Greedy Minimum-Energy Scheduling

Gunther Bidlingmaier(✉)

Department of Computer Science, Technical University of Munich, Munich, Germany
g.bidlingmaier@tum.de

Abstract. We consider the problem of energy-efficient scheduling across multiple processors with a power-down mechanism. In this setting a set of n jobs with individual release times, deadlines, and processing volumes must be scheduled across m parallel processors while minimizing the consumed energy. When idle, each processor can be turned off to save energy, while turning it on requires a fixed amount of energy. For the special case of a single processor, the greedy Left-to-Right algorithm [7] guarantees an approximation factor of 2. We generalize this simple greedy policy to the case of $m \geq 1$ processors running in parallel and show that the energy costs are still bounded by $2\,\mathrm{OPT} + P$, where OPT is the energy consumed by an optimal solution and $P < \mathrm{OPT}$ is the total processing volume. Our algorithm has a running time of $\mathcal{O}(nf \log d)$, where d is the difference between the last deadline and the earliest release time, and f is the running time of a maximum flow calculation in a network of $\mathcal{O}(n)$ nodes.

Keywords: Scheduling · Greedy Algorithms · Approximation Algorithms

1 Introduction

Energy-efficiency has become a major concern in most areas of computing for reasons that go beyond the apparent ecological ones. At the hardware level, excessive heat generation from power consumption has become one of the bottlenecks. For the billions of mobile battery-powered devices, power consumption determines the length of operation and hence their usefulness. On the level of data centers, electricity is often the largest cost factor and cooling one of the major design constraints. Algorithmic techniques for saving power in computing environments employ two fundamental mechanisms, first the option to power down idle devices, and second the option to trade performance for energy-efficiency by speed-scaling processors. In this paper we study the former, namely classical deadline based scheduling of jobs on parallel machines which can be powered down with the goal of minimizing the consumed energy.

This work was supported by the Research Training Network of the Deutsche Forschungsgemeinschaft (DFG) (378803395: ConVeY).

J. Byrka and A. Wiese (Eds.): WAOA 2023, LNCS 14297, pp. 59–73, 2023.
https://doi.org/10.1007/978-3-031-49815-2_5

In our setting, a computing device or processor has two possible states, it can be either *on* or *off*. If a processor is on, it can perform computations while consuming energy at a fixed rate. If a processor is off, the energy consumed is negligible but it cannot perform computation. Turning on a processor, i.e. transitioning it from the off-state to on-state consumes additional energy. The problem we have to solve is to schedule a number of jobs or tasks, each with its own processing volume and interval during which it has to be executed. The goal is to complete every job within its execution interval using a limited number of processors while carefully planning idle times for powering off processors such that the consumed energy is minimized. Intuitively, one aims for long but few idle intervals, so that the energy required for transitioning between the states is low, while avoiding turned on processors being idle for too long.

Previous Work. This fundamental problem in power management was first considered by [7] for a single processor. In their paper, they devise arguably the simplest algorithm one can think of which goes beyond mere feasibility. Their greedy algorithm *Left-to-Right* (LTR) is a 2-approximation and proceeds as follows. If the processor is currently busy, i.e. working on a job, then LTR greedily keeps the processor busy for as long as possible, always working on the released job with the earliest deadline. Once there are no more released jobs to be worked on, the processor becomes idle and LTR keeps the processor idle for as long as possible such that all remaining jobs can still be feasibly completed. At this point, the processor becomes busy again and LTR proceeds recursively until all jobs are completed. For a single processor, [3] develop an optimal dynamic program for unit jobs. [4] generalize this to general job weights with a running time of $\mathcal{O}(n^5)$, while [5] generalize it to multiple processors but again only unit jobs, increasing the complexity to $\mathcal{O}(n^7 m^5)$.

Obtaining good solutions for the general case of multiple processors and general job weights is difficult because of the additional constraint that every job can be worked on by at most a single processor at the same time. It is a major open problem whether the general multi-processor setting is NP-hard. It took further thirteen years for the first non-trivial result on this general setting to be developed. In their breakthrough paper, [1] develop the first constant-factor approximation for the problem. Their algorithm guarantees an approximation factor of $3 + \epsilon$ by relaxing an Integer Programming formulation of the problem. For making the rounded LP-solution feasible, they develop an additional extension algorithm *EXT-ALG*. This approximation factor is improved to $2 + \epsilon$ in [2] by incorporating into the Linear Program additional constraints for the number of processors required during every possible time interval. They also develop a combinatorial 6-approximation for the problem. As presented in the papers, all three algorithms run in pseudo-polynomial time. By using techniques presented in [1], the number of time slots which have to be considered can be reduced from d to $\mathcal{O}(n \log d)$, allowing the algorithms to run in polynomial time. More specifically, the number of constraints and variables of the Linear Programs reduces to $\mathcal{O}(n^2 \log^2 d)$. The running time of the EXT-ALG used by all three approxi-

mation algorithms is reduced to $\mathcal{O}(Fmn^3 \log^3 d)$, where F refers to a maximum flow calculation in a network with $\mathcal{O}(n \log d)$ nodes.

Contribution. In this paper we develop a greedy algorithm which is simpler and faster than the previous algorithms. The initially described greedy algorithm Left-to-Right of [7] is arguably the simplest algorithm one can think of for a single processor. We naturally extend LTR to multiple processors and show that this generalization still guarantees a solution of costs at most $2\,\mathrm{OPT} + P$, where $P < \mathrm{OPT}$ is the total processing volume. Our simple greedy algorithm *Parallel Left-to-Right* (PLTR) is the combinatorial algorithm with the best approximation guarantee and does not rely on Linear Programming and the necessary rounding procedures of [1] and [2]. It also does not require the EXT-ALG, which all previous algorithms rely on to make their infeasible solutions feasible in an additional phase.

Indeed, PLTR only relies on the original greedy policy of Left-to-Right: just keep processors in their current state (busy or idle) for as long as feasibly possible. For a single processor, LTR ensures feasibility by scheduling jobs according to the policy Earliest-Deadline-First (EDF). For checking feasibility if multiple processors are available, a maximum flow calculation is required since EDF is not sufficient anymore. Correspondingly, our generalization PLTR uses such a flow calculation for checking feasibility.

While the PLTR algorithm we describe in Sect. 2 is very simple, the structure exhibited by the resulting schedules is surprisingly rich. This structure consists of *critical sets of time slots* during which PLTR only schedules the minimum amount of volume which is feasibly possible. In Sect. 3 we show that whenever PLTR requires an additional processor to become busy at some time slot t, there must exist a critical set of time slots containing t. This in turn gives a lower bound for the number of busy processors required by any solution.

Devising an approximation guarantee from this structure is however highly non-trivial and much more involved than the approximation proof of the single-processor LTR algorithm, because one has to deal with sets of time slots and not just intervals. Our main contribution in terms of techniques is a complex procedure which (for the sake of the analysis only) carefully realigns the jobs scheduled in between critical sets of time slots such that it is sufficient to consider intervals as in the single processor case, see Sect. 4 for details. Here, we also show that our greedy policy leads to a much faster algorithm than the previous ones, namely to a running time $\mathcal{O}(nf \log d)$, where d is the maximal deadline and f is the running time for checking feasibility by finding a maximum flow in a network with $\mathcal{O}(n)$ nodes.

Formal Problem Statement. Formally, a problem instance consists of a set J of jobs with an integer release time r_j, deadline d_j, and processing volume p_j for every job $j \in J$. Each job $j \in J$ has to be scheduled across $m \geq 1$ processors for p_j units of time in the execution interval $E_j := [r_j, d_j]$ between its release time and its deadline. Preemption of jobs and migration between processors is allowed

at discrete times and occurs without delay, but no more than one processor may process any given job at the same time. Without loss of generality, we assume the earliest release time to be 0 and denote the last deadline by d. The set of discrete time slots is denoted by $T{:=}\{0,\dots,d\}$. The total amount of processing volume is $P{:=}\sum_{j\in J} p_j$.

Every processor is either completely off or completely on in every discrete time slot $t \in T$. A processor can only work on some job in the time slot t if it is in the on-state. A processor can be turned on and off at discrete times without delay. All processors start in the off-state. The objective now is to find a feasible schedule which minimizes the expended energy E, which is defined as follows. Each processor consumes 1 unit of energy for every time slot it is in the on-state and 0 units of energy if it is in the off-state. Turning a processor on consumes a constant amount of energy $q \geq 0$, which is fixed by the problem instance. In Graham's notation [6], this setting can be denoted with $m \mid r_j; \overline{d_j}; \text{pmtn} \mid E$.

Busy and Idle Intervals. We say a processor is *busy* at time $t \in T$ if some job is scheduled for this processor at time t. Otherwise, the processor is *idle*. Clearly a processor cannot be busy and off at the same time. An interval $I \subseteq T$ is a (full) *busy interval* for processor $k \in [m]$ if I is inclusion maximal on the condition that processor k is busy in every $t \in I$. Correspondingly, an interval $I \subseteq T$ is a *partial busy interval* for processor k if I is not inclusion maximal on the condition that processor k is busy in very $t \in I$. We define (partial and full) *idle intervals*, *on intervals*, and *off intervals* of a processor analogously via inclusion maximality. Observe that if a processor is idle for more than q units of time, it is worth turning the processor off during the corresponding idle interval. Our algorithm will specify for each processor when it is busy and when it is idle. Each processor is then defined to be in the off-state during idle intervals of length greater than q and otherwise in the on-state. Accordingly, we can express the costs of a schedule S in terms of busy and idle intervals.

For a multi-processor schedule S, let S^k denote the schedule of processor k. Furthermore, for fixed k, let $\mathcal{N}, \mathcal{F}, \mathcal{B}, \mathcal{I}$ be the set of on, off, busy, and idle intervals of S^k. We partition the costs of processor k into the costs $\mathsf{on}(S^k)$ for residing in the on-state and the costs $\mathsf{off}(S^k)$ for transitioning between the off-state and the on-state, hence $\mathsf{costs}(S^k) = \mathsf{on}(S^k) + \mathsf{off}(S^k) = \sum_{N\in\mathcal{N}} q + |N|$. Equivalently, we partition the costs of processor k into the costs $\mathsf{idle}(S^k){:=}\sum_{I\in\mathcal{I}} \min\{|I|, q\}$ for being idle and the costs $\mathsf{busy}(S^k){:=}\sum_{B\in\mathcal{B}} |B|$ for being busy. The total costs of a schedule S are the total costs across all processors, i.e. $\mathsf{costs}(S) = \sum_{k=1}^{m} \mathsf{costs}(S^k)$. Clearly we have $\sum_{k=1}^{m} \mathsf{busy}(k) = P$, this means for an approximation guarantee the critical part is bounding the idle costs.

Lower and Upper Bounds for the Number of Busy Processors. We specify a generalization of our problem which we call *deadline-scheduling-with-processor-bounds*. Where in the original problem, for each time slot t, between 0 and m processors were allowed to be working on jobs, i.e. being busy, we now specify a lower bound $l_t \geq 0$ and an upper bound $m_t \leq m$. For a feasible solution

to *deadline-scheduling-with-processor-bounds*, we require that in every time slot t, the number of busy processors, which we denote with $\mathsf{vol}(t)$, lies within the lower and upper bounds, i.e. $l_t \leq \mathsf{vol}(t) \leq m_t$. This will allow us to express the PLTR greedy policy of keeping processors idle or busy, respectively. Note that this generalizes the problem *deadline-scheduling-on-intervals* introduced by [1] by additionally introducing lower bounds.

Properties of an Optimal Schedule

Definition 1. *Given some arbitrary but fixed order on the number of processors, a schedule S fulfills the* stair-property *if it uses the lower numbered processors first, i.e. for every $t \in T$, if processor $k \in [m]$ is busy at t, then every processor $k' \leq k$ is busy at t. This symmetrically implies that if processor $k \in [m]$ is idle at t, then every processor $k' \geq k$ is idle at t.*

Lemma 1. *For every problem instance we can assume the existence of an optimal schedule S_{opt} which fulfills the stair-property.*

2 Algorithm

The *Parallel Left-to-Right* (PLTR) algorithm shown in Algorithm 1 iterates through the processors in some arbitrary but fixed order and keeps the current processor idle for as long as possible such that the scheduling instance remains feasible. Once the current processor cannot be kept idle for any longer, it becomes busy and PLTR keeps it and all lower-numbered processors busy for as long as possible while again maintaining feasibility. The algorithm enforces these restrictions on the busy processors by iteratively making the lower and upper bounds l_t, m_t of the corresponding instance of *deadline-scheduling-with-processor-bounds* more restrictive. Visually, when considering the time slots on an axis from left to right and when stacking the schedules of the individual processors on top of each other, this generalization of the single processor *Left-to-Right* algorithm hence proceeds *Top-Left-to-Bottom-Right*.

Once PLTR returns with the corresponding tight upper and lower bounds m_t, l_t, an actual schedule S_{pltr} can easily be constructed by running the flow-calculation used for the feasibility check depicted in Fig. 1 or just taking the result of the last flow-calculation performed during PLTR. The mapping from this flow to an actual assignment of jobs to processors and time slots can then be defined as described in Lemma 2, which also ensures that the resulting schedule fulfills the stair-property from Definition 1, i.e. that it always uses the lower-numbered processors first.

As stated in Lemma 2, the check for feasibility in subroutines `keepidle` and `keepbusy` can be performed by calculating a maximum α-ω flow in the flow network given in Fig. 1 with a node u_j for every job $j \in J$ and a node v_t for every time slot $t \in T$ including the corresponding incoming and outgoing edges.

Lemma 2. *There exists a feasible solution to an instance of deadline-scheduling-with-processor-bounds l_t, m_t if and only if the maximum α-ω flow in the corresponding flow network depicted in Fig. 1 has value P.*

Algorithm 1. Parallel Left-to-Right

$m_t \leftarrow m$ for all $t \in T$
$l_t \leftarrow 0$ for all $t \in T$
for $k \leftarrow m$ to 1 **do**
  $t \leftarrow 0$
  while $t \leq d$ **do**
    $t \leftarrow$ KEEPIDLE(k, t)
    $t \leftarrow$ KEEPBUSY(k, t)

function KEEPIDLE(k, t)
  find maximal $t' > t$ s.t. $\exists$ feasible schedule with $m_{t''} = k - 1$ for all $t'' \in [t, t')$
  $m_{t''} \leftarrow k - 1$ for all $t'' \in [t, t')$
  return t'
function KEEPBUSY(k, t)
  find maximal $t' > t$ s.t. $\exists$ feasible schedule with $l_{t''} = \max\{k, l_{t''}\}$ for all $t'' \in [t, t')$
  $l_{t''} \leftarrow \max\{k, l_{t''}\}$ for all $t'' \in [t, t')$
  return t'

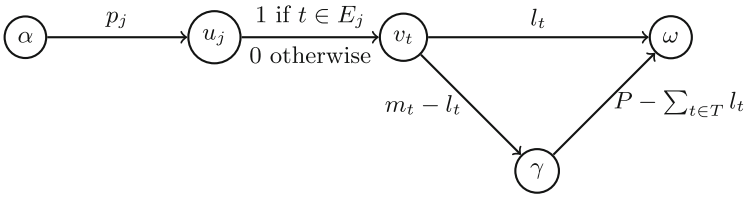

Fig. 1. The Flow-Network for checking feasibility of an instance of *deadline-scheduling-with-processor-bounds* l_t and m_t for the number of busy processors at $t \in T$. There are nodes u_j, v_t with the corresponding edges for every job $j \in J$ and for every time slot $t \in T$, respectively.

Theorem 1. *Given a feasible problem instance, algorithm* PLTR *constructs a feasible schedule.*

Proof. By definition of subroutines `keepidle` and `keepbusy`, PLTR only modifies the upper and lower bounds m_t, l_t for the number of busy processors such that the resulting instance of *deadline-scheduling-with-processor-bounds* remains feasible. The correctness of the algorithm then follows from the correctness of the flow-calculation for checking feasibility, which is implied by Lemma 2.

3 Structure of the PLTR-Schedule

3.1 Types of Volume

Definition 2. *For a schedule S, a job $j \in J$, and a set $Q \subseteq T$ of time slots, we define*

1. *the* volume $\mathsf{vol}_S(j, Q)$ *as the number of time slots of Q for which j is scheduled by S,*

2. *the* forced volume $\mathsf{fv}(j,Q)$ *as the minimum number of time slots of* Q *for which* j *has to be scheduled in every feasible schedule, i.e.* $\mathsf{fv}(j,Q) := \max\{0; p_j - |E_j \setminus Q|\}$,
3. *the* unnecessary volume $\mathsf{uv}_S(j,Q)$ *as the amount of volume which does not have to scheduled during* Q*, i.e.* $\mathsf{uv}_S(j,Q) := \mathsf{vol}_S(j,Q) - \mathsf{fv}(j,Q)$,
4. *the* possible volume $\mathsf{pv}(j,Q)$ *as the maximum amount of volume which* j *can be feasibly scheduled in* Q*, i.e.* $\mathsf{pv}(j,Q) := \min\{p_j, |E_j \cap Q|\}$.

Since the corresponding schedule S will always be clear from context, we omit the subscript for vol and uv. We extend our volume definitions to sets $J' \subseteq J$ of jobs by summing over all $j \in J'$, i.e. $\mathsf{vol}(J',Q) := \sum_{j \in J'} \mathsf{vol}(j,Q)$. If the first parameter is omitted, we refer to the whole set J, i.e. $\mathsf{vol}(Q) := \mathsf{vol}(J,Q)$. For single time slots, we omit set notation, i.e. $\mathsf{vol}(t) := \mathsf{vol}(J,\{t\})$. Clearly we have for every feasible schedule, every $Q \subseteq T, j \in J$ that $\mathsf{fv}(j,Q) \leq \mathsf{vol}(j,Q) \leq \mathsf{pv}(j,Q)$. The following definitions are closely related to these types of volume.

Definition 3. *Let* $Q \subseteq T$ *be a set of time slots. We define*

1. *the* density $\phi(Q) := \mathsf{fv}(J,Q)/|Q|$ *as the average amount of processing volume which has to be completed in every slot of* Q,
2. *the* peak density $\hat{\phi}(Q) := \max_{Q' \subseteq Q} \phi(Q')$,
3. *the* deficiency $\mathsf{def}(Q) := \mathsf{fv}(Q) - \sum_{t \in Q} m_t$ *as the difference between the amount of volume which has to be completed in* Q *and the processing capacity available in* Q,
4. *the* excess $\mathsf{exc}(Q) := \sum_{t \in Q} l_t - \mathsf{pv}(Q)$ *as the difference between the processor utilization required in* Q *and the amount of work available in* Q.

If $\hat{\phi}(Q) > k-1$, then clearly at least k processors are required in some time slot $t \in Q$ for every feasible schedule. If $\mathsf{def}(Q) > 0$ or $\mathsf{exc}(Q) > 0$ for some $Q \subseteq T$, then the problem instance is clearly infeasible.

3.2 Critical Sets of Time Slots

The following Lemma 5 provides the crucial structure required for the proof of the approximation guarantee. Intuitively, it states that whenever PLTR requires processor k to become busy at some time slot t, there must be some critical set $Q \subseteq T$ of time slots during which the volume scheduled by PLTR is minimal. This in turn implies that processor k needs to be busy at some point during Q in every feasible schedule. The auxiliary Lemmas 3 and 4 provide a necessary and more importantly also sufficient condition for the feasibility of an instance of *deadline-scheduling-with-processor-bounds* based on the excess $\mathsf{exc}(Q)$ and the deficiency $\mathsf{def}(Q)$ of sets $Q \subseteq T$. Lemmas 3 and 4 are again a generalization of the corresponding feasibility characterization in [1] for their problem deadline-scheduling-on-intervals, which only defines upper bounds.

Lemma 3. *For every* α-ω *cut* $(S, \bar{S})$ *in the network given in Fig. 1 we have at least one of the following two lower bounds for the capacity* $c(S)$ *of the cut:* $c(S) \geq P - \mathsf{def}(Q(S))$ *or* $c(S) \geq P - \mathsf{exc}(Q(\bar{S}))$*, where* $Q(S) := \{t \mid v_t \in S\}$.

Lemma 4. *An instance of deadline-scheduling-with-processor-bounds is feasible if and only if* $\mathsf{def}(Q) \leq 0$ *and* $\mathsf{exc}(Q) \leq 0$ *for every* $Q \subseteq T$.

Definition 4. *A time slot* $t \in T$ *is called an* engagement of processor k *if* $t = \min B$ *for some busy interval* B *on processor* k. *We say processor* k *is* engaged *at time* t *if* t *is an engagement of processor* k. *A time slot* $t \in T$ *is just called an* engagement *if it is an engagement of processor* k *for some* $k \in [m]$.

Lemma 5. *Let* $Q \subseteq T$ *be a set of time slots and* $t \in T$ *an engagement of processor* $k \in [m]$. *We call* Q *a* tight set for engagement t of processor k *if* $t \in Q$ *and*

$$\begin{aligned} &\mathsf{fv}(Q) = \mathsf{vol}(Q), \\ &\mathsf{vol}(t') \geq k - 1 && \text{for all } t' \in Q \text{ , and} \\ &\mathsf{vol}(t') \geq k && \text{for all } t' \in Q \text{ with } t' \geq t. \end{aligned}$$

For every engagement t *of some processor* $k \in [m]$ *in the schedule* S_{pltr} *constructed by* PLTR, *there exists a tight set* $Q_t \subseteq T$ *for engagement* t *of processor* k.

Proof. Suppose for contradiction that there is some engagement $t \in T$ of processor $k \in [m]$ and no such Q exists for t, i.e. every $Q \subseteq T$ containing t violates at least one of the three conditions in the Lemma. We show that PLTR would have extended the idle interval on processor k which ends at t. Consider the step in PLTR when t was the result of `keepidle` on processor k. Let $l_{t'}$, $m_{t'}$ be the lower and upper bounds for $t' \in T$ right after the calculation of t and the corresponding update of the bounds by `keepidle`. We modify the bounds by decreasing m_t by 1. Note that at this point $m_{t'} \geq k$ for every $t' > t$ and $m_{t'} \geq k - 1$ for every t'.

Consider $Q \subseteq T$ such that $t \in Q$ and $\mathsf{fv}(Q) < \mathsf{vol}(Q)$. Before our decrement of m_t we had $m_Q := \sum_{t' \in Q} m_{t'} \geq \mathsf{vol}(Q) > \mathsf{fv}(Q)$. The inequality $m_Q \geq \mathsf{vol}(Q)$ here follows since the upper bounds $m_{t'}$ are monotonically decreasing during PLTR. Since our modification decreases m_Q by at most 1, we hence still have $m_Q \geq \mathsf{fv}(Q)$ after the decrement of m_t. Consider $Q \subseteq T$ such that $t \in Q$ and $\mathsf{vol}(t') < k - 1$ for some t'. At the step in PLTR considered by us, i.e. when `keepidle` returned t on processor k, we hence have $m_{t'} \geq k - 1 > \mathsf{vol}(t')$. Before our decrement of m_t we therefore have $m_Q > \mathsf{vol}(Q) \geq \mathsf{fv}(Q)$, which implies $m_Q \geq \mathsf{fv}(Q)$ after the decrement. Finally, consider $Q \subseteq T$ such that $t \in Q$ and $\mathsf{vol}(t') < k$ for some $t' > t$. At the step in PLTR considered by us, we again have $m_{t'} \geq k > \mathsf{vol}(t')$, which implies $m_Q \geq \mathsf{fv}(Q)$ after our decrement of m_t. In summary, if for t no Q exists as characterized in the lemma, the engagement of processor k at t could not have been the result of `keepidle` on processor k.

Lemma 6. *We call a set* $C_k \subseteq T$ critical set for processor k *if* C_k *fulfills that*

- $C_k \supseteq C_{k'}$ *for every critical set for processor* $k' > k$,
- $t \in C_k$ *for every engagement* t *of processor* k,

- $\mathsf{fv}(C_k) = \mathsf{vol}(C_k)$,
- $\mathsf{vol}(t) \geq k-1$ *for every* $t \in C_k$, *and*
- $\phi(C_k)$ *is maximal.*

For every processor $k \in [m]$ *of* S_{pltr} *which is not completely idle, there exists a critical set* C_k *for processor* k.

Proof. We show the existence by induction over the processors $m, \ldots, 1$. For processor m, consider the union of all tight sets over engagements of processor m. This set fulfills all conditions necessary except for the maximality in regard to ϕ. Suppose that the critical sets $C_m, \ldots, C_{k+1}$ exist. Take $Q_k \subseteq T$ as the union of C_{k+1} and all tight sets over engagements of processor k. By definition of C_{k+1}, we have $Q_k \supseteq C_{k'}$ for all $k' > k$. By construction of Q_k, every engagement t of processor k is contained in Q_k. Finally, we have $\mathsf{fv}(Q_k) = \mathsf{vol}(Q_k)$ and $\mathsf{vol}(t) \geq k-1$ for every $t \in Q_k$ since all sets in the union fulfill these properties.

3.3 Definitions Based on Critical Sets

Definition 5. *For the critical set* C_k *of some processor* $k \in [m]$, *we define* $\mathsf{crit}(C_k) := k$. *Let* $\succeq$ *be the total order on the set of critical sets* C *across all processors which corresponds to* crit, *i.e.* $C \succeq C'$ *if and only if* $\mathsf{crit}(C) \geq \mathsf{crit}(C')$. *Equality in regard to* $\succeq$ *is denoted with* $\sim$. *We extend the definition of* crit *to general time slots* $t \in T$ *with* $\mathsf{crit}(t) := \max\{\mathsf{crit}(C) \mid C \text{ is critical set}, t \in C\}$ *if* $t \in C$ *for some critical set* C *and otherwise* $\mathsf{crit}(t) := 0$. *We further extend* crit *to intervals* $D \subseteq T$ *with* $\mathsf{crit}(D) := \max\{\mathsf{crit}(t) \mid t \in D\}$

Definition 6. *A nonempty interval* $V \subseteq T$ *is a* valley *if* V *is inclusion maximal on the condition that* $C \sim V$ *for some fixed critical set* C. *Let* $D_1, \ldots, D_l$ *be the maximal intervals contained in a critical set* C. *A nonempty interval* V *is a* valley *of* C *if* V *is exactly the valley between* D_a *and* D_{a+1} *for some* $a < l$, *i.e.* $V = [\max D_a + 1, \min D_{a+1} - 1]$. *By the choice of* C *as a critical set (property 1), a valley of* C *is indeed a valley. We define the jobs* $J(V) \subseteq J$ *for a valley* V *as all jobs which are scheduled by* S_{pltr} *in every* $t \in V$.

Definition 7. *For a critical set* C, *an interval* $D \subseteq T$ is a section of C *if* $D \cap C$ *contains only full subintervals of* C *and at least one subinterval of* C. *For a critical set* C *and a section* D *of* C, *the* left valley V_l *is the valley of* C *ending at* $\min(C \cap D) - 1$, *if such a valley of* C *exists. Symmetrically, the* right valley V_r *is the valley of* C *starting at* $\max(C \cap D) + 1$, *if such a valley of* C *exists.*

Lemma 7. *For every critical set* C, *every section* $D \subseteq T$ *of* C, *we have: if* $\phi(C \cap D) \leq \mathsf{crit}(C) - \delta$ *for some* $\delta \in \mathbb{N}$, *then the left valley* V_l *or the right valley* V_r *of* C *and* D *is defined and* $|J(V_l)| + |J(V_r)| \geq \delta$. *We take* $|J(V)| := 0$ *if* V *is not defined.*

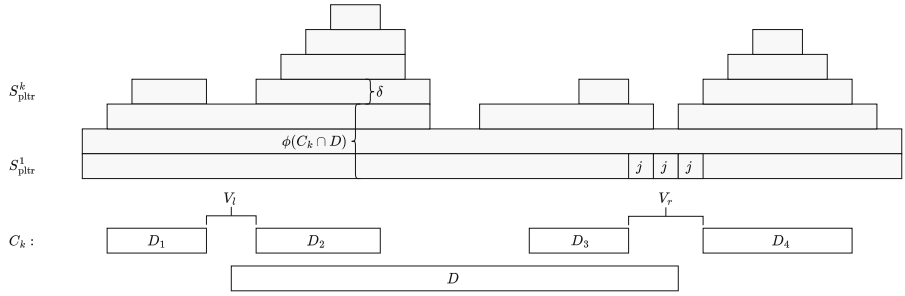

Fig. 2. The left and right valley V_l and V_r of the critical set C_k for processor k and a section D of C_k. Lemma 7 guarantees that δ jobs are scheduled at every slot of V_l or V_r.

Proof. Refer to Fig. 2 for a visual sketch of the lemma. By the choice of C as a critical set with $c{:=}\,\mathsf{crit}(C)$, we have $\mathsf{vol}(C \cap D) \geq (c-1) \cdot |C \cap D|$. If this inequality is fulfilled strictly, then with the premise $\mathsf{fv}(C \cap D)/|C \cap D| \leq c - \delta$ we directly get $\mathsf{uv}(C \cap D)/|C \cap D| > \delta - 1$. This implies that there are at least δ jobs j scheduled in $C \cap D$ with $\mathsf{uv}(j, C \cap D) > 0$. Such jobs can be scheduled in the part of C not contained in D, i.e. we must have $E_j \cap (C \setminus D) \neq \emptyset$ and hence the left valley V_l or the right valley V_r of C and D must be defined. Since these jobs j are scheduled in C only for the minimum amount possible, i.e. $\mathsf{vol}(j, C) = \mathsf{fv}(j, C) > 0$, they must be scheduled in every $t \in E_j \setminus C$ and are therefore contained in $J(V_l)$ or $J(V_r)$.

If on the other hand we have equality, i.e. $\mathsf{vol}(C \cap D) = (c-1) \cdot |C \cap D|$, then let t be an engagement of processor c. Since $\mathsf{vol}(t) > c - 1$, we must have $t \notin C \cap D$. By the same argument as before, we have that if $\mathsf{fv}(C \cap D)/|C \cap D| \leq c - \delta$, then $\mathsf{uv}(C \cap D)/|C \cap D| \geq \delta - 1$. Let $J'{:=}\{j \in J \mid \mathsf{uv}(j, C \cap D) > 0\}$. Since $\mathsf{uv}(j, C \cap D) \leq |C \cap D|$ for every $j \in J$, we have $|J'| \geq \delta - 1$. If this lower bound is fulfilled with equality, then every $j \in J'$ must be scheduled in every time slot of $C \cap D$ and hence $\mathsf{fv}(J', C \setminus D) = \mathsf{vol}(J', C \setminus D)$. Now suppose for contradiction that all jobs j scheduled during $C \setminus D$ which are not contained in J' have $E_j \cap C \cap D = \emptyset$. Then $\mathsf{fv}(C \setminus D) = \mathsf{vol}(C \setminus D)$ and we get $\phi(C \setminus D) > \phi(C)$ since by case assumption $\mathsf{vol}(C \cap D)/|C \cap D| = (c-1) < \phi(C)$. With $\mathsf{vol}(t) \leq c - 1$ for every $t \in C \cap D$, we know that $\mathsf{crit}(C \cap D) \leq c$ and therefore $C \setminus D$ is still a critical set for processor c but has higher density than C, contradicting the choice of C. Therefore, there must exist a job $j \notin J'$ scheduled in $C \setminus D$ with an execution interval intersecting $C \cap D$. In any case, we have at least δ jobs scheduled in C with an execution interval intersecting both $C \setminus D$ and $C \cap D$. This implies that the left valley V_l or the right valley V_r of C and D exists and that at least δ jobs are contained in $J(V_l)$ or $J(V_r)$.

4 Modification of the PLTR-Schedule for Analysis

In this section we modify the schedule S_{pltr} returned by PLTR in two steps. We stress that this is for the analysis only and not part of PLTR. The first step augments specific processors with auxiliary busy slots such that in every critical set C at least the first $\mathsf{crit}(C)$ processors are busy all the time. For the single processor LTR algorithm, the crucial property for the approximation guarantee is that every idle interval of S_{opt} can intersect at most 2 distinct idle intervals of the schedule returned by LTR. The second modification step of S_{pltr} is more involved and establishes this crucial property on every processor $k \in [m]$ by making use of Lemma 7. More specifically, it will establish the stronger property that $\hat{\phi}(B) > k-1$ for every busy interval B on processor k with $\mathsf{crit}(B) \geq 2$, i.e. that every feasible schedule requires k busy processors at some point during B. Idle intervals surrounded by only busy intervals B with $\mathsf{crit}(B) \leq 1$ are then handled in Lemma 12 with essentially the same argument as for the single processor LTR algorithm. By making sure that the modifications cannot decrease the costs of our schedule, we obtain an upper bound for the costs of S_{pltr}.

4.1 Augmentation and Realignment

We transform S_{pltr} into the *augmented schedule* S_{aug} by adding for every t with $k{:=}\,\mathsf{crit}(t) \geq 2$ and $\mathsf{vol}(t) = k-1$ an auxiliary busy slot on processor k. No job is scheduled in this auxiliary busy slot on processor k and it does also not count towards the volume of this slot. It merely forces processor k to be in the on-state at time k while allowing us to keep thinking in terms of idle and busy intervals in our analysis of the costs.

Lemma 8. *In S_{aug} processors $1, \ldots, \mathsf{crit}(t)$ are busy in every slot $t \in T$ with $\mathsf{crit}(t) \geq 2$.*

Proof. The property directly follows from our choice of the critical sets, the definition of $\mathsf{crit}(t)$, and the construction of S_{aug}.

As a next step, we transform S_{aug} into the *realigned schedule* S_{real} using Algorithm 2. We briefly sketch the ideas behind this realignment. Lemma 8 guarantees us that every busy interval B on processor k is a section of the critical set C with $C \sim B$. It also guarantees that the left and right valley V_l, V_r of C and B do not end within an idle interval on processor k. Lemma 7 in turn implies that if the density of B is too small to guarantee that S_{opt} has to use processor k during B, i.e. if $\hat{\phi}(B) \leq k-1$, then V_l or V_r is defined and there is some j scheduled in every slot of V_l or V_r. Let V be the corresponding left or right valley of C and D for which such a job j exists. Instead of scheduling j on the processors below k, we can schedule j on processor k in idle time slots during V. This merges the busy interval B with at least one neighbouring busy interval on processor k. In the definition of the realignment, we will call this process of filling the idle slots during V on processor k the *closing of valley V on processor k*. The corresponding subroutine is called $\texttt{close}(k, V)$.

The crucial part is ensuring that this merging of busy intervals by closing a valley continues to be possible throughout the realignment whenever we encounter a busy interval with a density too small. For this purpose, we go through the busy intervals on each processor in decreasing order of their criticality, i.e. in the order of $\succeq$. We also allow every busy slot to be used twice for the realignment (see variable $\sup_V$ in Algorithm 2) by introducing further auxiliary busy slots, since for a section D of the critical set C, both the right and the left valley might be closed on processor k in the worst case. This allows us to maintain the invariants stated in Lemma 9 during the realignment process, which correspond to the initial properties of Lemmas 7 and 8 for S_{aug}.

4.2 Invariants for Realignment

Lemma 9. *For an arbitrary step during the realignment of S_{aug} and a valley $V \subseteq T$, let the* critical processor k_V for V *be the highest processor such that*

- *processor k_V is not fully filled yet, i.e. $\texttt{fill}(k_V, T)$ has not yet returned,*
- *no $V' \supseteq V$ has been closed on k_V so far, and*
- *there is a (full) busy interval $B \subseteq V$ on processor k_V.*

We take $k_V{:=}0$ if no such processor exists. At every step in the realignment of S_{aug} the following invariants hold for every valley V, where C denotes the critical set with $C \sim V$.

1. *If $\phi(C \cap D) \leq k_V - \delta$ for some $\delta \in \mathbb{N}$, some section $D \subseteq V$ of C, then the left valley V_l or the right valley V_r of C, D exists and $\sup_{V_l} + \sup_{V_r} \geq 2\delta$.*
2. *For every $t \in C \cap V$, processors $1, \ldots, k_V$ are busy at t.*
3. *Every busy interval $B \subseteq V$ on processor k_V with $B \sim V$ is a section of C.*

Lemma 10. *The resulting schedule S_{real} of the realignment of S_{aug} is defined.*

Lemma 11. *For every processor $k \in [m]$ and every busy interval B on processor k in S_{real} with $\mathsf{crit}(B) \geq 2$, we have $\hat{\phi}(B) > k - 1$.*

Proof. We show that $\texttt{fill}(k, T)$ establishes the property on processor k. The claim then follows since $\texttt{fill}(k, T)$ does not change the schedules of processors above k. We know that on processor k busy intervals are only extended, since in $\texttt{fill}(k, T)$ we only close valleys for busy intervals B on k which are a section of the corresponding critical set C. Let $B \subseteq V$ be a busy interval on processor k in S_{real} with $B \sim V$ and $\mathsf{crit}(B) \geq 2$. No valley $W \supseteq V$ can have been closed on k since otherwise there would be no $B \subseteq V$ in S_{real}. Therefore, at some point $\texttt{fill}(k, V)$ must be called. Consider the point in $\texttt{fill}(k, V)$ when the while-loop terminates. Clearly at this point all busy intervals $B' \subseteq V$ with $B' \sim V$ on processor k have $\hat{\phi}(B') > k-1$. At this point there must also be at least one such B' for B to be a busy interval on k in S_{real} with $B \sim V$ and $B \subseteq V$. In particular, one such B' must have $B' \subseteq B$, which directly implies $\hat{\phi}(B) \geq \hat{\phi}(B') > k - 1$.

Algorithm 2. Realignment of S_{aug} for analysis only

$\sup_V \leftarrow 2|J(V)|$ for every valley V
for $k \leftarrow m$ to 1 **do**
  FILL(k, T)
  $\sup_V \leftarrow \sup_V - 1$ for every V s.t. some V' with $V' \cap V \neq \emptyset$ was closed on proc. k

function FILL(k, V)
  if $\mathsf{crit}(V) \leq 1$ **then**
    return
  let C be the critical set s.t. $C \sim V$
  while $\exists$ busy interval $B \subseteq V$ on processor k with $B \sim V$ and $\hat{\phi}(B) \leq k - 1$ **do**
    let V_l, V_r be the left, right valley for C and B (given B is a section of C)
    if V_l exists and $\sup_{V_l} > 0$ **then**
      CLOSE(k, V_l)
    else if V_r exists and $\sup_{V_r} > 0$ **then**
      CLOSE(k, V_r)
  for every valley $V' \subseteq V$ of C which has not been closed on k **do**
    FILL(k, V')
function CLOSE(k, V)
  for every $t \in V$ which is idle on processor k **do**
    if processors $1, \ldots, k-1$ are idle at t **then**
      introduce new auxiliary busy slot on processor k at time t
    else
      move busy slot t of highest processor $\leq k - 1$ to processor k

While with Lemma 11 we have our desired property for busy intervals B of $\mathsf{crit}(B) \geq 2$, we still have to handle busy intervals of $\mathsf{crit}(B) \leq 1$. To be precise, we have to handle idle intervals which are surrounded only by busy intervals B of $\mathsf{crit}(B) \leq 1$. We will show that this constellation can only occur in S_{real} on processor 1 and that the realignment has not done any modifications in these intervals, i.e. S_{pltr} and S_{real} do not differ for these intervals. With the same argument as for the original single-processor Left-to-Right algorithm, we then get that at least one processor has to be busy in any schedule during these intervals.

Lemma 12. *Let I be an idle interval in S_{real} on some processor k and let B_l, B_r be the busy intervals on k directly to the left and right of I with $\mathsf{crit}(B_l) \leq 1$ and $\mathsf{crit}(B_r) \leq 1$. Allow B_l to be empty, i.e. we might have $\min I = 0$, but B_r must be nonempty, i.e. $\max I < d$. Then we must have $k = 1$ and $\hat{\phi}(B_l \cup I \cup B_r) > 0$.*

Lemma 13. *For every processor k, every idle interval on processor k in S_{opt} intersects at most two distinct idle intervals of processor k in S_{real}.*

Proof. Let I_{opt} be an idle interval in S_{opt} on processor k intersecting three distinct idle intervals of processor k in S_{real}. Let I be the middle one of these three idle intervals. Lemma 12 and Lemma 11 imply that k busy processors are required during I and its neighboring busy intervals. This makes it impossible for S_{opt} to be idle on processor k during the whole interval I_{opt}.

4.3 Approximation Guarantee and Running Time

Lemma 13 finally allows us to bound the costs of the schedule S_{real} with the same arguments as in the proof for the single-processor LTR algorithm of [7]. We complement this with an argument that the augmentation and realignment could have only increased the costs of S_{pltr} and that we have hence also bounded the costs of the schedule returned by our algorithm PLTR.

Theorem 2. *Algorithm* PLTR *constructs a schedule of costs at most* $2\,\mathrm{OPT} + P$.

Proof. We begin by bounding $\mathsf{costs}(S_{\text{real}})$ as in the lemma. First, we show that $\mathsf{idle}(S^k_{\text{real}}) \leq 2\,\mathsf{off}(S^k_{\text{opt}}) + \mathsf{on}(S^k_{\text{opt}})$ for every processor $k \in [m]$. Let $\mathcal{I}_1$ be the set of idle intervals on S^k_{real} which intersect some off interval of S^k_{opt}. Lemma 13 implies that $\mathcal{I}_1$ contains as most twice as many intervals as there are off intervals in S^k_{opt}. Since the costs of each idle interval are at most q, and the costs of each off interval are exactly q, the costs of all idle intervals in $\mathcal{I}_1$ is bounded by $2\,\mathsf{off}(S^k_{\text{opt}})$. Let $\mathcal{I}_2$ be the set of idle intervals on S^k_{real} which do not intersect any off interval in S^k_{opt}. The total length of these intervals is naturally bounded by $\mathsf{on}(S^k_{\text{opt}})$.

We continue by showing that $\mathsf{busy}(S_{\text{real}}) \leq 2P$. By construction of S_{aug} and the definition of $\sup_V$ and `close`, we introduce at most as many auxiliary busy slots at every slot $t \in T$ as there are jobs scheduled at t in S_{pltr}. For S_{aug}, an auxiliary busy slot is only added for t with $\mathsf{crit}(t) \geq 2$ and hence $\mathsf{vol}(t) \geq 1$. Furthermore, initially $\sup_V = 2|J(V)|$ for every valley V and $\sup_V$ is decremented if some V' intersecting V is closed during $\texttt{fill}(k, T)$. During $\texttt{fill}(k, T)$ at most a single V' containing t is closed for every $t \in T$. Finally, auxiliary busy slots introduced by S_{aug} are used in the subroutine `close`. This establishes the lower bound $\mathsf{costs}(S_{\text{real}}) = \mathsf{idle}(S_{\text{real}}) + \mathsf{busy}(S_{\text{real}}) \leq 2\,\mathsf{off}(S_{\text{opt}}) + \mathsf{on}(S_{\text{opt}}) + 2P \leq 2\,\mathrm{OPT} + P$ for our realigned schedule.

We complete the proof by arguing that $\mathsf{costs}(S_{\text{pltr}}) \leq \mathsf{costs}(S_{\text{real}})$ since transforming S_{real} back into S_{pltr} does not increase the costs of the schedule. Removing the auxiliary busy slots clearly cannot increase the costs. Since the realignment of S_{aug} only moves busy slots between processors, but not between different time slots, we can easily restore S_{pltr} (up to permutations of the jobs scheduled on the busy processors at the same time slot) by moving all busy slots back down to the lower numbered processors. By the same argument as in Lemma 1, this does not increase the total costs of the schedule.

Theorem 3. *Algorithm* PLTR *has a running time of* $\mathcal{O}(nf \log d)$ *where* f *denotes the time needed for finding a maximum flow in a network with* $\mathcal{O}(n)$ *nodes.*

Acknowledgement. A comprehensive version of this paper, including all proofs, is available on arXiv: https://arxiv.org/abs/2307.00949. Thanks to Prof. Dr. Susanne Albers for her supervision during my studies. The idea of generalizing the Left-to-Right algorithm emerged in discussions during this supervision.

References

1. Antoniadis, A., Garg, N., Kumar, G., Kumar, N.: Parallel machine scheduling to minimize energy consumption. In: Proceedings of the Thirty-First Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2020, pp. 2758–2769. Society for Industrial and Applied Mathematics, USA (2020)
2. Antoniadis, A., Kumar, G., Kumar, N.: Skeletons and minimum energy scheduling. In: Ahn, H.K., Sadakane, K. (eds.) 32nd International Symposium on Algorithms and Computation (ISAAC 2021). Leibniz International Proceedings in Informatics (LIPIcs), vol. 212, pp. 51:1–51:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2021). https://doi.org/10.4230/LIPIcs.ISAAC.2021.51. https://drops.dagstuhl.de/opus/volltexte/2021/15484
3. Baptiste, P.: Scheduling unit tasks to minimize the number of idle periods: a polynomial time algorithm for offline dynamic power management. In: Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithm, SODA 2006, pp. 364–367. Society for Industrial and Applied Mathematics, USA (2006)
4. Baptiste, P., Chrobak, M., Dürr, C.: Polynomial time algorithms for minimum energy scheduling. In: Arge, L., Hoffmann, M., Welzl, E. (eds.) ESA 2007. LNCS, vol. 4698, pp. 136–150. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-75520-3_14
5. Demaine, E.D., Ghodsi, M., Hajiaghayi, M.T., Sayedi-Roshkhar, A.S., Zadimoghaddam, M.: Scheduling to minimize gaps and power consumption. In: Proceedings of the Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures, SPAA 2007, pp. 46–54. Association for Computing Machinery, New York (2007). https://doi.org/10.1145/1248377.1248385
6. Graham, R., Lawler, E., Lenstra, J., Rinnooy Kan, A.: Optimization and approximation in deterministic sequencing and scheduling: a survey. Ann. Discret. Math. **5**, 287–326 (1979). https://doi.org/10.1016/S0167-5060(08)70356-X
7. Irani, S., Shukla, S.K., Gupta, R.K.: Algorithms for power savings. In: Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms, 12–14 January 2003, Baltimore, Maryland, USA, pp. 37–46. ACM/SIAM (2003). http://dl.acm.org/citation.cfm?id=644108.644115