Green power aware approaches for scheduling independent tasks on a multi-core machine

Ayham Kassab, Jean-Marc Nicod, Laurent Philippe, Veronika Sonigo

Green Power Aware Approaches for Scheduling Independent Tasks on a Multi-core Machine*

Ayham Kassab, Jean-Marc Nicod, Laurent Philippe, Veronika Rehn-Sonigo
FEMTO-ST Institute, Université Bourgogne Franche-Comté / CNRS / ENSMM
F-25000 Besançon, France
[ayham.kassab|jean-marc.nicod|laurent.philippe|veronika.sonigo]@femto-st.fr

August 18, 2021

Abstract

The energy consumption of large Information and Communications Technology structures such as data and computation centers along with the corresponding carbon footprint are on the rise. Green computing has become an indispensable solution to face the resulting economical and environmental challenges. Powering these centers with renewable energy sources is however a challenge since these sources cannot guarantee a constant power supply due to their fluctuating power production. We here tackle the problem of scheduling independent tasks on a multi-core machine within a predicted renewable power envelope that varies over time. We evaluate the complexity of different instances of the problem from a theoretical point of view. We propose several heuristics, including genetic algorithms, and we conduct experiments to assess their performance. For some particular cases we compare the performance of these heuristics to optimal solutions.

Keywords Task scheduling - optimization - complexity - heuristics - renewable energy sources - parallel machines - green computing

*This work was supported in part by the Agence Nationale de la Recherche: DATAZERO (contract "ANR-15-CE25-0012") project and EIPHI Graduate school (contract "ANR-17-EURE-0002"). Computations have been performed on the supercomputer facilities of the Mésocentre de calcul de Franche-Comté – Besançon.

1 Introduction

The increasing demand on computational resources pushes computing and data center operators to expand their IT infrastructures, but the growing capacity of these centers is accompanied with rising concerns about their carbon footprint. As the electricity consumption of Information and Communications Technology is estimated to reach up to 20% of the world consumption by 2030[1], which will correspond to 5.5% of the global CO2 emissions, the resulting environmental impact underline the necessity of greener computing. Over the course of the last few years, many efforts have addressed the energy efficiency of IT infrastructures. The proposed solutions vary from reducing the power consumption of IT components, such as CPUs and hard disks, to addressing the energy efficiency on the whole facility level such as cooling and electricity transport systems as early mentioned by Khargharia et al. in [21]. On the other hand, as the nature of the used energy source highly impacts the carbon footprint, computing and data center operators are deploying renewable energy sources to support their increasing power demand, and, at the same time, to reduce their environmental impact.

Renewable sources can be used in different scenarios. Several operators are choosing to build their new data centers in northern countries, whose advanced

[1] https://theshiftproject.org

renewable production is attractive. Other operators sign a green energy contract with energy suppliers. This actually means that the consumed energy could be a mix of green and brown energy but enough green energy is being produced and added to the grid to off-set the supplied brown energy. Other high performance computing (HPC) and data centers use on-site renewable energy. This means that the energy is produced by on-site renewable energy sources.

The challenge with on-site energy production is the variability of the production of most renewable energy sources (e.g., solar panels and wind turbines) while, in order to perform any computation, enough power must be available to cover the power consumption of the environment (cooling, power distribution, etc.) and of the computation units. It follows that a system solely powered by on-site renewable sources may have different computational capacities at different intervals of time if no power storage units, like batteries, are used. On the other hand, using batteries to stabilize the power input over time, has a cost since between 15 and 20% of the power is lost when charging and discharging the batteries [22]. It is thus worth trying to run as much computations as possible directly with the produced power , which in turn implies that the workload can be scheduled over time depending on the available power, i.e., tasks do not have deadline. This is, for instance, the case of the workflow of a HPC application, composed of tasks with different computation densities that are run in batch mode and thus tolerate delay. Then workload management, scheduling, is needed to efficiently match the workflow with the variable green power production. Ideally, intensive computations should be run when the power production is high, and the less dense computations when the power production is lower. The issue is thus to run as many tasks as possible under the constraint of the available and variable power to finishing the tasks as soon as possible. We thus consider both the makespan and the total flowtime objectives.

The research scope of the paper is the problem of scheduling the execution of HPC applications on a computing center powered solely by renewable energy sources. As the whole problem is complex [28], it must be handled in several steps. In this paper, we concentrate on the classical optimization problem of running a set of independent sequential tasks on a multi-core machine, the $P||C_{\max}$ problem, with the additional constraint of a power supply that varies over time. So the main problem to be solved is to schedule tasks depending on the available power, which means that a task cannot be run when there is not enough power available, even if there is a free core. The objective is to assess the behavior of scheduling algorithms in this particular case. The contributions of the paper are: (i) a theoretical study to determine the complexity of several scenarios of this optimization problem, (ii) the proposition of several scheduling heuristics that integrate the power constraint, and (iii) a comparison of the performance of these heuristics based on simulations. Note that, to our knowledge, this paper is the first research work that addresses the theoretical aspect of scheduling tasks under power availability constraints.

The paper is organized as follows. In Section 2 we summarize the related work on energy, scheduling and computation. In Section 3 we define the model and the limitations of the tackled problem. In Section 4 we present a theoretical study to determine the complexity of several scenarios of this optimization problem. We demonstrate that the general case of this problem is NP-Hard in the strong sens. In Section 5 we propose several heuristics that take the power variation into account. In Section 6 we presents the experiments done to assess the heuristics performance and their results. Finally we conclude in Section 7. Note that this paper is an extended work from these two preceding papers: [19, 18]. In particular it presents a wider study on the heuristics performance, including a comparison of the distance from the results of the heuristics to the optimal solutions for specific scenarios.

2 Related Work

Researchers from many fields addressed the energy efficiency in big Information and Communication Technology infrastructures such as computating and data centers. The proposed solutions vary from switching off unused servers [29] to integrating renewable

sources into the power supply of the data center [1]. Many surveys such as [26], [10], [20], [27], [34] give a wide range of technologies and tools that are deployed at different levels of the center to reduce its energy consumption.

2.1 Combined software and hardware IT management

Many works propose solutions that control both the system software components, scheduling policies for example, and the hardware components such as processors and memory chips. Sheikh et al. [34] gave a comprehensive presentation of the main technique issues (hardware and software) for energy aware scheduling of workflows on different types of architectures (from single to parallel architectures). A common energy management technique on the hardware level is Dynamic Voltage and Frequency Scaling (DVFS) [36], which reduces the power consumption of a processor by lowering its clock rate frequency and supply voltage. As a result, the processor becomes slower and the execution time is thus longer. Since energy equals power times duration, if DVFS is not applied appropriately, in some cases, the overall energy consumption might be worse than running the same job on higher voltage and finishing the execution earlier. DVFS can be applied during processor slack periods, which is accomplished independently in modern processors using on-chip controllers [14, 16].

In [38], the authors combine DVFS with an energy-efficient scheduling algorithm that takes into account the Service Level Agreement (SLA) to decide when to run the processors at lower frequencies. Each job has a maximum and a minimum frequencies F_{max} and F_{min}, and by knowing the maximum and the minimum operating frequencies of each server, the algorithm selects the server that operates within the frequency limits required by the job and at the same time ensures that the job cannot overuse the resources. In [37], two scheduling algorithms are proposed. The proposed algorithms apply DVFS at slack times to reduce the energy consumption without increasing the total scheduling length. In addition, a green SLA is developed to offer users the option to consume less energy at the expense of increasing the execution time within an affordable limit.

High performance computing providers sign an energy contract with the electricity supplier agreeing on a certain level of power supply during the duration of that contract. If the power consumption exceeds the supply value specified in the contract, the price of electricity per additional Watt is penalized by the supplier. Therefore, most HPC centers deploy power capping techniques to avoid going past the power supply limit. A power cap is a value given in watt which the system power consumption must not exceed. Many techniques can be used to perform power capping such as DVFS, or simply switching off certain computational nodes [30]. Server level power capping can also be achieved by setting some of the server components to sleep mode, and throttling the CPU by inserting idle cycles. For multi-threaded workloads, thread packing can be used to control the number of active cores of a processor, therefore limiting its power consumption, [31]. In [7] the total energy consumption over a period of time is limited by setting an energy budget, regardless of the instantaneous power consumption levels during that time. Sheikh et al. in [33] proposed a comprehensive overview of thermal-aware task scheduling on multi-core architectures. Controlling processor temperature, by avoiding hotspots for instance, appears mandatory to optimize energy consumption of processors since, in some cases, energy-aware scheduling alone may not be efficient enough.

Zhang et al. in [39] presented a grid that is made out of several geographically distributed data centers powered by renewable sources. The idea is to overcome the downsides of using renewable sources, which are uncertainty and variation in available power levels, by taking advantage of the fact that different weather conditions in different locations would lead to different available power levels. By using inter data center virtual machine migration, data centers with extra workloads than their local energy production can export these jobs to other data centers with extra production, taking the network capacity into account.

Goiri et al. in [11] proposed a scheduler for parallel batch jobs in a data center powered by both a green energy source and the electrical grid (GreenSlot).

Prediction of the amount of solar energy that will be available in the near future is done using historical data and weather forecasts. Then, jobs are scheduled in a way that maximizes the green energy consumption while meeting their deadlines. By matching the workload with the predicted green energy level (scheduling more jobs at times where the green energy production level is high), and with the usage of brown energy is necessary to avoid the violation of job deadlines, it schedules these jobs at times where brown energy prices are cheap.

Khargharia et al in [21] proposed a scalable hierarchical framework dedicated to large scale e-business data centers using game theory to globally optimize power and performance at runtime considering *performance/watt* as a metric.

Compared to these works, we tackle the scheduling of tasks on only one infrastructure. This infrastructure is solely powered by renewable sources which introduces a variable power constraint in the scheduling process. We do not target reducing the energy consumption, as it is done with DVFS. We rather aim to exploit the available power as well as possible which transforms the problem into using as much power as possible, when the latter is available. The novelty of this work is that our problem is thus power constrained, with variations over time, rather than energy budget constrained or optimized. This simplifies the optimization problem since optimizing the power use requires to optimize only one objective (e.g. makespan or flowtime) while energy based problems are usually bi-criteria problems (e.g. energy and makespan).

2.2 IT management via software

Scheduling policies are a good example of software level solutions that can be used to tackle the energy efficiency in HPC systems. In [17] a 2-phase algorithm based on list scheduling considers two objectives, minimizing the makespan, which is the execution time of all the submitted jobs, and minimizing the total energy consumption. The results show a trade-off between the performance and the energy consumption. Lam et al. in [23] presented a trade-off between the energy consumption and the flowtime of a job, which is the time elapsed from the submission of the job until it is completion. In [3, 35], the authors addresse both the makespan and the total flowtime objectives while minimizing the energy consumption.

Genetic algorithms represent another common method for optimization problems. In [24], each gene represents the allocation of a task to a processor and the voltage of that processor. The proposed algorithm has two objectives, minimizing the makespan and minimizing the total energy consumption. In our problem, using renewable energy sources makes finding the best utilization of the instant power the focus of the work, rather than reducing the energy consumption.

Sheikh et al in [32] propose a multi-objective evolutionary algorithm that aims at optimizing simultaneously the makespan, the consumption and the temperature peaks. The proposed approach for scheduling tasks is able to obtain an optimization upon these three metrics. This process is not very time consuming contrary to what one can sometimes observe with genetic algorithms.

In [4], a genetic algorithm is proposed to minimize due date violations of batch tasks in a cloud data center while respecting the renewable power envelope and the resource constraints. Grange et al. [13] propose in 2018 an algorithm for scheduling batch tasks powered by renewable energy sources and at the same time connected to the electrical grid. Their algorithm takes into account the availability of the renewable energy and the cost of brown energy to lower the operational cost while respecting due date constraints. In our work we do not consider due dates, since HPC applications do not have due dates when submitted to a computation center. We focus on more HPC related objectives such as minimizing the makespan and the total flowtime.

3 Model

In this section, we define all the formal and technical requirements of the proposed model.

As previously said we concentrate in this work on the classical $P||C_{max}$ problem. The execution

resource is thus a parallel platform, where several identical execution units process independent tasks. Practically, we consider a parallel machine with multiple CPU cores as execution units. The platform thus consists of a set $\mathcal{C} = \{C_1, C_2, \ldots, C_c\}$ of c cores C_j that represent the execution units. Note that, since we consider a single parallel machine, network problems do not have to be taken into account.

We add to this problem the constraint that the power supply of the platform is provided solely by renewable energy sources. As such power supply is not stable and varies over time, the available power is represented at each time t by a value $\Phi^{available}(t)$. For technical reasons, a real power supply always provides a constant power supply, at least during a time interval. We thus assume that the available power is a constant value $\Phi_x^{available}$ over an interval of time Δ_x, (Cf. Figure 1). For a given time horizon $\mathcal{H}$, the available power is thus modeled by a list of X intervals Δ_x of length δ_x, such that $\sum_{x=1}^{X} \delta_x = \mathcal{H}$. So the technical requirement is to have such an inetrval list before running the scheduling algorithm. This list could be provided by the power management system as in [28] so that we have a static knowledge of available power, at least until the time horizon $\mathcal{H}$. This power is shared by all the cores of the machine.

In the $P||C_{max}$ problem, the tasks are modeled by a set $\mathcal{T} = \{T_1, T_2, \ldots, T_n\}$ of n sequential independent tasks. Each task T_i is characterized by an a priori known processing time p_i and the requirements are hence to statically know which task will be run and its processing time. To introduce the power dimension in this problem, we consider that running a task on one core generates an extra power consumption [9] which varies over time depending on whether the task intensively computes or not. In order to keep the task model simple and usable in an optimization problem, we approximate the power consumption as follows: we assume that each task T_i has a constant power demand, which is its largest power need φ_i over its lifetime. This way, we guarantee that the real power consumption fits in the allocated power envelope. When a task T_i is executed on a core C_j, the total power consumption of C_j hence increases by φ_i.

When the parallel machine is running, it consumes

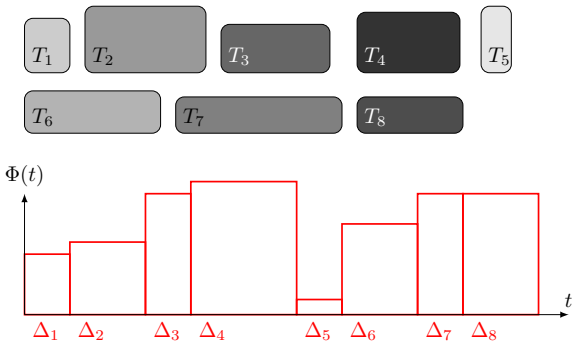


Figure 1: Illustrating example for the optimization problem: A set of tasks to be scheduled in the given power envelope.

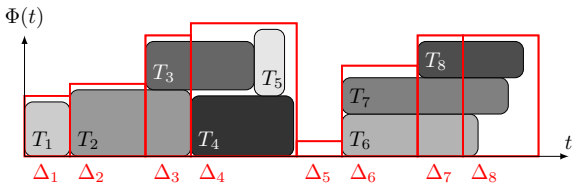


Figure 2: Example of a schedule

at any time at least its static power P^{stat}. With our task model, when the machine does not process any tasks, the power consumption is exactly P^{stat}, otherwise it increases depending on the executed tasks. The cores are not considered to have an independent power consumption when they are idle because they belong to the same machine. Since the static power has a constant value P^{stat} for the entire time horizon H, we can deduce for each period of time Δ_x the remaining power which is available for task computation, i.e., $\Phi_x = \max(\Phi_x^{available} - \Phi^{stat}, 0)$. Hence we only consider Φ_x as the available power to run the tasks in the scheduling problem.

Table 1 summarizes the notations used in the remainder of the paper. For interested readers, additional definitions for the model, used in the complexity proofs of the problems, are given in A.

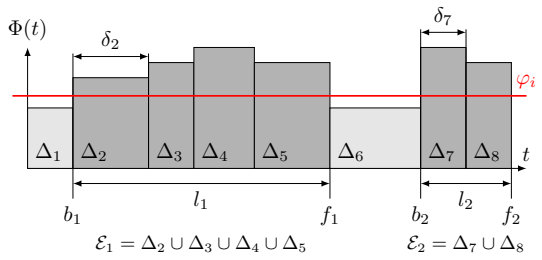


Figure 3: Illustrative example for intervals $(\Delta_1, \ldots, \Delta_8)$, available power on time and time slots ($\mathcal{E}_1$, $\mathcal{E}_2$) in which the task T_i, with power need φ_i, could be scheduled.

4 Optimization Problems and Complexity Study

Using the preceding model, we consider static optimization problems where the number of the tasks, their energy consumption, their duration and the available power are known in advance. Static problems are sometimes far from real practical cases but tackling such problems is however necessary to formally prove the complexity of the optimization problems related to real cases. Note that, for readability reasons, the theorem proofs are given in the Appendices.

4.1 Notations and objectives

Graham et al. defined the $\alpha|\beta|\gamma$ notation that characterizes a scheduling optimization problem [12]. In this notation the α value gives the characteristics of the execution platform: 1 for one machine, P, Q or R for parallel machines respectively identical, uniform or unrelated. The β value gives the task characteristics and/or constraints: $p_i = p$ for tasks of the same size, *prec* for precedence between tasks, *pmtn* if tasks can be preempted, etc. The γ value gives the criteria to be minimized as, for instance the makespan $C_{\max}$ or the (total) flowtime [2], $\sum C_i$, where C_i is the completion time of task T_i.

To express the constraint of limited available power, we propose to add $\varphi_i \leq \Phi_x$ for one machine problems and $\sum \varphi_i \leq \Phi_x$ for parallel machine problems to the Graham notation. This enforces that the power needed by one (φ_i) or several tasks ($\sum \varphi_i$) must be lower than the power provided by the energy sources (Φ_x). For example the problem $1|\varphi_i \leq \Phi_x|C_{max}$ is a one machine problem where we target makespan minimization for independent tasks, available power is not a constant over the time horizon and each task has a power need different from each other. If Φ_x variables are set to Φ, the available power is constant over the considered period and if φ_i variables are set to φ, every task needs the same power to run.

variable	definition
$\mathcal{T}$	set of tasks
T_i	task i
n	number of tasks
p_i	processing time of T_i
φ_i	power needed by T_i
$\mathcal{C}$	set of cores
C_j	jth core of the parallel machine
c	number of cores
Δ_x	interval with constant power
Φ_x	useful power in interval Δ_x
δ_x	length of Δ_x
X	number of consecutive intervals Δ_x
$\mathcal{H}$	total length of the interval set

Table 1: Summary of the notations

Considering computing and data centers, two main criteria are usually considered for minimization: the makespan ($C_{\max}$) targets the minimization of the completion time of a set of tasks and is thus relevant for computing centers where applications are composed of a set of tasks. In the case of several tasks launched by different users, as in data centers, then the flowtime ($\sum C_i$) is more relevant as minimizing this criterion leads to minimizing the mean finish time, which enforces a fair share of the resources between users.

4.2 One Machine Problems

We first tackle one machine problems as showing that these problems are NP-Hard is a way to prove that the more general parallel problems are NP-Hard as well. Note that we keep here the one machine name for the one core problem, actually a mono-core machine.

We consider the one machine problems for both objectives of makespan and flowtime and for the cases with or without preemption. These cases are simple without power constraint. We recall that each no delay schedule (i.e., schedule without delay between the tasks) is an optimal solution for the makespan objective and that the Shortest Processing Time (SPT) algorithm gives an optimal solution for the flowtime objective. We show here that, with power constraints, these problems are polynomial in the case of identical tasks (i.e., $p_i = p$, $1 \leq i \leq n$) and that the problems where tasks have different processing times are actually NP-Hard if preemption is not allowed.

We now consider different cases for the task processing time and objective functions.

4.2.1 Problems without preemption

In computing centers one node is usually dedicated to one user and no preemption is applied to tasks. We assess here the complexity of the one machine scheduling problem in that context.

Identical tasks $p_i = p$ and $\varphi_i \leq \Phi_x$ The most simple problem is when each task has the same computing time $p_i = p$ and the available power envelope is constant. To optimize our objective, we just have to choose as many tasks as possible in each time slot, considering the tasks in decreasing order of power requirements (largest power need first). If the interval length is not a multiple of the task size then the remaining time of this interval can be used to shift the next tasks. Obviously this solution is optimal for the makespan objective, as all tasks can be exchanged with each other. Changing the task order does not give a better solution and no place where a task could be placed is left empty. For the flowtime, as each task has the same processing time, any task permutation within the schedule leads to the same optimal flowtime.

Non-identical tasks The non-identical task problems, denoted respectively $1|\varphi_i \leq \Phi_x|C_{\max}$ and $1|\varphi_i \leq \Phi_x|\sum C_i$, are NP-Hard.

Theorem 1. *Minimizing the makespan of the schedule of a set of tasks ($1|\varphi_i \leq \Phi_x|C_{\max}$) to run in a set of intervals is NP-Hard in the strong sense if the tasks have different processing times p_i.*

The proof of Theorem 1 is given in appendix B.1.

Theorem 2. *Optimizing the flowtime of the schedule of a set of tasks ($1|\varphi_i \leq \Phi_x|\sum C_i$) to run in a set of intervals is NP-Hard in the strong sense if the tasks have different processing times p_i.*

The proof of Theorem 2 is given in appendix B.2.

4.2.2 Problems with preemption

In the case of data centers where the tasks to be processed are requests, these tasks can be preempted. We thus consider the impact of preemption on the scheduling problem complexity.

The $1|\varphi_i = \varphi \leq \Phi_x, pmtn|C_{\max}$ problem, where all tasks need the same power to run, accepts a polynomial solution. Remember that without power constraints non delay schedules are optimal. With power constraints it is however not possible to always have non delay schedules as some of the intervals Δ_x may not provide enough power Φ_x to schedule a task. The general idea is to avoid leaving intervals empty when there are still unscheduled tasks. For this purpose we schedule tasks with the following policy: at the beginning of a new interval or when a task is finished, we schedule the task (or the remaining part of a task) which wastes the less power ($\min(\Phi_x - \varphi_i)$). If another task than the current running task is selected, the running task is preempted and rescheduled later. We call this algorithm Less Wasting Remaining Task (LWRT).

Theorem 3. *Algorithm LWRT gives an optimal solution for the $1|\varphi_i \leq \Phi_x, pmtn|C_{\max}$ problem.*

The proof of Theorem 3 is given in appendix B.3.

Figures 4 and 5 illustrate the case where a LWRT task is or is not scheduled at each interval change or when a task is completed. On Figure 4 task T_2 is not preempted at the end of interval Δ_2. As a result task T_4 is scheduled later because of its large power need and interval Δ_5 is not used. On Figure 5 task T_2 is preempted at the end of interval Δ_2 and Task T_4 is executed instead. As Task T_2 needs less power to run it can be executed in interval Δ_5 which improves the makespan.

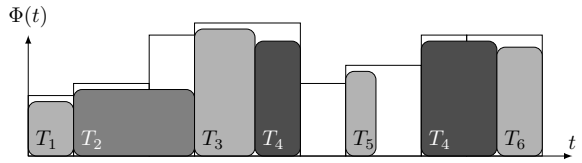


Figure 4: Illustrating example for the LWRT algorithm, T_2 is not the LWRT task for interval Δ_3, T_4 must be run here.

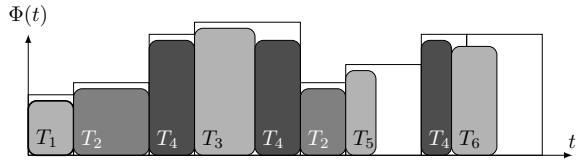


Figure 5: Illustrating example for the LWRT algorithm, part of T_4 has been swapped with T_2 which can be executed sooner than T_5, the makespan is optimal.

The complexity of the problem $1|\varphi_i \leq \Phi_x, pmtn|\sum C_i$ is still open. We have counter examples that SPT (Shortest Processing Time) does not always give the optimal result as due to power constraints it can be necessary to schedule longer tasks before short ones. Even if the complexity of this case remains an open problem, we suspect it to be NP-Hard.

4.3 Parallel Problems

We consider here the problem of scheduling a set of tasks on the cores of a parallel machine.

From the previous complexity results we can deduce that $P|\sum \varphi_i \leq \Phi_x|C_{\max}$ and $P|\sum \varphi_i \leq \Phi_x|\sum C_i$ problems are NP-Hard since parallel problems are generalizations of one machine problems. Problems with preemption must however be investigated. For the $P|\sum \varphi_i \leq \Phi_x, pmtn|C_{\max}$ problem, we have to schedule several tasks at the same time such that the sum of their power needs $\sum \varphi_i$ is lower than the available power Φ_x in each interval.

If the power needed by the tasks is the same ($P|\sum \varphi_i \leq \Phi_x, \varphi_i = \varphi, pmtn|C_{\max}$), then the problem is simple: in a given interval we execute as many tasks as possible in parallel provided that the power Φ_x and the constraint on the number of cores P are respected. Then, at the end of a task, we schedule another one and, at the end of the interval, we either stop tasks if there is less available power than before or start additional tasks if idle cores remain.

If the power needed by each task is different ($P|\sum \varphi_i \leq \Phi_x, pmtn|C_{\max}$), the problem is NP-Hard.

Theorem 4. *Minimizing the makespan of the schedule of a set of power heterogeneous preemptive tasks to run in a set of intervals ($P|\sum \varphi_i \leq \Phi_x, pmtn|C_{\max}$) is NP-Hard in the strong sense.*

For interested readers, the proof of Theorem 4 is given in appendix B.4.

Note that the proof highlights that the problem $P|\sum \varphi_i \leq \Phi, p_i = p, pmtn|C_{\max}$ is NP-Hard when the tasks have the same size ($p_i = p$).

For the flowtime objective, the $P|\sum \varphi_i \leq \Phi_x, pmtn|\sum C_i$ problem, we can differentiate the particular case where tasks have the same power need $\varphi_i = \varphi$ which is simple for the more general case where tasks have different power needs. In the $\varphi_i = \varphi$ case the SPT algorithm, modified to take both the available power and the number of core constraints into account, gives an optimal solution even if the tasks have different sizes. Then the problem where the tasks have different power needs is NP-Hard as the problem $P|\sum \varphi_i \leq \Phi_x, p_i = p, pmtn|\sum C_i$ is equivalent to $P|\sum \varphi_i \leq \Phi_x, pmtn|C_{\max}$ since the tasks do not need to be ordered as they are of the same size. This implies that the more general case $P|\sum \varphi_i \leq \Phi_x, pmtn|\sum C_i$ is NP-Hard too.

5 Heuristics

After conducting the complexity study for different scenarios of the problem in the previous section and since the parallel problems are proven to be NP-Hard, we propose in this section different heuristics to address these problems. We focus on two general problems $P|\varphi_i \leq \Phi_x|C_{\max}$ and $P|\varphi_i \leq \Phi_x|\sum C_i$, i.e., parallel problems without preemption because preemption is seldom used in the parallel application context. As presented in Section 3, we consider independent sequential tasks to concentrate on the energy concerns, without adding other constraints like communications. So the problem is to minimize the makespan or the flowtime of a set of tasks under variable power constraints.

All the proposed algorithms compute an ordered task list that is then executed on a first come first serve basis. We propose three types of heuristics that take power constraints into consideration. The first family is close to classical list based algorithms and the order in the list is based on a single criterion such as the processing time or the energy consumption. The second family uses more complex criteria to generate the list. The third family computes the task list using a genetic algorithm.

Algorithm 1: $PlaceTask(p_i, \varphi_i, x)$

Input:
p_i, φ_i: processing time, power consumption of task T_i
x: index of Δ_x, interval search starting point
Data:
Φ_x: useful power in interval Δ_x
δ_x length of interval Δ_x
b_x, f_x: the beginning and the finishing time of Δ_x
nc_x: number of available cores in Δ_x
$found$: boolean, initialized to $true$
$timeSlot$: list of intervals, initialized to $\emptyset$
Result:
TimeSlotAlloc: time slot allocation of task T_i

```
1  repeat
2      if (Φ_x < φ_i) ∧ (nc_x < 0) then
3          found ← false
4      else
5          found ← true
6          timeSlot ← timeSlot ∪ Δ_x
7          x ← x + 1
8  until (∑ δ_x = p_i) ∨ (!found)
9  if found then
10     TimeSlotAlloc.i ← timeSlot
11     for Δ_x ∈ timeSlot do
12         Φ_x-=φ_i
13         nc_x-=1}
14 return TimeSlotAlloc
```

5.1 Task scheduling

Once the task list is computed, the heuristics scan the interval list using the $Place\_Task()$ function (see Algorithm 1) to find for each task the first time interval where it can be scheduled.

The $Place\_Task()$ function takes a task, its computing time p_i and its power need φ_i, and a starting interval x to find the time when the task is run. It first iterates (Lines 1-6), starting at Δ_x, on the interval list to look for a time slot where the task can be placed, i.e., a list of consecutive intervals with enough power ($\varphi_i < \Phi_x$) and cores ($nc_x > 0$). If enough intervals are found then the task power need is removed from the time slot intervals (Lines 7-9). The function returns *true* if the task is placed, *false* otherwise.

The $Place\_Task()$ function thus schedules a task in the earliest possible time slot. Note that it is possible that a task is scheduled after another task that is ordered further in the list, due to the power constraints . The earliest possible time slot does not only mean the earliest time slot with a free processing unit, but also a time slot with enough available power to cover the power need of the task. For example, in Figure 2, let us suppose that list $L = [T_2, T_1, T_3, T_4, ...]$. We notice that T_2 comes before T_1 in the list, yet T_1 is scheduled first because the available power level at I_1 is not high enough to schedule T_2 which is hence delayed till I_2, the first interval with enough available power.

Note also that the $Place\_Task()$ function only takes integer values effacefor the computing time. The time slots are unitary and the task computing times are integer so that a task can only finish at the end of a time slot but not during a time slot. In a previous paper [18] we present experiments based on a $Place\_Task()$ function that takes real values for both the time slots and the task processing times. This leads to very large computing times, some ex-

periments lasting more than one week. . Since the results obtained on both experiment sets are not significantly different, using integer values is reliable enough for the experiments.

We detail the proposed heuristics in the rest of this section.

5.2 Simple priority based algorithms

List algorithms are fast and simple, they consist of two steps. In the first step, the tasks are sorted in a queue based on a priority value, and, in the second step, they are scheduled according to their order in the queue in a greedy manner, i.e., using the $Place\_Task()$ function.

For the first step we use classical priorities for the list algorithms. *Random* takes the task list as it is. This naive approach is used to compare the other algorithms with a non smart solution. LPT (Largest Processing Time) sorts tasks by decreasing processing times. This solution fosters long tasks which are more difficult to place and usually gives good results for the makespan minimization on parallel identical machines when the number of tasks exceeds 50, as shown in [5]. SPT (Shortest Processing Time) sorts tasks by increasing processing times p_i, as for flowtime minimization an increasing p_i order must be preferred. These three algorithms are implemented as they are defined in the literature. We just add the $Place\_Task()$ function to adapt them to the power constrained context.

These three algorithms do not take however the power constraints into consideration. We then propose new priorities for the list ordering that takes the power need of the tasks into consideration. LPN (Largest Power Need) sorts tasks by decreasing power need φ_i. Tasks with large power needs are difficult to place and scheduling them first may avoid using later slots. $LPTPN$ (Largest Processing Time Power Need) sorts tasks by decreasing values of $p_i \times \varphi_i$.

Due to their priority, the LPT, LPN and $LPTPN$ algorithms are rather makespan oriented and they do not produce appropriate solutions for the flowtime minimization, while SPT rather targets this criterion.

5.3 More complex priorities

To better address the bi-dimensional aspect of the problem we propose two algorithms that do not simply rely on one value but rather try to take into account both constraints, power and time.

As both the processing time p_i and the power need φ_i are important values for scheduling the tasks, taking them independently only fosters one and ignores the second. We thus propose algorithms that combine both values. *twoQs*, for two queues, tries to exploit the advantages of two priority assignments at the same time, in a try to giving the priority to tasks that might have high priority in one priority assignment but would be scheduled towards the end of the schedule in another. It sorts the tasks alternatively by processing time and power need: it computes the LPT and LPN lists and puts alternatively one of each in the final list.

In the tasks to be scheduled there may be some tasks that must absolutely be considered first because, if they are not, they will be placed in later intervals and they will weight badly on the optimization criterion. To avoid these cases, we propose a priority that fosters the tasks with only few possibilities of placement. This heuristics is called LPP for least possible places. For each task, a list of all possible intervals where it can be scheduled is computed and the tasks are sorted in the list by the number of possible places where they can be scheduled.

5.4 Genetic scheduling algorithms

Genetic algorithms (GA) are designed for such problems where the search space is large and they have proven to give good results in scheduling problems. Using GA also allows us to verify if simple solutions, such as our list based algorithm, can be improved by randomly changing part of it and how far it can be improved.

The first challenge in using GA is how to properly represent the solutions as a chromosome. A solution could be represented in many ways, for example, it can be presented as a task to machine allocation scheme, or task to time interval assignment, or simply by setting the start time for each task. These

different representations affect the efficiency of the GA. In a second step the solutions must be filtered thanks to a fitness value . In all the implemented solutions, the fitness is the optimization criteria, either the makespan or the flowtime, and the lower the fitness, the better the solution.

As a first try, we considered assigning each task to a time interval. A chromosome representing a schedule in this case consists of n genes, one for each task, and the value in the i-th gene expresses the time interval at which task T_i is scheduled for execution. Due to the power constraints of our scheduling problem, a solution is only valid if all tasks are scheduled in time intervals where enough power is produced and enough computing cores are available at that time, since tasks are scheduled in parallel. This approach leads to very long computing times and poor solutions so that we changed the chromosome representation.

The chosen solution considers the order of the task list to be a solution, a chromosome. The fitness value is computed from this list by generating the schedule with the $Place\_Task()$ function. One advantage of this approach is that all solutions are valid as long as the available power curve is big enough to execute all tasks, since the placement algorithm will always find a valid interval of time for each task , no matter where it is ordered in the list. Note that the implementation of this representation of the problem is also simpler and faster than in the preceding proposition, leading to shorter computation times.

The chromosome representation and its corresponding schedule is illustrated in Figure 6. A chromosome is a list of integers $[1 \rightarrow n]$ that represents the indices of all n tasks, and the place of an integer in the chromosome, represents the order of its corresponding task in the placement list. To create a new solution, it is enough to shuffle this list of integers. . This allows an easy implementation of most genetic operators.

Algorithm 2 illustrates the main genetic algorithm. The initial population size is set to 50. By setting X, the number of intervals, high enough, we assume that all initial chromosomes give a feasible schedule. Lines $[2 \rightarrow 7]$ show the generation of the initial generation. To improve the chances of finding a good solution, we add 5 individuals as seeds to the initial population.

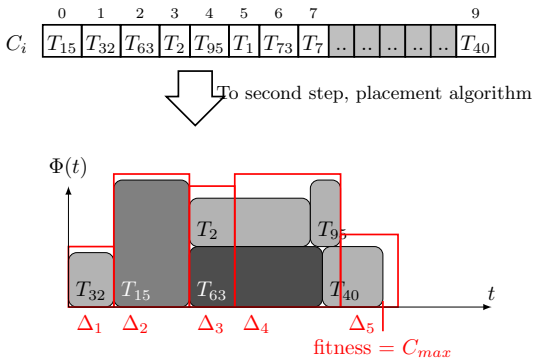


Figure 6: Illustrating example of a chromosome and the corresponding schedule

For the makespan objective, as shown in the algorithm, these five individuals are the priority queues of list based heuristics that showed good potential in early experiments: LPT, LPN, $LPTPN$, $twoQs$ and LPP. For the flowtime objective these individuals are $Random$, $twoQs$, SPT, LPN and LPP. The other 45 individuals are randomly generated by shuffling a list of integers $[1 \rightarrow n]$. In Line 8, the fitness of the population is calculated, and then, a population is sorted according to the fitness of its individuals in ascending order (Line 9). The elitism property is expressed in Line 13, where the best ten individuals are copied to the next generation. Then, 15 chromosomes are selected for 1-gene mutation $[14 \rightarrow 16]$, and another 15 chromosomes are selected for chunk mutation $[17 \rightarrow 19]$. Finally, 20 additional chromosomes are selected to perform 10 crossovers $[20 \rightarrow 23]$. At the end of each iteration, the fitness of the new offspring is calculated, and the new generation is again sorted according to the fitness values. This process is repeated from Line 10, until we go through nbI iterations without improvement.

The performance of genetic algorithms is based not only on the representation of the solution but also on the genetic operators, selection, mutation and crossing, implementation. In the literature, several propositions are available for these operators but, as we could not find any study that would answer the questions of which crossover to use for this kind of

Algorithm 2: *geneticAlgorithm*$(\mathcal{T}, \Delta, nbI)$

```
   Data: T, Δ: set of tasks, set of intervals
         nbI: number of iterations without enhancement
   Result: task list order
 1 stopCounter ← 0
 2 currentGeneration[0] ← LPT(T, Δ)
 3 currentGeneration[1] ← LPTPN(T, Δ)
 4 currentGeneration[2] ← twoQs(T, Δ)
 5 currentGeneration[3] ← LPN(T, Δ)
 6 currentGeneration[3] ← LPP(T, Δ)
 7 currentGeneration[5:50] ← 46 random solutions
 8 calculatePopulationFitness(currentGeneration)
 9 currentGeneration.sort()/* by ascending fitness   */
10 while stopCounter ≤ nbI do
11     oldBest ← currentGeneration[0]
12     nextGeneration ← []
13     nextGeneration[0:10] ← currentGeneration[0:10]
14     for i=1 to 15 do
15         mutant ←
           mutation(selection(currentGeneration))
16         nextGeneration.append(mutant)
17     for i=1 to 15 do
18         mutant ←
           chunckMutation(selection(currentGeneration))
19         nextGeneration.append(mutant)
20     for i=1 to 10 do
21         C1, C2 ← selection(currentGeneration)
22         newC1, newC2 ← crossOver(C1, C2)
23         nextGeneration.append(newC1, newC2)
24     calculatePopulationFitness(nextGeneration)
25     nextGeneration.sort()
26     currentBest ← nextGeneration[0]
27     currentGeneration ← nextGeneration
28     if oldBest − currentBest = 0 then
29         stopCounter++
30     else
31         stopCounter← 0
32 return currentBest
```

problems? Which selection to use for this kind of problems? We decide to assess several propositions. For the selection function we test two types of selections that are usually used in GA, wheel selection and random selection. For the crossover function we test three types of crossover: a one point crossover and two crossovers that use two points. The mutation operator raises the questions as it just needs to change one randomly chosen value. We present these operators in the following.

The 1-gene mutation operator, simply referred to as mutation in Algorithm 2, is illustrated in Figure 7. Two genes of the selected chromosome are randomly chosen, and their values are interchanged. The chunk mutation operator in Line 18 of the algorithm uses the same technique but two chunks of random size between 1 → 10 at two randomly selected points of the chromosome are swapped as shown in Figure 8.

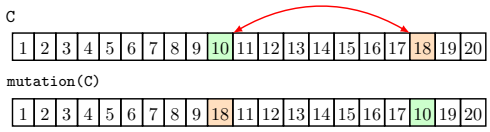


Figure 7: `mutation(C)`

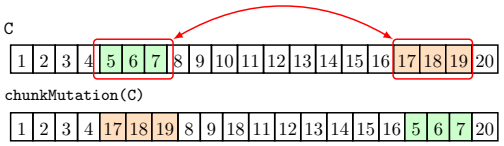


Figure 8: `chunkMutation(C)`

In this study, we evaluate four GA configurations and we test each configuration using both wheel and random selections. The first configuration applies only the mutation operators as the characteristics of our problem might suggest that applying too many modifications on a candidate solution might eventually be as arbitrary as randomly generating a new one. This approach is named noX, for no crossover, and, combined with the two selection operators, random (R) and wheel selection (W), we get the $noX-R$ and $noX-W$ algorithms.

Algorithm 3: `onePointCrossOver(C1, C2)`

```
   Data: C1 /* chromosome 1 with n tasks          */
 1       C2 /* chromosome 2 with n tasks          */
   Result: newC1, newC2: 2 new chromosomes each with n
           tasks
 2 n ← length(C1)
 3 p ← intRand(0, n)/* integer random value:  0 ≤p< n
                                                  */
 4 newC1 ← C1[0:p]/* p values between 0 and p-1   */
 5 newC2 ← C2[0:p]
 6 newC1 ← newC1 + C2 \ newC1
 7 newC2 ← newC2 + C1 \ newC2
 8 return newC1 , newC2
```

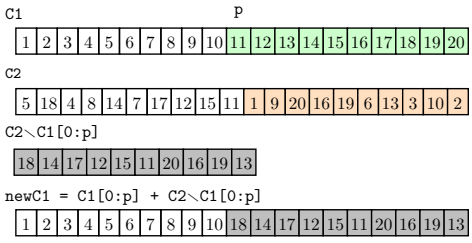


Figure 9: `OnePointCrossOver(C1, C2)`

Mutating only one gene or even a chunk of genes limits the exploration of the search space. Crossover, on the other hand, allows wider exploration of the search space by applying bigger changes on candidate solutions. In this paper we assess three different crossovers described in the literature. The first one is the common 1-point crossover illustrated in Algorithm 3. A crossover point is randomly selected where both parent chromosomes are split between the first, the head, and the second, the tail. In a typical 1-point crossover, the two chromosomes would swap tails with each other. In our case however, this cannot be done this simply, because the same task could appear in the head of one chromosome and in the tail of the other. After the crossover the task would be twice in one chromosome, while it completely disappears from the other. To overcome this difficulty, each child chromosome keeps its parent's head, while the genes of its tail (the remaining tasks) are ordered according to their order in the other parent. The 1-point crossover algorithms are named $1pX$. With the two selection operators we have $1pX-R$, for random selection, and $1pX-W$, for wheel selection.

In 1-point crossover, up to $n-1$ genes could change after a crossover, which could be arbitrary enough to lower the quality of a provided a good seed. 2-point crossover operators provide a solution for this problem. The changed part of a chromosome is limited by two points instead of just one which gives more control on the percentage of the chromosomes that gets modified.

The first 2-point crossover we test is called Order Crossover, OX. It is illustrated in Algorithm 4. Two crossover points p_1 and p_2 are randomly chosen in such a way that $p_1 < p_2$. Each parent chromosome C_{parent} passes its middle genes to the child $C_{parent}[p_1 : p_2] \rightarrow C_{child}[p_1 : p_2]$. Then, the genes in the edges of the child, starting from $C_{child}[p_2+1]$ circling back to $C_{child}[p_1-1]$, are reordered according to their order in the other parent starting from p_2+1 circling back to p_2. This algorithm is illustrated in Figure 10. The first new off-spring $newC_1$ is composed of the middle part of the first parent $C_1[p_1 : p_2]$. The subset of the rest of the genes of C_1 starting from $C_1[p_2+1]$: [18,19,20,1,2,3] is reordered as these genes appear in $C_2[p_2+1] \rightarrow C_2[p_2]$: [3,2,18,1,20,19]. The ordered subset is then added to the first off-spring $newC_1$ in the same circular man-

Algorithm 4: `orderCrossOver(C1, C2)`

```
   Data: C1 /* chromosome 1 with n tasks          */
1        C2 /* chromosome 2 with n tasks          */
   Result: newC1, newC2: 2 new chromosomes each with n
           tasks
2  n ← length(C1)
3  p1 ← ⌊n × rand(0, 1) × 0.15⌋
4  p2 ← ⌊n × (rand(0, 1) × 0.15 + 0.85)⌋
5  newC1 ← C1[p1:p2]
6  newC2 ← C2[p1:p2]
7  temp1, temp2 ← [], []
8  for i = 0 to n − 1 do
9      if C2[(i + p2)%n] ∉ newC1 then
10         temp1.append(C2[(i + p2)%n])
11     if C1[(i + p2)%n] ∉ newC2 then
12         temp2.append(C1[(i + p2)%n])
13 newC1 ← temp1[n−p2:n−p2+p1] + newC1 +
   temp1[0:n−p2]
14 newC2 ← temp2[n−p2:n−p2+p1] + newC2 +
   temp2[0:n−p2]
15 return newC1 , newC2
```

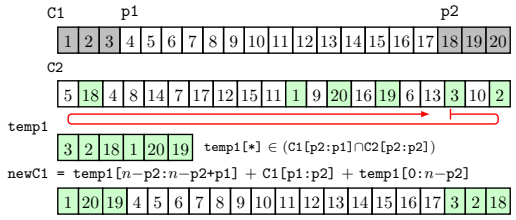


Figure 10: `orderCrossOver(C1, C2)`

Algorithm 5: `middleCrossOver(C1, C2)`

```
   Data: C1 /* chromosome 1 with n tasks                    */
1        C2 /* chromosome 2 with n tasks                    */
   Result: newC1, newC2: 2 new chromosomes each with n
           tasks
2  n ← length(C1)
3  p1 ← intRand(0, n)/* integer random value:
       0 ≤p1< n                                            */
4  p2 ← intRand(0, n)
5  if p1 < p2 then
6  |  newC1, newC2 ← orderCrossOver(C1, C2)
7  else
8  |  if p1 = p2 then
9  |  |  newC1 ← mutation(C1)
10 |  |  newC2 ← mutation(C2)
11 |  else
12 |  |  newC1 ← C1[0:p2] /* p2 task indices              */
13 |  |  newC2 ← C2[0:p2]
14 |  |  for i = 0 to n − 1 do
15 |  |  |  if C2[i] ∈ C1[p2:p1] then
16 |  |  |  |  newC1.append(C2[i])
17 |  |  |  if C1[i] ∈ C2[p2:p1] then
18 |  |  |  |  newC2.append(C1[i])
19 |  |  newC1 ← newC1+C1[p1:n]
20 |  |  newC2 ← newC2+C2[p1:n]
21 return newChromo1 , newChromo2
```

ner. The algorithms that use this crossover operator are named $OX - R$ and $OX - W$ when associated with random and wheel selections respectively.

Finally, we test a classical 2-point crossover. Each child has the same edges as its parent $C_{parent}[0 : p_1 - 1] \rightarrow C_{child}[0 : p_1 - 1]$ and $C_{parent}[p_2 + 1 : n] \rightarrow C_{child}[p_2+1 : n]$. The genes in the middle of the child $C_{child}[p_1 : p_2]$ are reordered according to their position in the parent C_{parent}. This operator is detailed in Algorithm 5. Figure 11 illustrates an example of this operator. The operator is named Middle Cross Over and the corresponding algorithm are $MX - R$ and $MX - W$ for random and wheel selections respectively.

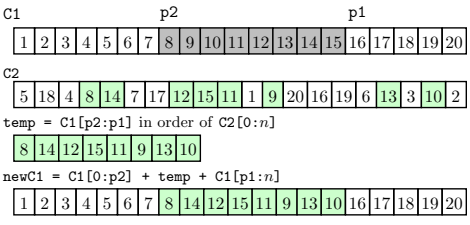


Figure 11: `middleCrossOver(C1, C2)`

It is worth noticing that, in a previous paper [18], we have also proposed binary search based scheduling algorithms. The idea behind this family of algorithms was to fix a time horizon and try to schedule the tasks in the time slots that better fit their power needs, to reduce the power waste. The time horizon was then reduced, when the algorithm found a schedule shorter than the current time horizon or increased otherwise, using a binary search technique. The family of algorithms however did not give good results while taking large computing times. For this reason we do not include them in this paper.

6 Experiments

In this section, we present the experiments conducted in order to assess the performance of the heuristics proposed in the previous section. We developed a python script[2] that implements the heuristics, calculates the resulting schedules and presents the performance of each heuristic depending on the studied objective. With the script we run experiments using different methods to generate the experimental input data sets, either based on real world data, or using random generation within a specified range and following a distribution law. Since the computation of a schedule by a heuristics is deterministic, the settings of the experiments only depend on the input, the experimental input data sets.

An experimental input data set consists of two parts, the workload and the power envelope available for the task executions (interval list). The workload is represented by a list of tasks that is passed to the scheduler for execution. The methods used to generate these values differ from one experiment to another and we detail them in the following. The overall aim of the different data generations is twofold: first to allow a performance evaluation of our heuristics under a broad spectrum of conditions; and second a

[2]The source files are available on GitHub `http://github.com/laurentphilippe/greenpower`

comparison to a known optimal solution.

Note that, for all the experiments, we consider that the computational platform consists of a parallel machine with 16 cores and we set the time unit to 6 minutes, 1/10 of an hour.

6.1 Task list models

We recall that each task T_i is characterized by two main values, its processing time p_i and its power consumption φ_i.

Feitelson model From the analysis of real workload logs, Lublin and Feitelson present in [25] a model of workload generation based on a hyper-gamma distribution. In the *Feitelson task model* based experiments, we use this distribution law to generate p_i values in order to produce tasks with processing times that are close to real data. The hyper-gamma law is based on two gamma laws, of parameters $\alpha = 4.2/\beta = 0.94$ for the first and $\alpha = 312/\beta = 0.3$ for the second, with a probability of 0.685 for the first gamma law following a uniform law. We thus first draw a uniform random value to decide which law to use when generating the p_i value and we repeat this for the whole task list. The resulting task lists contain 100 tasks per list, with a range of processing times between 1 and 500 time units. This model however does not include the power consumption of the tasks, so that we use random generation of φ_i with a uniform distribution law between 1 power unit and $\varphi_{i_{\max}}$. In the experiments $\varphi_{i_{\max}}$ ranges between 15 and 150 power units, by steps of 15.

Exponential model Feitelson's model although realistic is based on workload logs collected from only three sites which limits the experiments to the range of values collected for that sample of data. Therefore, to explore a wider set of parameters, we use for our second set of experiments synthetic workloads, with random generation for both p_i and φ_i. The φ_i generation parameters are kept similar to the previous experiment. The processing times of tasks however range between 1 time unit and $p_{i_{\max}}$. We use an exponential distribution law where most generated samples will fall around the mean value $p_{max}/2$, while less values fall towards both limits of the range. This means that tasks that are too hard or too easy to schedule occur with less probability.

The size of the generated tasks varies in this case on both time and power axes, representing tasks with different processing times and different computational density. We generate 100 groups of lists of tasks. For each 10 group, the maximum processing time in each task list $p_{i_{\max}}$ varies from 10 to 100 time units, with an increment of 10. While the maximum power consumption in a task list for each 10 group $\varphi_{i_{\max}}$ varies from 15 to 150 power units, with an increment of 15. In total, we test 100 combinations of ($p_{i_{\max}}$, $\varphi_{i_{\max}}$). For each combination, in order to get statistically stable results, 150 different lists of tasks are generated with consideration to the specified (p_{max}, φ_{max}) couple. Each task list is composed of 100 tasks.

"Tasks From Intervals" model Since it is not possible to find the optimal solution of an NP-Hard problem in polynomial time, we propose two methods to measure how far the proposed heuristics are from the optimal. The first method, named Tasks From Intervals (TFI), consists in generating a set of tasks starting from a given time interval set. Given a set of time intervals representing a power envelope, we generate a set of tasks that will totally fill the intervals using Algorithm 6.

The input of Algorithm 6 is a power interval list. In this list we randomly choose an interval as a starting point Δ_{start}. The power consumption φ_i of the generated task is randomly chosen in $[1, \Phi_{start}]$ and limited by $\varphi_{\max}$, and the processing time p_i is chosen between $[\Delta_{start} \rightarrow \Delta_{stop}]$, where Δ_{stop} is the last interval before the available power drops below φ_i. The task is added to the task set, and then its power consumption is remmoved from all the intervals involved in its generation. The process is repeated until all the available energy is used. Lower thresholds for φ_i and p_i are used to avoid generating too small and too many tasks. Therefore all the tasks can fit in the intervals and totally fill them. We hence have generated an optimal solution with a makespan of $\Delta_{\max}$. We also take more or less intervals to generate between

Algorithm 6: Tasks generation from intervals

```
Input: interList: power interval list
Data:
  inter: interval
  Δ_start, Δ_stop  begining and end of interval inter
  Φ_start: power available in inter
  φ_i, p_i: power and processing time of task t_i
Result: taskList: task list
1  begin
2      while length(interList) ≠ 0 do
3          inter ← random(0, length(interList))
4          Φ_start ← inter.power
5          φ_i ← min(φ_max, random(1, Φ_start))
6          Δ_stop ← getLastInterval(φ_i)
7          p_i ← random(Δ_start, Δ_stop)
8          t_i ← newTask(φ_i, p_i)
9          taskList.add(t_i)
10         for i ∈ [Δ_start, Δ_stop] do
11             i.power ← i.power − φ_i
12             if i.power = 0 then
13                 interList.remove(i);
14 return taskList
```

Algorithm 7: Intervals generation from tasks

```
Input: taskList: task list
Data:
  inter: interval
  s_i, e_i: start and end time of task t_i
  φ_i, p_i: power and processing time of task t_i
Result: interList: power interval list
1  begin
2      interList = newinterList(maxTime)
3      for inter ∈ interList do interList.add(inter(0)
4      for t_i ∈ taskList do
5          scheduled ← true
6          repeat
7              s_i ← random(0, H − p_i)
8              e_i ← s_i + p_i
9              for inter ∈ [s_i, e_i] do
10                 if inter.nbCores = 0 then
11                     scheduled ← false
12             if scheduled then
13                 for inter ∈ [s_i, e_i] do
14                     inter.nbCores ← inter.nbCores − 1
15                     inter.power ← inter.power + φ_i
16         until not scheduled
17 return taskList
```

90 and 110 tasks per list. Since the tested heuristics are expected to find a makespan that is longer than this optimal value, we repeat the power envelope two times in order to make sure that the C_{max} calculated by the heuristics would fit within the time horizon. The ratio between the C_{max} found by a heuristics and the optimal value is the distance from the optimal for this heuristics, $dis_{OPT} = C_{max} \div \Delta_{\max}$.

6.2 Power interval model

Let recall that the power envelope is discretized into time intervals Δ_x of length δ_x.

Realistic model A realistic model is used to generate the power envelope that represents the various renewable power supply. Based on real collected historical data [15], this model can produce a realistic power envelope that corresponds to a given number of wind turbines and a given area of solar panels in square meters. We consider that the generated power envelope is accessible by all computation units only for tasks execution. All the energy needed to power on the computation units and to support the IT infrastructure is thus previously deducted. We partition the power envelope into unified intervals Δ_x with length of $\delta = 1$ time unit. This interval length corresponds well with the fluctuation frequency seen in renewable power supplies. Each time interval list contains 10 000 intervals, representing in total the power envelope.

"Intervals From Tasks" model A second method, named Intervals From Tasks (IFT), to measure the distance of the heuristic generated schedules from optimal is is based on the generation of the power envelope from a set of tasks.

In this method, given in Algorithm 7, we take each task from a task set and randomly choose a starting point within the time horizon $\mathcal{H}$. We then check in the corresponding time intervals that last as long as p_i (between s_i and e_i) that enough cores are available. If a core is available for all that duration, the algorithm increases the level of available power by φ_i, or makes another try otherwise, until the task is scheduled. Since we consider a 16 cores platform, up

to 16 tasks can overlap over one or more time intervals, and the power level in such intervals is the sum of all the concerned over lapping tasks. The generated intervals have irregular lengths in this set of experiments. Similarly to the case of TFI, the total area under the generated power envelope equals the total area of the set of used tasks. Therefore, the optimal solution is the end of the last time interval $\Delta_{\max}$. For this data generation method, we use semi-synthetic workloads with p_i values based on Feitelson's traces and randomly generated φ_i.

6.3 GA settings

We evaluate the GAs with different types of crossovers, without crossover, and evaluate the effect of wheel vs. random selection. For each GA the stopping condition (nbI in Algorithm 2, number of generations without any improvement) is set to 50. Note that other computations have shown that a value of 10 gives poorer results. The fitness value used to evaluate each chromosome is either $C_{\max}$ or $\sum C_i$ depending on the optimization objective. All compared chromosomes are solutions for the same case, same sets of time intervals and sets of tasks, therefore, the $C_{\max}$ and $\sum C_i$ are fair fitness value to use for comparison.

6.4 Evaluation metrics

The finish time of the last task, the makespan, is a natural objective to optimize the power envelope use since its optimization allows to reduce the whole execution time of the task set. It cannot however be used as a reference metric for performance assessment since it depends on the processing times of the tasks. A set of larger tasks indeed always gives a longer makespan than a set of shorter ones. The makespan is also subject to the distribution of the available power φ_i in the intervals, and in particular intervals with low power. We thus propose normalized metrics that do not depend on the p_i and φ_i values.

We selected two metrics to characterize the performance of the studied algorithms. The first metric compares, experiment by experiment, the number of times an algorithm achieves the best $C_{\max}$. This comparison is fair since the experiment settings used to compute two makespan values are the same. The best $C_{\max}$ value allows to understand on which set of input values an algorithm behaves the best. In a similar manner we can compute the second and third best makespan. This metric however does not allow to quantify the distance between the algorithms.

To compare the distance between two heuristics on a same data set, we define a metric called NM for normalized metric. This metric takes the heuristic makespan and the best computed makespan and normalizes their difference to the power envelope size. $NM = \sum out \div \sum total$: we calculate the energy, $p_i \times \varphi_i$, of all the tasks executed after the best $C_{\max}$ ($\sum out$), and divide it by the total area of the set of tasks ($\sum total$). This metric is fair because all heuristics are compared to each other on the same data set and the result is normalized on both the processing time and the power need as it uses the power surface. The NM value allows to compute a qualitative result given by an algorithm since it gives a distance to the best known value. Note however, that this metric cannot be used for the fitness in the genetic algorithm as it is based on the best schedule which can only be computed once all the algorithms are run. We take the average of this metric for each heuristic over 150 execution. The lower the better.

Minimizing the sum of the completion times of the tasks, the flowtime, leads to minimize the mean finish time. We selected this metric since it fosters the fair share of the resources between users. To remove the dependency of the classic flowtime metric from input values, we define a normalized value $PERFLOW$ to evaluate the total flowtime as $PERFLOW = (\sum(C_i - useless_i))/\sum p_i$, where C_i is the completion time of task T_i and $useless_i$ is the sum of the lengths of the intervals, between 0 and C_i, where no task can be scheduled because there is not enough available power.

6.5 Results

In this section we present the results of our experiments. The whole computations took around 80 000 hours on the local computing center, mainly because the heatmaps and the genetic algorithms are very

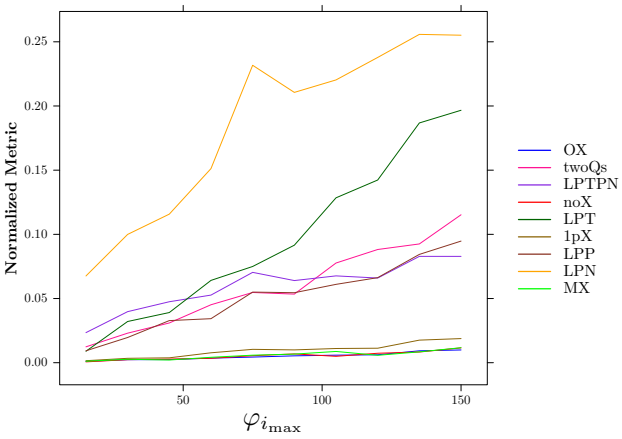


Figure 12: Average NM for experiment 6.5.1

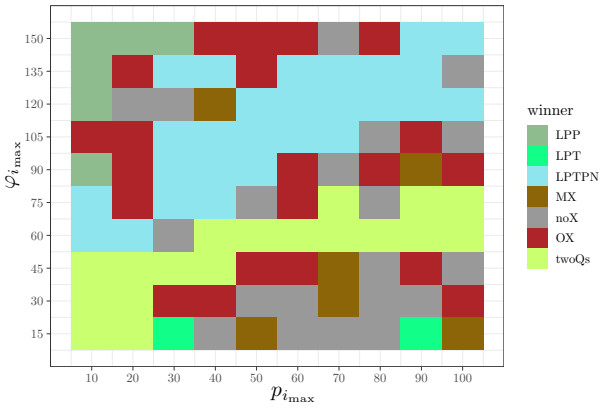


Figure 13: Best average NM for experiment 6.5.2

time consuming. The figures are generated using the R statistical environment.

6.5.1 Feitelson tasks and realistic power envelopes

In our first experiment we aim to reproduce real-world conditions. For this purpose we combine Feitelson model based tasks which emulate real workload traces in terms of processing times with realistic power envelopes. We use the hyper-gamma distribution of the Feitelson model (see Section 6.1) to generate 10 groups of 150 task lists with 50 tasks each. Between the 10 groups the p_i values of a task list are the same while $\varphi_{i_{\max}}$ ranges between 15 and 150 power units by steps of 15. The same 150 lists of 10 000 intervals generated using the realistic model were used across all ten experimental setups.

Figure 12 presents the average NM evaluation metric from 150 executions for each heuristic over 10 experimental setups. Note that we have removed the SPT heuristics to make the figure more readable since it gives poor results that flatten the rest of the curves. From the figure we can say that in general genetic algorithms perform better than list based algorithms. Between the list algorithms, we notice that the policy that orders the task list based on the least possible places for each task along with the policies that take in consideration both the processing time p_i and the power consumption φ_i of tasks like $LPTPN$ and *twoQs* outperform the policies that consider only one priority criteria such as LPT and LPN.

6.5.2 Exponential tasks and realistic power envelope

In order to test the versatility and the limitations of our heuristics, in this experiment we use a synthetic workload that is scheduled in a realistic set of intervals. We use this synthetic workload to explore the heuristics behavior on a broader set of tasks and, in particular, the heuristics performance depending on the tasks properties. Here $p_{i_{\max}}$ and $\varphi_{i_{\max}}$ vary between $10 \rightarrow 100$ and $15 \rightarrow 150$ with steps of 10 and 15 respectively. 100 combinations of $(p_{i_{\max}}, \varphi_{i_{\max}})$ couples are therefore tested, each one is represented as a square in Figure 13, for each combination 150 task lists are created, each task list contains 100 tasks. The maximum p_i and φ_i values that can occur in a task list are limited to the corresponding $(p_{i_{\max}}, \varphi_{i_{\max}})$ couple. The same 150 lists of 10 000 intervals generated using the realistic model were used across all 100 squares.

Since the results of the experiment are three dimensional, two input values (p_i and φ_i) and one result value (the best heuristics), we use a heatmap to represent them where the color of the square corre-

spond to the winner heuristics. Each square in Figure 13 represents the average NM over 150 executions at the corresponding $p_{i_{\max}}$ and $\varphi_{i_{\max}}$. We notice that, as the processing time of the tasks increases (higher $p_{i_{\max}}$ or towards the right on the heatmap), the more squares are won by genetic algorithms compared to list based algorithms, and that, as the power consumption of the tasks increases (higher $\varphi_{i_{\max}}$ or towards the top on the heatmap), a list based algorithm that considers both the time and power dimensions of tasks gives the best solution in most cases, while twoQs that gives an edge to the time dimension performs better when the power consumption of the tasks becomes lower. We also notice that LPP wins in the upper left corner of the heatmap (low $p_{i_{\max}}$ and high $\varphi_{i_{\max}}$) where tasks have high average power consumption, yet their short processing times do not put them in high priority using $LPTPN$. LPP favors those tasks that are harder to place regardless of their dimensions.

Algorithm	noX	OX	MX	1pX
Time (s)	1415.25	1714.32	1737.58	723.71
Algorithm	LPT	LPN	LPP	twoQs
Time (s)	0.099	0.093	13.67	0.126

Table 2: Average computation times in experiment 6.5.2 (sec)

Table 2 gives the average computation times for the heuristics. We see that the GA based heuristics take more than 1000 times to calculate the schedules than the simpler heuristics. Yet they do not outperform the much faster and much simpler list based algorithms. This is due to the fact that the fitness value used in GAs is $C_{\max}$, therefore, GAs focus on optimizing the solution on the time dimension only. This hypothesis is enforced by Figure 14. Yet some list algorithms are still able to provide better NM values because NM is a two dimensional evaluation metric. Recall that NM cannot be used as the fitness value in GA because it cannot be calculated until all other algorithms are run.

Figure 14 shows the number of times a heuristic finds the best $C_{\max}$ in the 150 executions. We can see that all the tested genetic algorithms find the best $C_{\max}$ much more often than the list algorithms although they do not win every case in the heatmap. This is explained by their fitness value that targets $C_{\max}$ instead of NM as previously explained. Note that the best solution can be found by several algorithms, which explains why the left right square (simple cases with small p_i and φ_i values) is red for almost all the heuristics. We calculated as well how many times each algorithm found the second and the third best C_{max} (not presented in the paper) and the results confirm that the genetic algorithm always finds shorter makespans than list based algorithms.

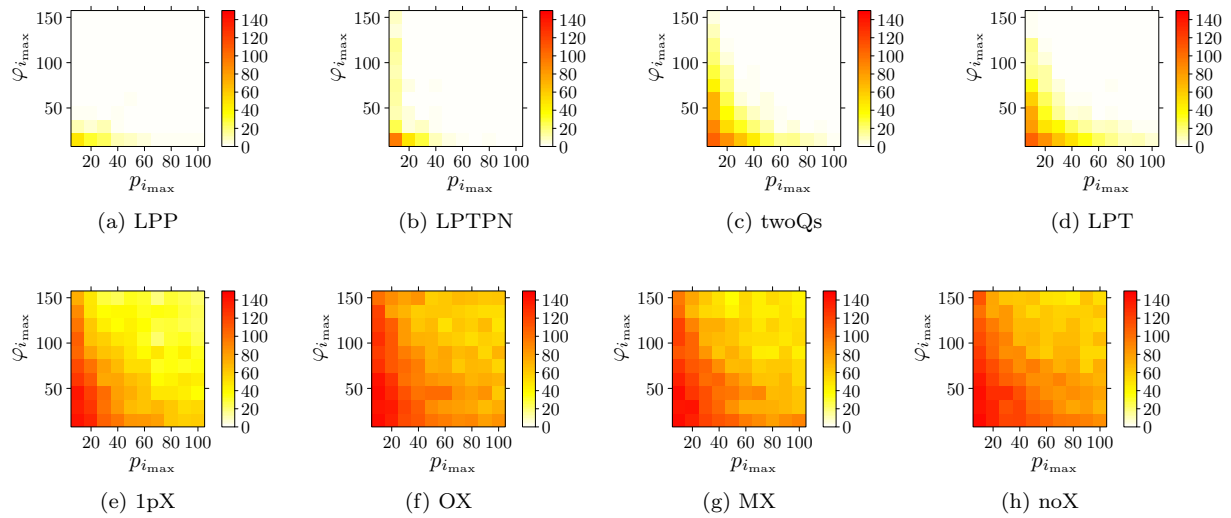
Our last experiment with this set of data assesses the heuristics performance regarding the total flowtime. All the runs are won by the SPT heuristics and we therefore do not present a heatmap but rather present in Figure 15 the distance of each list algorithm from the best $PERFLOW$.

As part of this experiment, we modify the fitness function in GA to consider $\sum C_i$ instead of C_{max} and we use as seeds solutions from heuristics that proved in previous experiments that they give better $PERFLOW$ results than others such as SPT. Our results show that the tested GA indeed improves the initial seed solution provided by SPT in all the tested cases. Figure 16 presents how far GA was able to improve the SPT average PERFLOW, we notice that the bigger the tasks get on both time and power axes, the less efficient SPT gets compared to GA.

6.5.3 Tasks From Intervals and realistic power envelopes

In this experiment we generate 150 lists of 10 000 intervals generated using the realistic model. Similarly to the previous experimental settings, both $p_{i_{\max}}$ and $\varphi_{i_{\max}}$ vary between $10 \rightarrow 100$ and $15 \rightarrow 150$ with steps of 10 and 15 respectively. 100 combinations of $(p_{i_{\max}}, \varphi_{i_{\max}})$ couples are therefore tested. For each combination 150 task lists are generated from the 150 realistic interval lists. The aim of this experiment is to create a special case of datasets in which the optimal solution is known in order to calculate how far the solutions found by the proposed heuristics are from the optimal.

Figure 17 presents the average distance from the



(a) LPP (b) LPTPN (c) twoQs (d) LPT

(e) 1pX (f) OX (g) MX (h) noX

Figure 14: Number of times each algorithm finds the best $C_{\max}$

Algorithm	noX	OX	MX	1pX
Time (s)	774.75	954.59	960.41	400.80
Algorithm	LPT	LPN	LPP	twoQs
Time (s)	0.017	0.015	3.39	0.039

Table 3: Average computation times in experiment 6.5.3 (sec)

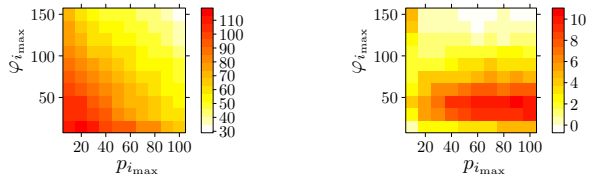
optimal over 150 executions for each heuristic using a unified scale to make the comparison clearer. We note that all GAs are closer to the optimal than the list based heuristics. This is however at a cost of much higher computation times as shown in Table 3. On the other hand the simple *twoQs* and *LPT* heuristics give close to optimal results, with less than 5% of difference in the cases where the power need φ_i is small and in most cases the difference is under 10% which is quite good.

Figure 18 shows the NM results for the heuristics in this experiment. Each square represents the average NM over 150 executions at the corresponding ($p_{i_{\max}}$ and $\varphi_{i_{\max}}$). From this figure we can see that in most cases the genetic algorithms outperform the list based algorithms for this experiment, except for some cases where *twoQs* finds better average NM.

6.5.4 Feitelson tasks and Intervals From Tasks

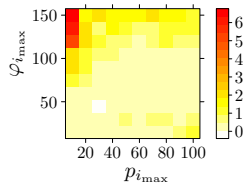
This experiment is designed to create another special case of datasets in which the optimal solution is known in order to calculate how far the solutions found by the proposed heuristics are from the optimal. We use the same 10 groups of 150 task lists generated in the experiment presented in Section 6.5.1 to generate 10 groups of 150 lists of intervals. Each interval list represents a power envelope under which the area (time times power) is equal to the area represented by the ensemble of the task list used to generate it $\sum p_i \times \varphi_i$ for $T_i \in T$

Figure 19 presents the average distance from the optimal from 150 executions for each heuristic over 10 experimental setups. We can note that the distance to the optimal value is larger in this experiment than in the previous one. This probably means that the optimal solution is more difficult to find in this case. This also shows that, in particular cases, the best algorithms are not able to find solutions closer than about 18% of the optimal one. Similar to the previ-

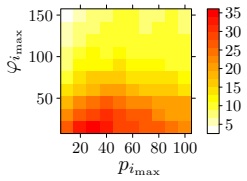


(a) LPP

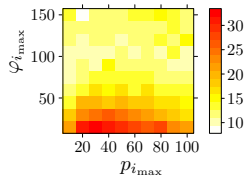
(b) LPTPN



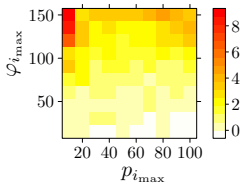
(c) twoQs



(d) LPN



(e) Random



(f) LPT

Figure 15: *PERFLOW* distance from the best *PERFLOW*

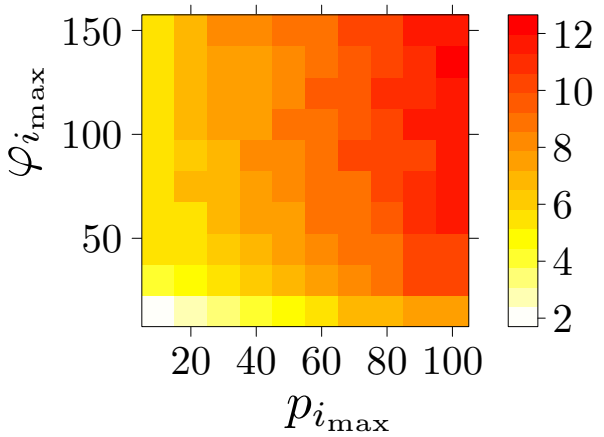
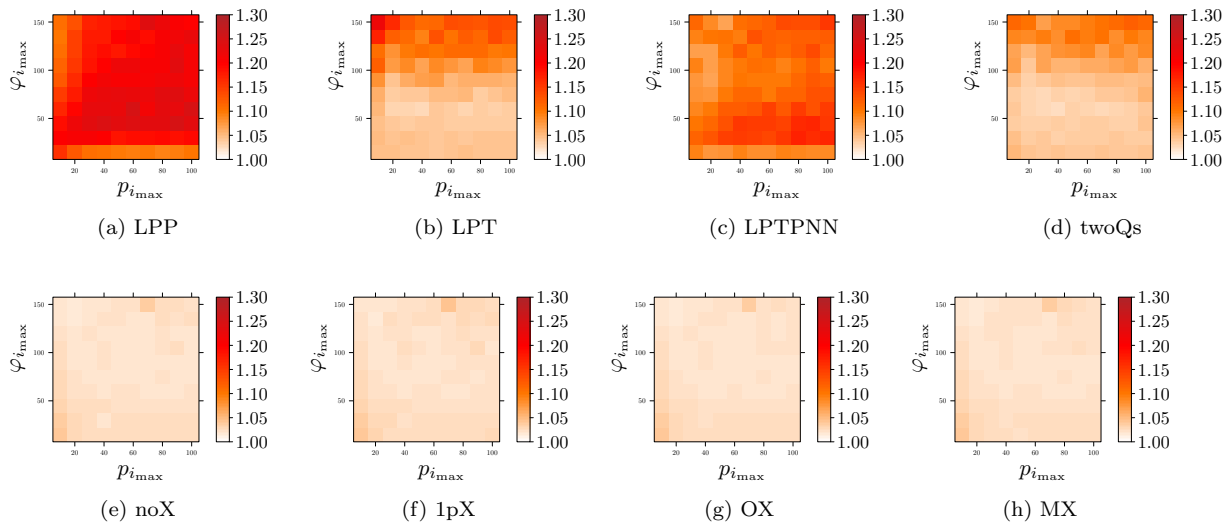


Figure 16: Average *PERFLOW* distance Between SPT and GA.

ous experiment, these best solutions come from the genetic algorithms. The difference between the list based heuristics and the genetic algorithms is about the same compared to the previous experiment, but it stays more stable when the power need varies.

Globally the results of these experiments are that the power dimension which is added to the classical scheduling problem makes the search for a good solution harder. This can be seen as the performance of the list based algorithms, in particular the one of *LPT* that usually has good performance on this problem, degrades when the power need increases. In the general case, where the tasks are generated from an exponential law, the list based algorithms behave better and the huge computation time taken by the genetic algorithms is not worth it. On the other hand, the findings of the experiments that consider particular cases (Feitelson, known optimal solution) show that genetic algorithms are closer to the optimal in these cases. This probably means that the heatmap contains some tricky cases which are better handled by a heuristics as *twoQs*. Finally, on all the realistic cases (Feitelson) the genetic algorithms allow to improve the schedule by 5 to 10 % at a cost of (very) long computation times. Moreover, the difference of computation times between both classes of



(a) LPP (b) LPT (c) LPTPNN (d) twoQs

(e) noX (f) 1pX (g) OX (h) MX

Figure 17: Distance from the optimal in experiment 6.5.3

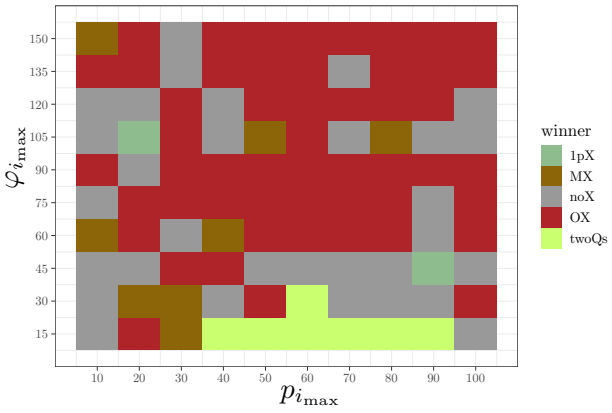


Figure 18: Best average NM for experiment 6.5.3

algorithms shows that there is only little interest in improving the genetic algorithm performance.

7 Conclusion

In this paper, we tackle the problem of scheduling the execution of a set of sequential tasks on a parallel machine powered solely by renewable energy sources.

The contribution of this work is a study on the problem of scheduling tasks on a parallel machine under power constraints. In particular we set the question of the theoretical complexity of the problem, of how to take the power constraints into account in the scheduling and of which scheduling algorithm is efficient.

To answer these questions, we provide formal complexity results on scheduling problems on one machine as well as on more general parallel problems. Our complexity results prove that the general case of both optimization problems of minimizing the makespan and the total flowtime is NP-Hard. Further, we propose new power aware scheduling heuristics that take power constraints into consideration and we conduct experiments based on simulations to

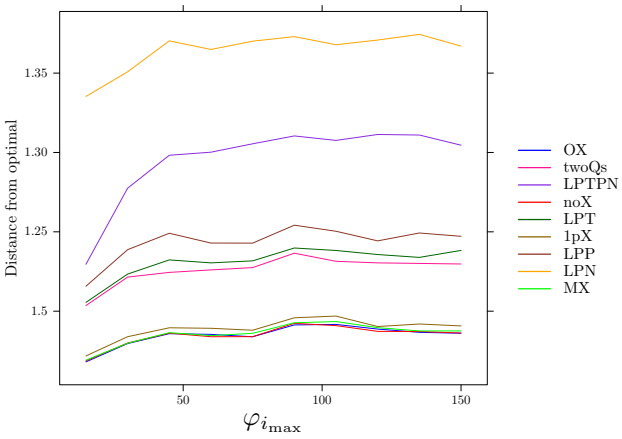


Figure 19: Distance from the optimal in experiment 6.5.4

evaluate and compare the performance of the proposed algorithms in different experimental settings.

The innovation of our experimental study is that we propose two experiments in a special case where the optimal solution is known. These experiments allow to calculate the distance of the proposed heuristics from the optimal solution. Whereas theoretically estimating the optimal solution in such complex optimization problems is a complicated and time consuming task.

Our results show that all tested GA configurations find shorter makespans than list based algorithms, between 5 to 10%, but are much more expensive in computations.

For future work, we are working on the multi-machine problem, with powering on/off machines, to takes execution platforms as clusters into consideration. We also work on the addition of batteries and their management policies to limit the loss of power in the intervals where no more tasks can be run. Several other directions will be explored in the future. A first direction could be to develop other algorithms, for instance using the results of [6]. Another is to implement the heuristics on a real platform to assess their behavior with real applications.

References

[1] Martin Arlitt, Cullen Bash, Sergey Blagodurov, Yuan Chen, Tom Christian, Daniel Gmach, Chris Hyser, Niru Kumari, Zhenhua Liu, Manish Marwah, et al. Towards the design and operation of net-zero energy data centers. In *Thermal and Thermomechanical Phenomena in Electronic Systems (ITherm), 2012 13th IEEE Intersociety Conference on*, pages 552–561. IEEE, 2012.

[2] Peter Brucker. *Scheduling Algorithms*. Springer Heidelberg, 2007.

[3] David P. Bunde. Power-aware scheduling for makespan and flow. *Journal of Scheduling*, 12(5):489–500, Oct 2009.

[4] Stephane Caux, Paul Renaud-Goud, Gustavo Rostirolla, and Patricia Stolf. It optimization for datacenters under renewable power constraint. In *European Conference on Parallel Processing*, pages 339–351. Springer, 2018.

[5] Stéphane Chrétien, Jean-Marc Nicod, Laurent Philippe, Veronika Rehn-Sonigo, and Lamiel Toch. Using a sparse promoting method in linear programming approximations to schedule parallel jobs. *Concurrency and Computation: Practice and Experience*, 27(14):3561–3586, 2015.

[6] Federico Della Croce and Rosario Scatamacchia. The longest processing time rule for identical parallel machines revisited. *Journal of Scheduling*, Dec 2018.

[7] Pierre-François Dutot, Yiannis Georgiou, David Glesser, Laurent Lefevre, Millian Poquet, and Issam Rais. Towards energy budget control in hpc. In *Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 381–390. IEEE Press, 2017.

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

[9] Y. Georgiou, D. Glesser, and D. Trystram. Adaptive resource and job management for limited power consumption. In *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*, pages 863–870, May 2015.

[10] Marco E. T. Gerards, Johann L. Hurink, and Philip K. F. Hölzenspies. A survey of offline algorithms for energy minimization under deadline constraints. *Journal of Scheduling*, 19(1):3–19, Feb 2016.

[11] Íñigo Goiri, Md E Haque, Kien Le, Ryan Beauchea, Thu D Nguyen, Jordi Guitart, Jordi Torres, and Ricardo Bianchini. Matching renewable energy supply and demand in green datacenters. *Ad Hoc Networks*, 25:520–534, 2015.

[12] R. L. Graham, E. L. Lawler, J. K. Lenstra, and A. H. G. Rinnooy Kan. Optimization and approximation in deterministic sequencing and scheduling: a survey. *Annals of discrete mathematics*, 5(2):287–326, 1979.

[13] Léo Grange, Georges Da Costa, and Patricia Stolf. Green it scheduling for data center powered with renewable energy. *Future Generation Computer Systems*, 86:99–120, 2018.

[14] Daniel Hackenberg, Robert Schöne, Thomas Ilsche, Daniel Molka, Joseph Schuchart, and Robin Geyer. An energy efficiency feature survey of the intel haswell processor. In *International Parallel and Distributed Processing Symposium Workshop (IPDPSW)*, pages 896–904. IEEE, 2015.

[15] M. Haddad, J/-M. Nicod, C. Varnier, and M.-C. Péra. Mixed integer linear programming approach to optimize the hybrid renewable energy system management for supplying a stand-alone data center. In *IEEE IGSC'19*, USA, oct 2019.

[16] Johannes Hofmann, Dietmar Fey, Jan Eitzinger, Georg Hager, and Gerhard Wellein. Analysis of Intel's Haswell Microarchitecture Using The ECM Model and Microbenchmarks. In *International Conference on Architecture of Computing Systems*, pages 210–222. Springer, 2016.

[17] Fredy Juarez, Jorge Ejarque, and Rosa M Badia. Dynamic energy-aware scheduling for parallel task-based application in cloud computing. *Future Generation Computer Systems*, 78:257–271, 2018.

[18] A. Kassab, J. M. Nicod, L. Philippe, and V. Rehn-Sonigo. Scheduling independent tasks in parallel under power constraints. In *46th International Conference on Parallel Processing (ICPP)*, pages 543–552, July 2017.

[19] Ayham Kassab, Jean-Marc Nicod, Laurent Philippe, and Veronika Rehn-Sonigo. Assessing the use of genetic algorithms to schedule independent tasks under power constraints. In *2018 International Conference on High Performance Computing & Simulation (HPCS)*, pages 252–259. IEEE, 2018.

[20] Tarandeep Kaur and Inderveer Chana. Energy efficiency techniques in cloud computing: A survey and taxonomy. *ACM Comput. Surv.*, 48(2):22:1–22:46, October 2015.

[21] Bithika Khargharia, Salim Hariri, Ferenc Szidarovszky, Manal Houri, Hesham El-Rewini, Samee Ullah Khan, Ishfaq Ahmad, and Mazin S Yousif. Autonomic power & performance management for large-scale data centers. In *2007 IEEE International Parallel and Distributed Processing Symposium*, pages 1–8. IEEE, 2007.

[22] George Kyriakarakos, Dimitrios D Piromalis, Konstantinos G Arvanitis, Anastasios I Dounis, and George Papadakis. On batteryless autonomous polygeneration microgrids: Investigation of the combined hybrid capacitors/hydrogen alternative. *Energy Conversion and Management*, 91:405–415, 2015.

[23] Tak-Wah Lam, Lap-Kei Lee, Isaac K. K. To, and Prudence W. H. Wong. Improved multiprocessor scheduling for flow time and energy. *Journal of Scheduling*, 15(1):105–116, Feb 2012.

[24] Hongtao Lei, Rui Wang, Tao Zhang, Yajie Liu, and Yabing Zha. A multi-objective

co-evolutionary algorithm for energy-efficient scheduling on a green data center. *Computers & Op. Research*, 75:103–117, 2016.

[25] Uri Lublin and Dror G. Feitelson. The workload on parallel supercomputers: modeling the characteristics of rigid jobs. *J. Parallel Distrib. Comput.*, 63(11):1105–1122, 2003.

[26] Anne-Cecile Orgerie, Marcos Dias de Assuncao, and Laurent Lefevre. A survey on techniques for improving the energy efficiency of large-scale distributed systems. *ACM Comput. Surv.*, 46(4):47:1–47:31, March 2014.

[27] Eduard Oró, Victor Depoorter, Albert Garcia, and Jaume Salom. Energy efficiency and renewable energy integration in data centres. strategies and modelling review. *Renewable and Sustainable Energy Reviews*, 42:429–445, 2015.

[28] J. Pierson, G. Baudic, S. Caux, B. Celik, G. Da Costa, L. Grange, M. Haddad, J. Lecuivre, J. Nicod, L. Philippe, V. Rehn-Sonigo, R. Roche, G. Rostirolla, A. Sayah, P. Stolf, M. Thi, and C. Varnier. Datazero: Datacenter with zero emission and robust management using renewable energy. *IEEE Access*, 7:103209–103230, 2019.

[29] Issam Raïs, Anne-Cécile Orgerie, Martin Quinson, and Laurent Lefèvre. Quantifying the impact of shutdown techniques for energy-efficient data centers. *Concurrency and Computation: Practice and Experience*, page e4471, 2018.

[30] Dineshkumar Rajagopal, Daniele Tafani, Yiannis Georgiou, David Glesser, and Michael Ott. A novel approach for job scheduling optimizations under power cap for arm and intel hpc systems. In *High Performance Computing (HiPC), 2017 IEEE 24th International Conference on*, pages 142–151. IEEE, 2017.

[31] Sherief Reda, Ryan Cochran, and Ayse K Coskun. Adaptive power capping for servers with multithreaded workloads. *IEEE Micro*, 5(32):64–75, 2012.

[32] Hafiz Fahad Sheikh, Ishfaq Ahmad, and Dongrui Fan. An evolutionary technique for performance-energy-temperature optimized scheduling of parallel tasks on multi-core processors. *IEEE Transactions on Parallel and Distributed Systems*, 27(3):668–681, 2015.

[33] Hafiz Fahad Sheikh, Ishfaq Ahmad, Zhe Wang, and Sanjay Ranka. An overview and classification of thermal-aware scheduling techniques for multi-core processing systems. *Sustainable Computing: Informatics and Systems*, 2(3):151–169, 2012.

[34] Hafiz Fahad Sheikh, Hengxing Tan, Ishfaq Ahmad, Sanjay Ranka, and Phanisekhar Bv. Energy-and performance-aware scheduling of tasks on parallel and distributed systems. *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, 8(4):1–37, 2012.

[35] Akiyoshi Shioura, Natalia V. Shakhlevich, Vitaly A. Strusevich, and Bernhard Primas. Models and algorithms for energy-efficient scheduling with immediate start of jobs. *Journal of Scheduling*, 21(5):505–516, Oct 2018.

[36] Giorgio Luigi Valentini, Walter Lassonde, Samee Ullah Khan, Nasro Min-Allah, Sajjad A Madani, Juan Li, Limin Zhang, Lizhe Wang, Nasir Ghani, Joanna Kolodziej, et al. An overview of energy efficiency techniques in cluster computing systems. *Cluster Computing*, 16(1):3–15, 2013.

[37] Lizhe Wang, Samee U Khan, Dan Chen, Joanna KołOdziej, Rajiv Ranjan, Cheng-Zhong Xu, and Albert Zomaya. Energy-aware parallel task scheduling in a cluster. *Future Generation Computer Systems*, 29(7):1661–1670, 2013.

[38] Chia-Ming Wu, Ruay-Shiung Chang, and Hsin-Yu Chan. A green energy-efficient scheduling algorithm using the DVFS technique for cloud datacenters. *Future Generation Computer Systems*, 37:141 – 147, 2014.

[39] Liang Zhang, Tao Han, and Nirwan Ansari. Renewable energy-aware inter-datacenter virtual

machine migration over elastic optical networks. In *Cloud Computing Technology and Science (CloudCom), 2015 IEEE 7th International Conference on*, pages 440–443. IEEE, 2015.

A Others definitions used in the model

To schedule a task T_i we need to guarantee that T_i can be completed before the available power Φ_x becomes lower than its need φ_i. To be able to exhibit such time slots we define the set $\mathcal{E}_j(\varphi_i) = \{\mathcal{E}_{1,i}, \mathcal{E}_{2,i}, \ldots, \mathcal{E}_{K_i,i}\}$ of K_i eligible time slots. Let $b_{k,i}$ and $f_{k,i}$ be respectively the beginning of the slot $\mathcal{E}_{k,i}$ and its finish time. Then, for $\mathcal{E}_{k,i} = [b_{k,i}, f_{k,i}[$, the available power must be greater than φ_i, with $b_{k,i} \leq t < f_{k,i}$ and $\Phi_k(t) \geq \varphi_i$. Formally, it exists two integer values x and s such that the kth time slot $\mathcal{E}_{k,i}$ is defined by $\mathcal{E}_{k,i} = \Delta_x \cup \Delta_{x+1} \cup \ldots \cup \Delta_{x+s}$ $(x + s \leq X)$ where at any time $t \in \mathcal{E}_{k,i}$, $\Phi(t) \geq \varphi_i$ and at any time $t \in \Delta_{x-1}$ $(x > 1)$ or $t \in \Delta_{x+s+1}$ $(x + s + 1 \leq X)$ $\Phi(t) < \varphi_i$. So $b_{k,i} = \sum_{x'=1}^{x-1} \delta_{x'}$ and $f_{k,i} = \sum_{x'=x+s}^{X} \delta_{x'}$ (see Figure 3). If, considering already scheduled tasks, it remains enough time to perform task T_i in the duration $l_{k,i} = f_{k,i} - b_{k,i}$ that time slot is an option to run T_i. When a time slot is chosen to schedule a task, the corresponding power is subtracted from available power in the intervals that compose this time slot.

Finally, we consider an allocation function $A(i, j) = k$ that returns in which time interval $\mathcal{E}_{k,i}$ the task T_i is scheduled on core C_j. Let $\mathcal{T}_{k,j}$ be a subset of task set $\mathcal{T}$ that contains the tasks scheduled in the time slot $\mathcal{E}_{k,i}$ on C_j. For every task $T_i \in \mathcal{T}_{k,j}$, we set $A(i, j) = k$. Note that $\sum_{i|T_i \in \mathcal{T}_{k,j}} p_i \leq f_{k,i} - b_{k,i} = l_{k,i}$.

Note that, in order to make the reading more understandable, the index j is removed from previous notations for the one-machine problems, i.e., $\mathcal{E}_{k,j}$ becomes $\mathcal{E}_k$.

Table 4 summarizes the notations used in the theorem proofs.

variable	definition
$\mathcal{E}_j(\varphi_i)$	time slot: set of eligible time intervals
$\mathcal{E}_{k,i}$	kth eligible interval of $\mathcal{E}_j(\varphi_i)$
$b_{k,i}$	beginning of time slot $\mathcal{E}_{k,i}$
$f_{k,i}$	finish time of time slot $\mathcal{E}_{k,i}$
K	number of time slots in $\mathcal{E}_j(\varphi_i)$
$l_{k,i}$	length of time slot $\mathcal{E}_{k,i}$

Table 4: Summary of the notations

B Theorem proofs

B.1 Proof of theorem 1

Theorem 1 addresses the complexity of the makespan optimization on one machine for a set tasks that have different power consumptions and different processing times.

Theorem 1. *Minimizing the makespan of the schedule of a set of tasks* $(1|\varphi_i \leq \Phi_x|C_{\max})$ *to run in a set of intervals is NP-Hard in the strong sense if the tasks have different processing times* p_i*.*

Proof. First note that, in the case where all tasks need the same power to run $\varphi_i = \varphi$, a time interval Δ_x either provides enough power to run a task or not. The real amount of power provided during this interval is not important as it is just a binary question of enough power or not. The NP-Hardness of the makespan minimization problem will be demonstrated by proving first the problem where each task needs a power φ $(1|\varphi_i = \varphi \leq \Phi|C_{\max})$ to be executed.

Intuitively the proof is based on a set of time slots that provide enough power to run all the tasks, a priori disjoint (the time slots between them do not provide enough power). Then we define a set of tasks such that it is necessary to solve a 3-Partition problem to schedule all the tasks in these intervals.

Let us consider the following decision problem: given a time Z, is there a schedule where the last task is completed before Z ? We assume that the allocation respects the constraints of the problem, i.e., every task allocated to one time slot has enough time

to be completed before the end of the slot and the power available into this time slot is greater or equal to the sum of power needed by the tasks scheduled in the time slot.

The problem is in NP: given a schedule it is easy to check in polynomial time whether it is valid or not before the time Z. The NP-Completeness is obtained by reduction from 3-PARTITION [8] which is NP-Complete in the strong sense.

Let us consider an instance $\mathcal{I}_1$ of 3-PARTITION: given an integer B and $3H$ positive integers $a_1, a_2, \ldots, a_{3H}$ such that for all $i \in \{1, \ldots, 3H\}$, $B/4 < a_i < B/2$ and with $\sum_{i=1}^{3H} a_i = HB$, does it exists a partition $I_1, \ldots, I_H$ of $\{1, \ldots, 3H\}$ such that for all $h \in \{1, \ldots, H\}$, $|I_h| = 3$ and $\sum_{i \in I_h} a_i = B$?

We build the following instance $\mathcal{I}_2$ of our problem with $\mathcal{E}(\varphi) = \{\mathcal{E}_1, \mathcal{E}_2, \ldots, \mathcal{E}_{2H-1}\} = \mathcal{E}_{odd} \cup \mathcal{E}_{even}$ $2H-1$ set of time slots $\mathcal{E}_x$ such that $\forall t \in \mathcal{E}_x$, $\Phi_x = \varphi$ if x is odd or $\Phi_x = 0$ otherwise. Each time slot $\mathcal{E}_x$ has a length is equals to $f_x - b_x = l_x = B$. Thus, there are H times slot $\mathcal{E}_h \in \mathcal{E}_{odd}$ (i.e., $|\mathcal{E}_{odd}| = H$). There are $3H$ tasks $T_i \in \mathcal{T}$ such that each T_i needs a power of φ to be executed and its processing time is $p_i = a_i$ for all $1 \leq i \leq 3H = n$. The problem is to find a task to time slot assignment such that all the tasks can be run in the defined time slots such that the makespan is equals to $(2H-1)B$. Clearly, the size of $\mathcal{I}_2$ is polynomial in the size of $\mathcal{I}_1$. We now show that $\mathcal{I}_1$ has a solution if and only if $\mathcal{I}_2$ does.

Suppose first that $\mathcal{I}_1$ has a solution. For $1 \leq h \leq H$, task T_i is assigned to time slot $\mathcal{E}_h = [b_h, f_h[$ with $i \in I_h$ within the period and $p_i = a_i$. Then, with $A(i) = h$ meaning that task T_i is assigned to time slot $\mathcal{E}_h$, we have $\sum_{i|A(i)=h} p_i = l = \sum_{i \in I_h} a_i = B$ and therefore the constraint on the processing time is respected for the H slots. We have a solution to $\mathcal{I}_2$.

Suppose that $\mathcal{I}_2$ has a solution. Let $\mathcal{T}_h$ be the set of tasks allocated to the slot $\mathcal{E}_h$ (we recall that if $T_i \in \mathcal{T}_h$, $A(i) = h$) such that for all tasks $T_i \in \mathcal{T}_h$ with $i \in I_h$, $\sum_{i \in I_h} p_i = l = B$. Because of $p_i = a_i$, $|\mathcal{T}_h| = |I_h| = 3$. The length of the time slot l in which the available power is φ has to be fully filled for all H periods to be sure to complete the last task within the slot $\mathcal{E}_H = [b_H, f_H[$ at time $t = f_H = Z$. Otherwise, an other slot has to be used to complete unprocessed tasks. Thus the solution is a 3-PARTITION and we have proven that the addressed decision problem is NP-Complete and thus minimizing the makespan $C_{\max}$ of a set of tasks with different processing times and the same power need to run on one machine is NP-Hard in the strong sense.

Since this problem $1|\varphi_i = \varphi \leq \Phi|C_{\max}$ is a special case of $1|\varphi_i \leq \Phi_x|C_{\max}$ it proves that this problem is also NP-Hard. This concludes the proof. □ □

B.2 Proof of Theorem 2

Theorem 2 addresses the complexity of the flowtime optimization on one machine for a set tasks that have different power consumptions and different processing times.

Theorem 2. *Optimizing the flowtime of the schedule of a set of tasks ($1|\varphi_i \leq \Phi_x| \sum C_i$) to run in a set of intervals is NP-Hard in the strong sense if the tasks have different processing times p_i.*

Proof. Let us consider the following decision problem: given a time Z is there a schedule where the sum of the task completion times is less than Z ? We assume that the allocation respects the constraints of the problem.

The problem is in NP: given a schedule, it is possible to confirm in polynomial time whether this schedule is valid or not and the sum of the task completion times is less than Z. The NP-Completeness is obtained by reduction from the $1|\varphi_i = \varphi \leq \Phi|C_{\max}$ problem that is proven NP-Complete in the strong sense in Theorem 1.

Intuitively, the proof is based on the definition of a set of disjoint intervals that all provide enough power to run the tasks. The last interval is so far from the previous one that, if we schedule a task in this interval, then the flowtime is always larger than if the tasks are scheduled in any order without using this last interval. On the other hand scheduling the tasks without using this last interval implies to solve a 3-PARTITION problem.

Let us consider an instance $\mathcal{I}_1$ of $1|\varphi_i = \varphi \leq \Phi|C_{\max}$ described within the paper: given $\mathcal{E}(\varphi) = \{\mathcal{E}_1, \mathcal{E}_2, \ldots, \mathcal{E}_H\}$ the set of H qualified time slots $\mathcal{E}_h$ to run tasks and whose length are all equal to

$f_h - b_h = l_h = l = B$ $(1 \leq h \leq H)$ and given $3H$ tasks $T_i \in \mathcal{T}$ such that each T_i needs the same power φ to be executed and its processing time is p_i for all $1 \leq i \leq 3H = n$ such that for all $i \in \{1, \ldots, 3H\}$, $B/4 < p_i < B/2$ and with $\sum_{i=1}^{3H} p_i = HB$. Does there exist a schedule $\mathcal{T}_1, \ldots, \mathcal{T}_H$ such that, for all $h \in \{1, \ldots, H\}$ and for all $T_i \in \mathcal{T}_h$, T_i is scheduled in $\mathcal{E}_h$ $(A(i) = h)$ and $C_{max} = f_H$? Obviously, $|\mathcal{T}_h| = 3$ with $1 \leq h \leq H$ considering p_i.

We build the following instance $\mathcal{I}_2$ of the problem addressed in the beginning of the proof: $1|\varphi_i = \varphi \leq \Phi| \sum C_i$ with the set $\mathcal{E}'(\varphi) = \mathcal{E}(\varphi) \cup \mathcal{E}_{H+1}$ of $H+1$ qualified time slots ($\mathcal{E}$ described for $\mathcal{I}_1$), the same set $\mathcal{T}$ of $3H = n$ tasks T_i with $1 \leq i \leq n = 3H$. $\mathcal{E}_{H+1}$ is defined as a valid time slot ($\varphi \leq \Phi(t)$ with $b_{H+1} \leq t < f_{H+1}$) such that $b_{H+1} = n \times f_H$. Considering this problem instance, does there exist a schedule with $Z = n \times f_H$?

The size of $\mathcal{I}_2$ is polynomial in the size of $\mathcal{I}_1$. Let us show now that $\mathcal{I}_1$ has a solution if and only if $\mathcal{I}_2$ does.

Suppose first that $\mathcal{I}_1$ has a solution. For $1 \leq h \leq H$, task T_i is assigned to time slot $\mathcal{E}_h = [b_h, f_h[$ if $T_i \in \mathcal{T}_h$. Then, we have $\sum_{i|T_i \in \mathcal{T}_h} p_i = l = B$ and therefore the constraint on the processing time is respected for the H slots and $|\mathcal{T}_h| = 3$. Considering the schedule given by $\mathcal{I}_1$, it is possible to minimize the flowtime within $\mathcal{E}_h$ ($F_h = \sum_{i|T_i \in \mathcal{T}_h} (C_i - b_h)$ with C_i the completion time of T_i) by sorting the 3 tasks by increasing processing time order. Then each time slot $\mathcal{E}_h$ has its own flowtime F_h. As $f_h - b_h = l_h = l = B$ for all $1 \leq h \leq H$, it is possible to exchange task allocations from one time slot $\mathcal{E}_{h1}$ to another time slot $\mathcal{E}_{h2}$ ($h1 \neq h2$ and $1 \leq h1, h2 \leq H$) without changing the value of the makespan. Consequently, by sorting F_h in increasing order and by reallocating tasks $T_i \in \mathcal{T}_h$ to the right time slot regarding its rank given by the sort, the obtained flowtime for the whole task set is the smallest possible. We have a solution to $\mathcal{I}_2$.

Suppose now $\mathcal{I}_2$ has a solution. If the flowtime of the schedule is less than $Z = n \times f_n$, $\mathcal{T}_{H+1} = \emptyset$, otherwise since $b_{H+1} = n \times f_n$, if one task T_i is in $\mathcal{T}_{H+1}$, the flowtime is not able to be less than $n \times f_n$ because the completion time of T_i is at least $C_i = b_{H+1} + p_i = n \times f_n + p_i$ which is greater than Z. Thus, all tasks are scheduled within $\mathcal{E}$. Since $\sum_{i|T_i \in \mathcal{T}} p_i = HB$ and since $f_h - b_h = l = B$ for all $1 \leq h \leq H$ and $|\mathcal{E}| = H$, the completion time of the last task is $f_H = C_{max}$. We have a solution to $\mathcal{I}_1$.

By using the same valid arguments than within the proof of Theorem 1, we can confirm that we have proven that minimizing the flowtime of scheduling a set of tasks with different processing times which need the same amount of power φ to be performed on one machine is NP-Complete in the strong sense.

Since this problem $1|\varphi_i = \varphi \leq \Phi| \sum C_i$ is a special case of $1|\varphi_i \leq \Phi| \sum C_i$, it is sufficient to prove the NP-Completeness of $1|\varphi_i \leq \Phi| \sum C_i$. This concludes the proof. □ □

B.3 Proof of Theorem 3

Theorem 3 states that the LWRT Algorithm is optimal.

Theorem 3. *The LWRT Algorithm gives an optimal solution for the* $1|\varphi_i \leq \Phi_x, pmtn|C_{\max}$ *problem.*

Proof. The optimally of the LWRT algorithm is demonstrated by contradiction.

We consider that an optimal schedule S^* does not always run LWRT at each interval, starting from $t = 0$. We assume that interval Δ_x is the first interval such that it includes task T_i $(S^*(T_i) = t)$ which is not the LWRT task and such that T_i', the LWRT task, runs later $(S^*(T_i') = t', t' > t)$. As T_i is not the LWRT task, then we have $\Phi_x - \varphi_i > \Phi_x - \varphi_i'$ and $\varphi_i < \varphi_i' \leq \Phi_x$. Since the power consumed by T_i' is higher than the power consumed by T_i and since T_i' fits in interval Δ_x because it is the LWRT tasks for this interval, then we can swap T_i and T_i' (or at least part of them). Moreover, since T_i needs less power than T_i', it could be scheduled before t' in an interval that was not exploited by T_i' with more power. After this step the resulting schedule is at least the same but it could also have been improved by moving T_i before. This result is a contradiction with the assumption that S^* is optimal and, given any schedule, we can do better if we respect the LWRT order. Thus the LWRT algorithm gives an optimal schedule, which concludes the proof. □ □

B.4 Proof of Theorem 4

Theorem 4 states the complexity of the parallel problem.

Theorem 4. *Minimizing the makespan of the schedule of a set of power heterogeneous preemptive tasks to run in a set of intervals* $(P|\sum \varphi_i \leq \Phi_x, pmtn|C_{\max})$ *is NP-Hard in the strong sense.*

Proof. The NP-Hardness of this problem will be demonstrated by proving that the simpler problem where the processing time of each task is one unit of time (ut) is NP-hard in the strong sense. The remainder of the proof is build on a similar pattern as used within the proof of the theorem 1.

Let us consider the following decision problem: given a horizon of K intervals of time Δ_k $(1 \leq k \leq K)$ where their length δ_k is equal to one unit of time and where the available power is $\Phi(t) = \Phi_k = \Phi$ $(1 \leq k \leq K)$ and given 3 cores that share the available power, is there a schedule that allocates tasks over time such that the power needed by the cores never exceeds Φ for every time interval Δ_k $(1 \leq k \leq K)$? In other words, if $\mathcal{T}_k \subset \mathcal{T}$ is the set of tasks that are scheduled within the time interval Δ_k, $\forall k \leq K$, is $\sum_{i|T_i \in \mathcal{T}_k} \varphi_i \leq \Phi_k = \Phi$? The problem is in NP: given a schedule of K time intervals, it is easy to check in polynomial time whether this schedule is valid or not. The NP-Completeness is obtained by reduction from 3-PARTITION.

Intuitively, we define a set of disjoint unit time intervals that provide a same available power and 3 cores. Then we build a set of unit time tasks such that we must solve a 3-Partition problem to be able to schedule all the tasks in the intervals.

Let us consider an instance $\mathcal{I}_1$ of 3-PARTITION: given an integer B and $3K$ positive integers $a_1, a_2, \ldots, a_{3K}$ such that for all $i \in \{1, \ldots, 3K\}$, $B/4 < a_i < B/2$ and with $\sum_{i=1}^{K} a_i = KB$, does there exist a partition $I_1, \ldots, I_K$ of $\{1, \ldots, 3K\}$ such that for all $k \in \{1, \ldots, K\}$, $|I_k| = 3$ and $\sum_{i \in I_k} a_i = B$?

We build the following instance $\mathcal{I}_2$ of our problem with K time intervals, each interval Δ_k having a length of time $\delta_k = 1$ and with an available power $\Phi_k = \Phi = B$ for $1 \leq k \leq K$. There are $3K$ tasks T_i in $\mathcal{T}$ with $p_i = 1\,ut$ and $\varphi_i = a_i$ for all $1 \leq i \leq 3K = m$. Clearly, the size of $\mathcal{I}_2$ is polynomial in the size of $\mathcal{I}_1$. We now show that $\mathcal{I}_1$ has a solution if and only if $\mathcal{I}_2$ does.

Suppose first that $\mathcal{I}_1$ has a solution. For $1 \leq k \leq K$, task T_i is assigned to $\mathcal{T}_k$ within the period k with $i \in I_k$ and $\varphi_i = a_i$. Then, we have $\sum_{i|T_i \in \mathcal{T}_k} \varphi_i = \Phi_k = \sum_{i \in I_k} a_i = B$ and therefore the constraint on the demand is respected for the K time intervals. We have a solution to $\mathcal{I}_2$.

Suppose that $\mathcal{I}_2$ has a solution. Let $\mathcal{T}_k$ be the set of cores allocated to the period k such that for all tasks $T_i \in \mathcal{T}_k$ with $i \in I_k$, $\sum_{i \in I_k} \varphi_i = \Phi_k = \Phi = B$. Because of φ_i, $|\mathcal{T}_k| = |I_k| = 3$. Since the available power Φ has to be consumed for the K time intervals to process the scheduled tasks, the solution is a 3-PARTITION.

We have proven that the problem where $\Phi_k = \Phi$ for every time interval Δ_k $(1 \leq k \leq K)$ and $p_i = 1$ for every task $T_i \in \mathcal{T}$ $(1 \leq i \leq n)$ is NP-Complete in the strong sense. Since this problem is a special case of the more general problem where the available power Φ_k during each of the time intervals Δ_k is different from each other and where the processing time p_i of each task T_i is also different from each other, it is sufficient to prove the NP-Hardness of this associated general optimization problem. This concludes the proof. □ □