outer loop, let v_i denote the number of iterations of the inner while loop in line 6. Since we add at most $q(T)$ gaps to the heap GH we have $\sum_{i=1}^{z} v_i \leq q(T)$. The time for extracting or adding one gap needs time at most $\mathcal{O}(\log q(T))$. In total we have in each iteration i one extraction of a large job, v_i additions of gaps, one extraction of a gap, and possibly one addition of a job. Since $m, q(T) \leq n$ we have $\sum_{i=1}^{z} 2\mathcal{O}(\log\lfloor m/4\rfloor) + (v_i + 1)\mathcal{O}(\log q(T)) \leq \sum_{i=1}^{n}(v_i + 3)\mathcal{O}(\log(n)) = 3n\mathcal{O}(\log n) + \sum_{i=1}^{n} v_i\mathcal{O}(\log(n)) = \mathcal{O}(n \log n)$. □

5 Packing remaining jobs

After the construction of the previous section, we are left with the minimal value UB such that we first have successfully packed almost all jobs, (all but total processing time $mUB/8$), which we have modified by Lemma 8 to a packing of all but total processing time $2mUB/8 = mUB/4$. Since the construction is valid for makespan $T = \mathrm{OPT}$, we know that the final $UB \leq \mathrm{OPT}$.

We will now schedule the remaining jobs in the interval $[UB, \frac{3}{2}UB]$ using a Next Fit heuristic as follows: for convenience, denote these jobs $j_1 \ldots, j_{n'}$. Partition the jobs into

$$J'_S(UB) := \{i \in \{1, \ldots, n'\} \mid p_{j_i} \in (UB/4, UB/2]\},$$
$$J''_S(UB) := \{i \in \{1, \ldots, n'\} \mid p_{j_i} \in (0, UB/4].\}$$

Then schedule each of the jobs in the set $J'_S(UB)$ on one machine. This machine will not considered again. For every remaining machine, we greedily assign jobs in $J''_S(UB)$ to it until its extra load would exceed $\frac{1}{2}UB$ or we run out of jobs. Clearly, the running time of this procedure is $\mathcal{O}(n)$.

Lemma 10. *If the total size of jobs to be scheduled in this way is at most $mUB/4$, all jobs can be assigned in the interval $[UB, \frac{3}{2}UB]$.*

Proof. Note that since $UB \geq \max_{j\in\{1,\ldots,k\}} p_j + s_j$, all machines are available during the entire interval. The algorithm assigns jobs to the machines in a greedy fashion, and once a machine is considered "full", it is closed and never reopened again, and the next machine is considered. We show that a machine is not closed unless its load larger than $UB/4$: then, assuming a job would need to be assigned to a $(m+1)$st machine, the total length of the jobs would be strictly larger than $mUB/4$.

The machines with a job in $J'_S(UB)$ have load larger than $UB/4$ since the processing time of one job in $J'_S(UB)$ has size larger than $UB/4$. Assume now that some machine m_i is closed because it cannot accommodate job j for $j \in J''_S(UB)$. This means m_i's current extra load is in the interval $(UB/2-p_j, UB/2]$. Since $p_j \leq UB/4$, our claim is true since $UB/2 - p_j \geq UB/4$. □

In total, this proves the correctness of the algorithm. As to the running time, note that for each iteration of the binary search, the running time is essentially $\mathcal{O}(n) + T_{MSSP}(n, 1/8)$. By Remark 5, the number of iterations is bounded by $\mathcal{O}(\log(n \max_{j=1,\ldots,n} p_j))$, so the overall running time is bounded by $\mathcal{O}(\log(n \max_{j=1,\ldots,n} p_j)(n + T_{MSSP}(n, 1/8)) + n \log n)$.

In total, we obtain:

Theorem 1. *Scheduling with fixed jobs admits an approximation algorithm with ratio 3/2 and running time $\mathcal{O}(n \log n + \log(n \max_{j=1,\ldots,n} p_j)(n + T_{MSSP}(n, 1/8)))$.*

6 Scheduling with Non-availability

In this section, we briefly discuss how the algorithm given above can be adapted to scheduling with *non-availability*. This setting is very closely related to scheduling with fixed jobs, the main difference is that where for fixed jobs, the makespan is given as

$$C_{\max} = \max_{j\in\{1,\ldots,n\}} s_j + p_j\,, \tag{2}$$

it is

$$C_{\max} = \max_{j\in\{k+1,\ldots,n\}} s_j + p_j \tag{3}$$

here, i.e. the "fixed jobs" are not proper jobs, but, for example, downtime needed for maintenance reasons.

This difference makes the problem slightly harder, as a reservation late in the schedule does not increase lower bounds on the optimal value. In [DJ09], it is shown that this can be exploited to prevent any constant approximation ratio (unless $\mathsf{P} = \mathsf{NP}$), as long as reservations can occur on all the machines. We can actually show a slightly stronger result: even if at any point in time, almost all machines are running, no good approximation is possible.

Theorem 2. *Scheduling with non-availability, even if at any time, there is only at most one unavailable machine, does not admit a polynomial time algorithm with a constant approximation ratio unless $\mathsf{P} = \mathsf{NP}$.*

Proof. Let $c \in \mathbb{R}$, $c \geq 1$. We aim at a contradiction and suppose that there is an approximation algorithm B with constant ratio c for scheduling with non-availability where for each time step there is only one unavailable machine. Without loss of generality, we assume that c is integer. We use a reduction from the following NP-complete problem Equal Cardinality Partition (ECP) [GJ79]. The construction is sketched in Figure 4.

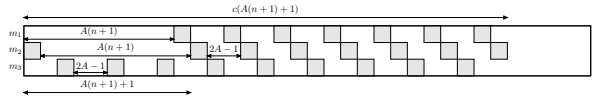

Figure 4: In the structure of intervals of non-availability of the generated instance I', for every time step there is at most one unavailable machine.

- *Given:* Finite list $I = (a_1, \ldots, a_n)$ of even cardinality with $a_i \in \mathbb{N}^*$ for each $i \in \{1, \ldots, n\}$, $A \in \mathbb{N}^*$ such that $\sum_{i=1}^{n} a_i = 2A$.
- *Question:* Is there a partition of the list I into lists I_1 and I_2 such that $|I_1| = n/2 = |I_2|$ and $\sum_{i \in I_1} a_i = A = \sum_{i \in I_2} a_i$?

Given an instance I of ECP we define an instance I' of scheduling with non-availability for arbitrary $m \geq 2$ where for each time step there is only at most one unavailable machine as follows. We may assume $m \leq A$ by suitable scaling of all a_i.

Each item a_i is copied to a job $k + i$ of length $2A + a_i$. Furthermore, the k periods of non-availability are defined as follows (also see Figure 4): for all $t = 0, \ldots, c(A(n+1)+1)$, there is a non-availability interval $[t, t+1)$ on machine m_1 iff $t \equiv 0 \pmod{2A}$ and $t > A(n+1)$, on machine m_2 iff $t \equiv 1 \pmod{2A}$ and $t > A(n+1)+1$, on machine m_i for $3 \leq i \leq m$ iff $t \equiv i - 1 \pmod{2A}$. Additionally, the interval $[0, 1)$ on machine m_2 is not available. It is easily seen that at any point, at most one machine is unavailable.

Note also that $k \leq cm(n+1)$, since there are at most m non-availabilities per time interval of length A, so k is polynomial in the size of the instance I.

By construction, on no machine, there is an available gap of length $\geq 2A$ that is at the same time entirely in the interval $[A(n+1)+1, c(A(n+1)+1))$, so no job of I' can be scheduled there. For machines $3, \ldots, m$, there even is no gap of size $\geq 2A$ in the interval $[0, c(A(n+1)+1))$. In particular, the makespan of any schedule of this instance is either at most $A(n+1)+1$ or strictly larger than $c(A(n+1)+1)$.

If I is a yes-instance to ECP, then there is a schedule in I' that has makespan (at most) $A(n+1)+1$: let $I_1 \cup I_2$ a suitable partition of I, then the corresponding partition $I'_1 \cup I'_2$ of I' satisfies that

$$\sum_{i \in I'_1} p_{i'} = \sum_{i \in I'_1} 2A + a_i = \frac{n}{2} 2A + \sum_{i \in I'_1} a_i$$
$$= (n+1)A = \sum_{i \in I'_2} p_{i'} \,,$$

so these two sets can be scheduled on m_1 and m_2 respectively in the intervals $[0, A(n+1))$ and $[1, A(n+1)+1)$. Also, our c-approximation B will deliver a schedule of length at most $c(A(n+1)+1)$.

On the other hand, if there is an optimal schedule of length at most $A(n+1)+1$, it must schedule all jobs either on machine m_1 in the interval $[0, A(n+1))$ or on m_2 in the interval $[1, A(n+1)+1)$. Since the total length of all jobs is $2(An+1)$, both intervals are filled exactly. Also, since $n/2+1$ jobs would have total length more than $(n/2+1)2A = (n+2)A$, neither interval can contain more than $n/2$ jobs, so both contain exactly $n/2$ jobs, which means they induce a solution to I. So, for no-instances to ECP, the optimal makespan is at least $c(A(n+1)+1)+1$, and algorithm B must return a solution that has at least this length.

In total, B decides in polynomial time whether I is a yes-instance or not, which is impossible unless $\mathsf{P} = \mathsf{NP}$. □

If we parametrize the problem by the fraction $\rho \in (0, 1)$ of machines that is guaranteed not to have any non-availability at all, Jansen and Diedrich [DJ09] again give a $3/2+\epsilon$ approximation (with running time doubly exponential in $1/\epsilon$ and $1/\rho$) and show that an approximation ratio of $3/2 - \epsilon$ is not possible unless $\mathsf{P} = \mathsf{NP}$.

Again, we can apply the algorithm described above for fixed jobs also to the case of unavailability. The relaxed decision procedure is virtually the same: first, "almost all" of the jobs are scheduled in the interval $[0, T)$ using the multiple subset sum problem as a subroutine. The remaining jobs are scheduled in the interval $[T, (3T/2))$, but only on the ρm machines which are not affected by reservations. To make this possible, the notion of "almost all" needs to slightly stronger, i.e. the total size of unscheduled jobs must be bounded by $\rho/8$ instead of $1/8$, which simply means we call the MSSP EPTAS subroutine with a higher accuracy $\rho/8$. After applying the exchange step of Lemma 8, unpacked non-large jobs of total area at most $\rho T/4$ remain. Then, Lemma 10 can be applied on the ρm machines which are guaranteed to be available after time T.

The only other consideration that needs to be made is the range over which the binary search is to be conducted: since the "fixed jobs" do not count towards the makespan, our bounds are different. Nevertheless, it is sufficient to use the trivial lower bound $0 \leq \mathrm{OPT}$ and the upper bound $\mathrm{OPT} \leq np_{\max}$ which is obtained by scheduling all jobs on one permanently available machines using Graham's List Schedule algorithm. Again, the range is pseudopolynomial in the instance size, so the number of binary search steps needed in the outer loop of the algorithm is polynomial in the input length. Hence, we obtain:

Theorem 3. *Scheduling with non-availability, as long as a constant fraction $\rho \geq 1/m$ of machines is always available, admits a $3/2$-approximation with running time $\mathcal{O}(n \log n + \log(n \max_{j=1,\ldots,n} p_j)(n + T_{MSSP}(n, \rho/8)))$.*

7 Conclusion

We have studied non-preemptive scheduling with fixed jobs where the objective is to minimize the makespan. For this problem, we obtain a polynomial time algorithm with ratio $3/2$, which is tight unless $\mathsf{P} = \mathsf{NP}$ holds. These techniques can also be used for the closely related setting of scheduling with non-availability; there, one needs to additionally assume that a constant percentage of the machines is permanently available.

In total, our approach yields a tight approximation result. However, our algorithm uses a very general MKP EPTAS for a fixed value of $\epsilon = 1/8$. It is an interesting open question if the more restricted problem MSSP admits a faster EPTAS or a faster combinatorial $7/8$-approximation that can be used to speed up our algorithm. So far, the best known non-PTAS result is a $3/4$-approximation due to Caprara et al. [CKP03] for the case of identical bin capacities.

References

[CK05] Chandra Chekuri and Sanjeev Khanna. A polynomial time approximation scheme for

the multiple knapsack problem. *SIAM Journal on Computing*, 35(3):713–728, 2005.

[CKP00] Alberto Caprara, Hans Kellerer, and Ulrich Pferschy. The multiple subset sum problem. *SIAM Journal on Optimization*, 11(2):308–319, 2000.

[CKP03] Alberto Caprara, Hans Kellerer, and Ulrich Pferschy. A 3/4-approximation algorithm for multiple subset sum. *J. Heuristics*, 9(2):99–111, 2003.

[DJ09] Florian Diedrich and Klaus Jansen. Improved approximation algorithms for scheduling with fixed jobs. In Claire Mathieu, editor, *SODA*, pages 675–684. SIAM, 2009.

[DJPT07] Florian Diedrich, Klaus Jansen, Fanny Pascual, and Denis Trystram. Approximation algorithms for scheduling with reservations. In Srinivas Aluru, Manish Parashar, Ramamurthy Badrinath, and Viktor K. Prasanna, editors, *HiPC*, volume 4873 of *Lecture Notes in Computer Science*, pages 297–307. Springer, 2007.

[DJPT10] Florian Diedrich, Klaus Jansen, Fanny Pascual, and Denis Trystram. Approximation algorithms for scheduling with reservations. *Algorithmica*, 58(2):391–404, 2010.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, New York, 1979.

[Gra69] Ronald L. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics*, 17(2):416–429, 1969.

[HLC05] Hark-Chin Hwang, Kangbok Lee, and Soo Y. Chang. The effect of machine availability on the worst-case performance of LPT. *Discrete Applied Mathematics*, 148(1):49–61, 2005.

[HS88] Dorit S. Hochbaum and David B. Shmoys. A polynomial approximation scheme for scheduling on uniform processors: Using the dual approximation approach. *SIAM Journal on Computing*, 17(3):539–551, 1988.

[Jan09a] Klaus Jansen. A fast approximation scheme for the multiple knapsack problem. Technical Report 0916, Institut für Informatik der Christian-Albrechts-Universität zu Kiel, 2009.

[Jan09b] Klaus Jansen. Parameterized approximation scheme for the multiple knapsack problem. *SIAM Journal on Computing*, 39(4):1392–1412, 2009.

[Lee96] Chung-Yee Lee. Machine scheduling with an availability constraint. *Journal of Global Optimization*, 9:363–384, 1996.

[Leu04] Joseph Y-T. Leung, editor. *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*. Chapman and Hall/CRC, 2004.

[LSL03] Ching-Jong Liao, Der-Lin Shyur, and Chien-Hung Lin. Makespan minimization for two parallel machines with an availability constraint. *European Journal of Operational Research*, 160:445–456, 2003.

[MMS] Nicole Megow, Rolf H. Mohring, and Jens Schulz. Decision Support and Optimization in Shutdown and Turnaround Scheduling. *Informs Journal on Computing*. to appear.

[Sch00] Mark Scharbrodt. *Produktionsplanung in der Prozessindustrie: Modelle, effiziente Algorithmen und Umsetzung*. PhD thesis, Fakultät für Informatik, Technische Universität München, 2000.

[SSW99a] Mark Scharbrodt, Angelika Steger, and Horst Weisser. Approximability of scheduling with fixed jobs. In *SODA*, pages 961–962, 1999.

[SSW99b] Mark Scharbrodt, Angelika Steger, and Horst Weisser. Approximability of scheduling with fixed jobs. *Journal of Scheduling*, 2(6):267–284, 1999.

[ST95] Guntram Scheithauer and Johannes Terno. The modified integer round-up property of the one-dimensional cutting stock problem. *European Journal of Operational Research*, 84(3):562 – 571, 1995.