Optimization of parallel scheduling and the use of renewable energy sources to power computing centers

Ayham Kassab

UBFC
UNIVERSITÉ
BOURGOGNE FRANCHE-COMTÉ

UNIVERSITÉ DE FRANCHE-COMTÉ

THÈSE DE DOCTORAT DE L'ÉTABLISSEMENT
UNIVERSITÉ BOURGOGNE FRANCHE-COMTÉ
PREPARÉE À L'UNIVERSITÉ DE FRANCHE-COMTÉ

Ecole doctorale n°37

SPIM

Doctorat d' informatique

PAR

Ayham KASSAB

Optimization of parallel scheduling and the use of renewable energy sources to power computing centers

Thèse presentée et soutenue à Besançon, le 14/11/2019

Composition du Jury :

Frédéric Desprez, Directeur de recherche, INRIA - Président
Anne Benoît, Maître de conférence HDR, ENS de Lyon - Rapporteur
Anne-Cécile Orgerie, chargée de recherche, CNRS - Examinateur
Marie-Cécile Péra, Professeur, Université de Franche-Comté - Examinateur
Veronika REHN-SONIGO, Maître de conférence, Université de Franche-Comté - Co Direc-teur de thèse
Jean-Marc NICOD, Professeur, ENSMM - Co Directeur de thèse
Laurent PHILIPPE, Professeur, Université de Franche-Comté - Directeur de thèse

Contents

1 Introduction **4**
- 1.1 Context of the thesis . . . 4
- 1.2 Raised research questions . . . 5
- 1.3 Manuscript outline . . . 6

I Context **7**

2 Green Computing **9**
- 2.1 IT to reduce energy consumption . . . 9
  - 2.1.1 IT power consumption: measure versus estimate: . . . 9
  - 2.1.2 Running Average Power Limit (RAPL) . . . 11
  - 2.1.3 DVFS . . . 12
  - 2.1.4 On/Off . . . 13
  - 2.1.5 Power capping . . . 13
  - 2.1.6 Cooling . . . 14
- 2.2 Green ICT . . . 15

3 Scheduling in green computing **17**
- 3.1 Parallel scheduling . . . 17
- 3.2 Resource constrained scheduling . . . 18
  - 3.2.1 Resource constrained project scheduling problem (RCPSP) . . . 19
  - 3.2.2 Continuous resources or Continuously divisible resources . . . 20
- 3.3 Energy-aware scheduling . . . 21
  - 3.3.1 Workload consolidation . . . 21
  - 3.3.2 Load balancing and thermal aware scheduling . . . 22
  - 3.3.3 Combined IT and Scheduling to reduce energy consumption . . . 23
- 3.4 Scheduling in green ICT . . . 24
  - 3.4.1 Geographically distributed data centers with on-site renewable sources . . . 24
  - 3.4.2 Green power aware scheduling . . . 25

II Contribution **26**

4 Problem formalization **28**
- 4.1 Task model . . . 29

4.2 Machine model . . . 29
4.2.1 Multi-core . . . 30
4.2.2 Multi-machine . . . 30
4.3 Power model . . . 31
4.4 Scheduling model . . . 31
4.5 Notations and objectives . . . 33

5 Complexity study **35**
5.1 Introduction . . . 35
5.2 One Machine Problems . . . 36
5.2.1 Problems without preemption . . . 36
5.2.2 Problems with preemption . . . 39
5.3 Parallel Problems . . . 41
5.4 Synthesis . . . 42

6 Scheduling strategies **44**
6.1 Place task . . . 44
6.1.1 *PlaceTaskCore* . . . 45
6.1.2 *PlaceTaskMachine* . . . 48
6.2 Planner heuristics . . . 52
6.2.1 List scheduling algorithm . . . 52
6.2.2 Genetic algorithm GA . . . 54
6.3 Power envelope partition based algorithms . . . 60
6.3.1 Best fit binary search . . . 61
6.3.2 One Processor On Binary search (OPOBS) . . . 62
6.3.3 Stripe . . . 65

III Experiments **66**

7 Multi-core platform experiment **68**
7.1 Data generation . . . 68
7.1.1 Task models . . . 68
7.1.2 Power interval model . . . 69
7.2 Evaluation metrics . . . 71
7.3 Simulator . . . 72
7.4 List scheduling versus binary search experiment . . . 74
7.4.1 Experimental settings . . . 74
7.4.2 Results . . . 75
7.4.3 Summary . . . 80
7.5 Assessing the use of a genetic algorithm experiment . . . 80
7.5.1 Experimental settings . . . 80
7.5.2 Results . . . 81
7.5.3 Summary . . . 84
7.6 Realistic power envelope model experiment . . . 84
7.6.1 Experimental settings . . . 84

7.6.2 Results . . . 85
7.6.3 Summary . . . 93

8 Multi-machine platform experiment **94**
8.1 Experimental settings and results . . . 94
8.1.1 Hyper-gamma tasks and realistic power envelopes . . . 95
8.1.2 Exponential tasks and realistic power envelope . . . 109
8.1.3 Summary . . . 113
8.2 Experiments synthesis . . . 114

IV Conclusion and perspectives **115**

9 Conclusion **116**
9.1 Evaluation . . . 117
9.2 Perspectives . . . 117
9.3 Publications . . . 118
9.3.1 Published or Accepted . . . 118
9.3.2 Under review . . . 119

Appendices **120**
9.A Useful definitions . . . 120
9.A.1 PUE . . . 120
9.A.2 Time/cost trade-off problem (TCTP) . . . 120
9.A.3 Payment scheduling problem (PSP) . . . 121
9.A.4 Partially elastic schedule . . . 121
9.A.5 Energetic reasoning . . . 121
9.A.6 left-shift/right-shift . . . 121
9.A.7 Multiple mode . . . 122

List of Figures **123**

List of Tables **125**

Chapter 1

Introduction

1.1 Context of the thesis

The growing dependency on Information and Communication Technology (ICT) services is pushing service providers to build bigger and more energy hungry ICT structures in order to maintain the quality of their services. Studies show that in 2015 ICT was responsible for about 4% of the global electricity consumption [2], this percentage is estimated to grow up to 20% by 2030 [1]. Besides the economic costs of this high consumption, environmental concerns are rising regarding the carbon footprint of this high consumption. Worst case scenario estimations indicate that ICT can be responsible for almost a quarter of the CO_2 emissions worldwide in 2030 [14], these concerns underline the necessity of greener computing to reduce ICT's greenhouse gas emissions. Establishing the Green500 [2] list which ranks High Performance Computing (HPC) systems according to their power efficiency [120] is an example of the increasing global awareness of the issue.

As a result, the energy efficiency of large scale ICT structures has been the focus of many research efforts over the last few years. Most of these efforts focus on reducing the energy consumption of different components of the system, such as using more energy efficient processors or more efficient cooling systems. However, deploying these solutions still allows the system to consume brown energy. Another "greener" approach consists of using renewable energy sources in the system's power supply, such as solar panels, wind turbines or fuel cells as these devices do not produce CO_2. The latter approach provides a more direct effect on the objective of reducing the carbon footprint.

Renewable sources can be used either by signing green energy contracts with an electricity provider or by building on-site renewable sources. The first scenario does not present a problem, as the electricity provider guarantees a constant level of power supply through the grid, the used electricity in this case however is a mixture of brown and green energy. We focus thus on the second scenario as it guarantees an exclusively green energy supply.

Using on-site renewable energy sources however presents a challenge, the power production level of most renewable sources is intermittent and it varies over time. Furthermore, the rate of computational demands submitted to the system varies over time as well, which means that at certain times extra power is produced with no computations demanding it, and at other times the submitted computational demands cannot be carried out due to lack of power production. Energy storage devices can be used to reduce the presence of such situations. This solution can be costly

[1] https://theshiftproject.org

[2] https://www.top500.org/green500/

and it requires a comprehensive analysis of the entire system to properly integrate such devices. Another solution consists of deploying workload management techniques for optimal utilization of the instantaneous available green power.

This thesis is supported in part by the DECALCO project and the ANR DATAZERO [3] project. The DATAZERO project presents a comprehensive study on how to efficiently design and operate a data center that is powered entirely by renewable energy sources. The study covers various aspects of the system such as power supply and energy storage management, workload management, networks, etc. It also introduces a negotiation infrastructure to make the IT needs and the power supply match. The DECALCO project addresses the problem of a HPC system which is solely powered by renewable energy sources, this project focuses on the workload management level of the system.

As part of these two projects, this work tackles the optimization problem of scheduling tasks on a parallel computational platform within a predicted power envelope that varies over time. Our approach is different from traditional energy aware scheduling approaches in that it does not target energy minimization itself but it rather targets to better use the available power. The optimization problem is thus rather to limit the energy waste, i.e., the produced energy that cannot be used, than finding ways to decrease the consumed energy. We tackle on the one hand computing center oriented problems where the optimization objective is the makespan to finish a set of jobs as soon as possible and, on the other hand, data center oriented problems where the optimization objective is the flowtime to reduce the mean waiting time.

The presented contributions are as follows:

- We show that most scheduling problems that consider a variable limited power supply constraint are complex. We provide formal complexity results on scheduling problems on one machine and extend them to more general parallel problems.
- We propose scheduling heuristics that take power constraints into consideration.
- We develop a simulator script to perform an experimental study of the proposed heuristics. The simulator offers a wide range of data set generation methods. It covers both the shared memory machine and the distributed memory platform cases, by offering two different computational platform models, namely the multi-core platform and the multi-machine platform.
- We improve the multi-machine platform's scheduler to mange the process of switching on/off the machines.
- We provide simulation results obtained with the simulator. These results show that simple algorithms may generate good schedules that are only 5 to 10% further from the optimal than the schedules generated using much more time complex algorithms.

1.2 Raised research questions

This work aims to answer the following questions:

- How to run tasks on a parallel computational platform powered by green sources?
- What is the complexity of the problem of scheduling tasks on a HPC system under limited power constraint?

[3] http://datazero.org

- How to take the limited power constraint into account when scheduling tasks on a HPC system?
- What algorithm is efficient to generate a schedule for running tasks on a HPC system under limited power constraints?

1.3 Manuscript outline

The document is organized as follows: a literature review about green computing and its deployed methods on the IT equipment level is presented in Chapter 2. Scheduling solutions in green computing are discussed in Chapter 3. Both Chapters 2 and 3 are divided into two categories:

- Solutions that deal with reducing energy consumption.
- Solutions that deal with using renewable energy.

In the second part of this document we present our contributions. The problem formalization including the proposed models that represent different elements of the problem, in addition to the problem statement are presented in Chapter 4. Chapter 5 presents a formal complexity study on different scheduling problems that can be derived from our optimization problem, we show that using renewable power sources increases the complexity of the scheduling problem. Chapter 6 describes the scheduler and the scheduling strategies that we propose as a solution for the problem, these scheduling strategies represent the planner part of the system's scheduler, while the executer part of the scheduler is represented by a function designed to handle the tasks allocation process. The third part of this document presents the experimental study that we carry out to evaluate the proposed scheduling strategies. We develop a simulator that allows to run different experimental setups on two computational platform models, namely the multi-core platform in Chapter 7 and the multi-machine platform in Chapter 8. Finally, in the fourth part of this document, Chapter 9 presents our conclusion and answers the raised research questions, in addition to an evaluation of our solutions and the proposed perspectives on how this work can be carried on in the future.

Part I

Context

The increasing demand on information and communication technology (ICT) services is accompanied with an increasing awareness of their carbon footprint. Current estimations predict that if this issue is not addressed, ICT can be responsible for up to 23% of the global greenhouse gas emissions in 2030 [14].

These environmental risks have pushed many efforts over the course of the last few years to reduce the environmental impact of large scale ICT structures, including data centers, computational grids, clouds and high performance computing (HPC) systems. These efforts can be grouped under the term Green Computing. Green computing can be achieved either by deploying energy efficient computing techniques that aim to reduce the system's energy consumption or by using renewable energy sources, which has a more direct effect on the system's carbon footprint. For this reason, this state of the art is divided into two chapters, and each chapter is then divided into efforts that aim to reduce the energy consumption and efforts that aim to optimize the usage of renewable sources in ICT structures. Chapter 2 presents hardware level solutions that are deployed on the system's infrastructure including IT equipment and supporting systems. Chapter 3 presents software level solutions represented by scheduling strategies.

Chapter 2

Green Computing

This chapter presents hardware solutions that are deployed on the system's infrastructure including IT equipment and supporting systems. The first step towards green computing is to understand the energy consumption of an ICT system, and to try to optimize the energy efficiency of its components. Studies on how to measure, control and reduce the energy consumption of the IT components of a large scale ICT system are discussed in Section 2.1. Section 2.2 presents the research work that investigates hardware level solutions for integrating renewable energy sources in the system's infrastructure.

2.1 IT to reduce energy consumption

Due to the burst of big ICT structures in the last few years such as computational grids, data centers, cloud systems, and HPC centers, the energy efficiency of such structures and their great number of components are becoming an important issue. Therefore, many works focus on the consumption of different components of large scale ICT systems. For example, Gurantne et al. [65] argue that an efficient device consumes energy proportional to its output or utility, thus, an idle or lightly utilized PC or Ethernet link should not consume the same energy as one that is highly utilized. They propose several methods to reduce energy consumption of computers, Ethernet links, and LAN switches. Work about the energy consumption of different components of large scale ICT systems can be found in many surveys such as [94, 102, 80, 103, 80] that give a wide range of technologies and tools that are deployed at different levels of the system to reduce its energy consumption.

2.1.1 IT power consumption: measure versus estimate:

In order to enhance the energy efficiency of any system, we must first understand the energy consumption of its components. The power consumption of IT components in large scale ICT architectures can be measured either by using hardware components or by applying software estimations or modeling [21]. Using measuring hardware such as intelligent Power Distribution Units (ePDU) installed on each rack or in-built sensors provides accurate measurements. However, these equipments are expensive to install on a large scale. A cheaper and more flexible approach is to use software to measure the IT power consumption. `Powerstat` is a Linux tool to measure the power consumption of a machine using the battery stats and Intel's Running Average Power Limit RAPL interface [4]. Powerstat gives actual and statistical information about the system's power consumption.

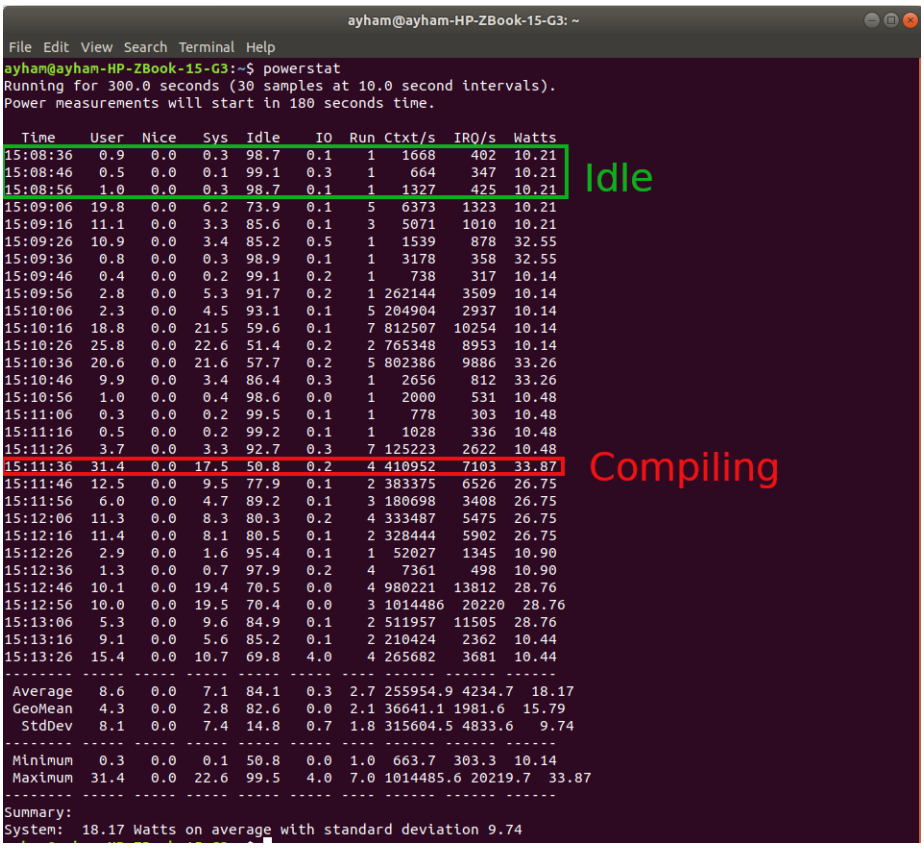


Figure 2.1 – `Powerstat` execution example

Figure 2.1 illustrates an example of one `Powerstat` run on a personal laptop. The first column from the right provides the power consumption in Watts at a 10 seconds step. We notice that the power drawn by the system is around 10 Watts when it is idle, but when executing a task on the CPU (the compilation of this manuscript for this test) the power consumption increases to about 30 Watts. Intel also provides `Powertop`, a similar tool that allows to identify applications with particular high power demands [5]. These tools only offer information about the processor's consumption, models on the other hand can be used to estimate the consumption of the entire system or some of its components. Several efforts have been carried out to model the IT power consumption [45, 37, 28, 123, 121]. The models in [45] and [37] are based on the CPU utilization. While in [123] a model is proposed to predict system-level power consumption starting from workload measures for a hybrid supercomputer. Prediction is obtained in two steps. The first step is to develop a relation between the power used by computing components and the power consumption of the whole system, including networking and IO system. The second step is to predict the power consumption of the computing components from the current workload data. The two steps are then combined to obtain the prediction of the system-level power starting from workload measures. Such system level power consumption models can be used in power-aware scheduling, where knowledge of the power consumption caused by the execution of a workload is necessary.

2.1.2 Running Average Power Limit (RAPL)

The power consumption of a processor is related to its frequency and to its supply voltage. Many tools have been developed over the years by processor chip makers to monitor and control these values on different levels, and for different purposes.

Processor chips are made of silicon and other semiconductors that can only operate within a certain range of temperature (risk of failure above a certain temperature). For this reason, to prevent IT components from overheating, manufacturers indicate the maximum amount of heat a component is expected to output when under load, so that the cooling system is designed accordingly. This value is called the thermal design power (TDP) [64]. A processor, however, can consume more power than its TDP for short durations without failure risks [71].

In single core processors, a job can utilize the processor's maximum computational capacity (maximum frequency) that does not exceed its TDP. Similarly, in dual core processors, a parallel job can utilize the maximum computational capacity of both cores. Running both cores on maximum frequency however can lead to TDP violation. Therefore, a lower maximum frequency limit is set for each core in such a way that the sum of those limits does not exceed the TDP. In the case of two cores for example, the maximum allowed frequency for each core would be only 50% of the maximum frequency of a single core processor. This way even when both cores are under full load, this will not risk overheating the chip. However, if one of the two cores is under higher workload than the other, it can operate at a higher frequency as long as the TDP allows it. This boost in performance above the declared maximum frequency is referred to as turbo mode. Deciding whether it is safe to run a processor's core on turbo mode or not depends on many factors, such as the processor's temperature and the number of active cores. In modern processors, this decision is carried out by a set of integrated power meters and registers. This configuration is referred to as the Running Average Power Limit (RAPL) interface. RAPL is a useful tool to acquire information about the processor's both power and energy consumption [44].

Several research works exploit RAPL readings to estimate the power consumption of a computation system [82, 44, 73, 66, 123, 127]. [82] for example presents a model for the server's total power

consumption based RAPL readings. [127] investigates in using RAPL to achieve energy-proportional operation for a server workload.

Getting an accurate estimation of the server's power consumption is useful for applying energy efficient techniques such as setting a power cap for better utilization of pricing variations, setting a power budget for the system, and managing the workload to match the system's consumption with a power constraint.

After understanding the energy and power consumption of ICT systems and their components, we now address several techniques that can be deployed to reduce this consumption. Most research efforts in this area focus on the processor's power and energy consumption, as it is one of the main sources for power consumption in a system, around 40% of the system's total power consumption is drawn by the processor [45]. As the processor's power consumption is subject to its frequency and its supply voltage, the first solution that comes to mind is to lower these two values.

2.1.3 DVFS

Dynamic voltage and frequency scaling (DVFS) is one of the most common methods used to reduce the processor's power consumption [45], it consists of dynamically setting the supply voltage and the frequency of a processor to one of its possible operational voltage and frequency levels, based on the requirements of the current workload. The different levels of voltage/frequency a processor can operate at while executing a job are called the performance states (**P-State**). P_0 expresses the combination of maximum possible voltage and frequency, the voltage and frequency in P_1 are less than in P_0 and in P_2 are less than inP_1, and so on.

Many works exploit DVFS to reduce the energy consumption of large scale ICT structures [56, 135, 83, 55, 134, 131, 140]. A processor's power consumption is decomposed into static and dynamic power consumption, where the static power is the power needed to run the processor when it is idle and the dynamic power is the extra power consumed by job execution. The dynamic power consumption of a processor is subject to its supply voltage V_{DD} and the frequency f [131].

$$P = V_{DD}^2 \times f \tag{2.1}$$

Therefore, reducing V_{DD} and f reduces the processor's power consumption. In addition, having a power higher than 1 for V_{DD} in the previous formula indicates that even though the instruction's execution under lower frequency lasts longer, the processor's total energy consumption over the execution time is inferior to the energy consumption when using higher voltage for a shorter execution time.

However, using DVFS will not necessarily lead to a direct reduction in the whole system's energy consumption. The energy consumed by the supporting systems during the additional execution time can indeed lead to an increase in the system's total energy consumption. The interest is thus to use DVFS to slow down jobs during slack times, when the processor would have stayed idle otherwise, therefore consuming energy anyway. In such cases using DVFS can lead to energy consumption reduction. This issue is rarely taken into consideration in the literature research studies that consider DVFS, as most of these studies focus on the processor due to a lack of models that describe entire systems.

In addition, modern CPUs have an on-chip controller that allows to set the frequency and voltage of each of their cores independently based on the density of the running computation and depending on the type of the application [67, 66, 73, 116]. For Intel processors before the Skylake architecture,

setting the processor's frequency and voltage was done by the operating system, Skylake architecture introduced Speed Shift, which gives the processor the control of P-state [13]. For this reason, really controlling the voltage and frequency of a processor at the application, or even system, level is not that simple to be implemented.

However, other techniques exist to reduce the energy consumption of other components, or even on the whole server level, such as switching unused servers or computation nodes off.

2.1.4 On/Off

A turned on server, even if idle, still consumes up to 66% of its peak power [31]. The reason of this high idle consumption is related to the leakage current issue in semiconductor circuits, which causes transistors designed for high frequencies to consume power even when they do not switch states.

In a cloud environment, the Service Level Agreements (SLAs) guaranty the client of a network service provider the quality and availability of the services, so that the client's demands would get good response time. To meet these SLAs, data center operators over-provision servers to cover the estimated peak load. In a same manner, HPC systems are not always fully loaded since they are usually designed to accept peak demand. In both cases the total energy consumption of the system however increases with the number of machines that are running.

Leaving the extra servers turned on when no workload demands them can lead to a low sever utilization ratio (5 to 20%) [50, 20]. A study done by McKinsey consulting firm on energy used by data centers in 2012 found that, on average, only 6 to 12% of the electricity consumed by their servers was used to perform computations. The rest was essentially used to keep servers idle and ready in case of sudden increase in the workload [3] and leads to great energy waste.

One solution to reduce this energy waste is hence to exploit the used servers to their maximum performance level as much of the time as possible, while the rest of the servers are turned off rather than being left idle [110]. Gandhi et al. [52] propose an intelligent dynamic capacity management to match the number of active servers with the current load. If no increase in the work-flow is expected in the near future, the idle servers could be either turned off to save power consumption, or released to a cloud computing platform to save money.

2.1.5 Power capping

Another common technique to reduce a system's power consumption is power capping. This technique can be used on the entire system level and on the IT components level as well. Power capping consists of setting a maximum threshold of power consumption that a system or some of its components should not exceed [45]. Setting a power cap over a processor for example can protect it from overheating.

Power capping an entire IT infrastructure is usually related to limiting the electricity bills. Large scale IT system operators buy their electricity via a fixed contract with a power provider. The contract guarantees a constant power supply that does not drop below a certain value in Watts. On the other hand, the electricity price per kW-h when the power consumption is over the contract value is penalized. Therefore, operators aim to keep their system's power consumption within the contract limit.

Power capping can be achieved using different techniques, including but not limited to:

- DVFS and dynamic P-state adjustment.

- Processor switch off.
- Idle injection (forced idling of the CPU to avoid overheating).
- RAPL (combines DVFS and clock throttling to keep the processor's power consumption below a threshold) [104].

Some of the easiest ways to force a power cap are to limit the number of running machines or to use DVFS to run the machines on lower frequencies [111]. Lefurgy et al. [87] propose peak system-level power consumption management, based on precise power measurements and using a feedback controller. The proposed solution can periodically select the highest performance state that keeps the server within the power constraint. If the server is running below the power supply capacity, then it runs at full speed. If the server power consumption exceeds the power supply capacity, then it runs at a slower speed to match its power consumption to the power supply capacity. Processor level power capping can be done by packing threads of the workload together on the same core to control the number of active cores of a processor [112]. Or by using RAPL, which can set power limits on processor packages and DRAM allowing to dynamically limit the maximum average power to match the expected power and cooling budget.

Note that, in systems that are powered by renewable energy sources, such as the problem addressed by this work, power capping can provide a useful tool to respect the limited variable green power production during an interval of time.

2.1.6 Cooling

One of the main reasons a data center can have high PUE, therefore, low energy efficiency, is due to the energy costs of its cooling system, which are responsible for high percentage of the energy consumption of large scale ICT structures (40 to 50 % [2]). Cooling costs are not limited to high electricity bills, large scale cooling system are also costly to build. To reduce both building and operating costs, data and computation center operators use several techniques. Many operators nowadays choose to build their new data centers in cool clime areas. Facebook for example built a data center in Lulea, northern Sweden in 2011. Google built a data center in Finland the same year. They use seawater from the bay of Finland to chill their servers. Verne Global's Icelandic data center is located at a former NATO base in the southwestern corner of Iceland, close to two geothermal generation facilities. The advantage of such locations for a data center exceeds the free cool air that flows through their racks. Building ICT structures in these locations also provides cheaper electricity, since electricity production in most of these countries relies heavily on renewable sources [93, 14].

Using efficient cooling systems such as free cold air cooling can lower the construction cost by up to 40% [84], which is considered to be a great amount of money, knowing that such IT facilities can cost around 15m$ to construct for every megawatt of power consumption capacity.

An innovative efficient cooling technique consists of submerging computer components in a thermally conductive liquid. Liquid submersion is a routine method of cooling large power distribution components such as transformers, yet, it is still rarely used for cooling IT Hardware. IT hardware cooled in this manner does not require fans and the heat exchange between the warm coolant and cool water circuit usually occurs through a heat exchanger such as a radiator. Some extreme density supercomputers such as the Cray-2 and Cray T90 use large liquid-to-chilled liquid heat exchangers

for heat removal [7]. Many modern supercomputer racks such as BullSequana X1000 use warm water (up to 40°C) to cool their critical components [6].

2.2 Green ICT

Deploying any of the techniques described above can reduce the carbon footprint of ICT by reducing the energy consumption of large scale ICT structures. However, they will still be responsible for CO_2 emissions. The main objective of this work is to achieve a large scale computational structure with zero CO_2 emissions. The solution consists of using green energy sources to power these structures.

One way to power ICT structures with green energy is to sign a green energy contract with an electricity provider. Google for example has announced that all of their servers are operating on 100% renewable energy. They achieved that by signing green energy contracts, which indicate that additional renewable energy is pumped into the grid to off-set their consumption. This solution however means that the main energy source is still the grid, which provides a mixture of brown and green energy. To avoid brown energy consumption, the solution is on-site renewable energy production.

Now that solar and wind energy are becoming the cheapest sources of energy in many countries due to the global increase in investments in renewable energy, deploying on-site renewable sources is not only feasible, but can also be profitable. However, due to the variable and intermittent nature of both the power production of on-site renewable sources and the system's work flow, guaranteeing the availability of the system when needed is a challenge. Luckily, driven by the necessity of green ICT, many research studies are investigating in using renewable sources to power large scale ICT structures such as data centers, clouds and HPC systems.

In [119, 15, 59] several models and prototypes of large scale ICT structures powered by on-site renewable sources are presented in detail, along with solutions to manage this kind of structures.

The ANR DATAZERO project [1] presents several solutions that can be deployed on different levels of the system for optimal management of a green data center with only renewable power supply. They investigate management solutions of different renewable energy sources and energy storage techniques such as batteries and fuel cells. In addition, they present a green negotiation model to find a trade-off between the system's scheduler and its electrical infrastructure [62].

Another solution consists of implementing workload management to match the computational demand with the renewable supply. According to [59], the use of on-site renewable generation is increasing among green data centers, and a workload management method that manages both the workload and the available power production can make such a system more reliable.

The DECALCO project (which supports this work) focuses on finding a solution of the problem that lays in workload management. In order to perform reliable workload management in systems that use renewable energy sources, it is essential to first understand the power requirements of the system, along with information about the predicted green energy production, as these information are assumed to be available and used as input to the system's scheduler [100]. Sahia et al [114] propose an energy prediction model using weather forecasts for optimal scheduling in HPC systems. Li et al. [90] investigate the sizing problem for the solar panels and batteries needed in a green date center, they aim to maximize the utilization of renewable energy by introducing a scheduling policy that shifts the workload to times where renewable energy is available.

The energy consumption of an ICT system depends on both the system's hardware and the workload that is being processed on it. Therefore, software optimization in workload management

should take part in the effort to lower the system's energy consumption, in addition, workload management plays an important role in operating a large scale ICT structure powered by renewable sources. In the following chapter, details about scheduling techniques that are used to lower ICT's both energy consumption and carbon footprint are presented.

Chapter 3

Scheduling in green computing

This work addresses HPC systems powered by renewable energy sources as a way to reduce ICT's carbon footprint. We focus on finding a solution using scheduling strategies to carry out the workload management process in such systems. Several scheduling research topics are thus connected to our problem:

- Scheduling is an optimization problem that deals with the process of task to resource allocation over a given period of time. Scheduling problems can have one or more optimization objectives [61, 32, 106, 105].
- HPC systems are parallel computational platforms for which numerous scheduling related research works have been done that we introduce here.
- Using renewable power supply imposes a constraint that limits the availability of the system's resources, therefore, we also present some of the research works that tackle resource constrained scheduling.
- Reducing the carbon footprint can be achieved either indirectly by reducing the energy consumption, or directly by using renewable energy sources which eliminates CO_2 emissions. Scheduling strategies that can be used in both scenarios are presented later in this chapter.

So the remainder of this chapter 3 presents software level solutions represented by scheduling strategies. It is organized as follows. A general review on parallel and resource constrained scheduling is presented in Section 3.1 and Section 3.2 respectively, as these two types of scheduling problems can be directly linked to our problem. Scheduling techniques that are used to reduce the energy consumption are presented in Section 3.3, while scheduling strategies in green ICT structures that are powered by renewable energy sources are presented in Section 3.4.

3.1 Parallel scheduling

We recall that Graham [61] defined the $\alpha|\beta|\gamma$ notation that characterizes a scheduling optimization problem. In this notation the α value gives the machine environment. In the case of a single stage problem, the characteristics of the execution platform can be, for computing systems: 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 processing time, *prec*

for precedence between tasks, *pmtn* if tasks can be preempted, etc. The γ value gives the objective function to be optimized as, for instance minimizing the makespan $C_{\max} = \max(C_i)$ or the total flowtime $\sum C_i$ (the flowtime for short), where C_i is the completion time of task T_i [25].

One of the fundamental parallel scheduling problems is the one that can be formally expressed as $(P||C_{max})$ which describes the problem of minimizing the makespan over multiple parallel identical machines. This problem is proven to be NP-Hard in [54], which gives an initial idea about the complexity class of our problem, since $P||C_{max}$ can be considered as a generalization of our problem if we assume that the renewable power production is guaranteed to be available, in other words by dismissing the power constraint. Many other parallel scheduling problems are thoroughly studied in the literature [46, 47, 49, 48, 25, 40, 91, 117, 118]. However, we focus on research work that tackles scheduling in large scale ICT structures, in particular, HPC systems.

The three most known types of large scale ICT structures are data centers, HPC systems and clouds. These systems differ from one another in architecture, IT equipment and types of submitted jobs. Some research work about scheduling in data centers and cloud environments can be found in [11, 24, 52, 95, 8, 78]. However, as mentioned before, the focus of our work is scheduling policies in HPC systems that are powered by renewable sources. HPC systems consist of a great number of computational resources, which offers the user high computational capacities. These systems can be used to speedup solving complex problems that require great computation times, and the speedup is proportional to the application's degree of parallelism (DOP). HPC systems are often considered as heterogeneous computational platforms, they usually consist of different types of CPUs and GPUs to adapt to different computational demands. In addition, either due to equipment update and replacement or due to an increase in the computational capacity, in both cases, operators tend to buy more modern processors than what is already installed in their systems. Therefore, many research works focus on scheduling problems on heterogeneous platforms [74, 10, 133, 130, 76, 22, 141]. However, some work consider a homogeneous platform as a specific case to test their proposed solutions on a less complex computational platform model [74, 132], we point out that in the experimental part of our work we consider a homogeneous system for the same reason.

Scheduling strategies have a great effect on the performance of a parallel system. A proper scheduling strategy can improve response times, throughput, and utilization. In [144], several scheduling techniques that can be used to improve the performance of large scale parallel systems are discussed, namely, backfilling, gang-scheduling and migration, and the effect of combining these techniques is analyzed. In [41] a bi-criteria algorithm for scheduling parallel tasks on cluster platforms is proposed. The two criteria are the makespan and the weighted minimal average completion time.

A multi-core processor can be considered as the base representation of a parallel computational platform, Saifullah et al. [115] tackle scheduling parallel tasks on multi-core processors. We point out that we use this representation of a parallel computational platform in our first set of experiments as it provides a low complexity platform to compare our proposed solutions on.

3.2 Resource constrained scheduling

Resource constrained scheduling deals with the case in real life when the resources necessary to carry out a job are limited. The resources can be execution units, workers, energy, etc. In some cases, other constraints should be considered as well, such as the precedence constraints in a DAG [16]. Parallel scheduling is an instance of resource constrained scheduling since the number of computing

resources is not infinite in practice. On the other hand, in our case, the use of green energy introduces a new constraint, the available power, in the scheduling problem. For that reason, in this section, we present some research work that tackles resource constrained scheduling problems.

We point out that our work differs from scheduling work that aims to reduce the energy consumption, as presented in Section 3.3, in that we consider that the available green energy (per Watt-h) is free to use. On the other hand its variability enforces an instantaneous power (per Watt) constraint that is function of the time. We, therefore, tackle a power constrained scheduling problem whose only optimization objective is minimizing the makespan or the flowtime.

Resource constrained scheduling problems are usually associated in the literature with project scheduling. Project scheduling deals with the execution of production activities over a limited number of resources [26]. Icmeli et al. [75] present a survey on three fundamental project scheduling problems: the time/cost trade-off problem (TCTP), the payment scheduling problem (PSP), and the resource constrained project scheduling problem (RCPSP). Lawler et al. [86] present an exhaustive complexity classification of RCPSP problems. They show that the problem of minimizing the maximum completion time over two parallel identical machines with unit processing times under resource constraints is solvable in polynomial time and that even when setting the number of resources to one, solving the same problem over three parallel identical machines is NP-Hard. They conclude that resource constrained project scheduling problems are NP-Hard when the number of used resources exceeds two.

3.2.1 Resource constrained project scheduling problem (RCPSP)

Many theoretical scheduling solutions deal with the time aspect in a schedule without consideration of resource restrictions. In real life, execution delays can occur when the required resources are not available during a certain time interval. This problem is known as the resource constrained project scheduling problem (RCPSP).

The resources in RCPSP problems have limited availability and the objective is to minimize the total execution time of a set of activities taking into consideration the resource constraints. Various extensions of the basic RCPSP exist, Hartmann et al. [70] present a survey that gives an overview over these extensions.

According to Slowinski et al. [124] and Weglarz et al. [137], three different types of resource can be found in project scheduling problems with multiple modes, renewable, nonrenewable, and doubly constrained resources.

Renewable resources: Renewable resources are resources that are available in each period with their full capacity, and they are limited on a per-period basis. Manpower and machines for example are considered as renewable sources. Fu et al [51] tackle the problem of job scheduling over parallel machines that are considered as renewable resources.

Nonrenewable resources: Unlike renewable resources which are limited to a certain value per period, nonrenewable resources have a limited capacity for the entire project, like the budget of the project for example. In green computing, an energy budget can be enforced over a specific time interval to limit the total energy consumption of the computational platform. Therefore, the total energy consumption during a period of time per Watt-h must not exceed a preset energy budget. Similar model is used in economic studies for dispensing a money budget during a certain period, in such cases, the following rules usually apply:

- The budget must not be spent too early and leave no money/energy for the rest of the project (this will unbalance the performance during the project).
- The total spent amount must not exceed the budget (no debts allowed).

Therefore, when enforcing an energy budget over a computational platform for example, before taking the decision of running a job, the availability of energy must be verified for the entire duration of job execution.

Doubly constrained resources: Doubly constrained resources are limited both for each period and for the whole project, both the cash flow and the budget of the project can be limited for example. Another example is energy, if a limited energy budget is fixed for the entire project, and the instantaneous power supply is limited as well.

3.2.2 Continuous resources or Continuously divisible resources

If the execution of an activity dedicates the entire resource for this activity we say that the resources are available in discrete quantities, such as manpower and processors when no time sharing is allowed over execution units. If a resource can be continuously divisible between several activities, we then have continuous resources, such as liquids or the electrical power in our case.

This approach can be applied to such resources as money, energy and some cases of manpower, if the constraints on resource usage are not considered in relation to the number of workers but in relation to the number of man-hours per day. This number, which is obtained in the optimal schedule, may be reached by changing the working time per day of several employees.

Weglarz et al. [138] present several methods for the optimal scheduling of activities over doubly constrained resources which are continuously divisible. Both the instantaneous and the total usage of resources is limited per time period and during the entire project duration, respectively. They represent the activities using mathematical models that consider the performance speed is a continuous function of resource amounts.

Artigues et al. [17] adapt the energetic reasoning and the left-shift/right-shift methods to the case of continuous resources for the Energy-Constrained Scheduling Problem (CECSP). They propose a model where the availability of a processor is continuous. The processor is thus considered as a continuously divisible resource, and its instantaneous usage is limited between a maximum and a minimum value. Job execution is limited within a time window (between its release date and its deadline), during which, the job's energy requirement should be met. In their model, the energy availability/consumption over a given time interval equals to the resource capacity/usage times the length of the interval. The objective in this scheduling problem is to compute a schedule that satisfies the energy constraint (limit per interval vs requirement per activity) while meeting the time window of each activity.

Nattaf et al. [101] also address energy constrained scheduling over continuous resource. They propose a Mixed Integer Linear Program (MILP) to find a schedule that respects both the instantaneous and the total resource usage constraints. They then propose a second method that combines their linear program with an adaptation of the left-shift/right-shift satisfiability test and a time window adjustment technique. The difference between this work and Artigues et al. [17] is that this work considers the general case where the energy is expressed as a linear function of the resource consumed, while in [17], they consider a particular case in which the resource consumed by a task is equal to the energy received by it.

As some of the previous works deal with the total available energy over a given time interval as a constraint, they consider this value to be limited by the capacity of the execution units times the length of the interval. In other words, they assume that the electrical power is available as long as there are machines to demand it and the energy constraint is rational to the number of processors. It is thus similar to a processor constraint problem in some sense, except that the task scheduling is based on the energetic reasoning and on the left/right shift defined in Appendix 9.A. As in our case, in addition to having a resource constraint represented by a limited number of processing units, the level of available green power during each time interval is limited as well.

3.3 Energy-aware scheduling

After presenting a general view about parallel and constrained scheduling, we go back to the issue of green computing. Reducing the carbon footprint of ICT structures can be done by reducing their energy consumption, as Section 2.1 talked about solutions to reduce the energy consumption on hardware level, in this section we present software level workload management techniques that can be deployed to reduce the energy consumption.

The energy consumption of a large scale computational system can be subject to its scheduler's decisions. Energy-aware scheduling consists of taking the energy consumption into consideration when computing the schedule. In heterogeneous multi-processor platforms for instance, one processor might be slower yet more energy efficient than the other. By selecting either the faster processor or the more energy efficient one, the scheduler is responsible for a trade-off decision between the energy consumption and the completion time [36, 79]. In a cloud environment for example, Tchernykh et al. [128] show that assigning tasks to processors with the lowest energy consumption reduces the system's energy consumption which offers financial benefits to the cloud provider. Yang et al. [142] propose an approximation scheme for task to processor allocation on heterogeneous processors to minimize the energy consumption as well. A similar energy-aware scheduling policy for distributed systems is proposed in [38], they propose deploying a high level scheduler that allocates tasks to a cluster, where a local low level scheduler allocates tasks to multi-core servers.

Energy-aware scheduling has been in the spotlight of academic research over the last few years [129, 128, 127, 43]. Other studies tackle this problem as a multi-objective optimization problem with the objective of reducing the energy consumption on one hand, and traditional scheduling objectives such as flowtime and makespan on the other [139, 85, 27, 122, 77].

In the following, two of the most common scheduling techniques that affect the system's energy consumption are presented in detail, namely the workload consolidation and load balancing techniques. We then analyze the combination of these techniques with IT management to reduce energy consumption.

3.3.1 Workload consolidation

The idea of workload consolidation started from the need to execute multiple jobs on the same physical machine. Doing so eliminated the need to increase the number of machines every time a new job was submitted, therefore, it reduces the cost of building a computational system. As IT energy consumption started to get more academic attention over the last few years, many research studies address using workload consolidation as an energy saving technique [125].

One of the main causes of energy waste in data centers is the idle power consumption. The idle

server consumes around 50% [94] and up to 66% [31] of its peak power consumption. Executing multiple jobs on a set of running machines, while turning the rest of the machines off, rather than putting them on idle mode, can reduce the energy waste resulting from idle consumption [125, 39].

Nevertheless, before switching off a machine, the delay it takes to switch it back on should be considered, as well as the power consumed to boot all its components during this delay, which is normally higher than the idle power consumption [42]. The rate at which jobs are submitted for execution is referred to as the work flow. If the work flow fluctuates frequently, it might be more energy efficient to leave additional machines running even when they are not needed for computation at the moment, yet an increase in the work flow is anticipated. In addition, the nature of the executed jobs forces workload consolidation to be more selective. In parallel jobs for example where there are dependency and concurrency relations between tasks, using more machines at lower performance mode might be more time and energy saving than using less machines at full performance mode.

In cloud environments, users carry out their businesses over virtual machines that are created within a remote computing environment. A virtual machine (**VM**) is basically a software that gives access to a remote platform which gives the client the feeling that he is using a real dedicated computer, where in reality, the cloud manager grants the user temporary access to some of the cloud's hardware.

Multiple virtual machines can run on the same physical hardware simultaneously. VM consolidation is another form of workload consolidation. Cloud management systems such as MUSE [30], Open stack [23, 33], delta cloud and Xen Cloud Platform XCP [23], exploit VM consolidation [24] to carry out the execution of multiple virtual machines on the same physical host, with the aim to reduce energy consumption and to minimize SLA violations. These management systems use a placement function that produces VM-to-server mappings and that minimizes a cost function, at the same time. The placement function must respect the physical limits of resources in order to avoid any infeasible VM consolidation schemes.

In some cases, the cloud manager can be forced to adjust the placement of a virtual machine between physical hardware. In this case, virtual machine migration (VMM) is deployed [63]. VMM can be used by the cloud manager to perform load balancing and VM consolidation, however, the migration of a VM over the network has a cost that should be taken into account.

Workload consolidation in HPC systems produces the risk of hot spots, due to extensive use of few computational resources in comparison to others. The presence of these hot spots forces the system operators to build bigger and more expensive cooling systems [9]. Balancing the workload equally between computational resources can reduce the risk of hot spots.

3.3.2 Load balancing and thermal aware scheduling

Load balancing consists of assigning the tasks to processors in a way that balances the workload equally across all the processors [107, 108]. Applying load balancing in HPC systems helps optimizing resource utilization and minimizing the system's response time [131].

In multi-processor systems, thermal-aware scheduling consists of distributing tasks over processors in such a way that prevents processors from overheating, if a task is causing a processor to overheat, it is then moved to a cooler processor. This approach is similar to load balancing where jobs are distributed evenly over the execution units, except that in thermal-aware scheduling jobs are assigned to processors in a way that evens out the temperature of the processors across the system (heat balancing). This technique is considered as a good solution for reducing cooling costs by preventing hot spots [12]. Merkel et al. [96] propose a scheduling policy to balance the work-

load over all processors evenly to prevent processors from overheating which decreases the system's energy consumption while reducing the need for throttling processors.

Power-aware load balancing uses the same concept to evenly balance the power consumption across all the processors [12], taking into account that tasks with different computational density produce different power consumption [126].

We point out a trade-off between reducing the static energy waste by applying workload consolidation and reducing extra cooling costs due to hot spots by balancing the workload over more running processors.

3.3.3 Combined IT and Scheduling to reduce energy consumption

Many studies combine energy efficiency methods that are deployed on the hardware level with a scheduling strategy. The result is a scheduling strategy that does not only perform task to processor or task to time interval allocation, but also controls some aspects of the IT equipment setting in coordination with the workload management decisions, with the aim of reducing the system's energy consumption. Many efforts that aim to reduce the energy consumption of large scale computation systems include IT level power management solutions in their system's scheduler, such as DVFS [135, 55] and power capping and machine shutdown [42].

Commonly used on the IT level power management, we recall that Dynamic Voltage and Frequency Scaling DVFS consists of scaling down the frequency and the supply voltage of the processor. Setting the processor to lower frequencies implies a slow down toll. DVFS thus presents a trade-off between task execution time and power consumption. This trade-off implies that when the power consumption is minimal, which means that the frequency $f = 0$ and the processor is not running, the task's execution time is maximal and it equals to infinity. One way to prevent the prolongation of task execution times from excessively increasing the overall computation time is to exploit slack times. Running non-critical tasks on lower frequencies during slack times can reduce the energy consumption without increasing the total execution time [98, 56, 135, 134, 18]. Wu et al. [140] propose a scheduling algorithm that creates VMs for allocating jobs to servers in a cloud environment, then, based on the scheduler's decisions, a DVFS controller sets the frequencies of servers in order to reduce their energy consumption.

Quan et al. [109] propose using a variable voltage processor in a real-time system to reduce its energy consumption. The jobs are scheduled using a fixed priority scheduling strategy and each job has a release date, a deadline and a required number of CPU cycles. The proposed solution consists of finding the minimum voltage needed to complete each job, a voltage schedule is thus produced by setting the supply voltage to different values at different times, taking advantage of the fact that when executing jobs on a variable voltage processor, the execution time of each job varies depending on the speed of the processor and its supply voltage.

Energy-aware scheduling can be addressed as an optimization problem. A common method for solving optimization problems is to use a genetic algorithm (GA) [97]. In scheduling problems, each solution of the problem is represented by a chromosome. The representation of the chromosome might differ from one problem to another, and from one GA to another. A bi-objective genetic algorithm for reducing the energy consumption and the makespan in computing systems is presented in [81], each chromosome represents a task to processor allocation schedule, in addition to a processor voltage management scheme, each gene of the chromosome thus assigns one task to a processor and sets its voltage. When genetic operators such as mutation and crossover are applied, the tasks are redistributed among the processors and the fixed voltage values. The chromosome that yields

lower energy consumption is considered as a good individual, it is therefore kept through the next generation. At the end of GA execution, the winner solution is the one that produces the minimum energy consumption and the shortest makespan.

3.4 Scheduling in green ICT

Energy-aware scheduling can reduce the carbon footprint of ICT systems by reducing energy consumption, however, as long as a system is dependent on the electrical grid, it will keep consuming brown energy which contributes to the carbon footprint. Using renewable energy sources such as solar and wind energy to power large scale ICT structures reduces their environmental impact, however, it imposes an instantaneous power constraint. Reducing the energy consumption is hence not an optimization objective in our problem, we rather aim to optimize the utilization of renewable energy with the objective of minimizing the total execution time. Scheduling can play an important role in optimizing the usage of renewable energy by managing the workload according to the intermittent renewable power supply.

3.4.1 Geographically distributed data centers with on-site renewable sources

The main drawback of using on-site renewable energy is that most renewable energy sources are intermittent and are not always accurate to predict. One solution is to interconnect several data centers that are geographically distributed, thus, they remain under different climate conditions. The advantage of such a model is that if the sun or the wind are running low over one data center, another one might still have good level of sun or wind speed. The idea is to deploy virtual machine migration VMM between the interconnected data centers in such a way that a data center with plenty of available power supply would perform extra jobs that were submitted to another data center, where the power supply is not enough to execute these jobs.

Zhang et al. [143], propose a model for a geographically distributed data center model. Each data center is powered by on-site renewable sources such as solar panels and wind turbines, in addition to the grid. The data centers are interconnected using optical networks. Since transmitting renewable energy via the power grid may introduce power loss, the proposed solution aims to maximize the utilization of renewable energy in a data center rather than pumping the extra energy into the grid. They propose two algorithms to perform renewable energy-aware inter-data-center VM migration with the objective of minimizing the total brown energy cost in all data centers, taking into account both the VM service constraints and the network resource constraints. The two algorithms are namely the Manycast with Shortest Path Routing (Manycast-SPR) and the Manycast Least-Weight Path Routing (Manycast-LPR). Their results show that Manycast-SPR saves about 15% cost of brown energy in comparison with the strategy without migration, while Manycast-LPR saves about 31% cost of brown energy in comparison with the strategy without migration.

Iturriaga et al. [78] address scheduling tasks in distributed heterogeneous data centers which are partially powered by renewable energy sources. For tackling this problem, they divide it into two scheduling subproblems, a higher-level scheduling problem for allocating tasks to data centers, and a lower-level scheduling problem for scheduling the tasks to the computing resources inside each data center. They propose two multi-objective evolutionary algorithms for solving both scheduling subproblems.

3.4.2 Green power aware scheduling

Using renewable energy sources makes the instantaneous power consumption a more important factor than the total energy consumption, as the renewable energy is theoretically free, while its power capacity per Watt is limited and variable.

In [29], a genetic algorithm is proposed to minimize tasks due date violations in a cloud environment, while respecting the renewable power envelope and the resource constraints.

GreenSLot [60, 58] is a scheduler for parallel batch jobs in a data center powered by both a green energy source and the electrical grid. A prediction of the available renewable energy is provided using historical data and weather forecasts. The jobs are divided into critical and non-critical. Non-critical jobs are the ones that have longer deadlines, therefore, they are more delay tolerant than critical jobs. The scheduler exploits the flexibility of non-critical jobs to produce a schedule where the workload power consumption matches the renewable power production. The system is however connected to the grid to avoid deadline violations when the renewable power supply is not high enough, in such case, when using brown energy from the grid is inevitable, the scheduler choses to use the grid electricity at times with the lowest pricing.

Lei et al. [88] propose a genetic algorithm for solving a multi-objective energy-efficient scheduling problem on a data center that is partially powered by renewable energy sources. The proposed algorithm addresses both minimizing the makespan and minimizing the total energy consumption objectives. A chromosome in the proposed model allocates tasks to processors, and then sets the supply voltage of each processor. With each iteration, the GA shuffles the processors part of the chromosome along with the voltage values part using different genetic operators such as mutation and crossover, while the order of tasks remains fixed according to their index. They show that their proposed GA outperforms the genetic algorithm proposed in [136].

Part II

Contribution

In this part we present the contribution of this work. We first formally describe out problem and the model proposed to represent its elements in Chapter 4. We provide formal complexity results on several scheduling problems that can be derived from our optimization problem in Chapter 5. We finally present our proposed scheduling solutions for this problem in Chapter 6.

Chapter 4

Problem formalization

The focus of this work is the optimization of scheduling independent sequential tasks on a parallel computational platform powered solely by green energy sources, such as solar panels and wind turbines. Since this scheduling problem is not studied in the literature, we propose our own theoretical model to evaluate our algorithmic study.

The computational platform in this model consists of parallel machines. Each machine requires a minimum amount of electrical power to operate, the machine's static power consumption. Executing a task over a machine adds an extra power consumption, the task's power consumption. We assume that tasks differ from each other in their processing times and power consumptions. We focus on the electrical power production/consumption rather than energy production/consumption because we aim at using the electrical production of the green sources at the time it is produced. We are not interested in reducing the consumption of what we consider *free* energy produced by renewable sources. This makes our work different from the studies mentioned in the previous chapter that address bi-objective scheduling strategies to reduce both energy consumption and the total execution time.

Our work thus is only concerned with execution time related objectives, the total execution time and the flowtime, while using renewable energy sources imposes a variable power constraint. We analyze here the elements of this constrained scheduling problem. Using the meaning of "renewable resource" defined in RCPSP (a resource that is available in each period with its full capacity, see 3.2.1), we consider the processing units as a renewable resource as they are available in their full capacity at the beginning of each time interval. Furthermore, the green power is considered as a continuous resource that can be continuously divisible between the available processing units. However, we do not consider the green power to be a renewable resource because it is not guaranteed to be available in full capacity at the beginning of each interval, nor it is considered a nonrenewable resource because consuming power in one interval does not reduce the power level in the next one. The green power constraint is thus a generalization case of continuously divisible resources problem with a various availability level that is function of time.

We finally point out that, in real life HPC systems, time sharing is not allowed. A processing unit is thus dedicated entirely for the execution of the job assigned to it until completion. Therefore, considering basic RCPSP in HPC systems, a main difference between power as a resource constraint and processor as a resource constraint is that the set of processors has a finite number of execution units, and each execution unit is dedicated to a task (discrete problem), while the available power (in Watt) can be continuously distributed among (continuous problem). Thus, the distribution of power

between tasks is on its own an optimization problem, which increases the complexity regarding the processor as a resource problem. In addition, due to the variability in the power production through time, the processing time of tasks should be taken into account to verify that both the power as a resource and the execution unit as a resource constraints are respected during the entire processing time of a task. This presents a two dimensional optimization problem where the horizontal axis represents time (Second) and the vertical axis represents power (Watt).

In the following, we present the models used to tackle optimization problem. The task model is presented in Section 4.1, the computational platform model is presented in Section 4.2, the available power envelope model is explained in Section 4.3, the scheduling model is presented in Section 4.4 and finally, Section 4.5 presents the objectives of our problem and the notations used in the rest of this work.

4.1 Task model

For the task model we consider a set $\mathcal{T} = \{T_1, T_2, \ldots, T_n\}$ of n tasks T_i characterized by their processing time p_i. These tasks are sequential independent tasks, other task models exist in the literature, such as moldable tasks, tasks with precedence relations, DAGs, etc., however the focus of this study is on a more simple task model, therefore, we consider sequential independent tasks where no communication nor precedence relations exists between tasks. Running a task on one machine generates an extra power consumption [126, 57] which varies over time depending whether the task intensively computes or not. It has to be approximated to be used in an optimization problem. We assume that each task T_i has a constant power demand, its largest power need φ_i over its lifetime. By taking the larger power consumption, we guarantee that the resulting schedule will fit in the power envelope. Consequently, when one task T_i is executed on one machine M_j, the power consumption of M_j hence increases by φ_i, this value differ from one task to another, a task that performs intensive computation for example should produce more power consumption than a light I/O task. Each task T_i is thus characterized by its *a priori* known processing time p_i and power consumption φ_i.

4.2 Machine model

The considered computing platform is parallel which means that several execution units are available to process the tasks. The platform thus consists of a set $\mathcal{M} = \{M_1, M_2, \ldots, M_m\}$ of m machines M_j that represent execution units.

The power consumption of each machine is divided into two parts. The first part is its static power consumption P^{static}, which is the power needed to run the machine when it does not process any tasks. This static power has a constant value for the entire considered computational time horizon $\mathcal{H}$. The second part is the dynamic power consumption $P_{CPU}^{dynamic}$, which is the processor's power consumption due to task execution and it equals to the sum of the executed tasks power consumption φ_i.

Starting from this general model we derive the case of one machine. The one machine model is used as a base of the theoretical complexity study presented in the next chapter. However, for our experimental study, we later on use two practical computational platform models, namely the multi-core and the multi-machine models, on which our proposed solutions are tested. In the following,

difference between these two models and their characteristics are presented, although the theoretical machine model remains the general one mentioned above.

4.2.1 Multi-core

We consider a parallel computational platform that consists of one multi-core machine. This machine comprises of nb_cores identical CPU cores as a representation of an identical parallel machine. This model of a shared memory parallel platform provides the possibility of scheduling multiple tasks in parallel simultaneously while allowing a straightforward evaluation of the proposed solutions as a first approach, while avoiding communication costs between different execution units.

This multi-core machine is assumed to be turned on all the time, and the static power necessary to turn it on is already deducted from the green power supply. The number of the machine's cores is nb_cores and at the beginning of the scheduling process, the number of the available cores av_cores through the computation time horizon $\mathcal{H}$ is set to nb_cores.

Note that in the multi-core model, the machines (cores) are not considered to have an independent power consumption when they are idle since they belong to the same parallel machine. When the parallel machine is running, it consumes at any time at least its static power P^{static}. When it does not process any tasks, the power consumption is exactly P^{static}, otherwise it increases depending on the executed task (Cf. the task model in Section 4.1).

Note that, in the multi-core model, the static power is assumed to have a constant value for the entire considered time horizon $\mathcal{H}$. So we consider for each period of time Δ_x only the remaining power which is available for task computation, i.e., $\Phi_x = \max(\Phi_x^{available} - P^{static}, 0)$. While in the multi-machine model, the static power of each running machine P^{static} must be deducted from the power availability curve $\Phi^{available}(t)$ by the scheduler at the moment of switching the machine on.

4.2.2 Multi-machine

The second examined computational platform is composed of several multi-core machines similar to the platform defined above.

Let a set $\mathcal{M} = \{M_1, M_2, \ldots, M_m\}$ of m machines M_j. Each machine M_j corresponds to a multi-core machine that has a set $\mathcal{C}_j = \{C_{j_1}, C_{j_2}, \ldots, C_{j_{nb\_cores}}\}$ of nb_cores_j cores. The cores of a machine M_j are available for computation only when this machine is powered on, therefore, the total number of available execution units (cores) varies over time, the number of available cores at a time t equals to the sum of the cores of the machines that are running during that time.Powering on a machine is not instantaneous, it takes a delay $t_j^{off \to on}$ during which the power consumption of the machine equals to $P_j^{off \to on}$ and while a machine is running it consumes at least its static power P_j^{static}. Switching off a machine also has a delay $t_j^{on \to off}$, during which the power consumption of the machine equals to $P_j^{on \to off}$.

The interest of this model is to examine the effect of switching available machines on and off under variable power constraints. To give a case study of the case when it is possible that the power supply is not enough to turn on all the m machines.

In real life, a main difference between the multi-core and the multi-machine models is that a multi-core platform is considered as a shared memory device, where the communication between cores theoretically has no cost. Whereas the multi machine platform is a distributed memory device, where communication between machines must have a cost. However, in both models we

neglect the communication cost, since we consider using independent tasks, therefore we assume that no communication occurs between machines.

4.3 Power model

As the power provisioning of the platform solely comes from green energy sources, its production is not stable and varies over time. The available power is represented at each time t by a curve $\Phi^{available}(t)$. To be able to optimize the usage of the power we assume that the available power $\Phi^{available}(t)$ is a constant value $\Phi_x^{available}$ over an interval of time Δ_x. We define the time horizon $\mathcal{H}$ as the length of the considered power envelop. Note that we only condider in the following time horizons long enough to schedule all the tasks. For a given time horizon $\mathcal{H}$ the available power is thus modeled by X intervals Δ_x of length δ_x, such that $\sum_{x=1}^{X} \delta_x = \mathcal{H}$. This power is shared by all the machines of the platform and it is assumed to be a constant value Φ_x during each interval of time Δ_x. In the case of multi-core platform, since it is useless to run a machine without processing tasks, we consider useful available power $\Phi_x = \Phi_x^{available} - \sum_{j=0}^{\mathcal{M}} \Phi_j^{stat}$ for the period of time Δ_x. If the platform consists of only one machine, $\Phi(t) = \Phi_x$. We point out that the characteristics of this power model impose a variable power constraint to our problem where the tasks power consumption during a time interval Δ_x must not exceed its level of available power Φ_x.

4.4 Scheduling model

A visual representation of the model's elements is presented in Figure 4.1, the red empty rectangles represent the power envelope that is distributed into time intervals, and the gray filled rectangles represent a set of submitted tasks. The horizontal axis represents the duration/processing times of the intervals/tasks, while the vertical axis represents the power availability/consumption of the intervals/ tasks.

We now consider the optimization problem of scheduling of these tasks under these variable power availability constraints over both of the computational platforms introduced above, and the objective is to find a schedule with minimum makespan that respects those constraints which would look like the example schedule of Figure 4.2.

We use a static scheduling model where all tasks are submitted for execution at the beginning of runtime. We define a time slot of length l as a group of l consecutive intervals. To schedule a given task T_i, the scheduler must find an eligible time slot during which $\Phi_x \geq \varphi_i$ for all the intervals that make up this time slot.

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 where task T_i can run. Let $b_{k,i}$ be the beginning of the slot $\mathcal{E}_{k,i}$ and $f_{k,i}$ be 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 $\Phi(t) \geq \varphi_i$. Formally, it exists two integer values x and s such that the kth time slot $\mathcal{E}_{k,i}$ is defined by $\mathcal{E}_{k,i} = \Delta_x \cup \Delta_{x+1} \cup \ldots \cup \Delta_{x+s}$ where at any time $t \in \mathcal{E}_{k,i}$, $\Phi(t) \geq \varphi_i$ and at any time $t \in \Delta_{x-1}$ or $t \in \Delta_{x+s+1}$ $\Phi(t) < \varphi_i$. So $b_{k,i} = \sum_{x'=1}^{x-1} \delta_{x'}$ and $f_{k,i} = \sum_{x'=x+s}^{X} \delta_{x'}$ (see Figure 4.3). When a time slot is chosen to schedule a task, the corresponding power is subtracted from the intervals that compose this time slot. If the task execution ends before the end of the last interval of the time slot, this interval is divided into two new intervals, and the available power level is modified only in the part of the interval where the task execution took place.

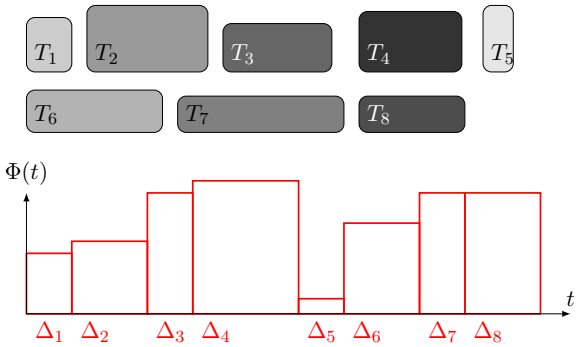


Figure 4.1 – Illustrating example for the optimization problem: A set of tasks to be scheduled in the given power envelop.

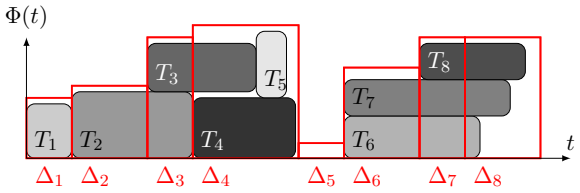


Figure 4.2 – Example of a schedule

Finally, we consider an allocation function $A(i,j) = k$ that carries on the task scheduling process. On the multi-core platform, this allocation function returns in which time slot $\mathcal{E}_{k,i}$ the task T_i is scheduled. Let $\mathcal{T}_{k,j}$ be a subset of task set $\mathcal{T}$ that contains the tasks scheduled in the time slot $\mathcal{E}_{k,i}$. For every task $T_i \in \mathcal{T}_{k,j}$, we set $A(i,j) = k$. Note that $\sum_{i|T_i \in \mathcal{T}_{k,j}} p_i \leq f_{k,i} - b_{k,i} = l_{k,i}$. When a task is allocated to a time slot, the number of available cores in each of the interval of that time slot is reduced by 1 ($av\_cores_k -= 1$). Thus, in order to schedule a task for execution in an interval Δ_x, a resource constraint must be met $av\_cores_x > 0$. Similarly to the power availability deduction, the number of available cores is reduced only in part of the last interval where the task execution took place. Note that, for simplicity reasons, in the case where there is only one machine the allocation function is simplified in $A(i) = k$, that returns in which time slot $\mathcal{E}_{k,i}$ the task T_i is scheduled. Similarly, when a time slot is eligible for all the tasks it will be noted $\mathcal{E}_k$, its begining and finishing times are b_k and f_k.

On the multi-machine platform, the allocation function performs two steps. It first allocates the task to a machine, then it allocates it to a time interval. The processing unit availability condition is thus different from the multi-core model. av_cores_x is the number of available cores in the time interval Δ_x, it equals to the sum of the cores of the machines that are turned on during Δ_x. At the begging of the scheduling process, all machines are assumed to be turned off, therefore $av\_cores_x = 0$ for $x \in [\![1, X]\!]$ where X is the number of intervals, and the resource constraint in this case is $av\_cores_x > 0$.

The scheduler manages turning the machines on/off according to the power availability and the computational demand in each interval Δ_x, taking into account that turning on and off a machine produces time and power costs. When a machine M_j is turned on during an interval Δ_x, then $av\_cores_x += nb\_cores_j$.

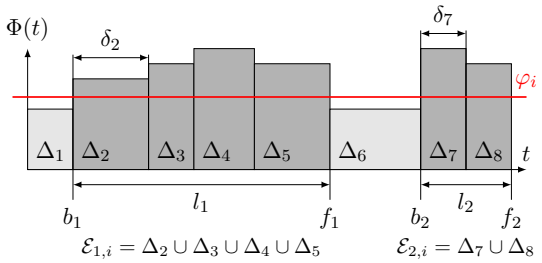


Figure 4.3 – Illustrative example for intervals $(\Delta_1, \ldots, \Delta_8)$, available power on time and time slots $(\mathcal{E}_1, \mathcal{E}_2)$ in which the task T_i could be scheduled when its power need is φ_i on a one machine platform.

4.5 Notations and objectives

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 3.1. This enforces that the power needed by one (φ_i) or several tasks $(\sum \varphi_i)$ must be lower than the power provided (Φ_x) by the power sources. 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 the Φ_x variable is set to Φ, the available power is constant over the considered horizon and if the φ_i variable is set to φ, each task needs the same power to run.

Considering computing and data centers, two main criteria are usually considered for minimization. The makespan $(C_{\max})$ targets the minimization of the running time for 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. This enforces a fair share of the resources between users.

Table 4.1 summarizes the notations used in the remainder of this work.

Table 4.1 – Summary of the notations

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
$\mathcal{M}$	set of machines
M_j	machine j
m	number of machines
nb_cores_j	number of cores of M_j
Δ_x	interval with constant power
δ_x	length of Δ_x
X	number of consecutive intervals Δ_x
$\mathcal{E}_(\varphi_i)$	time slot: set of eligible time intervals
$\mathcal{E}_{k,i}$	kth term 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 $\mathcal{E}_{k,i}$
$\mathcal{C}_j$	set of cores of M_j
av_cores_x	number of available cores during Δ_x

Chapter 5

Complexity study

5.1 Introduction

Using the preceding model we conduct a theoretical complexity analysis for several cases of our problem that consider static scheduling problems where information about the tasks and the available power are known in advance.

The complexity of scheduling problems has been well studied in the literature, We highlight briefly here some complexity results for problems that are close to ours. Minimizing the makespan and minimizing the flowtime problems on m uniform parallel machines are polynomially solvable if jobs are identical [35]. If jobs have different processing times however, minimizing the makespan on two or more parallel identical machines is NP-hard [89, 54]. Finally, scheduling independent tasks without preemption on a heterogeneous parallel platform is NP-hard [99].

The majority of the studied scheduling problems tackle traditional scheduling models where the power consumption of the entire computational platform or for each individual task is not taken into consideration, while in our work, the use of green power sources makes the power production level a constraint rather than an objective, and we focus on optimizing traditional scheduling metrics such as makespan and flowtime under the power constraints.

In this chapter, we examine the complexity of several cases of our problem using reduction from well studied problems but with consideration to the renewable power supply constraints. As far as we know, there does not exist other works that proof the complexity of these problems.

At first we tackle one machine problems as showing that these problems are NP-Hard implies that the more general parallel problems are NP-Hard as well, then we move on to the parallel version of the problem. We consider these problems for both cases of minimizing makespan and flowtime and for the cases with or without preemption.

We recall that NP-Complete problems are decision problems with "Yes" and "No" answers, whilst NP-Hard problems are optimization problems. Furthermore, an NP-Hard problem can be the optimization version of an NP-Complete decision problem, therefore, the optimization problems corresponding to NP-Complete problems are accordingly NP-Hard, it is thus enough to prove that decision version of the problem is NP-Complete to prove the NP-Hardness of the optimization problem. For that reason, even though our problem is an optimization one, the following proofs are NP-Completeness proofs and not NP-Hardness proofs.

5.2 One Machine Problems

The optimization problems of finding the optimal makespan and the optimal flowtime on one machine (with one processing unit or one core) are simple to solve without the power constraint. It exists simple polynomial time algorithms that give the optimal solution. We recall that all no delay schedules (i.e., schedules without delay between the tasks) are optimal solutions for the makespan objective and that the Shortest Processing Time (SPT) algorithm gives optimal solutions for the flowtime objective.

We show here that, with power constraints, these problems are polynomial in the case of identical tasks (i.e., tasks with same p_i) and become NP-Hard when tasks have different processing times and if preemption is not allowed.

As already said, the static power P_j^{static} consumed by the machine $M_j \in \mathcal{M}$ is assumed to be constant over time. Therefore, in the one machine problems, machine j cannot run at any time t where the power provided is lower than P_j^{static}. For that reason we assume that the static power needed by the machine is at least available in each interval in one machine a problems as well as in parallel problems using one multi-core machine. Therefore, we only consider $\Phi_x = \Phi_x^{available} - P_j^{static}$, the useful power to run tasks.

Note that, in the one machine problems, when the available power Φ_x is constant over the intervals, $\Phi_x = \Phi$, then the problem is just to check that every task power need φ_i is such that $\varphi_i \leq \Phi$. If this condition is true then the problem returns to the classical scheduling problem without power constraint. Otherwise, there is a task such that $\varphi_i > \Phi$ and the problem has no solution.

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

5.2.1 Problems without preemption

In computing centers the nodes are usually dedicated to the users and no preemption is applied to the tasks. We assess here the complexity of the scheduling problem in this context.

Time unit tasks $p_i = p$ and $\varphi_i \leq \Phi_x$

The most simple problems are the cases where every task has the same computing time $p_i = p$ (see figure 5.1). To optimize our objective, we just have to put as many tasks as possible in the earliest time interval possible, regardless of the order at which the tasks are scheduled, as long as the power constraint $\varphi_i \leq \Phi_x$ is respected. If the interval length is not a multiple of the task size then the remaining time of an interval can be used if the next tasks can be shifted of less than p. Obviously this solution is optimal for the makespan objective, as every task is interchangeable with another, changing the order will not give a better solution and we do not leave empty places where a task can be put. For the flowtime, as every task has the same processing time, none of them has a larger weight in the final sum and there is no need to put the tasks in a specific order.

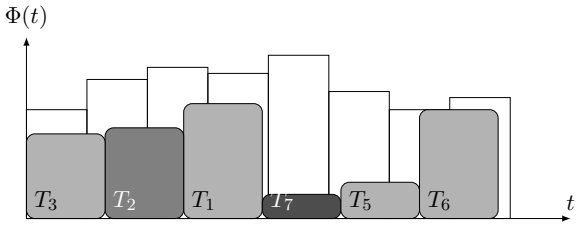


Figure 5.1 – $p_i = p$ and $\varphi_i \leq \Phi_x$ instance

Non-identical tasks and $\varphi_i \leq \Phi_x$

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. In the following, these complexity results are proven in a row.

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 1 *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 ($1|\varphi_i = \varphi \leq \Phi_x|C_{\max}$) 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 disjointed (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 ([53]) which is NP-Complete in the strong sense.

Let us consider an instance $\mathcal{I}_1$ of 3-PARTITION (see Figure 5.2): 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 $1|\varphi_i = \varphi \leq \Phi_x|C_{\max}$ 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. □

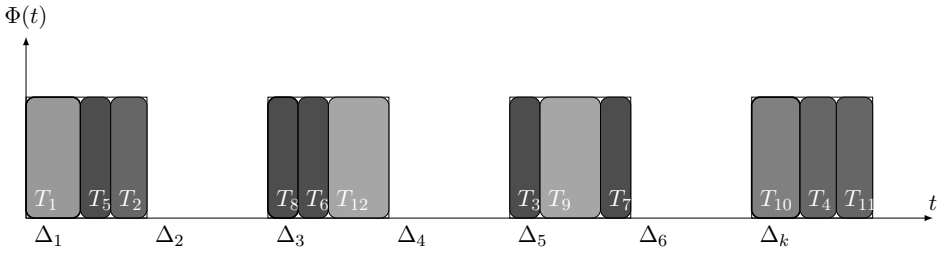


Figure 5.2 – An instance of 3-PARTITION

We now consider the flowtime objective.

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 2 *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 $1|\varphi_i = \varphi \leq \Phi_x|C_{\max}$ problem that is proven NP-Complete in the strong sense in Theorem 1.

Intuively, the proof is based on the definition of a set of disjointed 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 initerval. 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 $1|\varphi_i = \varphi \leq \Phi_x|C_{\max}$ 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 $i \in \{1,\dots,3H\}$, $B/4 < p_i < B/2$ and with $\sum_{i=1}^{3H} p_i = HB$. Does there exist a schedule $\mathcal{T}_1,\dots,\mathcal{T}_H$ such that, for all $h \in \{1,\dots,H\}$ 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 $1|\varphi_i = \varphi \leq \Phi_x|\sum C_i$ is a special case of $1|\varphi_i \leq \Phi_x|\sum C_i$, it is sufficient to prove the NP-Completeness of $1|\varphi_i \leq \Phi_x|\sum C_i$. This concludes the proof. □

5.2.2 Problems with preemption

In the case of data centers that process requests, preemption is allowed. We thus consider the impact of preemption on the scheduling problem complexity.

The $1|\varphi_i \leq \Phi_x, pmtn|C_{\max}$ problem 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, at the beginning of an interval, 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.*

Proof 3 *The optimally of the LWRT algorithm is proved 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, 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, 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.* □

Figures 5.3 and 5.4 illustrate the case where the LWRT task is or is not scheduled at each interval change or when a task is completed. On Figure 5.3 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 5.4 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.

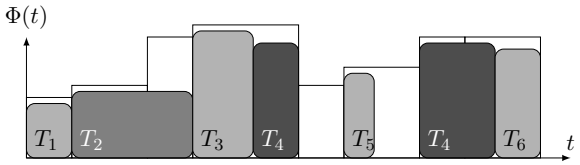


Figure 5.3 – Illustrating example for the LWRT algorithm, T_2 is not the LWRT task for interval Δ_3, T_4 must be run here.

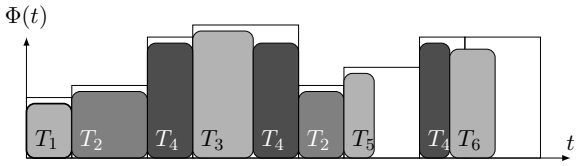


Figure 5.4 – Illustrating example for the LWRT algorithm, part of T_4 has been swapped with T_2 which can be executed sooner than T_5, the makespan is optimal.

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.

5.3 Parallel Problems

We consider now the problem of scheduling a set of tasks on a set of machines. Several sub-problems can be identified from the general parallel problem aside from the classical P, Q, R cases, such as shared memory (multi-core) problems and distributed memory (multi-machine) problems. In the shared memory problems only one machine is used and the tasks are processed by the different cores of the machine. As just one machine must be powered on, we do not need to take static power into consideration and, as for the one machine problems, we only take the task power consumption into account. In this case the machines are cores and we can limit the study to identical machines (P in the Graham notation), we focus in this complexity study on this model.

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 as parallel problems are generalizations of one machine problems. Note however that some simple cases are still polynomial. This, for instance the case when the power is constante, tasks have teh same processing times and need the same power to run, $P|p_i = p, \varphi_i = \varphi, \sum \varphi_i \leq \Phi|\sum C_i$. On the other hand as soon as the power needs differ the problem is NP-Hard.

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 is 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.*

Proof 4 *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 a processor with 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 unit time disjoined intervals that provide a same available power and a machine with three cores. Then we construct 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 machines 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.* □

Note that the proof highlights that the problem $P|p_i = p, \sum \varphi_i \leq \Phi, pmtn|C_{\max}$ is NP-Hard when the tasks have the same size ($p_i = p$) since there no interest to preempt a task.

For the flowtime objective, the $P|\sum \varphi_i \leq \Phi_x, pmtn|\sum C_i$ problem, we can differentiate the particular case where the tasks have the same power need $\varphi_i = \varphi$, which is simple, from the more general case where all the tasks have different power needs. First, in the $\varphi_i = \varphi$ case the SPT algorithm, completed to take both the available power Φ_x and the number of cores P constraints into account, gives an optimal solution even if the tasks have different processing times. Second the problem where the tasks have different power needs is NP-Hard. The problem $P|p_i = p, \sum \varphi_i \leq \Phi_x, pmtn|\sum C_i$ is indeed equivalent to $P|\sum \varphi_i \leq \Phi_x, pmtn|C_{\max}$ since the tasks do not need to be ordered as they have the same processing time. This implies that the more general case $P|\sum \varphi_i \leq \Phi_x, pmtn|\sum C_i$ is NP-Hard too.

We recall that the parallel problems examined above are represented by a multi-core machine model. For more general problems such as the multi-machine model both when each machine is a multi-core machine, like the one examined above, or a one core machine, they are generalization of the multi-core problem since they add the complexity of the static power consumption of machines and the switching on/off decision. For that reason, parallel problems that were proven above to be NP-Hard for the multi-core model are NP-Hard for the multi-machine model as well.

5.4 Synthesis

In this section, we presented the work done on the complexity of our scheduling problems. We examined and proved the complexity of different problems for both the one machine and parallel cases of our model. Tables 5.1 & 5.2 provide a summery of the results of this complexity study without and with the power constraint, respectively.

After proving that most cases of our problem are NP-Hard, we present in the following chapter several heuristics to solve the optimization problems of off-line scheduling independent sequential

	1 Machine		N Machines	
	no-pmtn	pmtn	no-pmtn	pmtn
C_{max}	P	P	NP-Hard	P
Flowtime	P	P	P	P

Table 5.1 – Complexity results without green power constraint

	1 Machine		N Machines	
	no-pmtn	pmtn	no-pmtn	pmtn
C_{max}	NP-Hard	P	NP-Hard	NP-Hard
Flowtime	NP-Hard	Open	NP-Hard	NP-Hard

Table 5.2 – Complexity results with green power constraint

tasks on a parallel platform under variable power constraint and the objective is either minimizing the makespan or minimizing the flowtime. We point out that all the heuristics presented in the following chapter are designed for no-preemption scheduling, as preemption is usually not allowed in a real life HPC system.

The trend toward server-side computing and the exploding popularity of Internet services due to the increasing of demand for networking, storage and computation has created a world-wild energetic problem and a significant carbon footprint. These environmental concerns prompt to several green energy initiative aiming either to increase data center efficiency and/or to the use of green energy supply. In this regard, As part of the ANR DATAZERO project, many researchers are working to define main concepts of an autonomous green data center only powered by renewable energies. Thus, the present paper proposes a mixed integer linear program to optimize the commitment of a hybrid energy system composed of wind turbines, solar panels, batteries and hydrogen storage systems. The approach is used to supply a data center demand and takes the weather forecasts into account at the time of optimization.

Chapter 6

Scheduling strategies

In this chapter we tackle the scheduling problem of minimizing the makespan and the flowtime under green power supply constraints. After proving this problem to be NP-hard in Chapter 5, we propose here scheduling heuristics that take power constraints into consideration. The two concerned models are the two computation platforms proposed in Section 4.2, i.e., a multi-core machine and a multi-machine computation platform.

To evaluate the scheduling heuristics equally, we developed a scheduler script that consists of two main parts, the planner and the executer. The scheduler takes as input a set of tasks and the green power production, expressed by a set of intervals of time with the available power level at each interval. As a first step, the planner which corresponds to the scheduling heuristics sorts the tasks in the task list based on a certain priority. Then, the executer carries out the schedule of the ordered tasks under the power availability constraint and calculates the makespan and the flowtime resulting from executing these tasks in this order. The executer part of the scheduler is the $PlaceTask$ function. As suggested by its name, this function *places* each task in the earliest available time slot where the execution of this task would not violate the power constraint.

6.1 Place task

Two versions of the $PlaceTask$ functions were created to address both the multi-core and multi-machine models respectively, namely the $PlaceTaskCore$ and the $PlaceTaskMachine$. We recall that the main difference between the two models is the availability of processing units. In the multi-core model, a processing unit is a processor's core, and the static power necessary to turn on the processor is already deducted from the green power supply. In this model, the executer is responsible for performing the task to time slot allocation, by finding the earliest suitable time slot for each task at a time. On the other hand, the multi-machine model consists of m identical multi-core processors, and the processing unit is a core of one of these m processors. In this model, in addition to performing task to time slot allocation, the executer is responsible for performing task to processor allocation, which includes deciding which machine to switch on/off and taking into account the static power and the time delay costs of this decision. Note that most of the planner's scheduling heuristics are designed for both computational platform models, however, some of them are only designed for the multi-machine model as they affect the switch on/off decision.

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

Input:
p_i, φ_i: processing time and power consumption of task T_i respectively
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
```

6.1.1 *PlaceTaskCore*

According to the order that is set by a scheduling policy, we pass each task at a time to the *PlaceTaskCore* function, along with the set of time intervals that represent the power production curve. A task T_i is represented by its processing time p_i and its power consumption φ_i. A time interval Δ_x is represented by its length δ_x and the amount of green power production during this time Φ_x.

In the first eight lines of Algorithm 1, we try to find a time slot that consists of a subset of sequential intervals at which enough power is available along with at least on free core. The searching process starts at an interval Δ_j, for example, and it adds one interval at a time to check if it is suitable for the task execution or not. If at some point an unsuitable interval arrives, then the searching process stops and returns the index of the interval at which the search has failed (lines 2-4), this way the next task placement attempt starts at interval $\Delta_{failed+1}$ rather than Δ_{j+1} knowing that trying to place the task before that will fail because of the presence of Δ_{failed}. This can lead to an important reduction in simulation times.

We point out that we use float values for interval length Δ_x and task processing times p_i in a first implementation that is used in the two multi-core experiments 7.4.1 and 7.5.1, which present our investigation in the multi-core platform using List algorithms and Genetic algorithms, respectively. As a result, several tasks can be scheduled for execution at the same interval, and a task can be executed across a time slot of several sequential intervals, on the condition that all intervals within

this time slot are suitable for the task execution. When a task is executed in an interval, the available power during that interval is reduced by the amount of the task's power consumption φ_i, and the number of free cores is reduced by 1. If the execution of the task ends before the end of the interval, to avoid unnecessary slack times, the interval in this case is split into two smaller intervals. The first new interval starts at the beginning of the original interval and ends with the end of the task's execution. The second new interval starts at the end of the task's execution and ends at the end of the original interval. Thus, the available power level and the number of free cores are only affected in the first new interval, where the task execution took place.

This process is no longer necessary in our second implementation that is used in the third multi-core experiment 7.6.1, which presents a comprehensive comparison between all the proposed heuristics (List, Binary search, Stripe and GA) on the multi-core platform, where integer values are used for the task processing times p_i and the length of intervals was unified to 1 time unit. This choice saves a lot of simulation time during the experiments, at the same time, it does not cause any significant difference from the results obtained while using float values. Figures 6.1 and 6.2 present a comparison between the results obtained using float and integer data sets respectively. Each figure represents the heuristics that produce the best average performance regarding the makespan objective with increasing values of $p_{i_{\max}}$ (over the horizontal axis) and $\varphi_{i_{\max}}$ (over the vertical axis). In the cases where each figure shows a different best heuristic the difference in the average performance value over 150 executions of both heuristics is less than 0.7%.

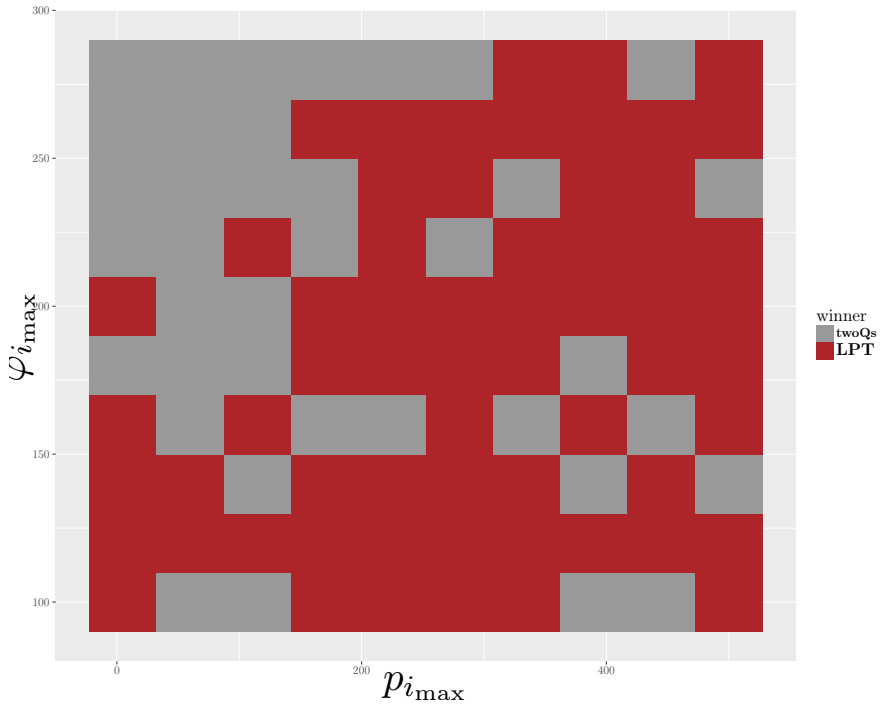


Figure 6.1 – Float data sets results

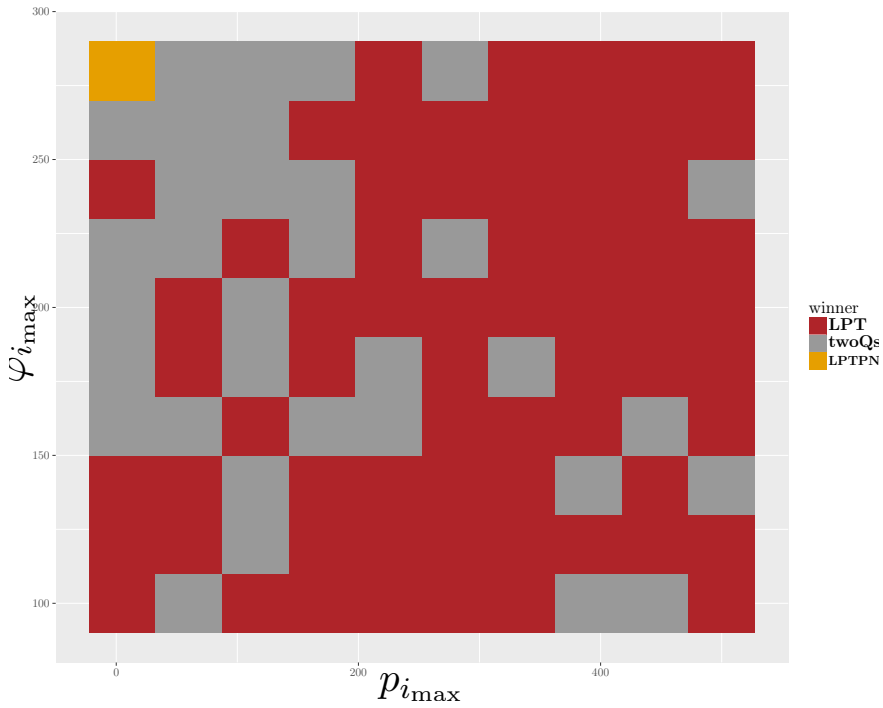


Figure 6.2 – Integer data sets results

6.1.2 *PlaceTaskMachine*

Algorithm 2: $PlaceTaskMachine(p_i, \varphi_i, x)$

Data:
$delay$: boolean, indicates a delay due to booting time
$found$: boolean, initialized to $true$
$ProcList$: set of available processors
On: a subset of $ProcList$ of "On" processors
Off: a subset of $ProcList$ of "Off processors
Son: a subset of $ProcList$ of Switching "On" processors
$Soff$: a subset of $ProcList$ of Switching "Off processors
$StartTime$ the time it takes to switch the processor on
$StaticPower$ the static power consumption of each processor
$nbCores$ number of available cores on each processor
$TotalnbCores$ sum of available cores from all processor in each interval
$StartPower$ the power drawn by the processor while being switched on
Result:
ProcAlloc: Task to machine allocation task T_i
TimeSlotAlloc: Task to time slot allocation task T_i

```
while Proc_chosen = NULL do
    Proc_chosen ← FindProc([On, Soff], [x → x + p_i])
    if Proc_chosen ≠ NULL then
        delay ← false
    else
        Proc_chosen ← FindProc([Son, off], [x + StartTime → x + StartTime + p_i])
        if Proc_chosen ≠ NULL then
            delay ← true
        else
            Proc_chosen = NULL
            Exit While loop
if Proc_chosen ≠ NULL then
    ProcAlloc.i ← Proc_m
    if delay = false then
        timeSlot = [x → x + p_i]
        for Δ_k ∈ timeSlot do
            if Φ_k ≥ (φ_i + StaticPower) then found ← true
    else
        timeSlot = [x + StartTime → x + StartTime + p_i]
        for Δ_k ∈ timeSlot do
            if Φ_k ≥ (φ_i + StaticPower & Φ_x ≥ StartPower) then found ← true
    if found then
        TimeSlotAlloc.i ← timeSlot
        for Δ_x ∈ timeSlot do
            Φ_x-=φ_i: Proc_chosen.State in Δ_x = On
            TotalnbCores_x-=1
            Proc_chosen.nbCores-=1
else
    Check next interval
return ProcAlloc and TimeSlotAlloc
```

Algorithm 3: $FindProc(ProcSubSet, [a \rightarrow b])$

```
Data:
  [a → b]: a search range
  ProcSubSet: a subset of the processor list
Result:
  Proc_chosen: the chosen processor initialized to NULL
1 while Proc_chosen = NULL do
2     for Proc ∈ ProcSubSet do
3         if Proc has at least one free core in [x → x + p_i] then
4             Proc_chosen ← proc
5 return Proc_m
```

In the multi-machine case, the $PlaceTaskMachine$ function is responsible for task to machine allocation in addition to task to time slot allocation. We recall that:

- Each machine is a multi-core processor and that the processing unit is a processor's core which is available for task execution only when that processor is turned on.
- A turned on processor consumes static power P^{static} regardless of whether it is executing tasks or not.
- A processor that is executing tasks also produces a dynamic power consumption $P^{dynamic}$ which is equal to the sum of the power consumption φ_i of the tasks running on it.

The $PlaceTaskMachine$ function takes one task at a time according to their order in the task list, and selects the processor on which the task will be executed, then it allocates it to the earliest available time slot possible. At the beginning of the run time, all processors are switched off, the decision to switch a processor on or off is made inside $PlaceTaskMachine$ based on two factors: the instantaneous computational demand and the instantaneous power supply constraint. In other words, if there is a need to turn on one more processor and if there is enough power to do so, the executer turns a processor on. In opposite, if a running processor finishes the execution of all its tasks in an interval where the computational demand does not require using this processor, or the available power does not allow the execution of any of the remaining tasks, the executer then switches the processor off.

A processor can be in one of four states at a given time interval: on, off, switching on (Son) or switching off (Soff). $PlaceTaskMachine$ is designed as a greedy function, in the sense that it schedules each task as early as possible on the first suitable processor under the following rules:

- No preemption nor job migration are allowed, therefore if a task starts executing on a machine, it should finish the execution on the same machine. The $PlaceTaskMachine$ function checks the availability of cores on each machine during enough time ahead to execute the task, before assigning the task to a machine. Then the available power level is checked during the affected intervals to verify that there is enough power for both the processor's static power consumption P^{static} and the task's power consumption φ_i.
- If a task is assigned to a processor that is performing a shutdown during the first interval of its execution Δ_{start}, the state of the processor is modified to "on", and the execution of the task then starts at Δ_{start}, as if the task is assigned to a processor that is already "on" during Δ_{start}.

- If a task is assigned to a processor that is switching on or is in an "off" state during Δ_{start}, then a delay is applied while the processor boots, and the execution of that task starts at $\Delta_{start+t_j^{off\rightarrow on}}$.
- Before shutting a processor off, the $PlaceTaskMachine$ function looks ahead to check if during the processor's shutdown time there is a switch on for that processor, then the processor is kept turned on.
- When deciding to switch a processor on at interval Δ_j for task execution, the $PlaceTask$-$Machine$ function looks back to check a shutting down process is ongoing in recent intervals, if this is the case, then its state is changed to "on" during the previous intervals instead of switching it off and then switching it right back on, this reduces the delay caused by unnecessarily rebooting the processor.
- The power consumption during the start up and the switch off of a processor is usually higher than its static power consumption. Therefore, in an interval where a processor is scheduled to be switched on or off, the switching on/off power consumption must be considered.

The $PlaceTaskMachine$ function, given in Algorithm 2 starts by examining each time interval of the passed power envelope starting with the earliest one. In each interval Δ_x if there are processors that are already turned on, we check if one of them has a free core during the entire duration of the task execution $\Delta_x \rightarrow \Delta_x + p_i$ ($l1 \rightarrow 4$ in Algorithm 2). The verification of core availability is done by calling the $FindProc$ function explained in Algorithm 3. If a free core is found, all time intervals in that range are checked to verify if the available power level is high enough for the task's power need $\Phi_k \geq \varphi_i$ for k $\in [x, x + p_i]$ ($l12 \rightarrow 21$ in Algorithm 2). The advantage of using an already running processor is that the static energy corresponding to the processor's static power consumption P_j^{static} during a task's processing time, is already lost for where the processor is already running. In addition, the switching on delay $t_j^{off\rightarrow on}$ is avoided. We point out that a processor's static energy consumption is considered a waste since it is not being used for task execution.

If both the core availability and power sufficiency conditions are met, then the task is scheduled for execution during the time slot $\Delta_x \rightarrow \Delta_x + p_i$ on the switched on processor who owns that core ($l22 \rightarrow 27$ in Algorithm 2). If the task cannot be executed over this time slot because of lack of power, then we resume the search at the time interval following the one that lacked the necessary power ($l29$ in Algorithm 2). If the power condition is met but no running processor is found suitable, the scheduler must decide whether to run a switched off processor, resume a processor that is scheduled for switch off in this interval or wait for a processor that is scheduled for switch on in this interval to finish booting. Using either a switching on or switching off processor is more power efficient than using a processor that is turned off in this interval, because the available power has already suffered the loss of $P_j^{off\rightarrow on}$ or $P_j^{on\rightarrow off}$ respectively. However we start by choosing a processor that is switching off if possible, because it is the only state between these two cases that does not cause a delay. The core availability check is the same as in the previous case, while the new power availability conditions are $\Phi_w \geq \varphi_i - (P_j^{static} - P_j^{on\rightarrow off})$ for w $\in [x, x + t_j^{on\rightarrow off}]$ and $\Phi_k \geq \varphi_i + P_j^{static}$ for k $\in [x + t_j^{on\rightarrow off} + 1, x + t_j^{on\rightarrow off} + 1 + p_i]$. If a switching off processor is found suitable, then the state of this processor is changed from switching off to on during $[x, x + t_j^{on\rightarrow off}]$, and the available power level in this range is modified accordingly: $\Phi_w = \Phi_w + P_j^{on\rightarrow off} - P_j^{static} - \varphi_i$ for w $\in [x, x + t_j^{on\rightarrow off}]$ and for the rest of the intervals $\Phi_k = \Phi_k - P_j^{static} - \varphi_i$ for k $\in [x + t_j^{on\rightarrow off} + 1, x + t_j^{on\rightarrow off} + 1 + p_i]$.

Both a processor that is switching on or one that is switched off produce a delay equals to $t_j^{off \to on}$. A processor that is switching on however is already running in the next intervals and its static energy consumption is thus already lost. Choosing a switching on processor is therefore more energy efficient than running a second processor and losing twice as static energy. The processors are checked for a free core in the time slot $\Delta_{x+t_j^{off \to on}} \to \Delta_{x+t_j^{off \to on}+p_i}$. If the chosen processor is already switching on in $[\Delta_x, \Delta_{x+t_j^{off \to on}}]$ then the power availability condition is $\Phi_k \geq \varphi_i$ for k $\in [x + t_j^{off \to on} + 1, x + t_j^{off \to on} + 1 + p_i]$.

If the chosen processor is off, then the decision is made to turn it on, thus to change its state in $[\Delta_x, \Delta_{x+t_j^{off \to on}}]$ to switching on. The power availability conditions in this case are $\Phi_w \geq P_j^{off \to on}$ for w $\in [x, x + t_j^{off \to on}]$ and $\Phi_k \geq \varphi_i + P_j^{static}$ for k $\in [x + t_j^{off \to on} + 1, x + t_j^{off \to on} + 1 + p_i]$.

On/off management performed by the $PlaceTaskMachine$ function

Since we assume using only renewable energy sources, our goal is not to reduce the power consumption rather than to better use this available power while finishing the job as early as possible. Theoretically, keeping all the processors turned on all the time should give the best performance and thus yield the shortest makespan, by avoiding the delays caused by booting and shutdown operations.

However, the intermittent green power supply produces cases where at some intervals of time the available power is not high enough to turn on all the processors. How many processors should be turned on then? and which ones? In addition, a running processor consumes static power even without executing tasks and since the instantaneous green power supply during an interval of time is limited, it does not make sense then to turn on too many processors and waste unnecessary static power consumption when not enough computational demand is present to keep the running processors busy. Therefore, we assume that all processors are off, and the $PlaceTaskMachine$ function turns them on one by one only when necessary, i.e., when all running processors are busy and there is still enough power to execute at least one additional task, keeping in mind that the objective is to finish the task execution as early as possible.

As mentioned above, the planner part of the scheduler consists of several scheduling heuristics. The first set of scheduling heuristics deals only with the task list, they are thus responsible for setting the order of the task list. The second set of scheduling heuristics combines the task list sorting step with a search range limiting technique that limits the schedule to only a part of the power envelope. Dealing with only a fraction of the power envelope rather than the entire time horizon $\mathcal{H}$ allows more complex scheduling decisions to be made without increasing the time complexity.

This work's third contribution (after the complexity study and the $PlaceTask$ function) lays in proposing and evaluating several scheduling heuristics, some of which deploy different sorting priorities to the task list, and others deploy different search range limiting techniques. Our initial results show that different sorting policies produce different makespans and flowtimes, therefore even the search range limiting heuristics are tested using all the proposed sorting policies.

The output of the planner is thus a sorted task list and either the entire power envelope represented by a set of intervals, or a segment of that power envelope represented by a subset of intervals. Therefore, we can classify the tested heuristics into two main approaches, namely heuristics that generate a sorted set of tasks passed to $PlaceTask$ and heuristics that generate power envelope partitions passed to $PlaceTask$.

6.2 Planner heuristics

The *PlaceTask* function in both of his versions represents the executer part of our scheduler. The following scheduling heuristics make up the planner part of our scheduler, the planner is responsible for generating the input data, a sorted set of tasks, used by the executer.

Our scheduling process is thus carried out on two steps:

- A scheduling heuristic within the planner sorts the task list, either based on a priority criteria like in list algorithms or using a scoring function of different orders like in GA.
- The ordered task list is then passed to the executer one task at a time, based on the order set by the scheduling heuristic, the greedy function *PlaceTask* allocates one task at a time in the earliest suitable time slot, taking into consideration the availability of power and processing units.

The first step is performed by the set of scheduling algorithms presented in this section, where the whole time horizon of the power production, in other words the entire interval list is available for task execution. Whereas, in the set of scheduling algorithms presented in the next section combine between the sorting features presented in this section, with some techniques to limit the search space to parts of the power production time horizon, in other words to a subset of the interval list.

6.2.1 List scheduling algorithm

List scheduling is one of the most common algorithms in scheduling theory. It can give good solutions within low time complexity for many scheduling problems. A list algorithm consists of two steps. The first step is to assign a priority value to each task in the scheduler queue. The priorities are assigned statically before scheduling tasks begins. The second step is to allocate the tasks, starting with the task with the highest priority, to a resource and schedule it for execution as early as possible. The second step is identical for all the tested list algorithms, and it is carried out by the *PlaceTask* function. The only difference between the list algorithms is which of the task's characteristics are used to assign the priorities.

It is worthwhile noting that the discretization of the time in intervals increases the complexity of the algorithms. The worst case complexity of the *PlaceTask* function depends on X, the number of intervals. As the algorithms iterates on the intervals to schedule a task then their complexity depends on X^2. Considering that the complexity of a list based algorithm depends on the list ordering ($O(n\log(n))$, n the number of tasks), the complexity of the list based algorithms with power constraints becomes $O(X^2 n\log(n))$. It turns out that the complexity of placing tasks in intervals heavily weighs on the computation times: running the whole set of experiments to compute the heatmaps based on 250 intervals takes 2 days and more than one week when based on 500 intervals.

Our problem can be described as an optimization problem whose objectives (the makespan and the flowtime) lay on a time axis, while the scheduling constraints (the power availability versus the power consumption) lay on a power axis, it is thus a two dimensional problem (time and power dimensions). For this reason we test priorities that consider the time axis, priorities that consider the power axis and priorities that consider both axis simultaneously. The list based heuristics that use those priorities are the following:

LPT, for Largest Processing Time first, is the first tested list algorithm because it is known in the literature to give good solutions for makespan in classical scheduling problems (without

the green power constraints). As suggested by its name, LPT sorts the tasks in the scheduler queue starting with the task with the highest processing time p_i.

LPN, for Largest Power Need first, is the second list based heuristic. It sorts the tasks in the scheduler queue starting with the task with the highest power need (φ_i). The heuristic is sheer power dimension oriented.

LPTPN, for Largest Processing Time times Power Need first, is an attempt to prioritize both the processing time and the power consumption of tasks, and therefore consider both dimensions of the problem. The tasks are thus sorted in the queue starting with the task with the highest ($p_i \times \varphi_i$) value.

SPT, for Shortest Processing Time first, is a typical optimal solution for the flowtime minimization objective in classical scheduling problems. We therefore test its performance regarding the minimizing the flowtime objective in our problem.

TwoQs, for Two Queues, try to balance the problems of the **LPTPN** algorithms in which we cannot tell, if a task T_i has high priority because it has a long processing time p_i or a high power consumption φ_i. Since the objective lays in the time dimension, we search for a way to give an edge for longer tasks. For this reason we propose the TwoQs heuristic, an adaptation of list scheduling that alternates between two queues. The first is LPTPN's queue that takes into consideration both dimensions of the problem, and the second is LPT's queue to give an edge for longer tasks. Once a task is scheduled, it is removed from both queues. This heuristic exploits the advantages of two priority assignments at the same time, privileging tasks that would have low priority in queue but might have high priority in the other.

LPP, for Least Possible Places first. The characteristics of our problem makes it possible to have some tasks that are more critical than others. Such tasks might have fewer possible places in the time horizon $\mathcal{H}$ where they can be executed, therefore they are harder to schedule. If a suitable place for a critical task is given to another task due to the used sorting priority, the next suitable place might be far away down the time horizon, which weighs badly on the optimization criterion. To privilege tasks that are harder to schedule regardless of whether it is because they have long processing times p_i or high power consumption φ_i or both, we add another step to the list algorithm. At first, we calculate the number of time slots in which it is possible to execute each task, then, we sort the tasks in the queue starting with the task with the Least Possible Places first.

Random, test the case where the task list is left in its original random order. This heuristic is used as a control case that ordering the tasks is worth it.

Although list algorithms are simple and not designed for complex problems, their low time complexity is convenient for testing different sorting policies and analyzing the effect of the task list's order on optimization objectives. Our initial results on list algorithms 7.4.2 demonstrate that the characteristics of the task list play an important role in determining which sorting policy to use. On the other hand more sophisticated heuristics have to be assessed so as to challenge the list algorithms since we do not know how far they can be improved and what is the cost of these improvements. The genetic algorithm proposed in the following section is designed to find good solutions regardless of the characteristics of the task list.

6.2.2 Genetic algorithm GA

Genetic algorithm (GA) is an approach used for solving optimization problems based on natural selection, the main concept in evolution. GA can be applied to solve optimization problems that are not well suited for standard optimization algorithms, including problems in which the objective function is discontinuous, non differentiable, stochastic, or highly nonlinear. If we consider that each solution is an individual of a population and a set of possible solutions represent a population, GA repeatedly applies genetic operators to modify a selected subset of its individuals. First, a selection process is performed to chose individuals from the current population. These chosen individuals then become parents used to produce the children for the next generation. The selection process can be done completely randomly or it can favor the good individuals by giving them a higher probability in the selection function. In addition, elitism selection can be used to pass the best individuals of a population to the next generation. With each new generation, the population evolves toward an optimal solution. The three main genetic operators are:

- Selection, selects the parents that are used to produce the next generation.
- Mutation, modifies randomly one parent to produce one child of the next generation.
- Crossover, two parents are paired to switch parts of their genes to produce two children of the next generation.

For our problem, we consider that a solution or an individual is the order of the task list in which the tasks are passed to the executer one at a time, which is a classical approach in genetic algorithms for task scheduling. The Genetic algorithm is iterative. At each iteration, it applies the three genetic operators to generate new orders of the task list, new children, that are all evaluated and given a fitness value. An individual is considered as a good solution if it has a good fitness value. At the end of each iteration, only the good solutions are passed to the next generation. Then, at the end of the GA execution, the solution that yields the minimum C_{max} is used to order the task list for the makespan minimization problem, and the solution that yields the minimum $\sum(C_i)$ is used to order the task list for the flowtime minimization problem.

Schedule representation

The first issue to be solved when designing a genetic algorithm is how to formalize the problem as a genetic problem: what does a chromosome and each of its genes represent, how to calculate the fitness value? Generally the fitness choice corresponds to the optimization objective. For our problem for example, the fitness equals to the makespan or flowtime. Therefore, since the objective is to minimize these two values, we consider that the lower the fitness, the better the corresponding individual. There are however numerous ways to code a solution and the representation choices highly impact the results so that several solutions must be explored.

As a first try, each chromosome represents a possible schedule. A chromosome is made of n genes with the i-th gene being the time interval where task T_i is scheduled for execution. The time interval selected for each task is chosen from a subset of all suitable intervals for that task ($\Phi_x > \varphi_i$). This subset is calculated for all tasks once before generating the initial population. Since tasks are scheduled in parallel, several genes can have the same time interval, i.e., several tasks can be scheduled in the same time interval. The algorithm tries to schedule the task list according to

the schedule presented by each chromosome, each task is thus executed at the time interval defined in its corresponding gene. If the schedule is feasible, the makespan or flowtime resulting from it is set as the fitness of that chromosome, if not, then its fitness value is infinity.

This approach has several drawbacks. The process of calculating the time interval possibility subsets for each task is time consuming and the generation of non-feasible schedules when applying the mutation and cross-over operators leads to time losses when generating and evaluating the chromosomes.

Our initial results regarding the list based heuristics showed that the order in which the tasks are passed to the executer part of the scheduler makes a big difference in the resulting makespan and that, depending on the characteristics of the task list, using one priority criteria or another impacts the performance of the resulting schedule. Therefore, an other GA was developed where each chromosome represents an ordered list of tasks that is then passed to the executer that, in turn, computes the corresponding schedule. The first obvious advantage is that all combinations of the task list order are valid if the available power constrains of the interval set allows it. The coding and implementation is also simpler and faster than in the previous proposition due to avoiding infeasible individuals, and thus solutions can be found in reasonable computation time.

GA description

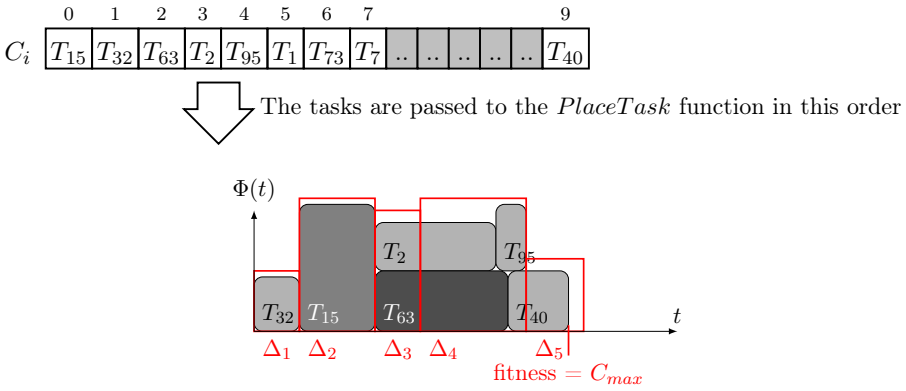


Figure 6.3 – Illustrating example of a chromosome and the corresponding schedule

Figure 6.3 illustrates the second proposed GA approach. The chromosome representation is a simple ordered list of the tasks. To be valid it must contain all the tasks of the list and not twice the same task. The chromosomes are then passed one at a time to the greedy function $PlaceTask$ which schedules the tasks based on their order in each chromosome and depending on the green power availability constraint. It then returns the corresponding makespan or flowtime of each schedule to compute the fitness of all chromosomes. In Figure 6.3 for instance, T_{15} is the first task to be scheduled, the second task is T_{32}, and T_{40} is the last task to be scheduled. Due to power constraints task T_{15} cannot be scheduled first although it is the first in the list, then, with respect of the power availability constraint, task T_{32} is executed in the first interval Δ_1 while task T_{15} is delayed until the second interval Δ_2 where there is enough power available.

This chromosome representation makes generating a new individual faster than in the first one, the GA only needs to shuffle a list of integers $[1 \rightarrow n]$ that represent the indexes of all n tasks. This

allows the exploration of bigger populations than in the first GA representation.

Algorithm 4: `geneticAlgorithm(`$\mathcal{T}$`, `Δ`, 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[4:50] ← 46 random solutions
 7 calculateFitnessOfPopulation(currentGeneration)
 8 currentGeneration.sort()/* fitness:  in increasing order                    */
 9 while stopCounter ≤ nbI do
10     oldBest ← currentGeneration[0]
11     nextGeneration ← []
12     nextGeneration[0:10] ← currentGeneration[0:10]
13     for i=1 to 15 do
14         mutant ← mutation(selection(currentGeneration))
15         nextGeneration.append(mutant)
16     for i=1 to 15 do
17         mutant ← chunckMutation(selection(currentGeneration))
18         nextGeneration.append(mutant)
19     for i=1 to 10 do
20         C1, C2 ← selection(currentGeneration)
21         newC1, newC2 ← crossOver(C1, C2)
22         nextGeneration.append(newC1, newC2)
23     calculateFitnessOfPopulation(nextGeneration)
24     nextGeneration.sort()
25     currentBest ← nextGeneration[0]
26     currentGeneration ← nextGeneration
27     if oldBest.fitness − currentBest.fitness = 0 then
28         stopCounter++
29     else
30         stopCounter← 0
31 return currentBest
```

Algorithm 4 gives the main genetic algorithm used to manage the chromosomes and to generate the schedules. In this GA the population size is set to 50, and by setting the number of intervals X high enough, it is valid to assume that all initial chromosomes give a feasible order of the task list.

Four individuals of the initial population are produced by the priority queues of four list scheduling algorithms: LPT, LPN, LPTPN and twoQs. We recall that the LPT algorithm is the classical Largest Processing Time list algorithm that sorts the tasks with the largest p_i first. The LPN algorithm does the same but using the task's power consumption φ_i as priority. The LPTPN algorithm uses the product of processing time and power consumption $p_i \times \varphi_i$ to take both properties into account. Finally the twoQs algorithm creates two queues, one sorted by p_i and the other by $\varphi_i \times p_i$, and takes in turn one task in each queue. The other 46 individuals are randomly generated by shuffling a list of integers $[1 \rightarrow n]$ that represents the indexes of all n tasks.

Selection

Another important factor in a genetic algorithm is the individual selection process. When choosing which individuals to apply the genetic operators on, we have implemented both random selection

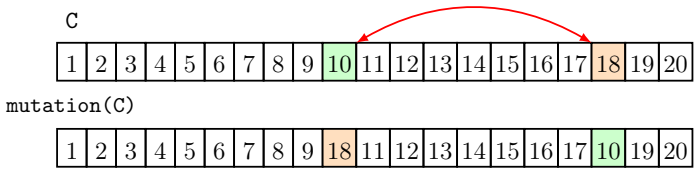


Figure 6.4 – `mutation(C)`

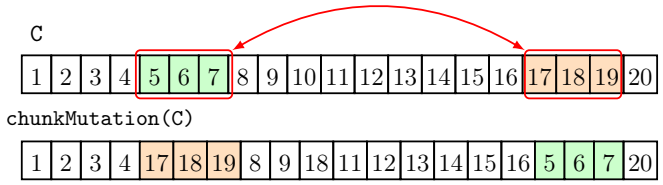


Figure 6.5 – `chunkMutation(C)`

and wheel selection. Wheel selection, also known as roulette wheel selection or fitness proportional selection, assigns a selection probability to the individual proportional to their fitness.

We assume that a crossover operator makes bigger changes in a chromosome than the mutation operator, therefore we first test a configuration of GA that only applies mutation. In Algorithm 4, first, the best ten individuals are copied to the next generation (elitism). Then, based on the used selection (wheel or random), 15 chromosomes are selected for 1-gene mutation (see $[l13 \rightarrow l15]$ in Algorithm 4) and another 15 chromosomes are selected for chunk mutation ($[l16 \rightarrow l18]$).

As for the rest of the evaluated GAs, 20 additional chromosomes are selected to perform 10 crossovers. In this study, we evaluate three crossover techniques. For each evaluation, one of these three crossovers is applied at $l21$ in Algorithm 4. Finally, the fitness of the new off-springs is calculated, and the entire process is repeated for the next generation until the GA runs nbI steps without improvement.

The mutation and crossover operators are explained in the following.

Mutation

Two different mutation operators are used in the GA algorithm. In the 1-gene mutation operator, two points on the chromosome are randomly picked, and the values of the two corresponding genes are interchanged as illustrated in Figure 6.4. The chunk mutation operator uses the same concept as 1-gene mutation, only except swapping two "1-gene"s, it swaps 2 "chunk"s of random size between $1 \rightarrow 10$ of the chromosome at two randomly selected points (Figure 6.5).

Due to the characteristics of our problem, it is interesting to study the effect of small modifications on a given solution. Applying too many modifications might eventually be as arbitrary as a random solution. Hence, the first GA we evaluate applies only 1-gene or chunk mutation genetic operators (it skips lines $[l19 \rightarrow l22]$ in Algorithm 4). This algorithm is named noX, for no crossover, algorithm. With the two different selections, random (R) and wheel based (W), we define the two algorithms: noX-R and noX-W.

Algorithm 5: `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
```

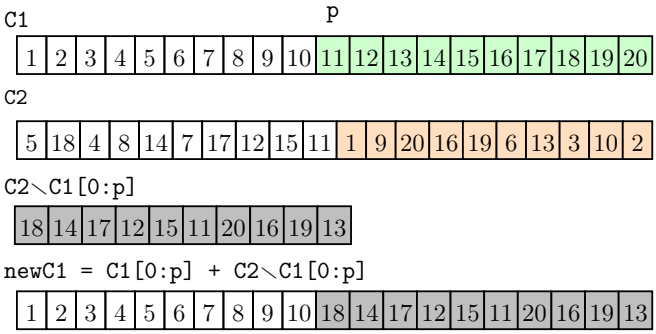


Figure 6.6 – `OnePointCrossOver(C1, C2)`

One point crossover

Crossover is one of the fundamental operators of GA and many crossover mechanisms are described in the literature. We here evaluate the effect of using three different crossovers. First, we evaluate the most common one point crossover. A randomly selected crossover point splits both parents into two parts. We cannot however directly exchange the chromosome parts as it is usually done because, doing this, the same task could appear twice in one chromosome, which would be a non-sense for a schedule. In our crossover we generate two new chromosomes, each one keeping its parent's head, while the genes of its tail (the remaining tasks) are ordered according to the order of the other parent. Algorithm 5 details the one point crossover operator.

Figure 6.6 illustrates this crossover operator. The crossover point is set to p. Chromosome $newC1$ takes the head of $C1$ and orders the remaining tasks according to the task order of $C2$. For instance, task 5 that is already present in the head of $C1$ is not duplicated in the tail of $newC1$ and thus task 18 appears first in the second part of the chromosome.

The one point crossover algorithms are named 1pX. With the two selection operators we have 1pX-R, for random selection, and 1pX-W, for wheel selection.

Two point crossovers

Considering that the one point crossover generates large changes in the chromosomes we have also implemented two other crossover operators that use two points.

The first operator is a two point crossover called Order Crossover (OX). Two crossover points are randomly drawn. Each parent keeps its middle part $C_i[p_1 : p_2]$. Then, the genes of its edges starting from $C_i[p_2 + 1]$, circling back to $C_i[p_1 - 1]$ are ordered according to their order in the other

Algorithm 6: `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
```

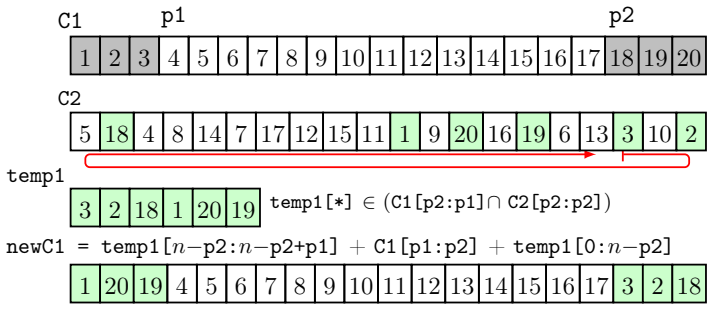


Figure 6.7 – `OrderCrossOver(C1, C2)`

parent starting from $p_2 + 1$. The Order Crossover operator is given by Algorithm 6.

For example in Figure 6.7, the first new off-spring cC_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 cC_1 in the same circular manner.

The algorithms that use this crossover operator are named OX. We thus have OX-R and OX-W depending on the associated selection operator.

The second two point crossover operator is a classical two point crossover. Each off-spring has the same edges as its parent $C_i[0 : p_1 - 1]$ and $C_i[p_2 + 1 : n]$. The genes in the middle of each parent $C_i[p_1 : p_2]$ are reordered in the off-spring according to their order in the other parent. This operator is detailed in Algorithm 7. Figure 6.8 illustrates an example of this operator. The operator is named Middle Cross Over and the corresponding algorithm is identified as MX (MX-R and MX-W with their selection operators) in the following.

The GA is designed to keep producing new generations until no further improvement is found between two successive generations, which means that an "as good as possible" solution was found. However, there is a high chance that this is a local optima. To reduce the possibilities of stopping when a local optima is found, the GA keeps producing new generations even if the quality of the new

Algorithm 7: `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
```

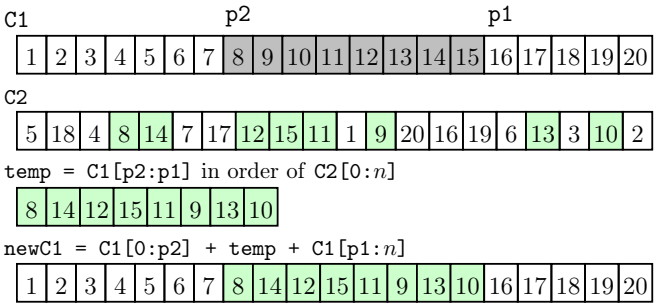


Figure 6.8 – `MiddleCrossOver(C1, C2)`

proposed solutions is not improved during fifty iterations, or until a better local optimal solution is found. Several stopping points were tested during our experiments (fifty and ten iterations without improvement). Our simulations showed that the further the stopping point is, the better the quality of the final solution, however, the more time it takes to find this solution. GA thus, presents a trade-off between time and performance.

6.3 Power envelope partition based algorithms

In this section we present algorithms that generate power envelope partitions, or search space/range of the power production time horizon, and use $PlaceTask$ in these partitions.

6.3.1 Best fit binary search

For the multi-core platform, the number of processing units represented by CPU cores is limited. We define the power waste as the available power left in an interval after deducting the power consumption of the tasks scheduled for execution in it. Power waste can occur either because all cores are busy, or the power level left is not high enough to execute any of the remaining tasks. Even though our objective is to minimize the makespan rather than reducing the power consumption or even the power waste, we argue that better utilization of the available power might lead to a shorter makespan in some cases. For example, Figure 6.9 illustrates the case of two tasks T_1 and T_2, with power consumptions equal to $\varphi_1 = 10$ and $\varphi_2 = 15$ power units, and with processing times equal to $p_1 = 10$ and $p_2 = 5$ time units. We recall that a time slot is defined as a set of successive intervals, and we assume the scenario where there is only one time slot available for the execution of T_2, let it be $TS_1 = \Delta_3 \rightarrow \Delta_4$ with a minimum power production of 16 power units through both intervals. Since this time slot is early enough in the runtime, using both LPT and LPTPN might schedule T_1 in this time slot, since these policies give higher priority to T_1 than T_2, and this time slot happens to be the first or earliest fit for T_1. This solution obviously would yield an invalid schedule, in which T_2 is not scheduled.

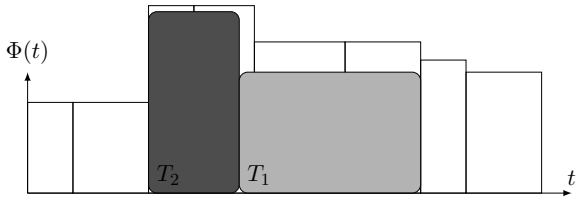


Figure 6.9 – Best fit BS

A solution is hence to place each task in its best fit time slot rather than on the first fit. We call best fit the time slot which has the closest minimum power production level to the task's power need φ_i. In the previous example, there is another time slot $TS_2 = \Delta_{30} \rightarrow \Delta_{40}$ with minimum power production of 12, but since it occurs after TS_1, it was not considered by LPT nor LPTPN for the execution of T_1. However, since 12 is closer to 10 than 16, between these two time slots, TS_2 is considered to be the best fit for T_1. In this example, using the best fit keeps TS_1 free for the execution of T_2, which makes the schedule valid.

Reducing the power waste also has potential to improve the performance in the multiple machines platform, where the saved power could be enough for activating an additional processor, and for executing an additional task.

A key problem here is how to avoid scheduling a task towards the end of the runtime, just because a time slot over there produces less power waste than many earlier ones. In other words, how to find the best fit for each task, without decreasing the quality of our solution (increasing the makespan). Binary search provides a solution that minimizes the makespan, and keeps the power waste at minimum level at the same time by scheduling each task in its best fit, while limiting the search space for the best fit to a minimum point by which the execution of tasks can be over. To find this minimum point, we propose the binary search algorithm presented in Algorithm 8, which provides the same behavior of the dual approximation algorithm proposed in [72].

BSPW, which stands for Binary Search considering the Power Waste, uses at first an initial

search limit:

$$midpoint = minpoint + maxpoint/2$$

with $minpoint = \sum(p_i)/nbCores$, a lower boundary of makespan without the power constraints in the multi-core platform, and $maxpoint$ is the end of the power envelope. BSPW then tries to schedule the task list within the subset of interval $[\Delta_1 \rightarrow midpoint]$ one task at a time, while placing each task in its best fit. If all tasks find their best fits within this range, we restart our search with a maximum limit $maxpoint = midpoint$, and a new midpoint is calculated accordingly. If the tested range is too short for the execution of all the tasks, we continue our search with a new minimum limit $minpoint = midpoint$, and a new midpoint is calculated accordingly. The search stops when the max and min limits are equal (typical binary search stopping point), and the last successful schedule before stopping is executed.

The binary search algorithm is combined with the sorting policies tested in the list algorithm, to examine the impact of different sorting priorities on the performance of this algorithm. We thus use all the variants of task ordering used with the list algorithm. As a result, the following set of algorithms is tested: BSPW-LPT, BSPW-LPTPN, BSPW-twoQs, BSPW-LPN, BSPW-R, where the order of the task list in BSPW-R is kept random.

In the multi-machine model, the variation in power production from one interval to another makes the number of processors that can be active simultaneously vary as well. This produces a decision making problem to determine how many processors are running in each interval. If the power production during an interval is not enough to run any processor, then we say that this interval is useless, and no scheduling nor on/off decisions are necessary during that interval. Thus, we only consider valid intervals with at least enough power to activate one processor and, execute one task.

Since the condition for a valid interval is the ability to run at least one processor in it, we examine the case where one processor is turned on through the entire runtime, in an aim to lower the complexity of the on/off decision problem. In addition, leaving a processor on reduces the delays and energy losses caused by its booting and shutdown. On the other hand, keeping a processor on for too long might lead to unnecessary static power waste. The processor should stay turned on just enough time to finish the execution of all tasks, the issue here is thus how to determine in advance how long that period is before executing the tasks.

The next two methods are used to limit the range in which a processor is constantly active.

6.3.2 One Processor On Binary search (OPOBS)

Binary search can be used to find a location of a list element in less time complex manner than linear search. For our scheduling problem, binary search can be used to find the point of the time horizon at which all tasks can finish their execution, this information can be exploited to apply the schema mentioned above, which consists of keeping at least one processor active during task execution. Binary search thus provides an acceptably time costly way to determine the point at which this processor is switched off, which corresponds to the minimum range of the time horizon under which all tasks can be executed.

We set a minimum point $minpoint = \Delta_0$ and a maximum end of the search range $maxpoint = \Delta_X$ where X is the total number of time intervals. We try to fit all tasks between Δ_0 and $midpoint = (minpoint + maxpoint) \div 2$, while keeping one processor running during this segment of the time horizon. If the execution of all tasks is successful, we retry after setting the maximum limit of the

Algorithm 8: Binary Search Power Waste algorithm

Data:
Task list: $\mathcal{T}$
minpoint: init to $(\sum_{i=0}^{N} p_i)/nbCores$
maxpoint: init to $\mathcal{H}$
schedule: a list of tasks with their time allocation, init to $\emptyset$
Result: A schedule
Order $\mathcal{T}$ in the chosen order
while $maxpoint > minpoint$ **do**
  $midpoint \leftarrow \frac{(maxpoint\text{-}minpoint)}{2}$
  $placed \leftarrow true$
  repeat
    Take the next task T_i in $\mathcal{T}$
    repeat
      $\Delta_x \leftarrow$ next(Intervals from 1 to *midpoint*, ordered by power waste)
      $placed \leftarrow PlaceTaskCore/PlaceTaskMachine(p_i, \varphi_i, x)$ //{depending on the platform}
      if *placed* **then**
        $schedule \leftarrow schedule \cup \{T_i, result\}$
    until $placed \vee endofintervallist$
    if *placed* **then** $finished \leftarrow true$ **else** $finished \leftarrow false$
  until $finished \vee endoftasklist$
  if *placed* **then**
    $maxpoint \leftarrow midpoint$
  else
    $minpoint \leftarrow midpoint$
return *schedule*

range $maxpoint = midpoint$ and recalculating $midpoint$. If the range is too short for task execution then we set $minpoint = midpoint$ and recalculate $midpoint$. Remember that during each tested search range, the first processor is kept turned on, which on the one hand limits the switch on/off operations of that processor, reducing, therefore, the time and energy losses they cause, and on the other hand it reduces the on/off decision complexity. Other processors are powered on on demand when scheduling the remaining tasks.

Figures 6.10, 6.11 and 6.12 illustrate an example of this binary search method. In this example, the minimum time range where all tasks can fit is found in only 3 steps using binary search, while it would have been more time complex to try to find it in a linear manner starting from the first interval and increasing the search area by one interval at a time.

To evaluate the effect of task list's order on this approach, the same sorting policies used in the list algorithm are applied here for sorting the task list. Therefore, **BSLPT**, **BSLPN**, **BSLPTPN**, **BSLPP** and **BSTwoQs** apply **LPT**, **LPN**, **LPTPN**, **LPP** and **twoQs** respectively. The ordered task list and the selected minimum search range of the power production time horizon are then passed to $PlaceTask$.

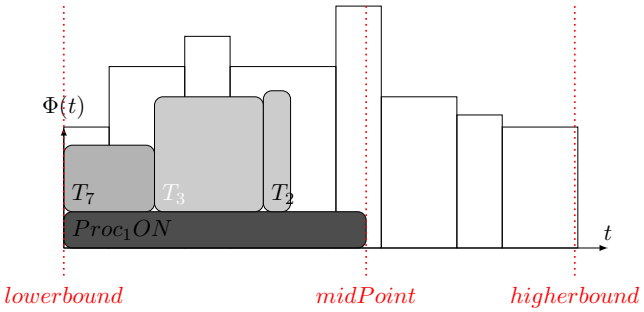


Figure 6.10 – OPOBS 1st step

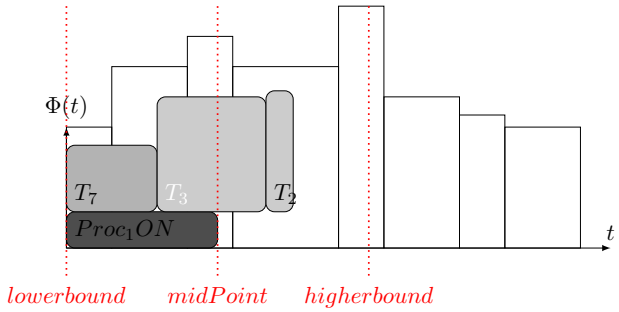


Figure 6.11 – OPOBS 2nd step

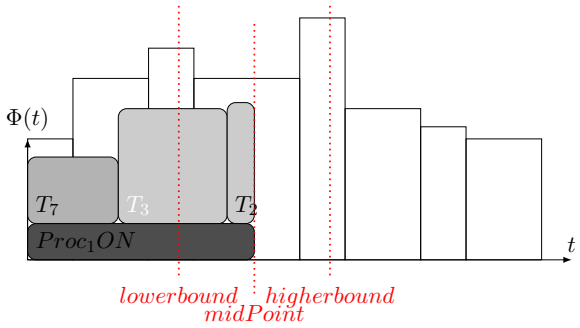


Figure 6.12 – OPOBS 3rd step

6.3.3 Stripe

The second technique used to limit the search range is illustrated in Figure 6.13. The idea is to start with a minimum space, and gradually increase the range until the execution of all tasks can fit within it. We start by scheduling the task with the longest processing time T_i. This guarantees that the rest of the tasks can fit (on the time axis) in this range (stripe) as well. The $PlaceTask$ function places this task in the earliest suitable interval, let it be Δ_{start}, so that the task execution finishes at $C_i = \Delta_{start} + p_i$. Then we try to fill the power envelope of the stripe T_i ($\Delta_{start} \rightarrow C_i = TH_1$) with as much tasks as possible. When no more task can fit before TH_1, the longest of the remaining tasks T_k is then scheduled at the earliest time slot starting from interval Δ_{TH_1+1}, creating the second search limit $TH_2 = C_k$. We again schedule as much tasks as possible in this stripe until no more task can fit in, and so on until all the tasks are scheduled. As usual the processors are only kept on during the time needed to execute the tasks they are running.

To fit as many tasks as possible within each stripe different sorting policies are applied on the task list. **StripeLPT** orders the task list according to the longest processing time first, it means that while filling each search range, we start by the longest tasks first. **StripeLPTPN**, **StripeTwoQs** and **StripeLPP** sort the task list similarly to **LPTPN**, **twoQs** and **LPP** respectively during the search range filing step, while the creation of each new time horizon TH_x remains using the longest of the remaining tasks. **Stripe** takes advantage of more than one priority criteria, it scans the time horizon twice while trying to fill the space above the longest task, once while using **LPTPN** and once using **twoQs**.

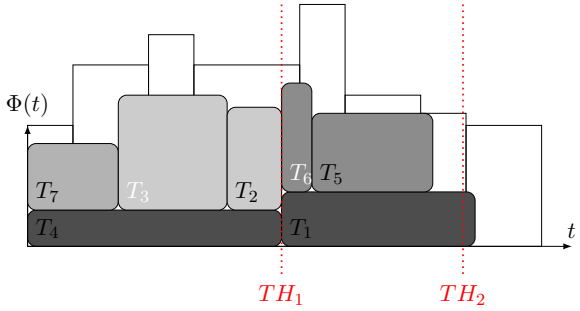


Figure 6.13 – Stripe heuristic

Part III

Experiments

In this part of the work, we perform an experimental study of the proposed heuristics. We develop a simulator script that offers a wide range of experimental setups on two computational platform models, namely the multi-core platform in Chapter 7 and the multi-machine platform in Chapter 8. Computations have been performed on the supercomputer facilities of the Mésocentre de calcul de Franche-Comté – Besançon. The presented results took over 580,000 hours of computation time over this supercomputer.

Chapter 7

Multi-core platform experiment

To evaluate and compare the proposed solutions mentioned in the previous chapter, we carry out two sets of experimental studies, the first study deals with the multi-core computational platform described in Section 4.2.1, and the second study deals with the multi-machine computational platform described in Section 4.2.2. In this chapter we present three experiment sets based on the multi-core platform. The presented experiments were realized using simulation rather than a real platform as running lots of real life experiments is costly and hence does not allow to explore a wide range of parameters. The choice to use a multi-core platform is to provide an initial evaluation of the proposed solutions on a less complex model than a more general HPC system representation.

7.1 Data generation

To carry out our simulations we must produce the data that represents the two main elements of the scheduling problem, namely the task list and the power envelope. Several methods are used to generate this data, some of which are synthetic and generated randomly based on different statistical laws, and some of which are based on realistic models.

7.1.1 Task models

We recall that we deal with sets of independent sequential tasks, each task T_i is characterized by two main values, its processing time p_i and its power consumption φ_i.

Hyper-gamma model

Lublin and Feitelson [92] model the distribution of HPC tasks based on real workloads and date logs using a hyper-gamma distribution with ($\alpha_1 = 4.2,\ \beta_1 = 0.94$) and ($\alpha_2 = 312,\ \beta_2 = 0.03$). We use this distribution law to generate p_i values, this model however does not include the task power consumption values, thus, we use random generation of φ_i with a uniform distribution law between $\varphi_{\min}$ and $\varphi_{\max}$ power units .

Exponential model

Lublin and Feitelson's model, although realistic, is based on workload logs collected from only three sites. This limits the experiments to the range of values collected for that sample of data. Therefore,

to explore a wider set of parameters, we use synthetic workloads with random generation for both p_i and φ_i. We use an exponential distribution law to generate p_i values between $p_{\min}$ and $p_{\max}$ time units, where most generated samples will fall around the mean value $p_{max}/2$ and fewer values fall towards both limits of the range. This means that tasks that are too hard or too easy to schedule have less probability. As for the tasks' φ_i generation, a uniform distribution is used.

Task from interval model

Since it is not possible to find the optimal solution of a 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 of generating a set of tasks starting from the time intervals set. Given a set of time intervals representing a power envelope, we generate a set of tasks that will totally fill the intervals. For that we randomly chose 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 deducted 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 and we generate an optimal solution with a makespan of Δ_{max}. We also take more or less intervals to generate between 90 and 110 tasks per list. Since the tested heuristics are expected to find a makespan that is longer than this optimal point, we repeat the power envelope two times in order to make sure that the calculated C_{max} would fit within the time horizon. The ratio between the C_{max} found by a heuristics and the optimal point is the distance from the optimal for this heuristics, $dis_{OPT} = C_{max} \div \Delta_{max}$.

7.1.2 Power interval model

We recall that the power envelope is discretized into time intervals Δ_x of length δ_x and each interval has an available green power value Φ_x.

Bell shape

To explore solar panel like power generation, we generate sets of intervals with a bell shape, as shown on Figure 7.1. We define five power levels and we randomly generate the available power for each interval inside the level. To generate the bell shape we give a higher probability to increase (resp. decrease) the level when we are in an increasing (resp. decreasing) phase. The used random law in the levels is uniform.

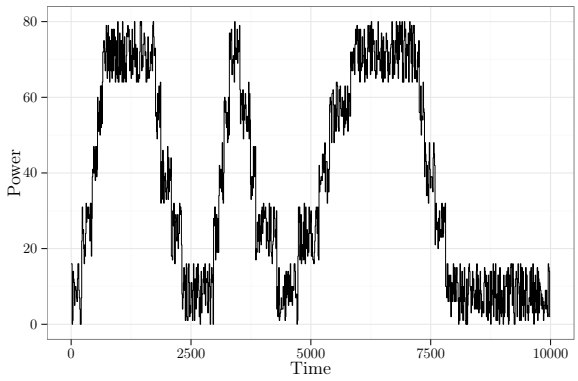


Figure 7.1 – A bell shape interval list generation.

Realistic models

A realistic model is used to generate the power envelope that represents the various renewable power supply based on real collected historical data. This model [68, 69] 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. We partition the power envelope into unified intervals Δ_x with length of δ, this interval length corresponds well with the fluctuation frequency seen in renewable power supplies. The available power Φ_x values are generated based on the realistic model. Figure 7.2 illustrates an example of a power envelope generated used the realistic model that considers a mixture of solar and wind sources with an average Φ_{max} of 350.

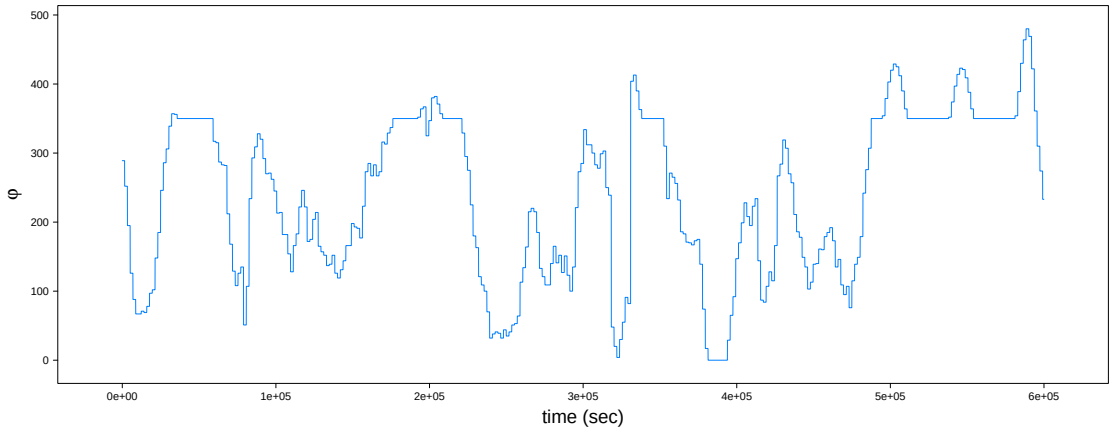


Figure 7.2 – Realistic power envelope $\Phi_{max} = 350$.

Interval from tasks

The second method that we use to measure the distance of the schedules generated by the proposed heuristics from the optimal solution is called Intervals From Tasks (IFT), and it is achieved by generating the power envelope from a set of tasks. In this method we take each task from a task

set and randomly place it within the time horizon $\mathcal{H}$. At that point, we create the corresponding time interval that lasts as long as p_i and with a level of available power that equals to φ_i. Since we consider a sixteen cores platform, up to sixteen tasks can overlap over one or more time intervals, and the power level in such intervals is the sum of all the concerned overlapping 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 Δmax. For this data generation method, we use semi-synthetic workloads, with p_i values based on Feitelson's traces, and randomly generated φ_i.

7.2 Evaluation metrics

To compare the results of simulations over different data sets with different characteristics we need normalized metrics. Raw makespan or flowtime values cannot be compared as they depend on the considered set of tasks and intervals. A set of long tasks always gives a longer makespan than a set of shorter ones. Therefore we use the following metrics to compare the schedules produced by different heuristics.

We define a normalized metric for makespan performance $PERMAK$ as

$$PERMAK = (makespan - useless)/\sum p_i$$

where $makespan$ is obtained by the schedule and $useless$ is the sum of the length of the intervals, between 0 and the end of the schedule, where no task can be scheduled because of too low available power.

In a same way we define $PERFLOW$ for flowtime performance evaluation 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 of too low available power.

Even if we applied a normalization on the makespan and flowtime results, we noticed, after a first set of experiments, that both $PERMAK$ and $PERFLOW$ are dependent on the characterizations of the power envelope. Therefore, for our last results, we propose two other metrics that provide a more fair comparison regardless of the power envelope.

We first propose an evaluation metric that compares the number of times an algorithm achieves the best $C_{\max}$. This comparison is done over one experiment at a time and is thus fair since the settings are the same. In a same manner we can compute the second and third best makespan. This metric does however 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 heuristics' makespan and the best computed makespan for that data set and normalizes their difference to the power envelop size.

$$NM = \sum out \div \sum total$$

Where $\sum out$ is the sum of the $p_i \times \varphi_i$ area of all the tasks that are scheduled after the best $C_{\max}$ found during that simulation, and $\sum total$ is the total $p_i \times \varphi_i$ area of the task list. 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 consumption as it uses the $p_i \times \varphi_i$ surface. 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 executions. The lower this average value, the better is the solution.

We point out that no similar normalized metric is proposed for the flowtime objective. The reason is that the calculation of NM depends on $C_{\max}$. The minimum $C_{\max}$ found among all the heuristics during one run is used as a reference point on the computational time horizon after which the $\sum out$ value in the NM formula is calculated. Since the flowtime is calculated based on the completion time C_i of each task, we do not have one reference point to use as a starting point for $\sum out$ calculation.

7.3 Simulator

In order to evaluate the performance of our solutions we create a scheduler script in Python [1] that allows us to compare the makespan and flowtime resulting from deploying each heuristic on a tested instance of the problem.

The simulator is designed in a way that allows adding a new heuristic or a new data generation method without the need to modify the whole code. The simulator's code is distributed among several integrated folders, methods for power envelope generation for example are stored in one folder, while another folder contains the task list generation methods, and each heuristic is written in a separate file in the *Algos* folder. Any addition or modification in one of these folders can be easily integrated in the rest of the simulator. A main file uses a prespecified generation method for task and interval list generation, and applies the selected heuristic to sort the generated task lists, which represents the planner part of the simulator. It then calls the *PlaceTask* function to compute the schedule, which represents the executer part of the simulator. Figure 7.3 outlines the simulator's components.

Note that both the planner and the executer parts of the scheduler are in the python block in Figure 7.3. Finally, the results are presented in different diagrams using R, in order to provide a clear comparison between different performances of the algorithms. The resulting schedule of each single execution can optionally be produced using Python's Turtle graphical tool to have a detailed view of the schedule, an example of one schedule presentation with turtle is given in Figure 7.4.

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

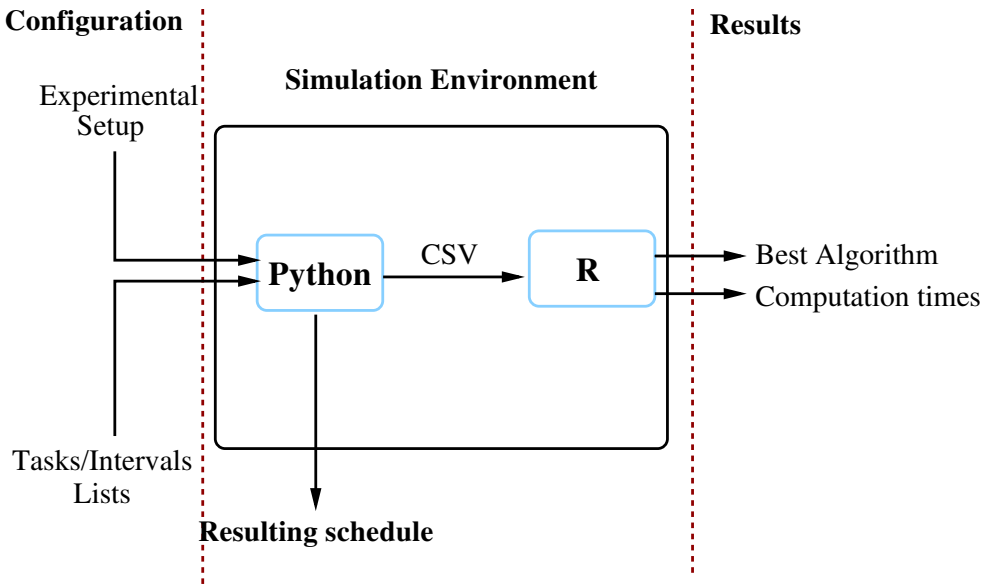


Figure 7.3 – Simulator

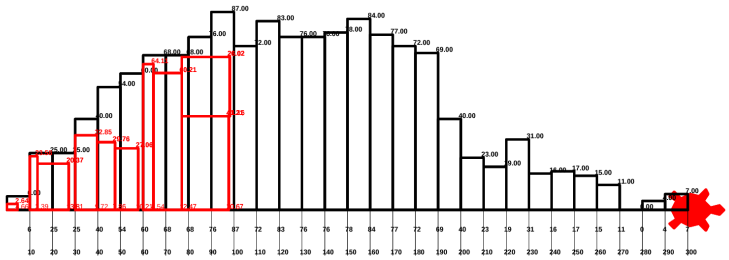


Figure 7.4 – Turtle example

The tasks are represented by a set of independent objects with two attributes each, the task's processing time p_i and its power consumption φ_i.

The power envelope is represented by a set of sequential linked rectangles, the horizontal side of a rectangle represents an interval's duration δ_x while the vertical side represents its green power availability Φ_x.

For the multi-core platform experiments, we assume that the computation unit is a processor's core and that all available cores belong to one processor that is constantly switched on, and that all the energy needed to power on the computation units and to support the IT infrastructure is thus previously deducted, in other words, the entire available power in each interval is dedicated for tasks execution. We carry out three experiments on this platform model, in the first experiment several adaptations of the list algorithm are designed to take the variant green power production into consideration and compared to a power waste aware binary search technique that we developed. In the second experiment we assess the use of a genetic algorithm for our problem, we test and compare several configurations of the genetic algorithm. In the third experiment we conduct a comprehensive

comparison between all the adaptations of the list algorithm evaluated in the first experiment and all the configurations of the genetic algorithm evaluated in the second experiment, we introduce in this experiment new methods for data generation including a realistic power envelope model and two methods for measuring the distance of a solution from optimal.

7.4 List scheduling versus binary search experiment

In this experiment two classes of algorithms that differ in complexity are tested. The first class of algorithms is list scheduling algorithm, see 6.2.1. Within the list class different priorities are used to sort the task list. The aim of this experiment is to answer the following question: Which task attribute is the most important to use as priority value in such a task model? In other words, which has a higher impact on the schedule, the processing time p_i or the power consumption φ_i of tasks?

The second class of algorithms belongs to the Binary Search class described in 6.3.1. Binary search allows to find the minimum range of the power envelope under which all tasks can fit in less time complexity than linear search. Once we find that range we can schedule each task in its best fit, which is the time slot that has the closest available power level to the task's power consumption. Limiting the search range using binary search reduces the risk of excessively delaying the task execution due to a high energy gain provided by using the best fit in a late time interval.

7.4.1 Experimental settings

The tasks p_i values are randomly generated using the suggested hyper-gamma law. We also use random values following an exponential law for p_i in parts of the experiments to simplify them. When we use the exponential law for task generation, we define an upper bound p_{max} and we set the mean value of this law to half of the p_{max}. In the experiments p_{max} ranges from 10 to 100 time units, by steps of 10.

The power consumption of the tasks is given in power units. As we were lacking values for the power consumption of the tasks, we choose to use a random generation of φ_i with a uniform law between 0.1 and φ_{max}. φ_{max} ranges between 4 and 40 power units, by steps of 4.

For the experiments where the exponential law based task generation is used, we generate 250 intervals per set and we generate 600 intervals per set for the experiments that use the hyper-gamma law based task generation. Note that there is no guarantee when we use a set of intervals that a schedule can be found. For that reason we use large numbers of intervals.

The results of this experiment are calculated as the average of 100 simulations, therefore we generate 100 different sets of intervals that are all subject to the same parameters and for each couple of p_{max} and φ_{max} values, we generate 100 different sets of tasks that are all bound by these two values. 10 000 experiments where thus performed, 100 for each (p_{max}, φ_{max}) couple, each with a different task set and the same interval set is used for each (p_{max}, φ_{max}) couple.

The number of cores is set to 8, which means that up to 8 tasks can be scheduled in the same interval. To assess the impact of the available power on the algorithms performance we use two values for the maximum available power, 40 and 80 power units. As the φ_{max} value ranges from 4 to 40, this means that the tasks may require up to 320 power units to run without constraint in the case where $\varphi_{max} = 40$.

For the time interval generation, we choose intervals of equal length of 10 time units. To explore solar panel like power generation, we generate sets of intervals with a bell shape, as shown on

Figure 7.1, the maximum power that can be provided by the sources is 80 power units and each level has a height of 16 power units.

We point out that in order to produce statistically concrete results, we run 100 simulation in the first experiment and present the results as the average of 100 simulation runs. In the second experiment we increase this number to 200 runs to have higher statistical accuracy. Finally, for the third experiment, we compare the averages of 50, 100, 150 and 200 simulations for several test cases, and we observe that the results started to show concurrency and stability after 150 runs, for this reason, the results of the third experiment are the average of 150 runs.

7.4.2 Results

In this section we present the results of the first multi-core experiments.

We assess all the algorithms regarding both the makespan performance *PERMAK* and the flowtime performance *PERFLOW*. Figure 7.5 presents the best algorithm for each value of p_i and φ_i. The best algorithm is defined as the algorithm that has the best mean *PERMAK* on the 100 simulation runs for a couple of values (p_{max}, φ_{max}). As we can see on Figure 7.5, the best algorithm depends on both the values p_i and φ_i. Unsurprisingly when the power consumption is low, the power is not a constraint and the LPT algorithm that fosters long jobs, gives the best results. We are, in this case, close to the classical $P||C_{\max}$ problem, which is efficiently solved using LPT. However when φ_i is higher, algorithms that take power consumption into consideration achieve better results. Moreover, many algorithms get the best results depending on the power consumption. When the processing time is small, the LPN algorithm is the best but when it increases, LPTPN, which takes both processing time and power consumption into account, is better. For the case of a medium power consumption and small processing times, the BSPW family of algorithms finds good schedules.

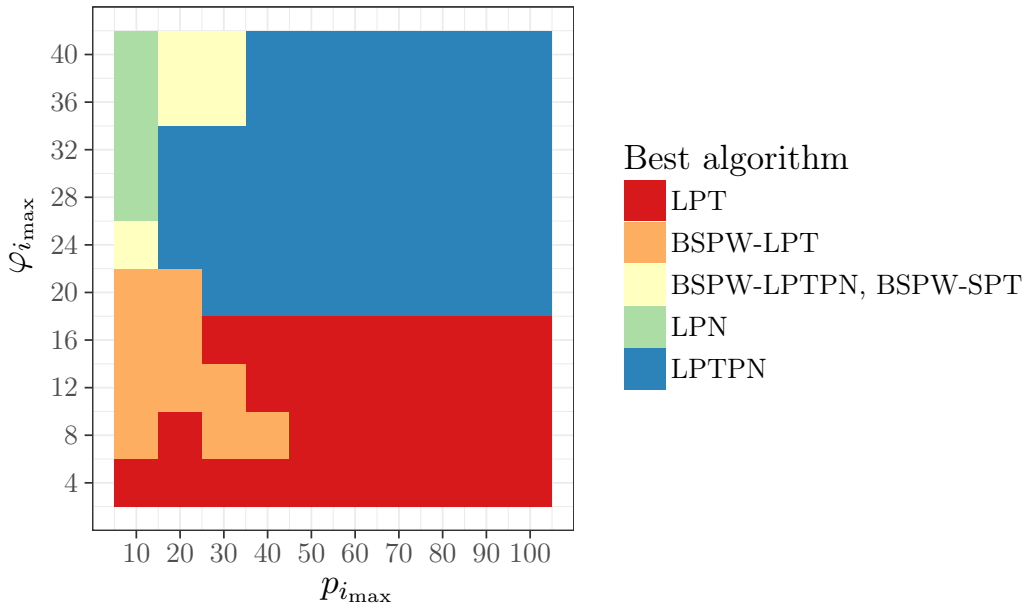
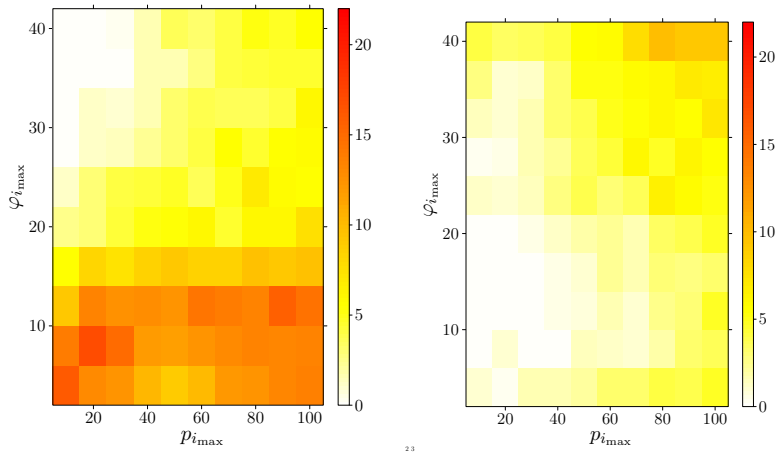


Figure 7.5 – Heat map of the best average *PERMAK*, $p_{max} \in [10 \to 100]$, $\varphi_{max} \in [4 \to 40]$.

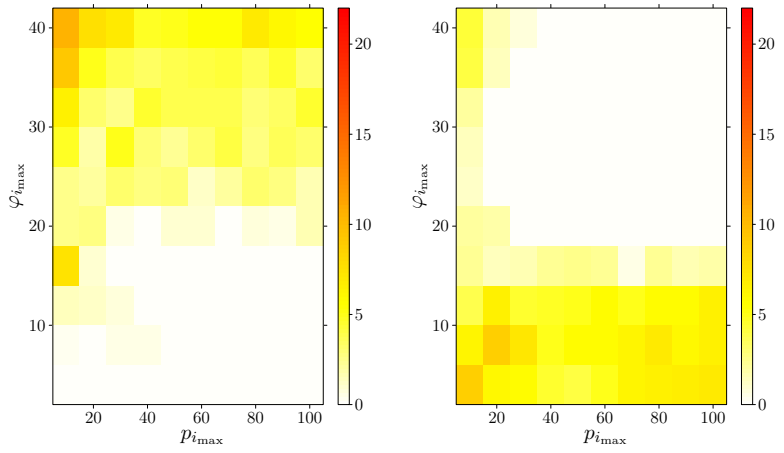
We also compare the makespan performance of our heuristics to each other, more precisely we

compute the makespan distance of each algorithm to the best solution found by one of them. BSPW-LPN, BSPW-LPT, LPT and LPTPN give more often the best performance and we present their distances in Figure 7.6. From Figures 7.6b and 7.6d we can see that the LPTPN and the BSPW-LPT algorithms generate schedules with makespan never farther than 6 % from the best one. This makes them good candidates for a global solution. Between them, the LPTPN algorithm gives more often the best makespan. Unsurprisingly, the LPT algorithm which gives the best makespan when the power consumption is low, generates its worst schedules when the power consumption is high and the processing time of tasks are small. It is however never worse than 11 %. On the other hand Figure 7.6a shows that the LPN algorithm gives its best solutions for short tasks with high power consumption.



(a) Makespan distance of LPN

(b) Makespan distance of BSPW-LPT



(c) Makespan distance of LPT

(d) Makespan distance of LPTPN

Figure 7.6 – Distance from the best *PERMAK*.

Figure 7.7 gives the standard deviation of the makespan performance *PERMAK* for ($p_{max} = 100$, $\varphi_{max} = 20$). As can be seen, the variation is low, ranging between 0.3 and 0.38, only few

measurements are noticed out of this range and they do not reach higher than 0.8.

We experimentally assessed the performance of our algorithms regarding flowtime *PERFLOW*. We tested all the algorithms for p_{max} ranging from 10 to 100 and φ_{max} ranging from 4 to 40. SPT has always produced the lowest flowtime, which is not surprising since SPT gives the optimal flowtime solution in classic scheduling problems.

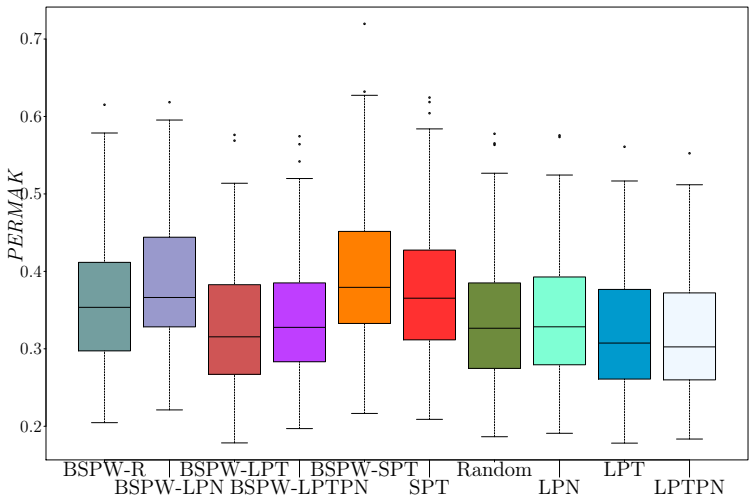
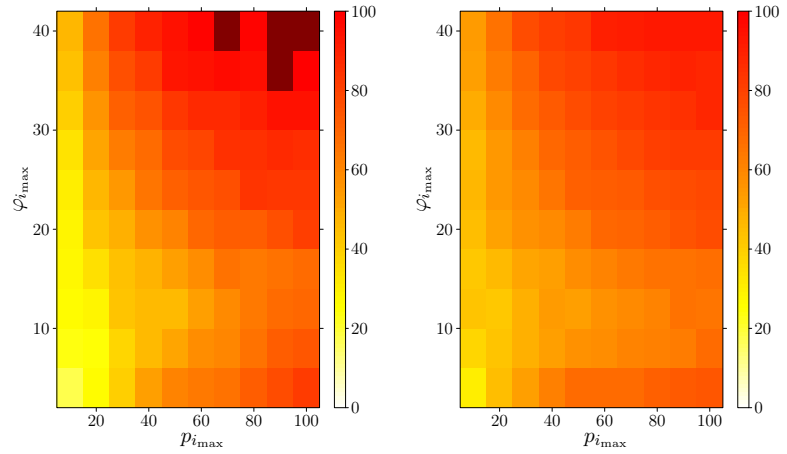
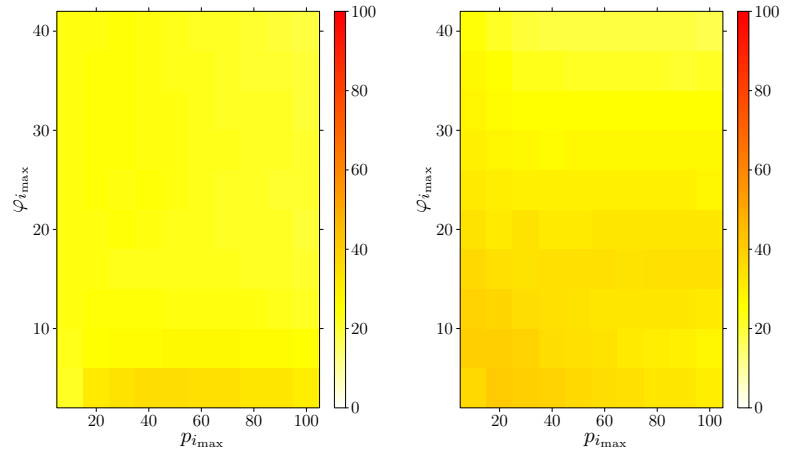


Figure 7.7 – Standard deviation of *PERMAK* for $p_{max} = 100$ and $\varphi_{max} = 20$.

Similarly to the makespan performance assessment, we analyze the distance between the flowtime performance *PERFLOW* of each algorithm from the best performance. The distances for BSPW-SPT, BSPW-R and Random, LPN are presented on Figure 7.8. From this figure we can note that a list algorithm with random tasks order has better probability to produce a good flowtime than BSPW algorithms or list algorithms with power consumption based task list order. These poor performance shows that the tasks ordering criterion largely impacts the flowtime. We can also note that the BSPW based algorithms have poorer performance. This is because the BSPW algorithms first searches for the shortest time horizon in which all the tasks can be scheduled, thus favoring the makespan, before taking the tasks order into account. These results highlight that the tasks order should be considered first to lower the flowtime.



(a) Flowtime distance of BSPW-SPT (b) Flowtime distance of BSPW-R



(c) Flowtime distance of Random (d) Flowtime distance of LPN

Figure 7.8 – Distance from best PERFLOW.

Figure 7.9 gives the standard deviation for the flowtime for all the algorithms used in the experiments. The SPT algorithm gives the lowest deviation, lower than 10 %. As for the *PERFLOW* performance measure, the Random algorithm is ranked second with less than 12%. Globally the list based algorithms give lower variations than Binary Search algorithms. It means that the proposed algorithms are inefficient in this case.

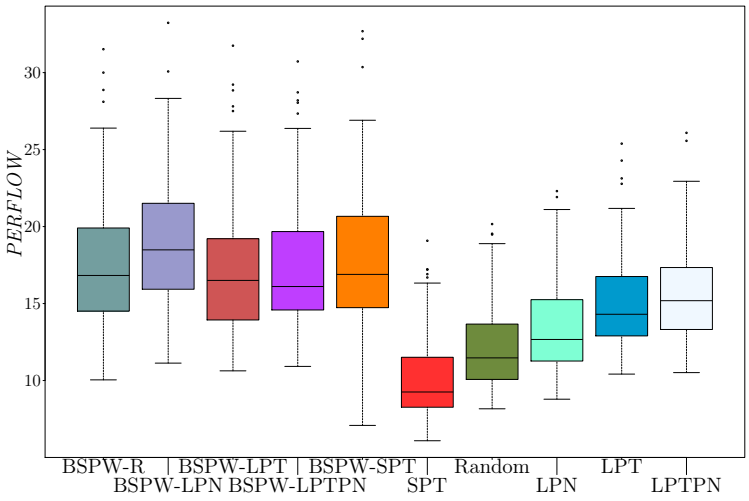


Figure 7.9 – standard deviation of $PERFLOW$ for $p_{max} = 100$ and $\varphi_{max} = 20$.

Figure 7.10 gives the mean computation time for the 100 runs done with one given value of the power consumption φ_{max} (20). Note that the LPTPN algorithm is barely visible on the plot as it gets the same running times as SPT and the curves overlap. If we except the BSPW-R algorithm, the power consumption value only slightly impacts the computation time that slightly decreases when the φ_i value increases. On the contrary when p_i increases, the computation time increases too. It is also clear that the BSPW family of algorithms has a larger computation time than the list based family. Indeed as they iterate on the horizon value and, at each iteration, they apply the same type of algorithm as the list based algorithms. From Figure 7.10, we can see that BSPW-R generates the longest execution times, 10 times more than the fastest algorithm, i.e., LPT.

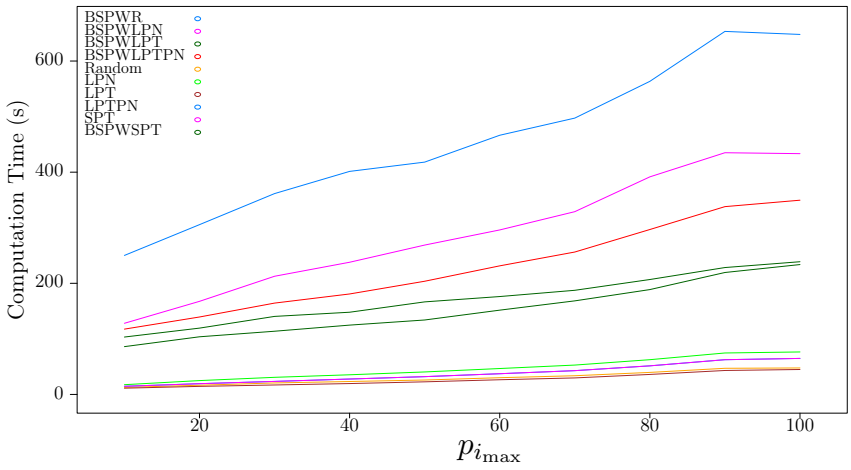


Figure 7.10 – Computation time of the heuristics for $\varphi_{max} = 20$.

7.4.3 Summary

Our experiment show that using a list algorithm that deploys the right priority criteria provides fast and good results, and that determining which list scheduling algorithm is the best depends on the characteristics of the task list and on the optimization objective. LPTPN generates the best performance for the makespan objective on many task list characteristics values and it generates schedules never worse than 5 % of the best ones while keeping reasonable computation times. The tested list algorithms are fast enough to be valid scheduling strategies in HPC systems even when performing on-line scheduling. The tested binary search scheduling algorithms are however more complex than the list algorithms, therefore we do not recommend using them in real life systems that require fast computation of the schedule.

7.5 Assessing the use of a genetic algorithm experiment

The second multi-core experiment is conducted over a set of evolutionary algorithms to assess if they can be a valid solution to our problem. The results show that those algorithms outperform the less time complex list algorithms, but at a much higher time coast. A trade-off between the schedule computation time and the performance is presented in the results section.

7.5.1 Experimental settings

The experimental settings are the following: The number of available cores is set to 8, which means that we cannot run more than 8 tasks in parallel even if there is still unused power. The performance of the algorithms is computed with different values of maximum task length that ranges from 10 to 100 with a step of 10 time units, and maximum task power consumption that ranges from 4 to 40 with a step of 4 power units. For each couple of (p_{max}, φ_{max}) values we run 200 simulations with different sets of 100 tasks and sets of 1000 intervals where $\Phi_{max} = 80$. For the tasks, the p_i and φ_i values are randomly chosen with, respectively, an exponential law and an uniform law. The performance of the algorithms depends on their obtained makespans. An algorithm may however not always get the best result, depending on the experimental parameters. Algorithms are hence rather compared based on their mean makespan. However a simulation with larger task is hardly comparable with another using small tasks. For these reasons we measure the algorithm's performance with their *PERMAK*, where $PERMAK = (makespan - useless)/\sum p_i$, which normalizes the raw makespan value, where *useless* is the sum of intervals with $\Phi_x < \min_i(\varphi_i)$, the available power is less than the minimum task power.

We evaluate the GAs with different types of crossovers, without cross over, and evaluate the effect of wheel versus random selection. For each GA the stopping condition (*nbI* in Algorithm 4, number of generations without any improvement) is set to 50. Note that other computations have shown that a value of 10 gives faster results but with lower quality. 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 metrics to use for comparison without the need to use a normalized metric.

7.5.2 Results

Figure 7.11 shows the best algorithm, the one with the lowest mean *PERMAK*, with each $p_i max$ and $\phi_i max$ values presented as a square on the heat map. From the figure we can say that in general, wheel selection gives better results than random selection. This probably means that better solutions are rather found by slightly modifying initially good solutions than searching at a wider distance from these initial solutions. The only cases where random selection gives better results is when the power demand of tasks is high, 36 or higher. From this figure we can also notice that the 1-point crossover does not show any best result in the heat map. This is because changing a big chunk (90%) of a solution, has almost the same probability of producing a good solution as generating a completely random solution, even if the parent is a good solution. In the same way we can state that limiting the size of the chunk to be changed in the crossover leads to better solutions.

Figure 7.11 also shows that the characteristics of the tasks play a rule in determining which GA configuration is the best. Since the heat map is relatively split diagonally between two GA configurations, we can conclude that GA with no crossover performs better for small tasks (the rectangle area $= p_i \times \phi_i$ is small), while it is better to use OX for big tasks.

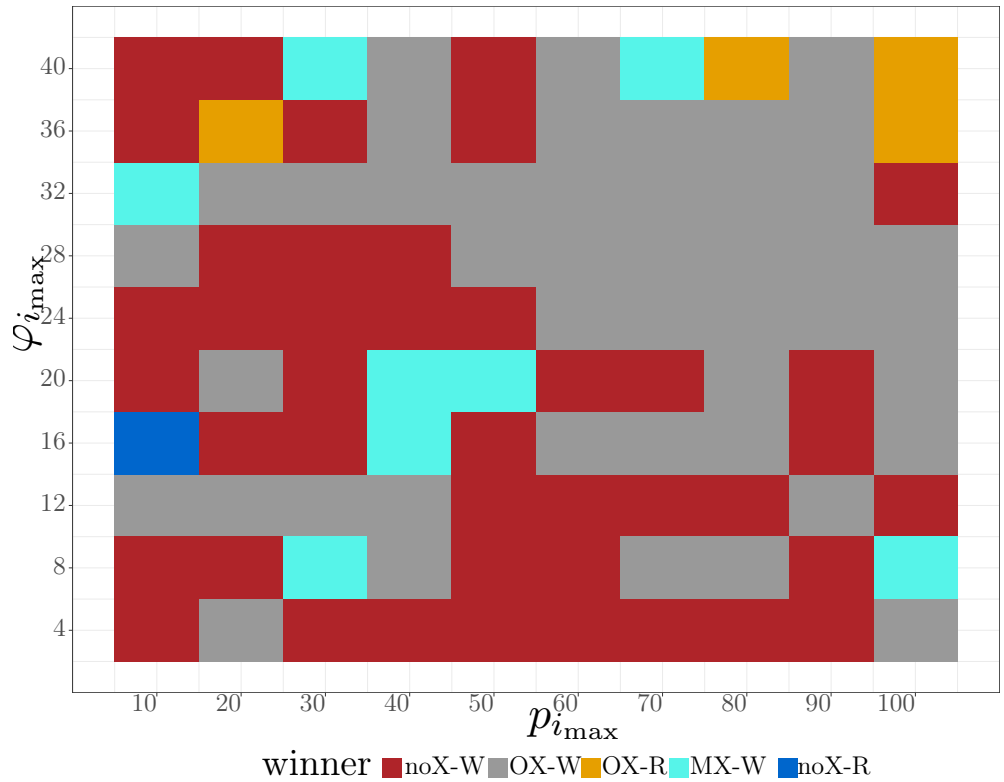
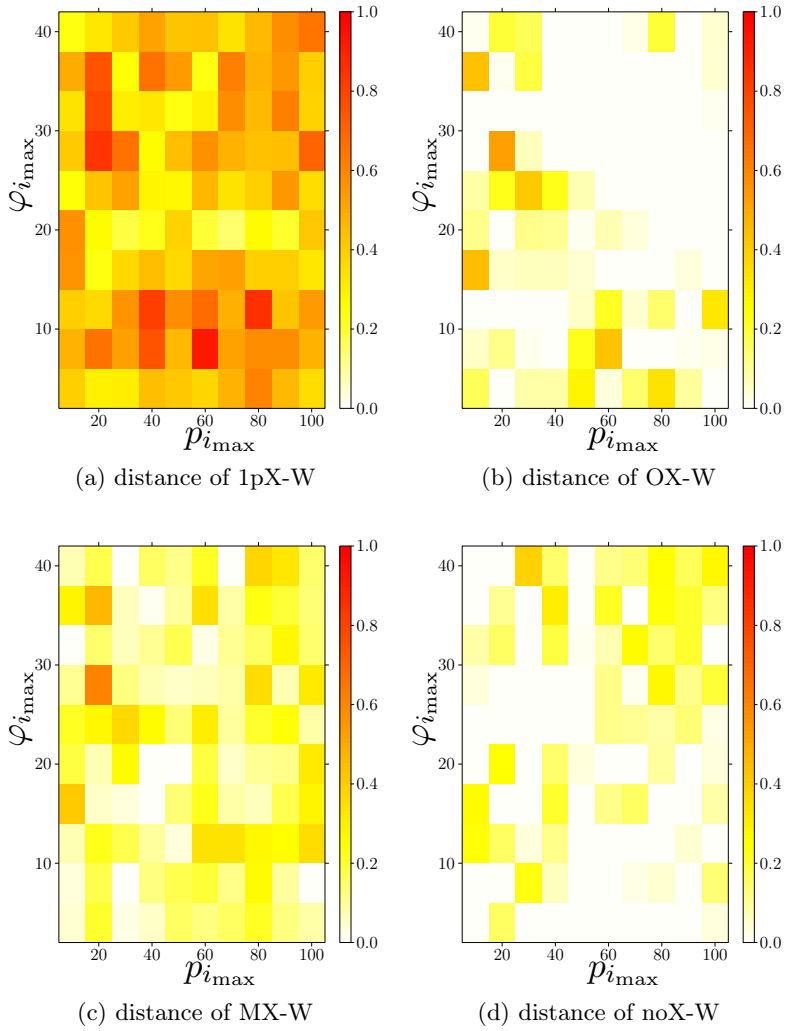


Figure 7.11 – Best heuristic regarding *PERMAK*

Figure 7.12 shows the distance in percent of the performance of the four algorithms using wheel selection to the best algorithm for each square in the heat map. We can see that the algorithms are never more than 2% worse than the best solution and that, except for the 1-point crossover, the algorithms are generally less than 0.5% worse than the best one, even for the noX-W algorithm.

The same analysis on random selection shows that it does not provide as good results as the wheel one. Note that, even in the cases where random selection provides the best result, OX-W is not farther than 0.3%. We conclude that OX-W may be used in almost all cases.



(a) distance of 1pX-W

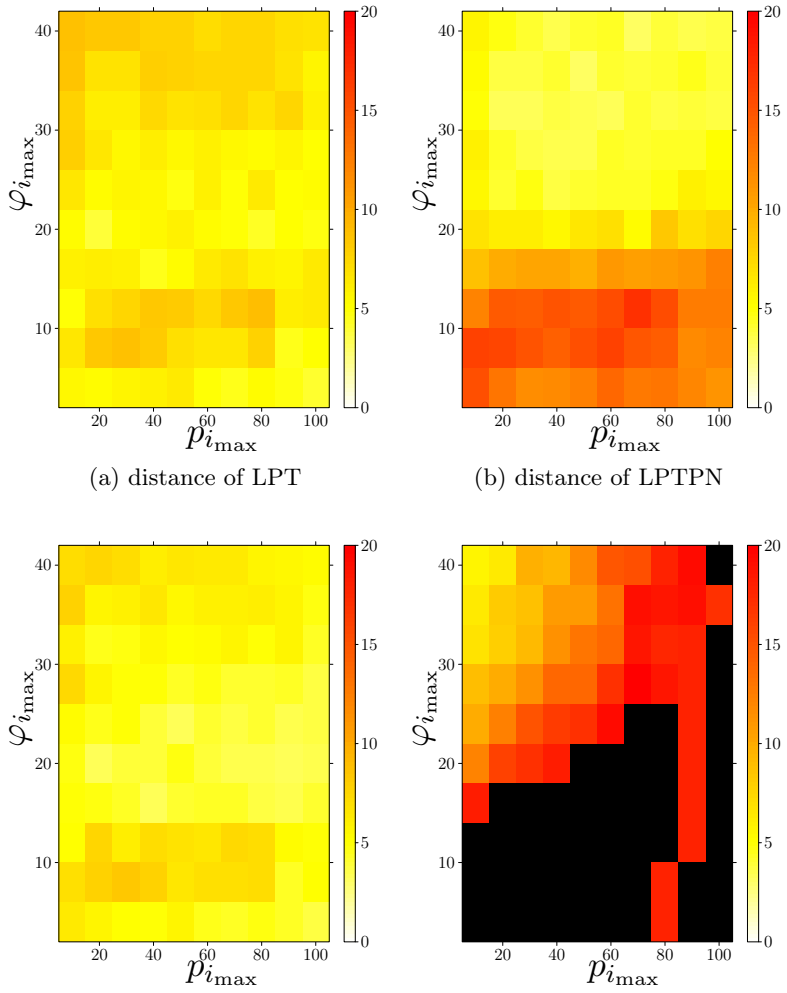
(b) distance of OX-W

(c) distance of MX-W

(d) distance of noX-W

Figure 7.12 – Distance of 1pX-W, OX-W, MX-W, and noX-W from the best algorithm

Figure 7.13 presents the distance of the list algorithms used as initial population of the different GAs from the best average *PERMAK* value. The figure shows how far the GAs are able to improve these initial schedules. Preceding results showed that TwoQs and LPT were the best solutions of the list algorithms which is confirmed here as we observe less red squares on their heat maps. Note that the black squares on the LPN heat map mean that the distance exceed the upper value of 20 %, reaching up to 40 % in some cases. Generally the best improvement of the GAs from those two solutions is around 5% and never better than 10%. This means that all GAs improve the good initial schedules but the improvement is not that significant.



(a) distance of LPT

(b) distance of LPTPN

(c) distance of TwoQs

(d) distance of LPN

Figure 7.13 – Distance of list algorithms from the best algorithm

Table 7.1 – Computation time

Algo	noX-W	OX-W	MX-W	1pX-W	LPT
Time (s)	239.34	301.83	300.11	128.6725	0.02
Algo	noX-R	OX-R	MX-R	1pX-R	TwoQs
Time (s)	287.28	356.08	339.58	334.93	0.04

Table 7.1 answers the question of the cost of the improvement as it gives the mean computation times for each algorithm. Note that the computation times stay almost constant whatever the value of p_i and φ_i. There is a huge difference in the computation times between list algorithms and GAs. If nbI, the number of iterations without improvement is set only to 10, the mean computation time falls to around 25%. The results also show that using random selection leads to bigger computation times, especially for 1-point crossover, where big changes are applied on the chromosomes, and

randomly selecting bad individuals for crossover would delay the convergence comparing to selecting good individuals by the wheel selection, and improving them through next generations. Given these results we can see that the list algorithms provide pretty good results for the time used and the improvements are costly. So the question is how much time can we wait for a schedule? Note that the algorithms are implemented in Python and can run faster in a compiled language.

Further analysis of the results also shows that all tested GAs have a relatively stable standard PERMAK deviation to each other, around 0.31, with ranging between a minimum around 0.18 and maximum around 0.45. This means that the PERMAK measure is stable for the GAs. On the other hand list based algorithms have bigger variations and higher standard deviation up to around 0.36.

7.5.3 Summary

In this experiment we evaluated the interest of using a genetic algorithm (GA) to find a solution to the optimization problem of scheduling a set of independent tasks on a parallel platform powered solely by renewable energy sources. We examine the effect of applying different types of crossover operators on the performance of GA. We compare the performance of these four GA configurations to previously tested list algorithms. We also investigate the impact of applying the wheel selection in comparison with random selection. Extensive simulations show that implementing any tested GA configuration outperforms all tested list algorithms, by improving known good solutions through small size genetic modifications. Even though the superiority in the performance of GA is not proportional to the time loss compared to list scheduling algorithms, the computation time of GA is still within acceptably limits.

7.6 Realistic power envelope model experiment

In the third multi-core experiment we evaluate and compare both the list scheduling based heuristics with the different configurations of the genetic algorithm. We introduce new data generation methods, some of which are based on a realistic model, and some of which are designed to provide the distance from optimal.

7.6.1 Experimental settings

In the third set of experiments on the multi-core machine model, the time intervals are unitary and the task computing time are integer so that a task can only finish at the end of an intervals but not during one, which means that the process of splitting an interval into two intervals in order to accurately apply the power availability changes due to the execution of a task is no longer necessary, which lowers the time complexity of our scheduler. However, the whole computations took around 80 000 hours on the university's local computing center, mainly because the variation of the upper bounds of the attributes of tasks and the genetic algorithms are very time consuming.

We run this experiment using different methods to generate the experimental datasets. An experimental dataset consists of two parts, the workload represented by the task list and the power envelope available for the task executions represented by the time interval list. The methods used to generate these values differs from one test to another and we detail them in the following.

Note that, for all the following experiments, we consider that the computational platform consists of a parallel machine with 16 cores, which means that up to 16 tasks can be scheduled simultaneously

in the same interval and we set the time unit to 6 minutes = 1/10 hour, which is also the unified duration intervals δ_x.

7.6.2 Results

Hyper-gamma tasks and realistic power envelops

For this experiment, we use the hyper-gamma distribution of the Feitelson model (see 7.1.1) to generate 10 groups of 150 task lists with 50 tasks each. Between the 10 groups the $p_{i_{\max}}$ values of each task list are the same while $\varphi_{i_{\max}}$ values range 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 7.14 presents the average NM evaluation metric from 150 executions for each heuristics over 10 experimental setups. Note that we have removed the SPT heuristic to make the figure more readable since it gives poor results that flatten the rest of he 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.

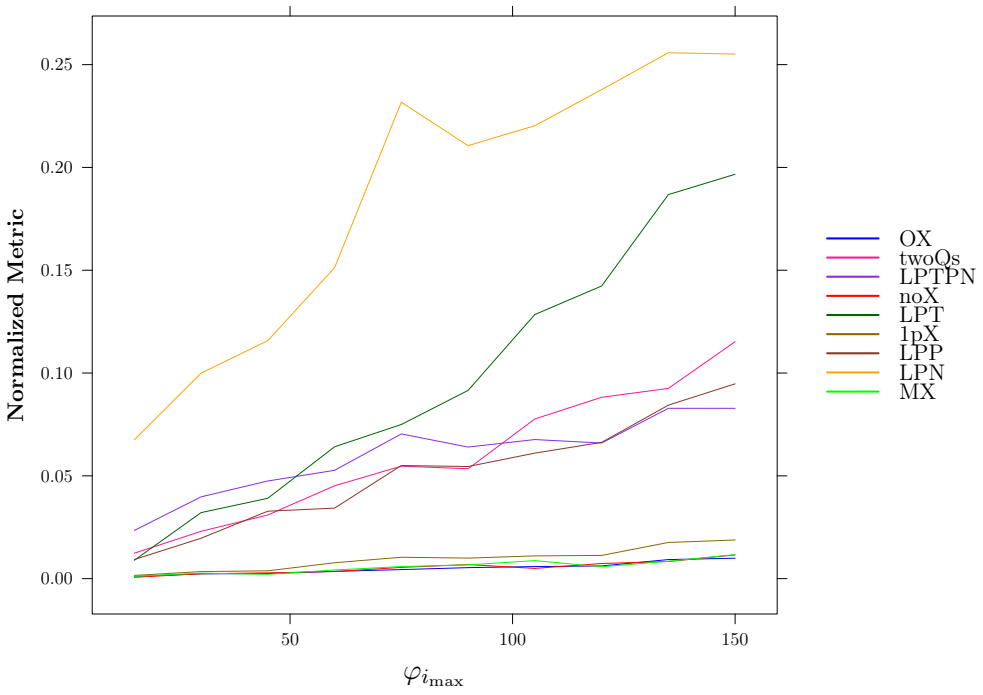


Figure 7.14 – Average NM.

Exponential tasks and realistic power envelope

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 Fig 7.15, 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.

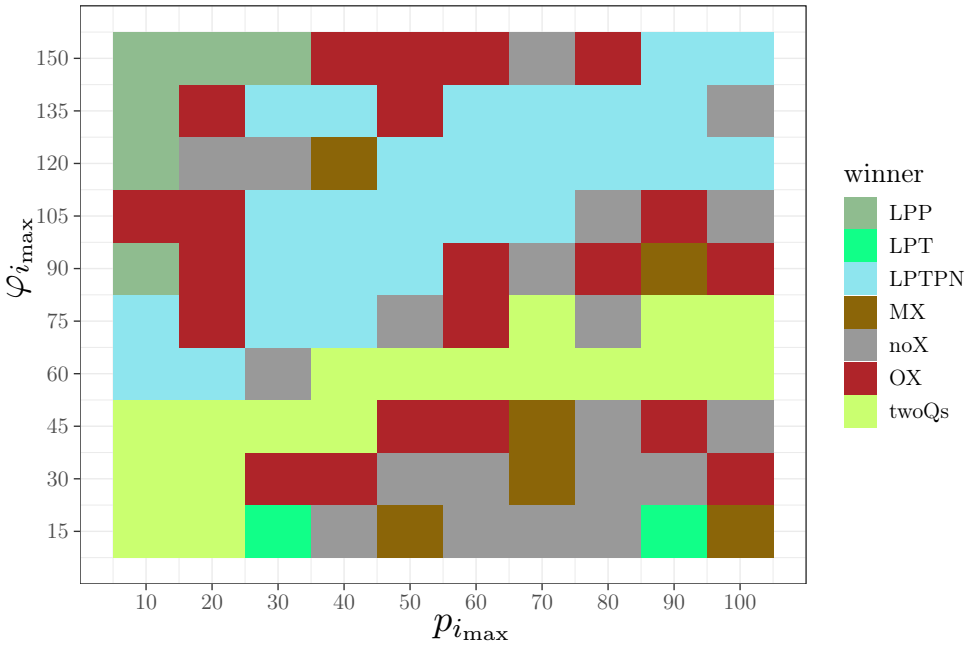


Figure 7.15 – Best average NM for experiment.

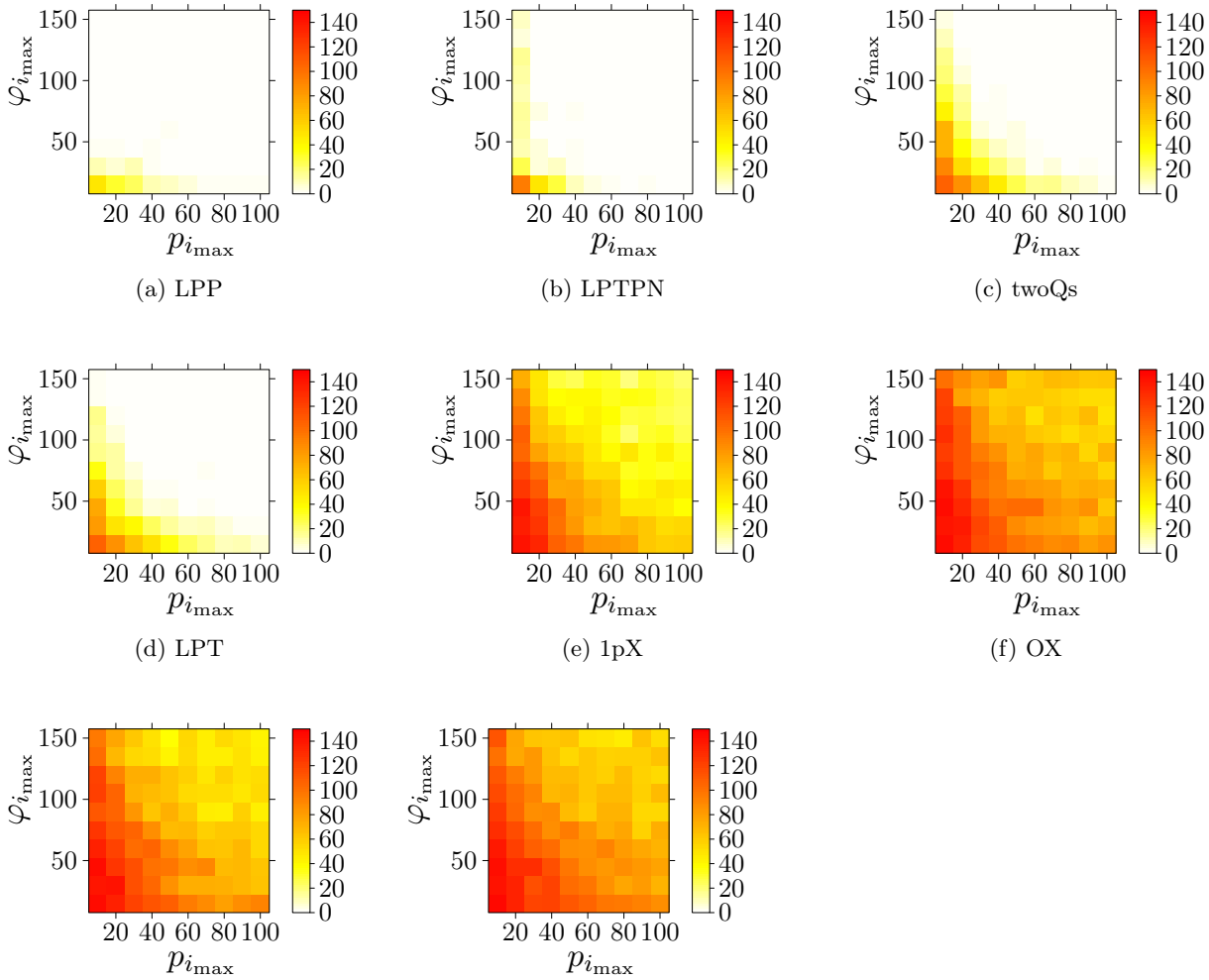
Each square in Figure 7.15 represents the average NM over 150 executions at the corresponding ($p_{i_{\max}}$ and $\varphi_{i_{\max}}$. We notice that as the processing time of the tasks increases (higher $p_{i_{\max}}$ or towards the right on the heat map), the more cases are won by genetic algorithms 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 heat map) 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 heat map (low $p_{i_{\max}}$ and height $\varphi_{i_{\max}}$) 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.

Table 7.2 – Average computation times (sec).

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 7.2 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 7.16. Yet some list algorithms are still able to provide better NM values because NM is a two dimensional evaluation metric. We recall that NM cannot be used as the fitness value in GA because it cannot be calculated until all other algorithms are run.

Figure 7.16 shows the number of times a heuristics 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 cases 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 bottom right square (simple cases with small p_i and φ_i values) is red for almost all the heuristics. In this work we calculate as well how many times each algorithm found the second and the third best C_{max} (not presented in the manuscript) and the results confirm that the genetic algorithm always finds shorter makespans than list based algorithms.



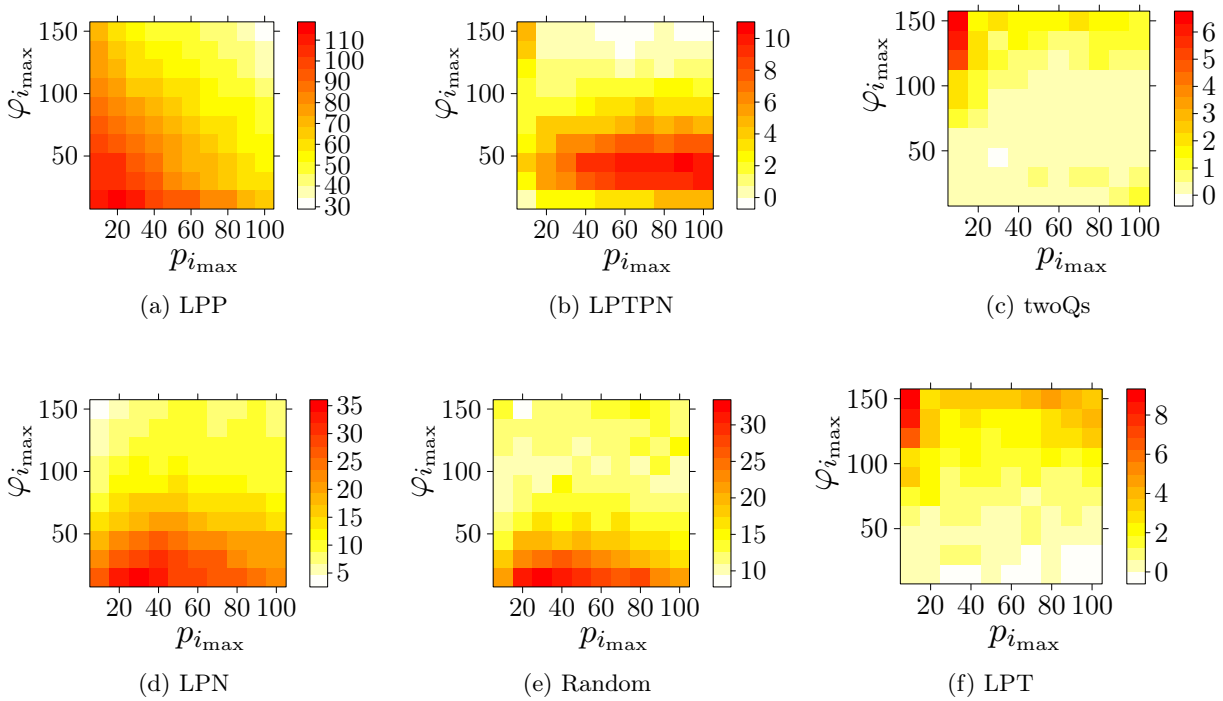
(a) LPP (b) LPTPN (c) twoQs

(d) LPT (e) 1pX (f) OX

(g) MX (h) noX

Figure 7.16 – Number of times each algorithm finds the best $C_{\max}$

Our last experiment with this set of data assesses the heuristics performance regarding the total flowtime. All the runs are won by the *SPT* heuristic, therefore, we do not present a heatmap but rather present in Figure 7.17 the distance of each list algorithm from the best *PERFLOW*, which is produced by *SPT*.



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

(d) LPN (e) Random (f) LPT

Figure 7.17 – *PERFLOW* distance from the average *PERFLOW* of SPT

In part of this experiment, we modify the fitness function in GA to consider $\sum C_i$ instead of C_{max} and we *SPT* order as a seed since it gives good *PERFLOW* results. The GA settings in this test are set to stop after 50 iterations without improvement and the GA uses the order crossover OX. The goal of this experiment is to examine is a well configured GA can be used for total flowtime optimization as well as the makespan optimization. Our results show that the tested GA indeed improves the initial seed solution provided by SPT in all the tested cases. Figure 7.18 presents how far GA was able to improve SPT's average *PERFLOW*, we notice that the bigger the tasks get in both time and power axises, the less efficient SPT gets compared to GA.

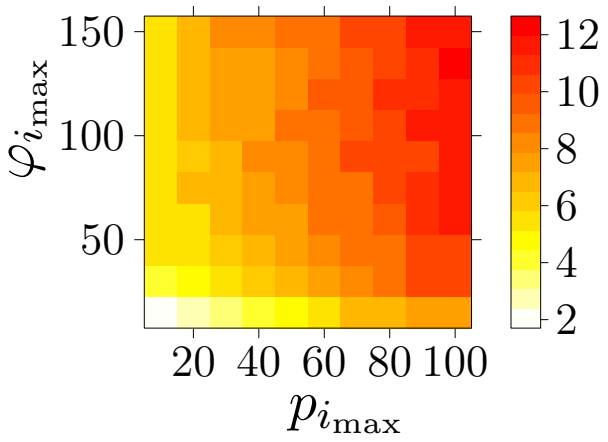
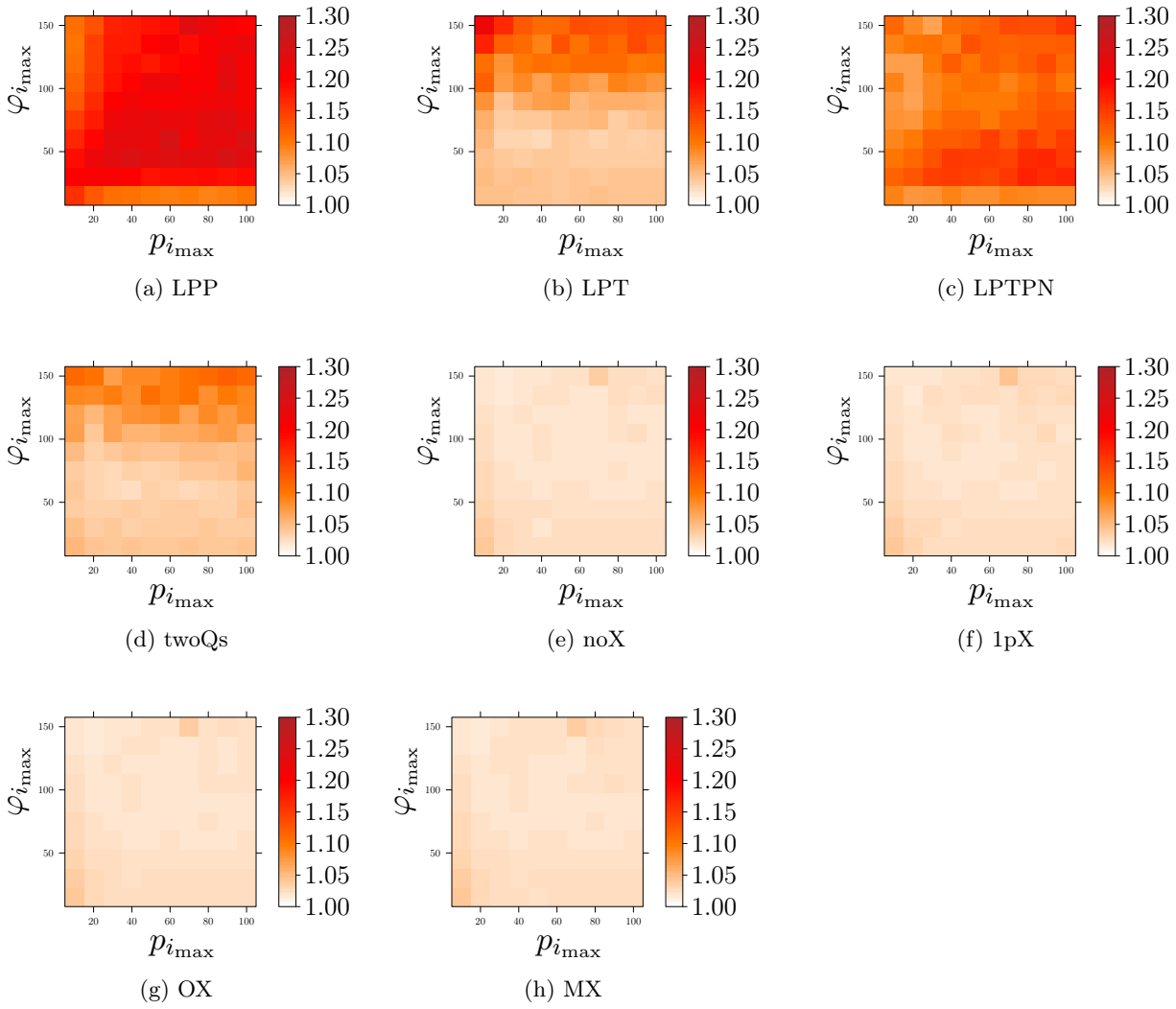


Figure 7.18 – Average *PERFLOW* distance Between SPT and GA.

Task from interval and realistic power envelops

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.



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

(d) twoQs (e) noX (f) 1pX

(g) OX (h) MX

Figure 7.19 – Distance from the optimal.

Figure 7.19 presents the average distance from the optimal over 150 executions for each heuristics using a unified scale to make the comparison clearer. We note that all GAs are closer to the optimal than list based heuristics. This is however at a cost of much higher computation times as shown in Table 7.3. On the other hand the *twoQs* and *LPT* simple heuristics give close to optimal results, with less than 5% of difference in these cases, when the power consumption φ_i is small and in most case this difference is under 10% which is quite good.

Table 7.3 – Average computation times (sec).

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

Figure 7.20 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 the 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.

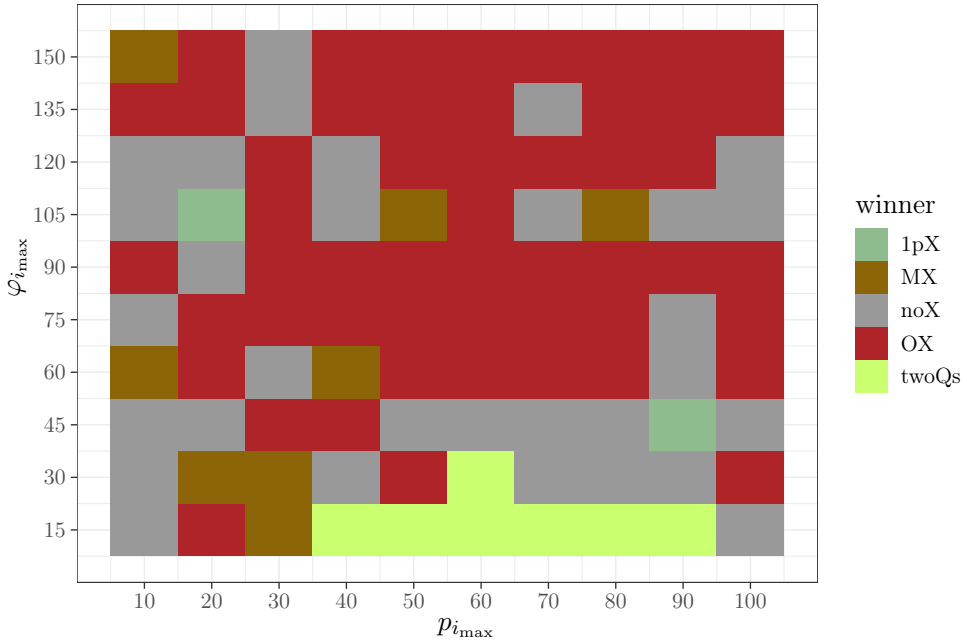


Figure 7.20 – Best average NM.

Hyper-gamma tasks and interval 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 7.6.2 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 7.21 presents the average distance from the optimal from 150 executions for each heuristics 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 previous 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 consumption varies.

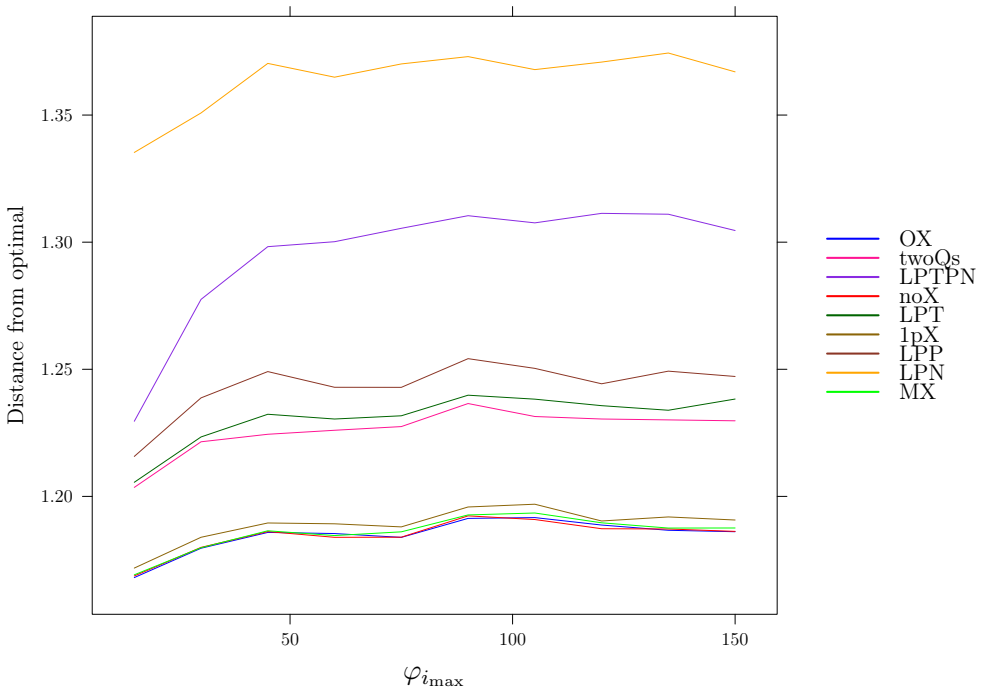


Figure 7.21 – Distance from the optimal.

7.6.3 Summary

The results of these experiments show 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 *LPT* that usually has good performance on this problem, degrade when the power consumption 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 where the optimal solution is known, show that genetic algorithms can be 5 to 10 % closer to the optimal solution, but at a cost of longer computation times.

Chapter 8

Multi-machine platform experiment

As mentioned in 4.2.2, the multi-machine computational platform consists of identical parallel machines. Each machine is a multi-core processor that consumes static power P^{static} when it is running. In this experiment we cannot compute the optimal solution because of the machine switch on/off process, since we cannot know in advance how many machines will be running during the makespan, it is hence impractical to generate a power envelope with the same ($power \times time$) area as a set of tasks and vise versa, without knowing how much energy will be lost on running the machines. Therefore we do not conduct the Intervals From Tasks (IFT) and Tasks from Intervals (TFI) experiments that are designed to measure the distance of a solution from the optimal solution. Thus, for this computational platform model we only conduct two experiments, namely the hyper-gamma tasks combined with realistic power envelopes and the exponential tasks combined with realistic power envelope. To evaluate the performance of the proposed heuristics on this platform, we use the NM and $PERFLOW$ metrics presented in Section 7.2 for the makespan and flowtime objectives respectively. We recall that:

$$NM = \sum out \div \sum total$$

Where $\sum out$ is the sum of the $p_i \times \varphi_i$ area of all the tasks that are scheduled after the best $C_{\max}$ of that simulation and $\sum total$ is the total $p_i \times \varphi_i$ area of the task list, and that:

$$PERFLOW = (\sum (C_i - useless_i)) / \sum p_i$$

8.1 Experimental settings and results

The platform consists of several identical multi-core processors. The following experimental parameters of each processor are set based on the values measured in [42]. The static power consumption of each processor P^{static} is set to 95 Watts. The time to switch a processor on $t_j^{off \to on} = 150$ seconds and the time to switch a processor off $t_j^{on \to off} = 6$ seconds. We assume that the number of processors available in the platform equals to the length of the task list. This way the availability of processors is guaranteed which exempts the processor as a resource constraint, and we focus on the variable green power constraint. Other experiments can be done in the future to consider both constraints at the same time. The number of processor cores in each processor is $nb - cores = 4$.

The realistic power envelope used in both experiments is similar to the one described in Section 7.1.2. In order to examine the effect of the power availability on the heuristics' performance,

we conduct several tests with increasing maximum available power Φ in each set of power envelopes generated by the realistic model. We start by setting the power generation parameters in the realistic model as follows: solar panels total area = $2000m^2$, pvEfficiency = 0.163, number of wind turbines = 1, and turbinePowerNominal = 350, where pvEfficiency and turbinePowerNominal are the efficiency factors of the used solar panels and wind turbines respectively. These settings yielded power envelopes with a maximum available power Φ of about 350 Watts, we then multiply the values of these settings by 2, 5 and 10 times to yield a maximum Φ of around 700, 1750 and 3500 Watts respectively. We partition the power envelope into unified intervals Δ_x with length of δ = 1 time unit = 1/10 hour. However, the data used in this realistic model is collected every hour, therefore, each 10 consecutive intervals have the same available power level Φ_x that corresponds to the available power at that hour. Each time interval list contains 10000 intervals, representing in total the power envelope.

8.1.1 Hyper-gamma tasks and realistic power envelopes

In this experiment we use hyper-Gamma distribution law to generate p_i values in order to produce tasks with processing times that are close to real data. The resulting task lists contain 50 tasks per list, with a range of processing times between 1 and 500 time units. We set the time unit to 6 minutes = 1/10 hour. As we were lacking real data values for the power consumption of the tasks, we use random generation of φ_i with a uniform distribution law between 1 power unit and $\varphi_{i_{\max}}$. In this experiment $\varphi_{i_{\max}}$ ranges between 15 and 150 power units, by steps of 15.

Normalized metric results

Figure 8.1 presents the value of NM for each heuristics with increasing value of maximum task power consumption φ_{max} on the horizontal axis where the maximum available power Φ is 350 Watts. Remember that the lower the NM the better the performance. We notice that with the increase in φ_{max}, the performance of most heuristic decreases, especially the heuristics that use LPT to sort the task list (blue lines). We also point out that heuristics that take the task power consumption into consideration when sorting the task list such as twoQs (red lines), LPTPN (green lines) and LPP (brown lines) outperform the LPT based heuristics when the power constraint is strict, which indicates that the power constraint is the most important in this case. The reason is that the available power is not enough to execute all tasks simultaneously, which makes the decision of which task to execute first very important.

Figure 8.2 presents the case where the power constraint is the least strict. In this case, the tasks are less concurrent for power due to high level of available power. This problem is thus closer to the classical scheduling problem without power constraints, in which case LPT provides good performance regarding minimizing the makespan. We notice a degradation in the performance of heuristics that take the tasks power consumption into consideration such as LPN (purple lines) and LPTPN (green lines).

Considering Figures 8.1 and 8.2 there are some remarks that are applicable to both. The light green dash-dotted line in both previous figures represents the NM of the tested GA, which uses the Order Crossover(OX) with wheel selection. We notice that this line always has the value of $NM = 0$, since the NM value of each heuristic during one simulation is calculated in relation to the best C_{max} produced in that simulation, which means that GA always produces the shortest or at least as short makespan as all other heuristics. The difference in performance between GA and

the other heuristics increases as the power constraint becomes stricter, on the other hand, when the power envelope provides plenty of power for task execution, all the heuristics that use LPTPN, twoQs, LPT and LPP orders provide a very close performance level to GA for tasks with low power consumption, and for tasks with higher power consumption, the performance of Stripe based heuristics degrades in comparison to BS and list based heuristic as we notice that LPT, BSLPT, twoQs, BSTwoQs and LPP keep the closest distance to GA's performance line. In addition, even list and BS heuristics that use LPTPN for sorting the task list provide worse performance when the tasks power consumption gets higher than 50 power units. We can then draw the conclusion that in the least strict power constraint case, the GA does not significantly improve the solution used in its initial seeds which is provided by LPT, twoQs and LPP for tasks that does not require a lot of power, and the tasks processing time is a more important prioritizing criteria than the tasks power consumption even for tasks with high power consumption. We finally point out that LPN which only prioritize the tasks power consumption does not perform well for the makespan minimization objective, neither does SPT which is designed for flowtime minimization, in fact, other than SPT, all the tested heuristics improve the produced schedule which is found when leaving the task list in its original random order, which proves the necessity for smart workload management in this problem.

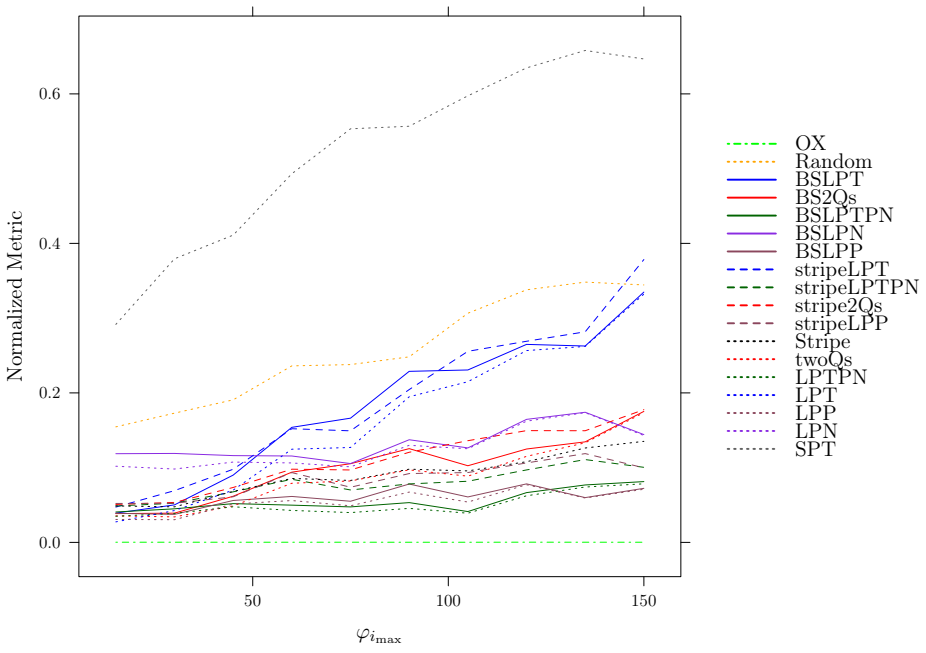


Figure 8.1 – Average NM, $\Phi_{max} = 350$ (with Random and LPN).

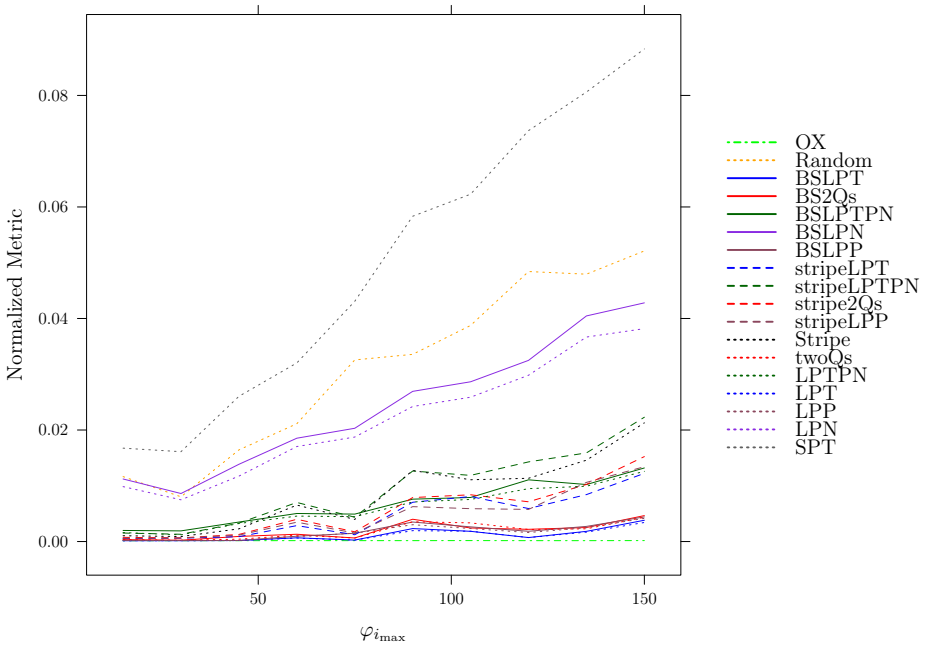


Figure 8.2 – Average NM, $\Phi_{max} = 3500$ (with Random and LPN).

The next four Figures 8.3, 8.4, 8.5 and 8.6 present the value of NM for each heuristic with increasing value of the maximum available power Φ of 350, 700, 1750 and 3500 Watts respectively. We point out that we leave out heuristics that have much worse performances than others in order to zoom in to compare the other heuristics, as a result Figures 8.3 and 8.6 are the same as Figures 8.1 and 8.2 only without the performances of SPT, LPN and random order heuristics. We also point out that we only run GA on the two cases that present the most and the least strict power constraint (maximum available power Φ is 350 and 3500), as it is time costly.

We notice that increasing the power availability affects the behavior of the proposed heuristics. When the power constraint is less strict (Figure 8.6), the problem approximates the classic scheduling problem of minimizing makespan, in which case heuristics that are based on the traditional makespan minimization solution (LPT) provide good schedules (the blue lines get lower), while the performance of all Stripe heuristics and heuristics that consider the task power consumption such as LPTPN based heuristics decreases (the green get higher). The stricter the power envelope gets, the more important the power consumption of tasks becomes, therefore the performance of heuristics that consider the tasks power consumption such as LPTPN and BSLPTPN increases (the green lines get lower), while the LPT based heuristics provide the worst performance when the power constraint is the most strict (Figures 8.3). We notice that heuristics that sort the task list based on the hardest task to schedule first (LPP) provide good schedules consistently through the variation of the power envelope, while the Stripe heuristics (all the dashed lines) under-perform both BS and list based heuristics that use the same sorting policy.

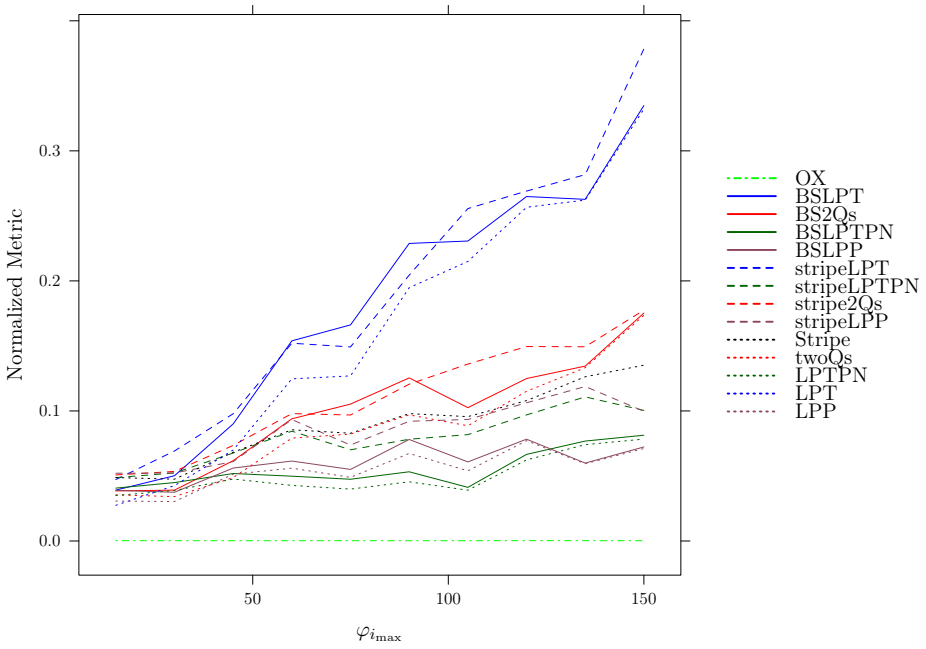


Figure 8.3 – zoom in Average NM, $\Phi_{max} = 350$ (without SPT, Random and LPN).

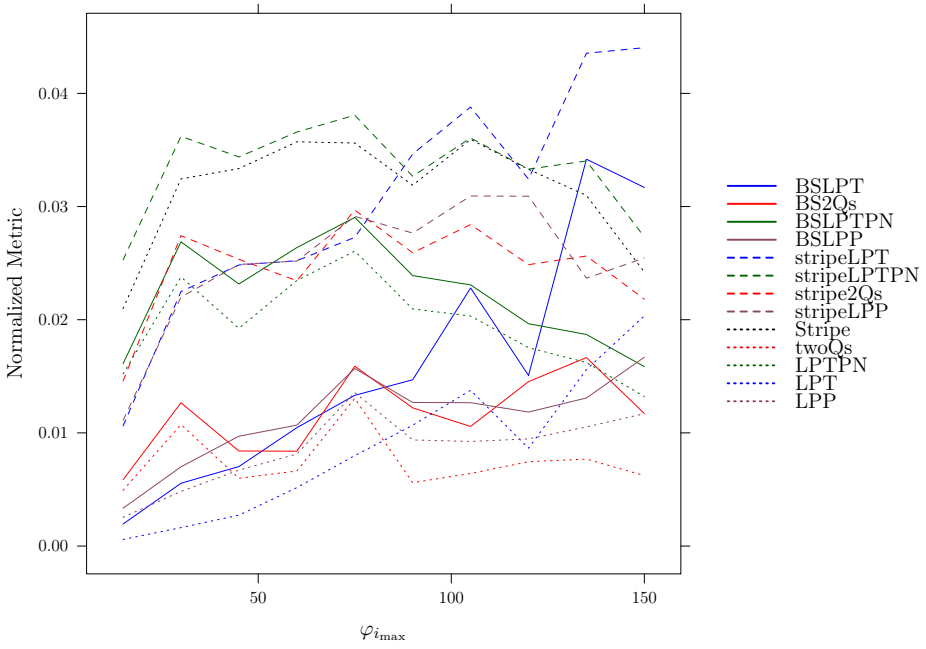


Figure 8.4 – zoom in Average NM, $\Phi_{max} = 700$ (without SPT, Random and LPN).

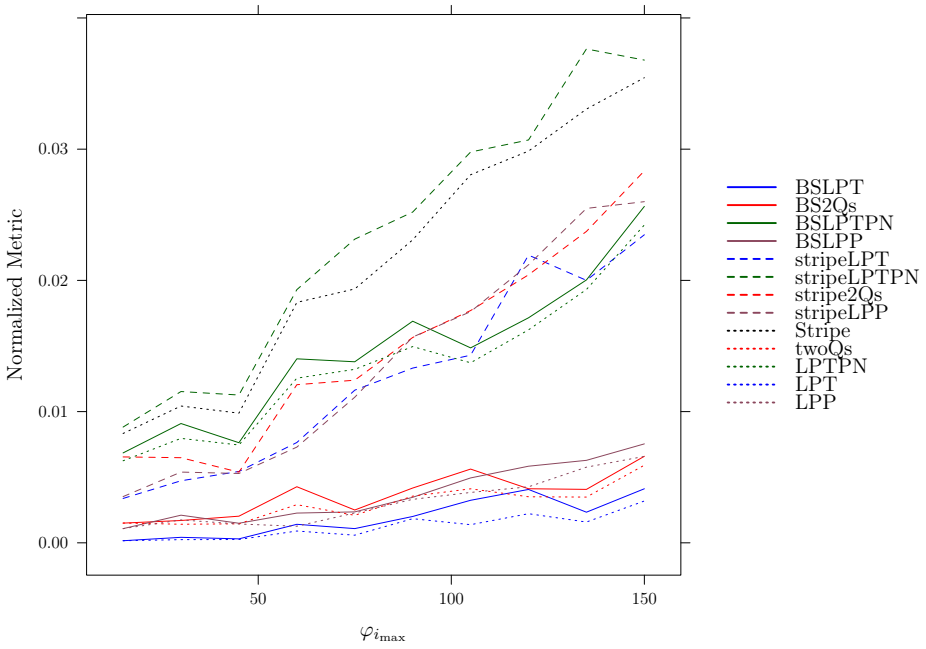


Figure 8.5 – zoom in Average NM, $\Phi_{max} = 1750$ (without SPT, Random and LPN).

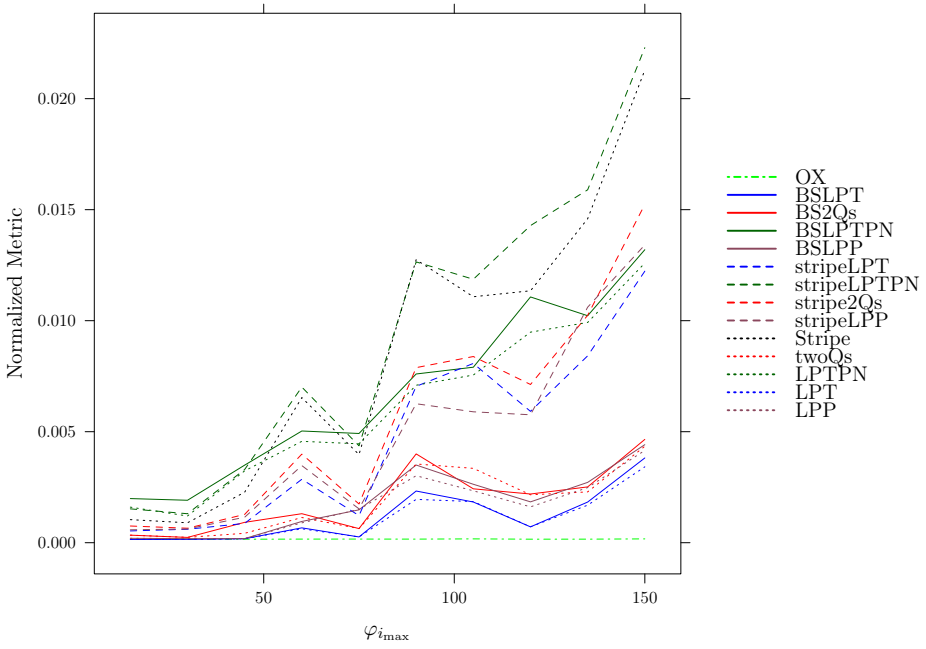


Figure 8.6 – zoom in Average NM, $\Phi_{max} = 3500$ (without SPT, Random and LPN).

Computation time (time to find a schedule)

Tables 8.1 and 8.2 present the average computation time of heuristics in both cases where GA is tested. We chose LPT and twoQs orders as a representation of each heuristic class to compare to the tested GA and to LPP. We notice that GA takes much higher computation time to find a schedule. We also point out that the computation time of GA in the case where it performs much better than the other heuristics ($\Phi_{max} = 350$) is up to 5 times higher than its computation time in the case where its performance is closer to other heuristics ($\Phi_{max} = 3500$), this means that when the scheduling problem is easier (less strict power constraint), the GA reaches its stopping conditions (25 iterations without improvement) faster. In general, even fast list based heuristics need ten times the computation times when the power constraint is strict to place all tasks in comparison to their computation time in the ($\Phi_{max} = 3500$) case. We finally point out that LPP is the only heuristic that does not show significant difference in computation time between the two cases of power envelope, however, it still takes 10 times the time as any other list based heuristic, due to the process of calculating in how many time slots each task can fit, before sort the task list. This process increases the time complexity of LPP which is however indifferent of the data set characteristics. In general, if we compare the computation time plots to the average NM plots, we notice that the computation time is rational to the length of the schedule, which is logical because the longer the schedule, the more time intervals are explored to find it. This is coherent with the observation that both LPP's performance and computation time are consistent with the power envelope variability.

Table 8.1 – Computation time $\Phi_{max} = 350$ (with GA)

Algo	OX	BSLPT	StripeLPT	LPT
Time (s)	24674.80	87.08	67	3.39
Algo	LPP	BSTwoQs	StripeTwoQs	TwoQs
Time (s)	35.68	89.83	65.34	3.59

Table 8.2 – Computation time $\Phi_{max} = 3500$ (with GA)

Algo	OX	BSLPT	StripeLPT	LPT
Time (s)	6157.18	49.04	0.71	0.32
Algo	LPP	BSTwoQs	StripeTwoQs	TwoQs
Time (s)	34.59	49.35	0.81	0.35

In order to compare the computation times of other heuristics, we leave GA out of the next four Figures 8.7, 8.8, 8.9 and 8.10 which present the computation times of all heuristics with increasing value of maximum available power Φ of 350, 700, 1750 and 3500 Watts respectively. We notice that the performance of a heuristic affects its computation time, in other words, the order of the task list affects how much time is needed to find a valid schedule, that is mainly because worse performance means longer makespan, which means that the heuristic must explore more time intervals in order to place all the tasks. However, when the same order is used, we notice that in most cases, binary search based heuristics are slower than Stripe based heuristics, yet they give better results. This is due to the higher time complexity of the binary search algorithm in comparison to the Stripe approach. In addition we point out that list based heuristics can find a schedule 60% faster than Stripe heuristics and up to 99% faster than binary search heuristics. The tested GA on the other

hand is around 18000 times slower than list heuristics while it offers a 0.05% improvement over the performance of the best list based heuristic.

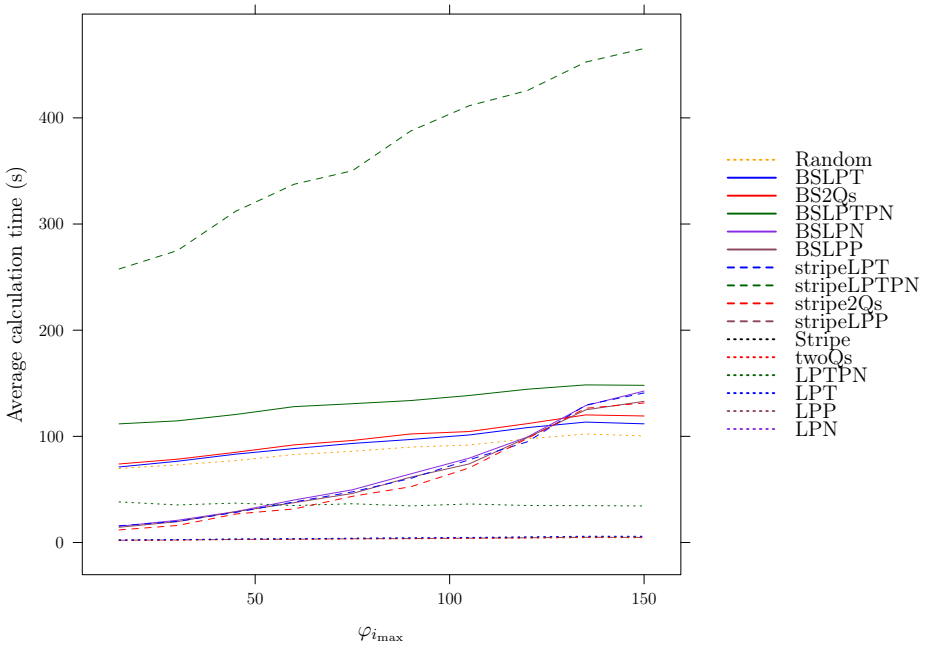


Figure 8.7 – Computation time, $\Phi_{max} = 350$.

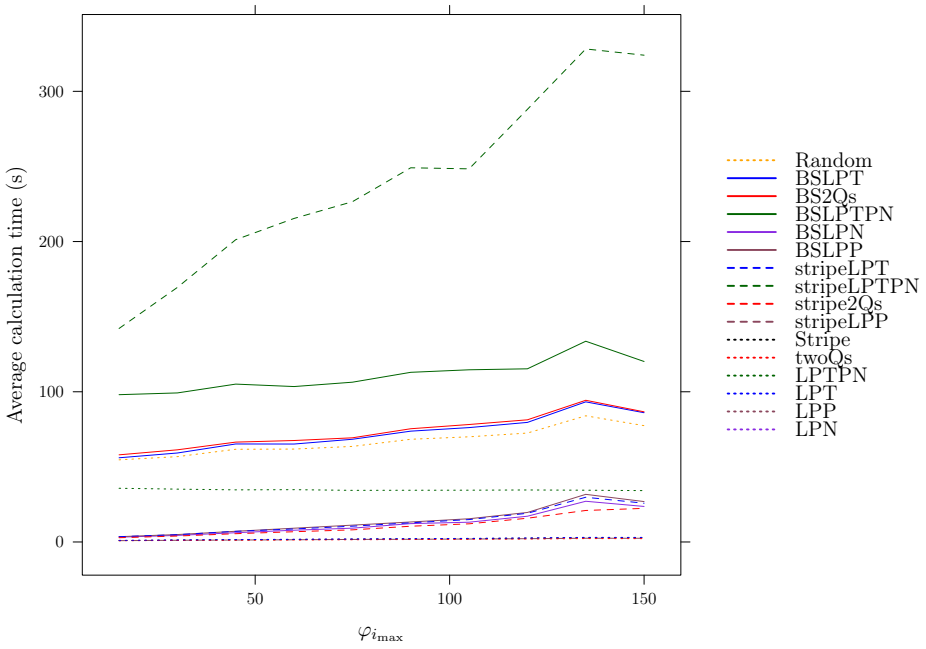


Figure 8.8 – Computation time, $\Phi_{max} = 700$.

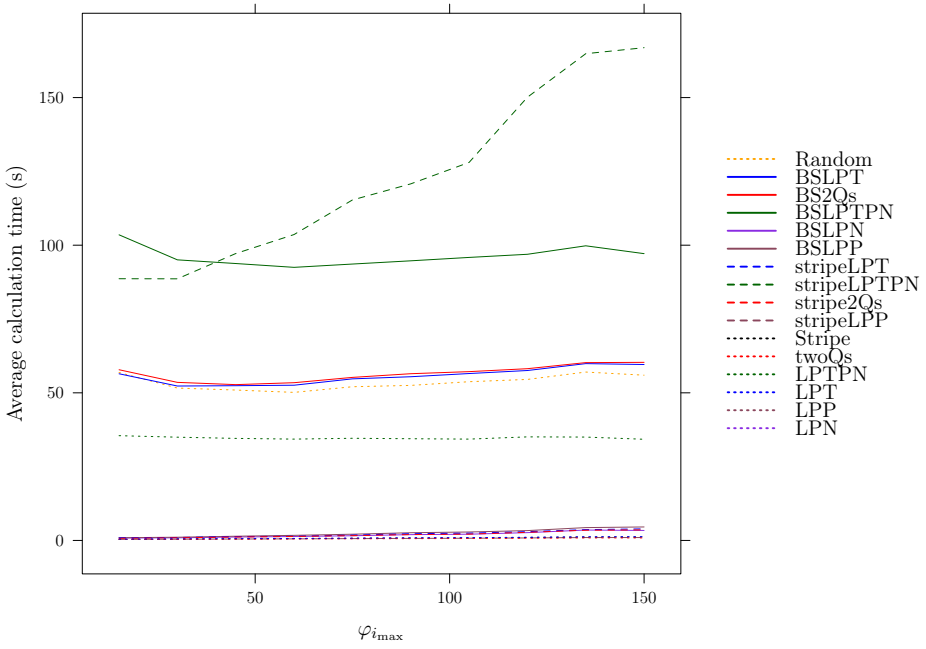


Figure 8.9 – Computation time, $\Phi_{max} = 1750$.

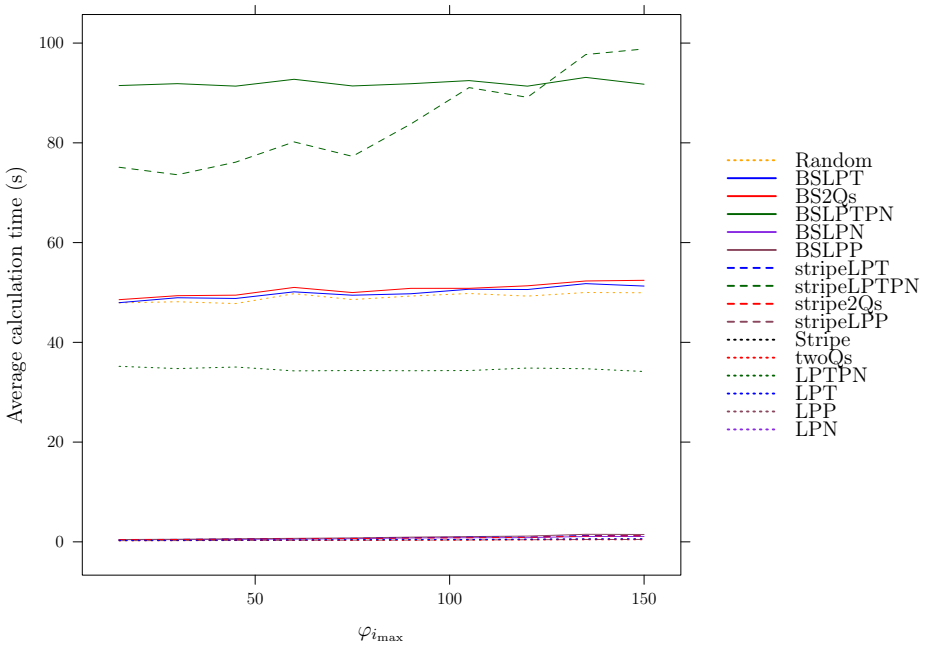


Figure 8.10 – Computation time, $\Phi_{max} = 3500$.

on/off

The next four Figures 8.11, 8.12, 8.13 and 8.14 present how many times a heuristics switches a processor on and off on average.

We point out that when the available power is much higher than the tasks average power consumption, more power is available to switch on more processors. In Figures 8.12 and 8.13 we notice that the number of on/off operations decreases when the tasks power consumption increases. However, no such slop is noticeable in Figure 8.11 because in this case the available power level is not enough to run more processors even when the tasks power consumption is at its lowest.

In comparison to the average NM results, we notice that the heuristics that perform less on/off operations provide better NM.

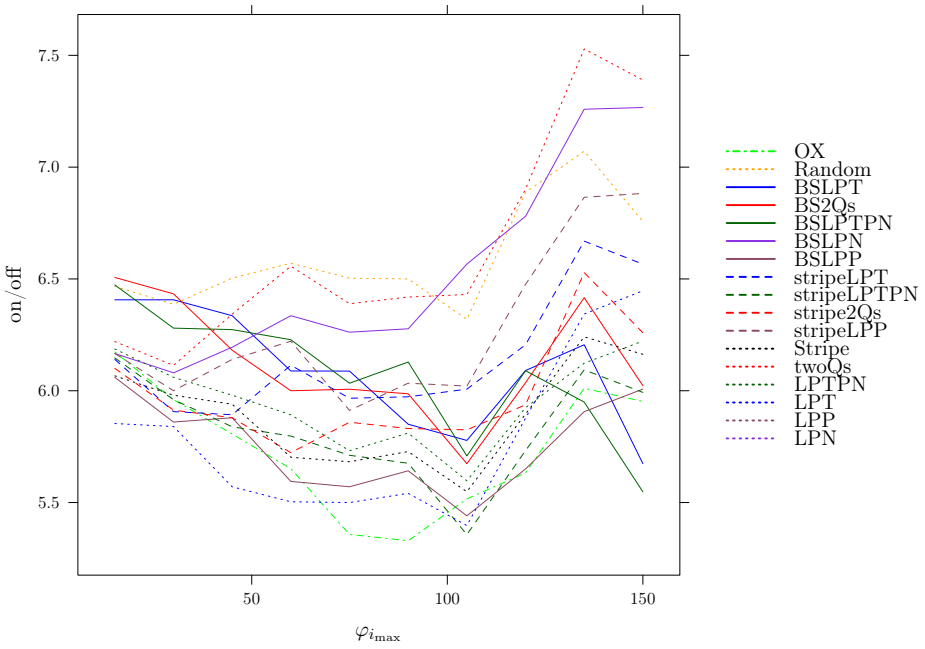


Figure 8.11 – on/off, $\Phi_{max} = 350$.

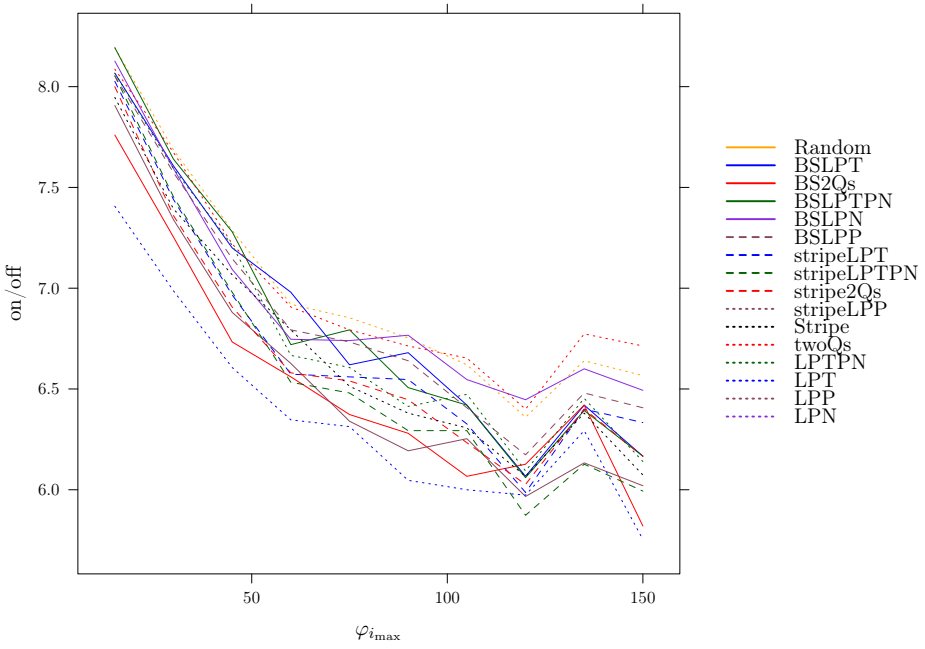


Figure 8.12 – on/off, $\Phi_{max} = 700$.

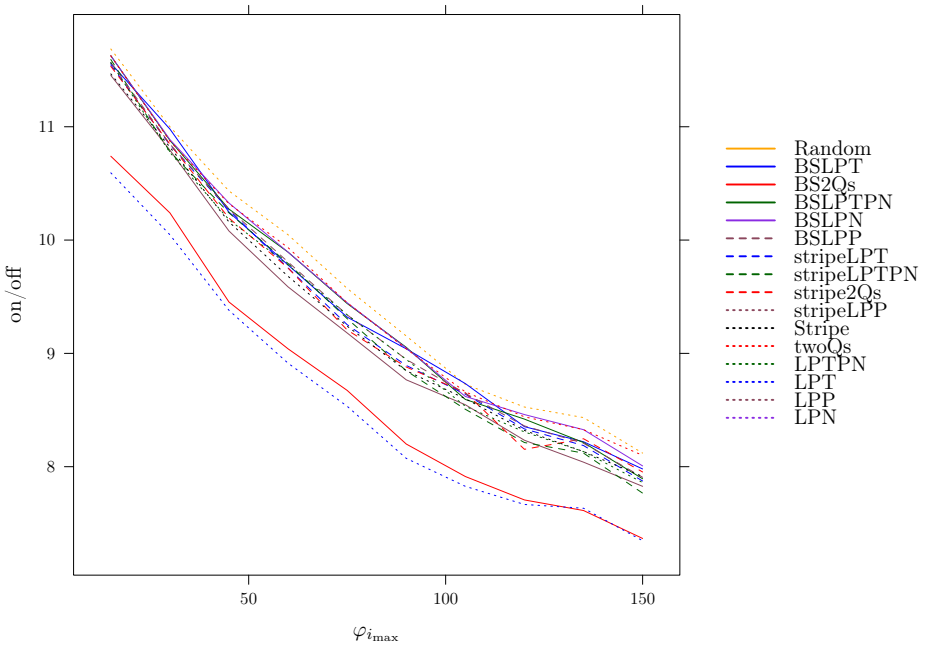


Figure 8.13 – on/off, $\Phi_{max} = 1750$.

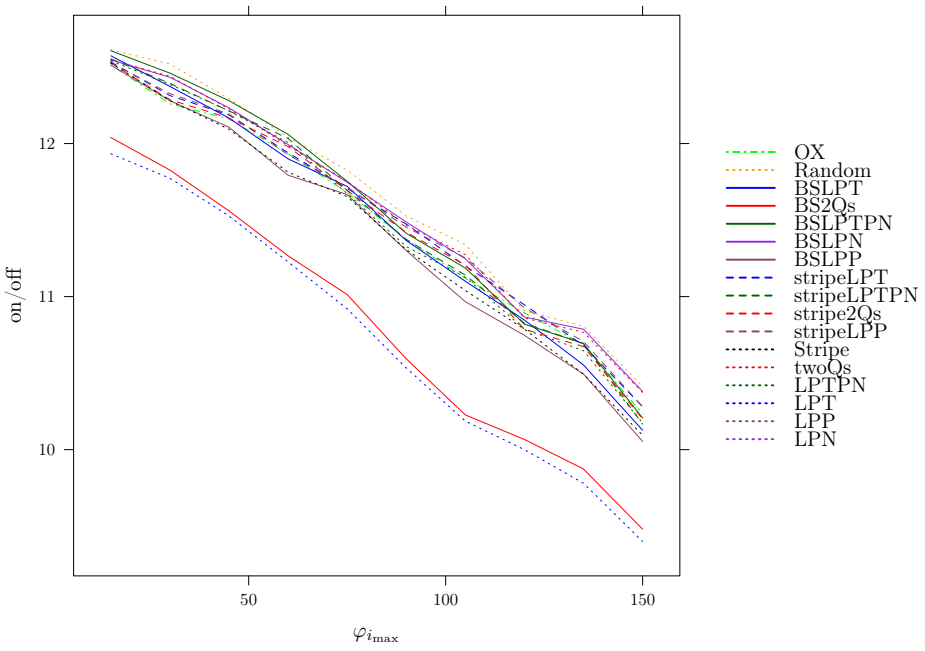


Figure 8.14 – on/off, $\Phi_{max} = 3500$.

Energy waste

The next four Figures 8.15, 8.16, 8.17 and 8.17 present the average energy waste of each heuristic. We define energy waste as the sum of the energy ($power \times time$) within the makespan that is left in used intervals or unused intervals that have at least enough power available to switch one processor on and the energy cost of switching processors on and off, and the static energy consumption of processors.

We notice that when the power constraint is strict ($\Phi_{max} = 350$), when tasks have lower power consumption, they are easier to schedule. It means that it is more probable to find a time slot of successive intervals with enough power to execute the tasks. Therefore, less energy is wasted in unused intervals. When the task power consumption increases, it gets harder to schedule the tasks: Even if an interval with enough available power to execute a task is found, it is less probable to find a time slot of successive intervals with enough power to execute that task, the interval remains thus unused, and the energy during this interval is wasted according to our definition of energy waste.

On the other hand, with the increase in the available power level, at some point, tasks with high φ_i become as easy to schedule as tasks with lower power consumption. Yet, they would fill the power envelope better in the scene that less power remains unused in a used interval, therefore, in cases where the power constraint is not strict, the power waste curve begins to incline when φ_{max} increases.

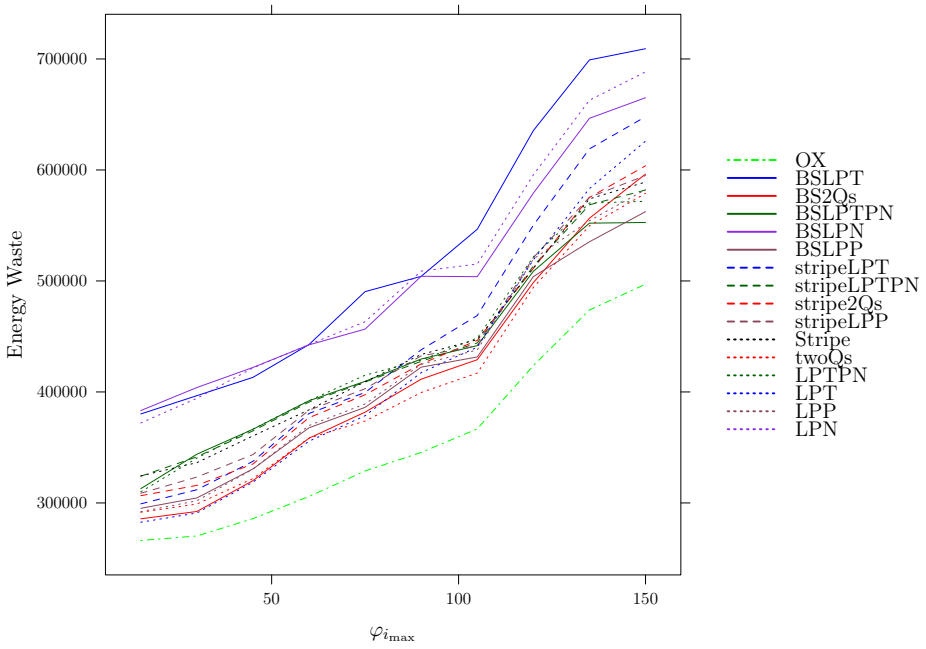


Figure 8.15 – Energy waste, $\Phi_{max} = 350$.

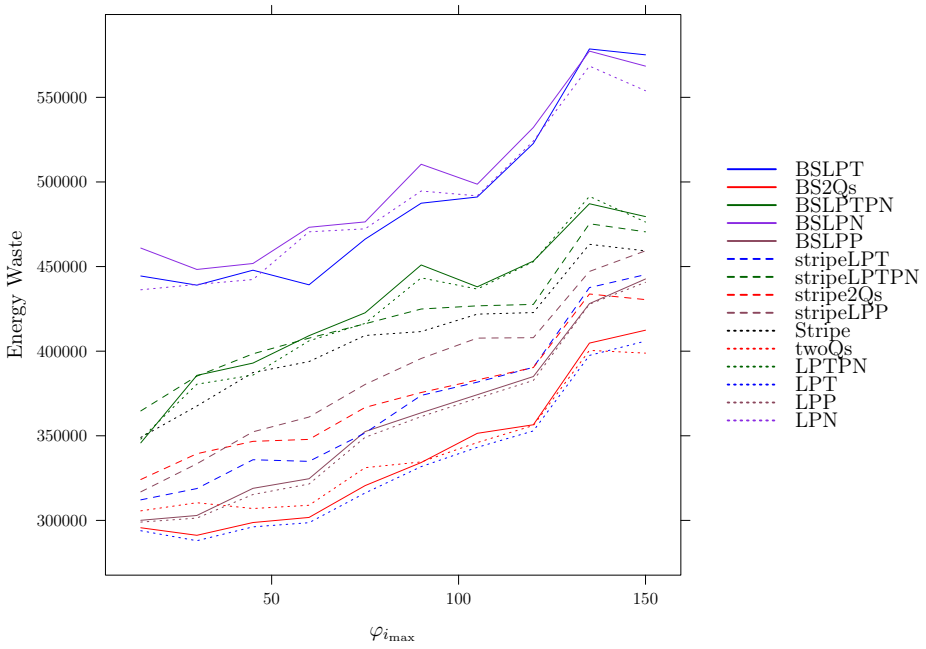


Figure 8.16 – Energy waste, $\Phi_{max} = 700$.

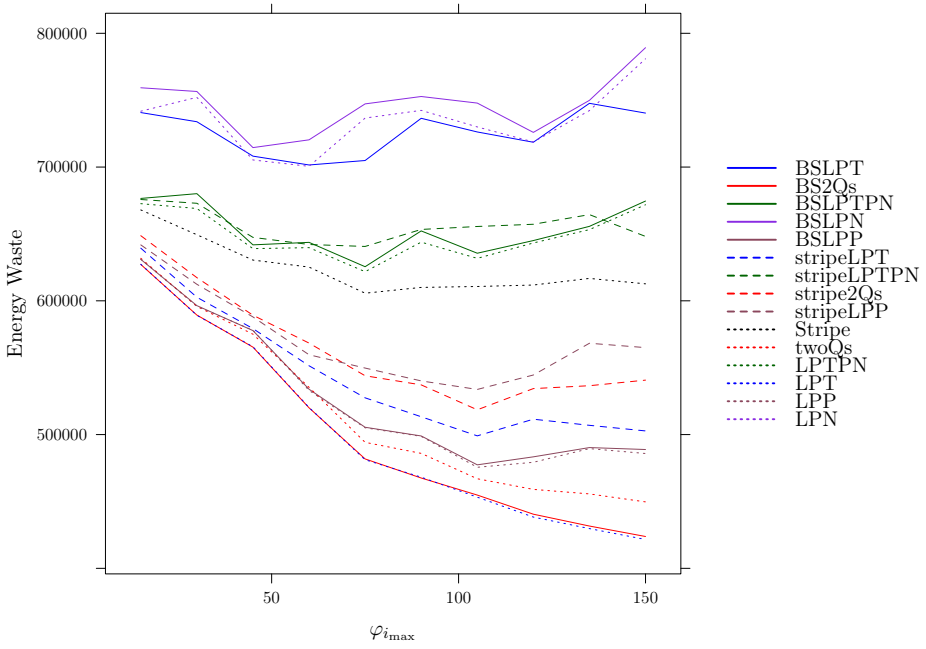


Figure 8.17 – Energy waste, $\Phi_{max} = 1750$.

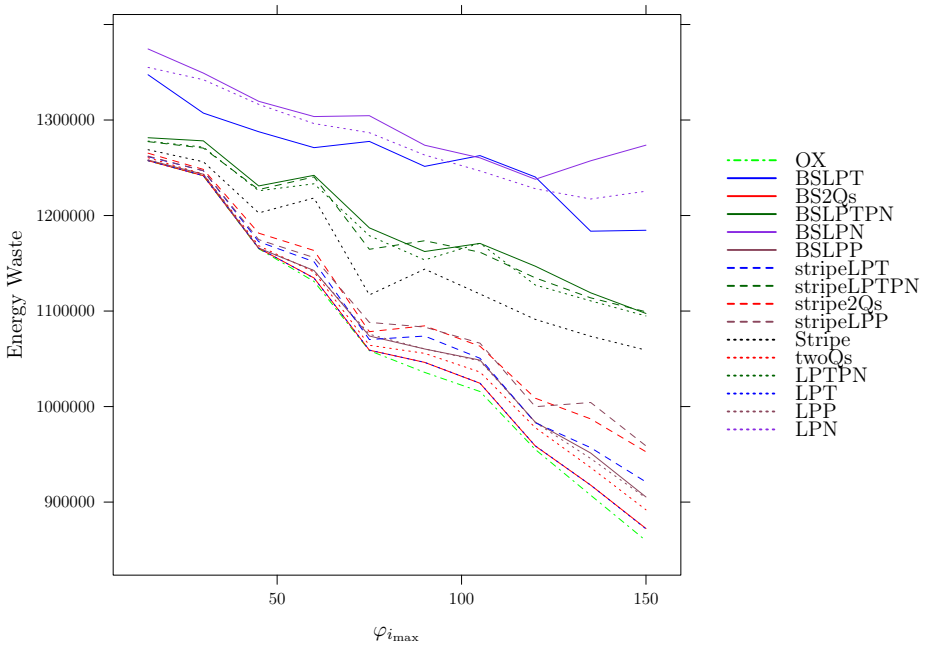


Figure 8.18 – Energy waste, $\Phi_{max} = 3500$.

8.1.2 Exponential tasks and realistic power envelope

The simulation using tasks with processing times p_i that use exponential random law presents more test cases than the simulation using hyper-gamma distribution, due to the variation in the maximum processing time in a set of tasks p_{max}, as a result, it is much more time costly. Therefore, we only evaluate in this case the proposed heuristics where the maximum available power in the realistic power envelopes Φ is 350 and 700 Watts.

In both experiments, the tested GA which is the genetic algorithm that uses the order crossover (OX) and wheel selection gives the best average NM in all the tested cases. Therefore we only present a comparison in average NM between the rest of the heuristics, then we show how far those heuristics are from GA.

Normalized metric results

Figure 8.19 presents the value of NM for each heuristics with increasing value of maximum task power consumption φ_{max} on the vertical axis and increasing value of maximum task processing time p_{max} on the horizontal axis where the maximum available power Φ is 350 Watts. We can distinguish two areas on this heat-map figure. When the task power consumption $\varphi_{max} \leq 75$, LPN order based heuristics give the best performance in most cases, the reason LPN is able to provide good results in this experiment while it performed badly in the hyper-gamma tasks experiment is because the tasks that are generated using hyper-gamma distribution have an average processing time of around 500 time units, while in this experiment tasks have shorter processing times, which makes the tasks power consumption a more important criteria. The right lower pink corner represents the case where the processing time increases enough to give the Stripe heuristics using LPTPN order the edge, this indicates that the Stripe heuristic outperforms the BS and list heuristic for tasks that have the longest processing time and the lowest power consumption. On the other hand, for the tasks with the shortest processing times, BS outperforms list and Stripe heuristics (the left "saumon" column).

When the task power consumption $\varphi_{max} > 75$ the power constraint becomes too strict that only the heuristics that order the task list based on the tasks Least Possible Places (LPP) first are able to find the best makespan, again, for shorter tasks, BS outperforms list and Stripe heuristics in this case as well.

Figure 8.20 presents the value of NM for each heuristics with increasing value of maximum tasks power consumption φ_{max} on the vertical axis and increasing value of maximum tasks processing time p_{max} on the horizontal axis where the maximum available power Φ is 700 Watts. We notice that the area where LPTPN based heuristics provide the best performance expands in comparison to the little pink corner from the previous figure, this area covers now the diagonal right half of the heat-map, while the performance of the LPT sorting policy drastically decreases, this indicates that under a less strict power constraint, the tasks power consumption becomes less relevant against the tasks processing time. The Stripe heuristic using LPTPN order outperforms the list based LPTPN when the tasks power consumption is closer to their processing time, we can say then that the Stripe heuristic performs better with square shaped tasks (remember the rectangular representation of tasks in Figure 4.1).The upper left half of the heat-map is mostly dominated by LPP where tasks have high power consumption and processing time $p_i \leq 60$, and when the processing time is below 25 time units, StripeLPP gives the best performance in most of the test cases. We notice a degradation in the performance of BS in comparison to the case of $\Phi = 350$, we can then say that when the power constraint is less strict, lower complexity heuristics such as list algorithm and Stripe algorithm

provide a good solution, however, in the hard cases of the problem, more complex algorithms can give the best solution in some test cases. This once more asserts the need for advanced workload management solutions in an HPC system powered solely by renewable power. However, the time complexity of the solution should be taken into account when finding the schedule in these systems, the computation times of the proposed heuristics are discussed later in this chapter.

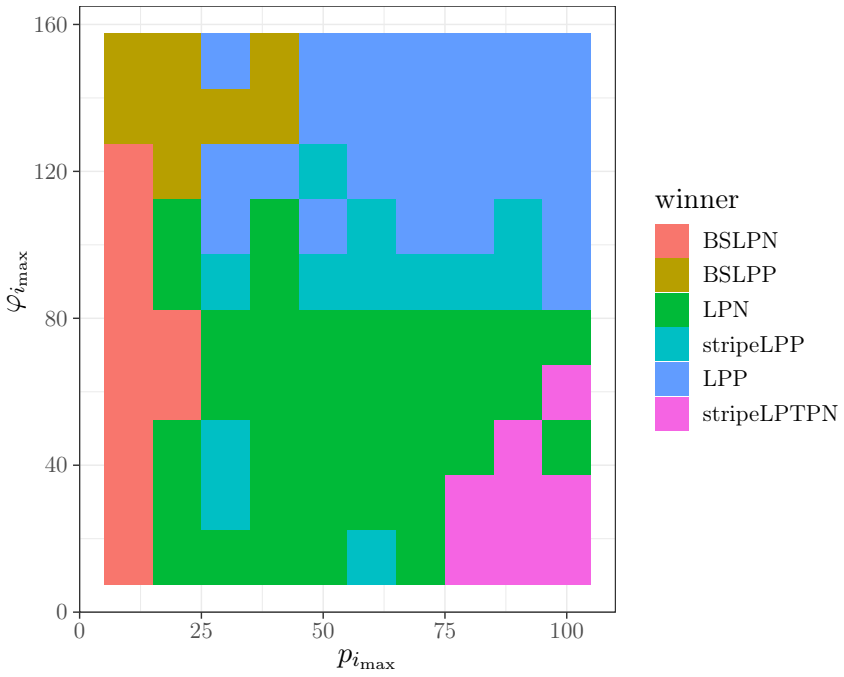


Figure 8.19 – NM, $\Phi_{max} = 350$.

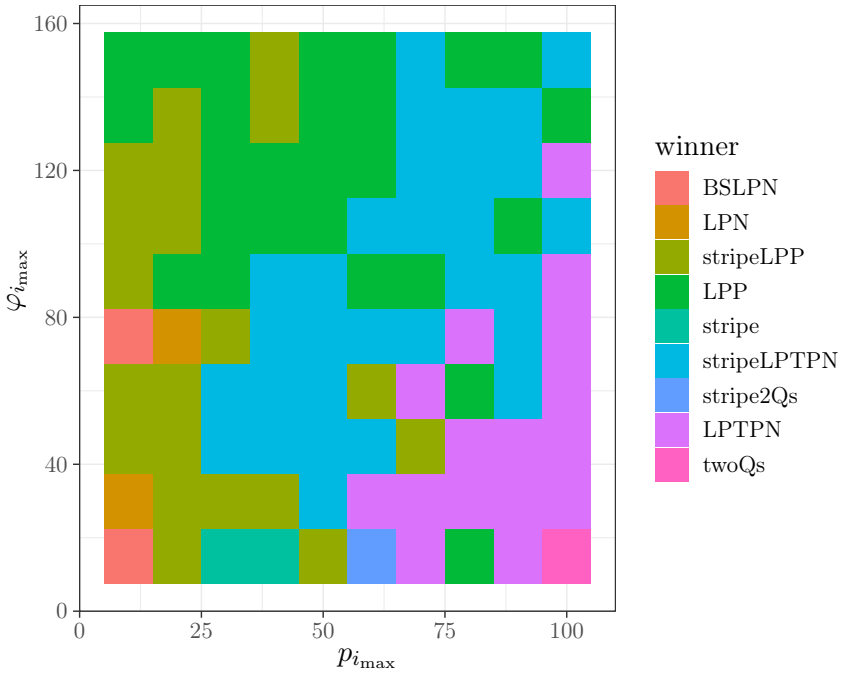
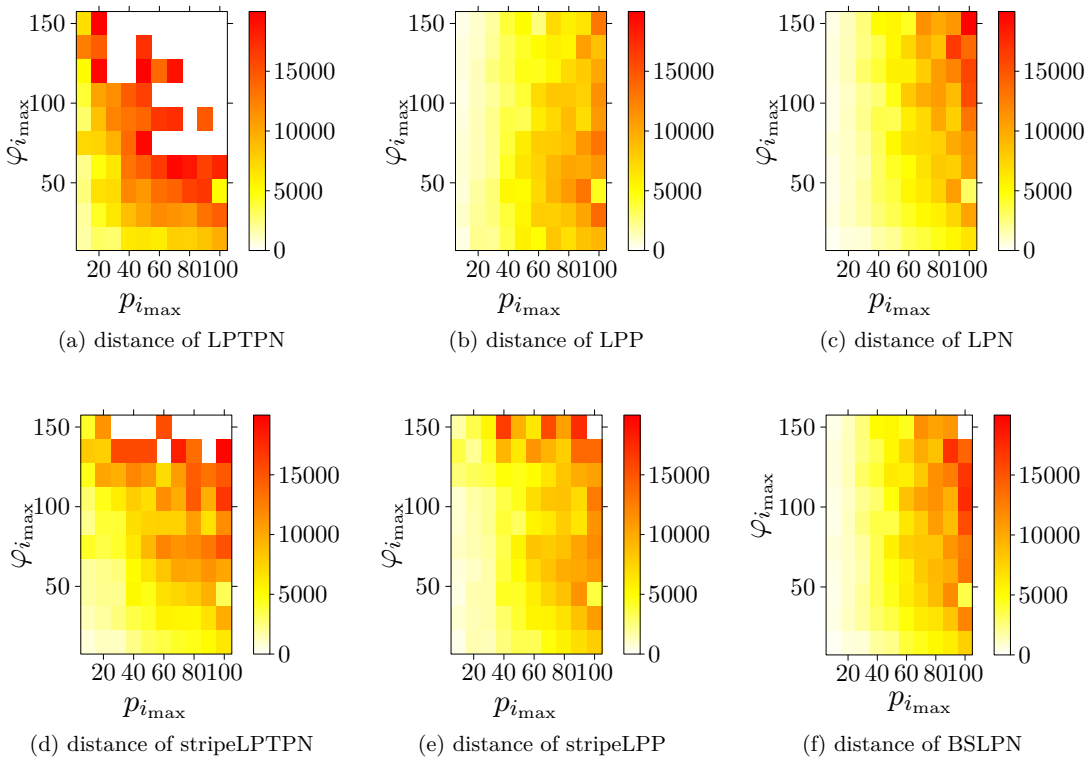


Figure 8.20 – NM, $\Phi_{max} = 700$.

Distance from GA

The previous two heat-maps presented in Figures 8.19 and 8.20 show the cases at which a heuristic gives the best average performance. In order to find out if there is a heuristic that does not necessarily give the best performance rather than giving a good performance but in a wider area of the heat-map, in each case (heat-map square) we calculate the distances between the average NM of each heuristic and the NM of the best heuristic, which is the tested GA.

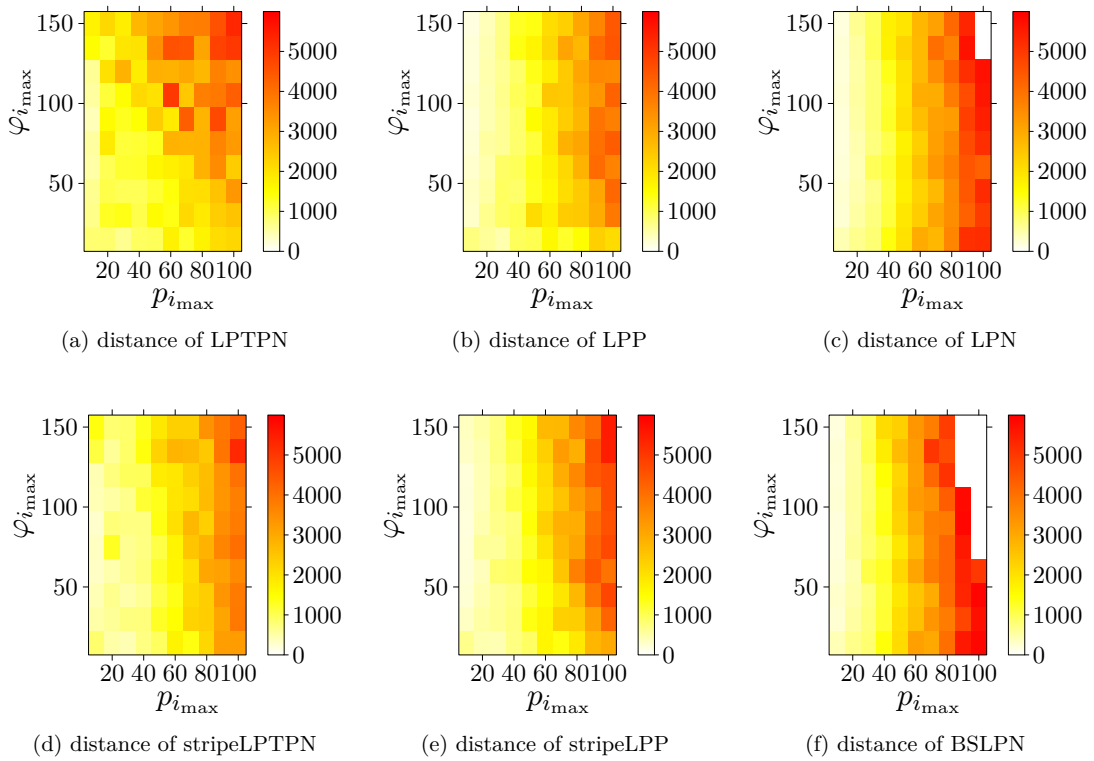
Figures 8.21 and 8.22 present the difference in average NM between several proposed heuristics and GA, with maximum available power Φ equals to 350 and 700 Watts respectively. We notice that in some cases, the GA gives up to 60000% better solutions than other tested heuristics when the power constraint is the strictest ($\Phi_{max} = 350$) and it gives up to 9000% better solutions than other tested heuristics when the power constraint is less strict ($\Phi_{max} = 700$). We point out that in both figures, the white areas represent values that are out of the unified scale illustrated to the right of each plot, they thus express the worst performance.



(a) distance of LPTPN (b) distance of LPP (c) distance of LPN

(d) distance of stripeLPTPN (e) distance of stripeLPP (f) distance of BSLPN

Figure 8.21 – Distance from OX unified scale $\Phi_{max} = 350$ unified.



(a) distance of LPTPN (b) distance of LPP (c) distance of LPN

(d) distance of stripeLPTPN (e) distance of stripeLPP (f) distance of BSLPN

Figure 8.22 – Distance from OX unified scale $\Phi_{max} = 700$ unified.

8.1.3 Summary

In this experiment, we showed that, as expected, the performance of the proposed heuristics depends on how strict the power constraint is, in other words, how high the green power availability level is in relation to the power consumption of tasks. LPT based heuristics give good performance when the problem is closer to the classic makespan optimization problem without a power constraint, and it is no surprise that heuristics that prioritize the tasks that are harder to schedule give the best performance when the power envelope implies a strict power constraint.

We noticed from Figure 8.3 that the order of the task list is more important than the used heuristic due to the color grouping of lines of different types, on the other hand, the lines are grouped based on their type rather than color in Figure 8.7 which indicates that the order of the task list has low effect on the heuristics calculation times. We point out that list based heuristics are faster than Stripe and binary search based ones, however, deploying the more time complex binary search and Stripe heuristics can give better performance than list heuristics when the same sorting policy is used.

As for the *PERFLOW* results, once again, SPT provides the best performance in all the tested cases as expected. We did not modify GA to consider the flowtime objective here as we already established in previous experiments that it can optimize the flowtime objective in comparison to

other tested heuristics.

We finally concluded that when the available power level is not high enough to run all the machines of a platform simultaneously, deploying a good on/off mechanism does not only save static energy consumption, but also can lead to better makespan due to less rebooting delays.

8.2 Experiments synthesis

In both the multi-core and multi-machine experiments we have tested several scheduling heuristics to solve the problem of scheduling independent sequential tasks on a parallel computational platform with variable power constraint. We deploy several data generation methods with different range values for both tasks and power envelope generation in order to cover a wide range of different instances of the problem. In the multi-core experiment we were able to produce two methods to measure the distance from the optimal solution. In the multi-machine platform we develop the scheduler to control the process of switching the machines on and off taking into account both time and power costs that corresponds to this process.

Our results show that the order of the task list has a great effect on the quality of the output schedule, and that choosing the right order depends on the characteristics of both the submitted task list and the power envelope.

For the flowtime minimization objective, using a traditional solution such as SPT outperforms all non GA solutions. The tested configurations of GA however are able to optimize both makespan and flowtime objectives and provide higher quality solutions than the other proposed heuristics.

For the makespan minimization objective, using a traditional solution such as LPT is only possible when the power production if sufficient to make the power constraint negligible, however, when the power production level is low in relation to the average power consumption of tasks, more concurrence for power is present between tasks, and the tasks power consumption must be taken into account when sorting the task list. The heuristics that prioritizes the tasks that are harder to schedule regardless of their characteristics (whether they have long processing times or high power consumption) outperforms the rest of the other list based heuristics when the power constraint is most strict. Furthermore, GA can be used to sort the task list in a way that yields a shorter makespan than all other sorting tested policies, we even show in some specials cases that GA produces schedules that are closer to the optimal solution than the other heuristics .

In general, most of the tested heuristics were able to produce shorter schedules in comparison to the schedules found when leaving the task list in its original random order, which proves the necessity for smart workload management in order to optimize the performance of a parallel computational platform under variable power constraint.

Part IV

Conclusion and perspectives

Chapter 9

Conclusion

The growing energy consumption of large scale ICT structures and the carbon footprint corresponding to this consumption underline the necessity for green computing. Many academic efforts tackle this problem by presenting solution on both the system's hardware and software levels, the proposed solutions include reducing the system's energy consumption or reducing its carbon footprint directly by using renewable energy sources.

Within the DECALCO and DATAZERO projects, this work addresses the optimization problem of task scheduling on parallel computational platforms powered solely by renewable energy sources. The considered objectives are minimizing the total execution time and minimizing the mean wait time. The motive behind this work is to reduce the environmental impact of HPC systems.

This work answers the following questions:

- **Q**: How to run tasks on a parallel computational platform powered by green sources?

  A: Deploying variable power constrained scheduling algorithms.

- **Q:** What is the complexity of the problem of scheduling tasks on an HPC system under limited power constraint?

  A: We conducted a theoretical complexity study on different scheduling problems that express different scenarios of the problem, and we showed that adding a variable power constraint to the scheduling problem makes the optimization problem NP-Hard.

- **Q:** How to take the limited power constraint into account when scheduling tasks on an HPC system?

  A: Several power aware scheduling heuristics were proposed in this work to solve this optimization problem. An experimental evaluation was conducted to compare the performance of the proposed heuristics. To carry out these experiments we developed a simulator script which offers a variety of experimental setups. The heuristics were tested over two different computational platform models, namely the multi-core platform and the multi-machine platform. The $PlaceTaskMachine$ function, which represents the executer part of the multi-machine platform's scheduler, manages the process of switching machines on and off. This presents a great usefulness in cases where the power production level can be not high enough to switch all the platform's machines on, which is a possible scenario in parallel computational platforms powered by a variable limited power supply.

- **Q:** What algorithm is efficient to generate a schedule for running tasks on an HPC system under limited power constraint?

  A: A well configured genetic algorithm (GA) can outperform all the other tested algorithms regarding both the makespan and flowtime minimization objectives, however list scheduling algorithms provide much faster schedules than GA (17800 to 20000 times faster in some cases), while they generate schedules never worse than 5 to 10% further from optimal than GA.

9.1 Evaluation

Regarding the makespan minimization objective:

Our results showed that deploying the right sorting policy in list algorithm can provide a good schedule in a fraction of the second. Which sorting policy is best to use depends on the characteristics of the submitted set of tasks and the power envelope. Furthermore, the tested configurations of Genetic Algorithm (GA) showed that the choice of the selection, mutation and crossover genetic operators affects the performance of GA in this problem. However, generally, all the tested GAs outperformed the other tested heuristics and they found solutions that are closer to the optimal solution, however, GA presents a performance/time trade-off between the quality of the schedule and the time needed to find it. As for the list based heuristics, both the task processing times and the task power consumption should be taken into account when sorting the task list, the weight of each of these values on the schedule depends on the characteristics of the tested data set, however, when the power constraint is most strict, the heuristics that prioritizes the tasks that are harder to schedule,regardless of whether they have long processing times or high power consumption, outperform the rest of the other list based heuristics.

Regarding the flowtime minimization objective, the solution of the classic version of this problem (SPT) outperforms all non GA solutions. However, a GA configuration that is designed to optimize the flowtime is able provide higher quality solutions than the other proposed heuristics.

9.2 Perspectives

This work is a first attempt towards the optimization of task scheduling in green HPC systems, and it can be carried on in future work addressing the following aspects:

- List algorithm can be further developed to make it provide good schedules regardless of the characteristics of the tasks set, possible solutions. Similarly to the concept of the twoQs heuristic, 3Qs, 5Qs.. etc., can be tested. Another possible solution is to sequentially try all the proposed sorting orders and then chose the order that yields the best schedule (which is feasible due to the short computation time of list algorithm).

- Machine learning can be used to train the scheduler with datasets (task lists and power envelops) whose characteristics are known and see which sorting policy is the best for each group of datasets that have similar characteristics. Then, when a new dataset is submitted to the scheduler, it should be able to compare it to the closest group of datasets that were stored during the training and use the sorting policy that gave the best schedule for that group.

- Advances in weather forecast technologies can help make our model more reliable.

- Further development in GA might make it faster, either by code optimization or by redesigning a faster GA.
- Investigating the use of batteries or other energy storage devices to address the variability to renewable power production.
- The task model and the machine model can be expanded to represent more realistic HPC systems, such as parallel tasks, malleable tasks, moldable tasks and DAGS in which case tasks have communication and precedence relations between them, which implies taking the communication cost and delay into consideration.
- Producing realistic HPC workload traces. Since we were not able to find real world traces that provide power consumption information about HPC tasks, we suggest measuring the power consumption of different types of HPC application. The first step of this objective is already done by installing the power measurement tool OmegaWatt wireless logger on one of Mesocentre's computational nodes, and as far as we know it might provide the first real world traces that provide power consumption information about HPC tasks.
- More detailed theoretical study might be able to estimate the optimal solution in the multi-machine model where we could not calculate the optimal solution because of the on/off process.
- Theoretical analysis using approximation schemes or linear programming can be explored to solve this problem.

9.3 Publications

9.3.1 Published or Accepted

International conferences

- Scheduling Independent Tasks in Parallel under Power Constraints.

  46th International Conference on Parallel Processing (ICPP 2017) - Bristol, United Kingdom (2017, Pages 543 - 552).

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

- Assessing the Use of Genetic Algorithms to Schedule Independent Tasks Under Power Constraints.

  16th International Conference on High Performance Computing & Simulation (HPCS 2018) - Orleans , France (2018, Pages 252 - 259).

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

- Green power constrained scheduling for sequential independent tasks on identical parallel machines.

  The 17th IEEE International Symposium on Parallel and Distributed Processing with Applications (ISPA 2019) – Xiamen, China (accepted).

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

French conferences

- Ordonnancement de taches independantes sur machine parallele sous contraintes d'energie renouvelable.

  Compas 2018 - Toulouse, France

Workshops

- New Challenges in Scheduling Theory.

  April , 2018 - Aussois, France.

- Green Days2018

  July, 2018 - Toulouse, France.

9.3.2 Under review

Journal Publications

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

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

  Journal of scheduling - (submitted).

- Survey on energy management in Green Datacenters with Renewable Energy Supply.

  (collective work within the ANR DATAZERO project)

  ACM Computing Surveys - (submitted).

Appendix

9.A Useful definitions

Some of the terms that are mentioned in the state of the art are defined here in order to facilitate the process of finding their definition for the reader. We point out that the following definitions are taken form the cited sources without personal contribution or modification from the authors of this work.

9.A.1 PUE

In most HPC systems, large percentage of the energy consumption is related to supporting systems that keep the processing units running, such as the cooling system, the power transfer system and the network. Since the purpose of HPC systems is to perform computing tasks. The energy consumption due to the system's components other than the actual processing units (CPUs and GPUs) is considered as a waste of energy. A data center that would perform the same amount of computation using less energy is considered to be more energy efficient. Many metrics are used to measure the energy efficiency of a data center. Power usage effectiveness (PUE) for example is one of the most used metrics [34].

$$PUE = Total power consumption / IT power consumption$$

. That the ideal PUE therefore equals to 1, in the theoretical case where the entire power consumption is due to its IT components. However, PUE only measures the efficiency of the building infrastructure supporting a given data center and indicates nothing about the efficiency of the IT equipment itself. Several efforts are proposing alternative metrics that can link the amount of computations a data center performs to its energy use, in order to help the industry to better understand where there are opportunities to reduce energy consumption. Such metrics include Green Grid's Data Center Productivity (DCP) and Data Center energy Productivity (DCeP), the Uptime Institute and McKinsey's Corporate Average Data center Efficiency (CADE), and JouleX's Performance per Watt (PPW) [113]. Nevertheless, PUE is still so far the dominant metric observed in the data center industry.

9.A.2 Time/cost trade-off problem (TCTP)

The time/cost trade-off problem is a project scheduling problem that concerns reducing the duration of some activities by allocating more resources to them. Which results in higher direct activity costs and shorter project durations. The question in this problem is to decide which activity's duration

to should be modified to shorten the total project duration when the duration of the project exceeds its predetermined limit? while aiming to minimize the additional cost [75].

9.A.3 Payment scheduling problem (PSP)

The payment scheduling problem is a project scheduling problem that takes the cash inflows and outflows throughout the life of a project into consideration. And the project manager schedules the project activities in such a way that the net instantaneous value of the cash flows is maximized. Given a project described by a network, the payment scheduling problem involves identifying a schedule that maximizes the instantaneous value of all transactions [75].

9.A.4 Partially elastic schedule

A partially elastic schedule achieves the following condition: Let pes(t, i) be an integer function such that: $\forall i, \forall t \notin [r_i, d_i], pes(t, i) = 0$. In other words, the execution of a partially elastic task/activity can be shifted but within the range between its release date r_i and its deadline d_i [19].

9.A.5 Energetic reasoning

The required energy consumption of a partially elastic activity A_i over an interval [t1, t2] $W_{PE}(Ai, t1, t2)$ is defined as follows: $W_{PE}(Ai, t1, t2) = c_i \times \max(0, p_i - \max(0, t1 - r_i) - \max(0, d_i - t2))$. Where c_i is the resource requirement of task/activity A_i, p_i is its processing time and r_i and d_i are its release date and its deadline respectively.

The last relation can be explained as the following: The minimum energy consumption of a task over a range [t1, t2] equals to its resource requirement times the remaining time units in which the execution of the task takes place within the range [t1, t2] after executing as much of the task as possible outside of the range [t1, t2], yet between its release date and its deadline. i,e., in (t1 - r_i) and (d_i - t2). In other words, the part of the task that is executed within [t1, t2] is the part that could not have been shifted left not right out of the range (advanced or delayed). The objective of this energetic reasoning is thus to advance or delay as much as possible of the task's execution in order to minimize its power consumption within a given time range [19].

9.A.6 left-shift/right-shift

Given an activity A_i and a time interval [t1, t2], the left-shift/right-shift required energy consumption of A_i over [t1, t2] is its resource requirement c_i times the minimum of the three following durations.

- t2 - t1 , the length of the interval.
- The number of time units within [t1, t2] used to execute A_i if it was left-shifted (scheduled as soon as possible between its release date and its deadline).
- The number of time units within [t1, t2] used to execute A_i if it was right-shifted (scheduled as late as possible between its release date and its deadline).

In our problem, we do not want to delay (right-shift) a task because our objective is to minimize the makespan, while in this model tasks have deadlines, and they can right-shift a task as long as

it does not violate its deadline. However, in work where left-shift/right-shift is used they also aim to minimize the total execution duration. Therefore, a right-shift might be usable in our problem as well [19].

9.A.7 Multiple mode

An activity in standard RCPSP can be executed in only one way which is determined by a fixed duration and fixed resource requirements. In project scheduling problems with multiple modes the activity could have several alternatives or modes in which it can be performed. Each mode represents a combination of a duration and resource requests that to execute the activity [70].

List of Figures

2.1 `Powerstat` execution example 10

4.1 Illustrating example for the optimization problem: A set of tasks to be scheduled in the given power envelop. 32
4.2 Example of a schedule 32
4.3 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 could be scheduled when its power need is φ_i on a one machine platform. 33

5.1 $p_i = p$ and $\varphi_i \leq \Phi_x$ instance 37
5.2 An instance of 3-PARTITION 38
5.3 Illustrating example for the LWRT algorithm, T_2 is not the LWRT task for interval Δ_3, T_4 must be run here. 40
5.4 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. 40

6.1 Float data sets results 46
6.2 Integer data sets results 47
6.3 Illustrating example of a chromosome and the corresponding schedule 55
6.4 `mutation(C)` 57
6.5 `chunkMutation(C)` 57
6.6 `OnePointCrossOver(C1, C2)` 58
6.7 `OrderCrossOver(C1, C2)` 59
6.8 `MiddleCrossOver(C1, C2)` 60
6.9 Best fit BS 61
6.10 OPOBS 1st step 64
6.11 OPOBS 2nd step 64
6.12 OPOBS 3rd step 65
6.13 Stripe heuristic 65

7.1 A bell shape interval list generation. 70
7.2 Realistic power envelope $\Phi_{max} = 350$. 70
7.3 Simulator 73
7.4 Turtle example 73
7.5 Heat map of the best average *PERMAK*, $p_{max} \in [10 \rightarrow 100]$, $\varphi_{max} \in [4 \rightarrow 40]$. 75
7.6 Distance from the best *PERMAK*. 76
7.7 Standard deviation of *PERMAK* for $p_{max} = 100$ and $\varphi_{max} = 20$. 77

7.8 Distance from best PERFLOW. 78
7.9 standard deviation of *PERFLOW* for $p_{max} = 100$ and $\varphi_{max} = 20$. 79
7.10 Computation time of the heuristics for $\varphi_{max} = 20$. 79
7.11 Best heuristic regarding *PERMAK* 81
7.12 Distance of 1pX-W, OX-W, MX-W, and noX-W from the best algorithm 82
7.13 Distance of list algorithms from the best algorithm 83
7.14 Average NM. 85
7.15 Best average NM for experiment. 86
7.16 Number of times each algorithm finds the best $C_{\max}$ 88
7.17 *PERFLOW* distance from the average *PERFLOW* of SPT 89
7.18 Average *PERFLOW* distance Between SPT and GA. 90
7.19 Distance from the optimal. 91
7.20 Best average NM. 92
7.21 Distance from the optimal. 93

8.1 Average NM, $\Phi_{max} = 350$ (with Random and LPN). 96
8.2 Average NM, $\Phi_{max} = 3500$ (with Random and LPN). 97
8.3 zoom in Average NM, $\Phi_{max} = 350$ (without SPT, Random and LPN). 98
8.4 zoom in Average NM, $\Phi_{max} = 700$ (without SPT, Random and LPN). 98
8.5 zoom in Average NM, $\Phi_{max} = 1750$ (without SPT, Random and LPN). 99
8.6 zoom in Average NM, $\Phi_{max} = 3500$ (without SPT, Random and LPN). 99
8.7 Computation time, $\Phi_{max} = 350$. 101
8.8 Computation time, $\Phi_{max} = 700$. 102
8.9 Computation time, $\Phi_{max} = 1750$. 102
8.10 Computation time, $\Phi_{max} = 3500$. 103
8.11 on/off, $\Phi_{max} = 350$. 104
8.12 on/off, $\Phi_{max} = 700$. 104
8.13 on/off, $\Phi_{max} = 1750$. 105
8.14 on/off, $\Phi_{max} = 3500$. 105
8.15 Energy waste, $\Phi_{max} = 350$. 106
8.16 Energy waste, $\Phi_{max} = 700$. 107
8.17 Energy waste, $\Phi_{max} = 1750$. 107
8.18 Energy waste, $\Phi_{max} = 3500$. 108
8.19 NM, $\Phi_{max} = 350$. 110
8.20 NM, $\Phi_{max} = 700$. 111
8.21 Distance from OX unified scale $\Phi_{max} = 350$ unified. 112
8.22 Distance from OX unified scale $\Phi_{max} = 700$ unified. 113

List of Tables

4.1 Summary of the notations 34

5.1 Complexity results without green power constraint 43
5.2 Complexity results with green power constraint 43

7.1 Computation time 83
7.2 Average computation times (sec). 87
7.3 Average computation times (sec). 91

8.1 Computation time $\Phi_{max} = 350$ (with GA) 100
8.2 Computation time $\Phi_{max} = 3500$ (with GA) 100

Bibliography

[1] ANR DATAZERO. https://www.irit.fr/ Georges.Da-Costa/Yerevan2017DataZero.pdf. 15

[2] Energie consommée par les data centers. https://www.planetoscope.com/electronique/230-energie-consommee-par-les-data-centers.html. 4, 14

[3] Power, pollution and the internet. https://www.nytimes.com. 13

[4] powerstat: a tool to measure power consumption. https://github.com/ColinIanKing/powerstat. 9

[5] Powertop primer. https://software.intel.com/en-us/articles/powertop-primer-1. 11

[6] Supercalculateurs bullsequana x1000 les supercalculateurs de classe exascale. https://atos.net/fr/produits/calcul-haute-performance-hpc/supercalculateurs-bull-sequana-x/gamme-bullsequana-x1000. 15

[7] What is immersion cooling? https://submer.com/blog/what-is-immersion-cooling. 15

[8] Dr Agarwal, Saloni Jain, et al. Efficient optimal algorithm of task scheduling in cloud computing environment. *arXiv preprint arXiv:1404.2076*, 2014. 18

[9] Faraz Ahmad and TN Vijaykumar. Joint optimization of idle and cooling power in data centers while maintaining response time. In *ACM Sigplan Notices*, volume 45, pages 243–256. ACM, 2010. 22

[10] Imtiaz Ahmad and Muhammad K Dhodhi. Multiprocessor scheduling in a genetic paradigm. *Parallel Computing*, 22(3):395–406, 1996. 18

[11] Ishfaq Ahmad and Yu-Kwong Kwok. On exploiting task duplication in parallel program scheduling. *Parallel and Distributed Systems, IEEE Transactions on*, 9(9):872–892, 1998. 18

[12] Abdulla M Al-Qawasmeh, Sudeep Pasricha, Anthony A Maciejewski, and Howard Jay Siegel. Power and thermal-aware workload allocation in heterogeneous data centers. *IEEE Transactions on Computers*, 64(2):477–491, 2015. 22, 23

[13] Ittai Anati, David Blythe, Jack Doweck, Hong Jiang, Wen-fu Kao, Julius Mandelblat, Lihu Rappoport, Efraim Rotem, and Ahmad Yasin. Inside 6th gen intel® core: New microarchitecture code named skylake. In *2016 IEEE Hot Chips 28 Symposium (HCS)*, pages 1–39. IEEE, 2016. 13

[14] Anders SG Andrae and Tomas Edler. On global electricity usage of communication technology: trends to 2030. *Challenges*, 6(1):117–157, 2015. 4, 8, 14

[15] 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. 15

[16] Christian Artigues, Sophie Demassey, and Emmanuel Neron. *Resource-constrained project scheduling*. Wiley Online Library, 2008. 18

[17] Christian Artigues and Pierre Lopez. Energetic reasoning for energy-constrained scheduling with a continuous resource. *Journal of Scheduling*, 18(3):225–241, 2015. 20

[18] Guillaume Aupy, Anne Benoit, Paul Renaud-Goud, and Yves Robert. Energy-aware algorithms for task graph scheduling, replica placement and checkpoint strategies. In *Handbook on Data Centers*, pages 37–80. Springer, 2015. 23

[19] Ph Baptiste, Claude Le Pape, and Wim Nuijten. Satisfiability tests and time-bound adjustmentsfor cumulative scheduling problems. *Annals of Operations research*, 92:305–333, 1999. 121, 122

[20] Luiz André Barroso and Urs Hölzle. The case for energy-proportional computing. 2007. 13

[21] Shajulin Benedict. Energy-aware performance analysis methodologies for hpc architectures-an exploratory study, vol. 35, no. 6, j. of n. and computer applications, 2012. 9

[22] Anne Benoit, Loris Marchal, Jean-Francois Pineau, Yves Robert, and Frederic Vivien. Scheduling concurrent bag-of-tasks applications on heterogeneous platforms. *Computers, IEEE Transactions on*, 59(2):202–217, 2010. 18

[23] Meenakshi Bist, Manoj Wariya, and Amit Agarwal. Comparing delta, open stack and xen cloud platforms: A survey on open source iaas. In *2013 3rd IEEE International Advance Computing Conference (IACC)*, pages 96–100. IEEE, 2013. 22

[24] Tom Brey and Larry Lamers. Using virtualization to improve data center efficiency. *the green grid, whitepaper*, 19, 2009. 18, 22

[25] Peter Brucker. *Scheduling Algorithms*. Springer Heidelberg, 2007. 18

[26] Peter Brucker, Andreas Drexl, Rolf Möhring, Klaus Neumann, and Erwin Pesch. Resource-constrained project scheduling: Notation, classification, models, and methods. *European journal of operational research*, 112(1):3–41, 1999. 19

[27] David P. Bunde. Power-aware scheduling for makespan and flow. *Journal of Scheduling*, 12(5):489–500, Oct 2009. 21

[28] Andrea Castagnetti, Cecile Belleudy, Sebastien Bilavarn, and Michel Auguin. Power consumption modeling for dvfs exploitation. In *2010 13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools*, pages 579–586. IEEE, 2010. 11

[29] 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. 25

[30] Jeffrey S Chase, Darrell C Anderson, Prachi N Thakar, Amin M Vahdat, and Ronald P Doyle. Managing energy and server resources in hosting centers. In *ACM SIGOPS Operating Systems Review*, volume 35, pages 103–116. ACM, 2001. 22

[31] Gong Chen, Wenbo He, Jie Liu, Suman Nath, Leonidas Rigas, Lin Xiao, and Feng Zhao. Energy-aware server provisioning and load dispatching for connection-intensive internet services. In *NSDI*, volume 8, pages 337–350, 2008. 13, 22

[32] Richard Walter Conway, William L Maxwell, and Louis W Miller. *Theory of scheduling*. Courier Corporation, 2003. 17

[33] Antonio Corradi, Mario Fanelli, and Luca Foschini. Vm consolidation: A real case based on openstack cloud. *Future Generation Computer Systems*, 32:118–127, 2014. 22

[34] Tugrul Daim, Jay Justice, Mark Krampits, Matthew Letts, Ganesh Subramanian, and Mukundan Thirumalai. Data center metrics: An energy efficiency model for information technology managers. *Management of Environmental Quality: An International Journal*, 20(6):712–731, 2009. 120

[35] Mohamed I Dessouky, Ben J Lageweg, Jan Karel Lenstra, and Steef L van de Velde. Scheduling identical jobs on uniform parallel machines. *Statistica Neerlandica*, 44(3):115–123, 1990. 35

[36] Cesar O Diaz, Mateusz Guzek, Johnatan E Pecero, Gregoire Danoy, Pascal Bouvry, and Samee U Khan. Energy-aware fast scheduling heuristics in heterogeneous computing systems. In *High Performance Computing and Simulation (HPCS), 2011 International Conference on*, pages 478–484. IEEE, 2011. 21

[37] Thanh Do, Suhib Rawshdeh, and Weisong Shi. ptop: A process-level power profiling tool, 2009. 11

[38] Bernabé Dorronsoro, Sergio Nesmachnow, Javid Taheri, Albert Y Zomaya, El-Ghazali Talbi, and Pascal Bouvry. A hierarchical approach for energy-efficient scheduling of large workloads in multicore distributed systems. *Sust. Computing: Informatics and Systems*, 4(4):252–261, 2014. 21

[39] Ronald P Doyle, Jeffrey S Chase, Omer M Asad, Wei Jin, and Amin Vahdat. Model-based resource provisioning in a web service utility. In *USENIX Symposium on Internet Technologies and Systems*, volume 4, pages 5–5, 2003. 22

[40] Maciej Drozdowski. *Scheduling for parallel processing*. Springer, 2009. 18

[41] Pierre-François Dutot, Lionel Eyraud, Grégory Mounié, and Denis Trystram. Bi-criteria algorithm for scheduling jobs on cluster platforms. In *Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 125–132. ACM, 2004. 18

[42] 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. 22, 23, 94

[43] Truong Vinh Truong Duy, Yukinori Sato, and Yasushi Inoguchi. Performance evaluation of a green scheduling algorithm for energy savings in cloud computing. In *Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, pages 1–8. IEEE, 2010. 21

[44] Ryan Elmore, Kenny Gruchalla, Caleb Phillips, Avi Purkayastha, and Nick Wunder. Analysis of application power and schedule composition in a high performance computing environment. Technical report, National Renewable Energy Lab.(NREL), Golden, CO (United States), 2016. 11

[45] Xiaobo Fan, Wolf-Dietrich Weber, and Luiz Andre Barroso. Power provisioning for a warehouse-sized computer. In *ACM SIGARCH Computer Architecture News*, volume 35, pages 13–23. ACM, 2007. 11, 12, 13

[46] Dror G Feitelson and Bill Nitzberg. Job characteristics of a production parallel scientific workload on the nasa ames ipsc/860. In *Job Scheduling Strategies for Parallel Processing*, pages 337–360. Springer, 1995. 18

[47] Dror G Feitelson and Larry Rudolph. Toward convergence in job schedulers for parallel supercomputers. In *Job Scheduling Strategies for Parallel Processing*, pages 1–26. Springer, 1996. 18

[48] Dror G Feitelson and Larry Rudolph. Metrics and benchmarking for parallel job scheduling. In *Job Scheduling Strategies for Parallel Processing*, pages 1–24. Springer, 1998. 18

[49] Dror G Feitelson, Larry Rudolph, Uwe Schwiegelshohn, Kenneth C Sevcik, and Parkson Wong. Theory and practice in parallel job scheduling. In *Job scheduling strategies for parallel processing*, pages 1–34. Springer, 1997. 18

[50] Armando Fox, Rean Griffith, Anthony Joseph, Randy Katz, Andrew Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, and Ion Stoica. Above the clouds: A berkeley view of cloud computing. *Dept. Electrical Eng. and Comput. Sciences, University of California, Berkeley, Rep. UCB/EECS*, 28(13):2009, 2009. 13

[51] Bin Fu, Yumei Huo, and Hairong Zhao. Approximation schemes for parallel machine scheduling with availability constraints. *Discrete Applied Mathematics*, 159(15):1555–1565, 2011. 19

[52] Anshul Gandhi, Mor Harchol-Balter, Ram Raghunathan, and Michael A Kozuch. Autoscale: Dynamic, robust capacity management for multi-tier data centers. *ACM Transactions on Computer Systems (TOCS)*, 30(4):14, 2012. 13, 18

[53] M. R. Garey and D. S. Johnson. *Computers and Intractability, a Guide to the Theory of NP-Completeness*. W.H. Freeman & Co, 1979. 37

[54] Michael R Garey and David S Johnson. "strong"np-completeness results: Motivation, examples, and implications. *Journal of the ACM (JACM)*, 25(3):499–508, 1978. 18, 35

[55] Saurabh Kumar Garg, Chee Shin Yeo, Arun Anandasivam, and Rajkumar Buyya. Environment-conscious scheduling of HPC applications on distributed cloud-oriented data centers. *Journal of Parallel and Distributed Computing*, 71(6):732 – 749, 2011. 12, 23

[56] Rong Ge, Xizhou Feng, and Kirk W Cameron. Performance-constrained distributed dvs scheduling for scientific applications on power-aware clusters. In *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 34. IEEE Computer Society, 2005. 12, 23

[57] 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. 29

[58] 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. 25

[59] Íñigo Goiri, William Katsak, Kien Le, Thu D Nguyen, and Ricardo Bianchini. Parasol and greenswitch: Managing datacenters powered by renewable energy. In *ACM SIGARCH Computer Architecture News*, volume 41, pages 51–64. ACM, 2013. 15

[60] Inigo Goiri, Kien Le, Md E. Haque, Ryan Beauchea, Thu D. Nguyen, Jordi Guitart, Jordi Torres, and Ricardo Bianchini. Greenslot: Scheduling energy consumption in green datacenters. *2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC 2011)*, 00:1–11, 2012. 25

[61] 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. 17

[62] 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. 15

[63] Laura Grit, David Irwin, Aydan Yumerefendi, and Jeff Chase. Virtual machine hosting for networked clusters: Building the foundations for autonomic orchestration. In *Proceedings of the 2nd International Workshop on Virtualization Technology in Distributed Computing*, page 7. IEEE Computer Society, 2006. 22

[64] Part Guide. Intel® 64 and ia-32 architectures software developer's manual. *Volume 3B: System programming Guide, Part*, 2, 2011. 11

[65] Chamara Gunaratne, Ken Christensen, and Bruce Nordman. Managing energy consumption costs in desktop pcs and lan switches with proxying, split tcp connections, and scaling of link speed. *International Journal of Network Management*, 15(5):297–310, 2005. 9

[66] Daniel Hackenberg, Robert Schöne, Thomas Ilsche, Daniel Molka, Joseph Schuchart, and Robin Geyer. An energy efficiency feature survey of the intel haswell processor. In *IEEE IPDPS Workshop*, 2015. 11, 12

[67] 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. 12

[68] M Haddad, J.-M. Nicod, and M.-C. Péra. Hydrogen infrastructure: data-center supply-refueling station synergy. In *14th IEEE VPPC 2017*, page 6, Belfort, France, dec 2017. 70

[69] 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. 70

[70] Sönke Hartmann and Dirk Briskorn. A survey of variants and extensions of the resource-constrained project scheduling problem. *European Journal of operational research*, 207(1):1–14, 2010. 19, 122

[71] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011. 11

[72] Dorit S. Hochbaum and David B. Shmoys. Using dual approximation algorithms for scheduling problems theoretical and practical results. *J. ACM*, 34(1):144–162, January 1987. 61

[73] 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. 11, 12

[74] Edwin SH Hou, Nirwan Ansari, and Hong Ren. A genetic algorithm for multiprocessor scheduling. *IEEE Transactions on parallel and distributed systems*, 5(2):113–120, 1994. 18

[75] Oya Icmeli, S Selcuk Erenguc, and Christopher J Zappe. Project scheduling problems: a survey. *International Journal of Operations & Production Management*, 13(11):80–91, 1993. 19, 121

[76] E Ilavarasan and P Thambidurai. Low complexity performance effective task scheduling algorithm for heterogeneous computing environments. *Journal of Computer sciences*, 3(2):94–103, 2007. 18

[77] Santiago Iturriaga, Sergio Nesmachnow, Bernabé Dorronsorro, and Pascal Bouvry. Energy efficient scheduling in heterogeneous systems with a parallel multiobjective local search. *Computing and Informatics*, 32(2):273–294, 2013. 21

[78] Santiago Iturriaga, Sergio Nesmachnow, Andrei Tchernykh, and Bernabé Dorronsoro. Multiobjective workflow scheduling in a federation of heterogeneous green-powered data centers. In *Cluster, Cloud and Grid Computing (CCGrid), 2016 16th IEEE/ACM International Symposium on*, pages 596–599. IEEE, 2016. 18, 24

[79] 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. 21

[80] 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. 9

[81] Yacine Kessaci, Mohand Mezmaz, Nouredine Melab, El-Ghazali Talbi, and Daniel Tuyttens. Parallel evolutionary algorithms for energy aware scheduling. In *Intelligent Decision Systems in Large-Scale Distributed Environments*, pages 75–100. Springer, 2011. 23

[82] Kashif Nizam Khan, Zhonghong Ou, Mikael Hirki, Jukka K Nurminen, and Tapio Niemi. How much power does your server consume? Estimating wall socket power using RAPL measurements. *Computer Science-Research and Development*, 31(4):207–214, 2016. 11

[83] Kyong Hoon Kim, Anton Beloglazov, and Rajkumar Buyya. Power-aware provisioning of virtual machines for real-time cloud services. *Concurr. Comput. : Pract. Exper.*, 23(13):1491–1505, September 2011. 12

[84] Jonathan Koomey, Stephen Berard, Marla Sanchez, and Henry Wong. Implications of historical trends in the electrical efficiency of computing. *IEEE Annals of the History of Computing*, 33(3):46–54, 2011. 14

[85] 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. 21

[86] Eugene L Lawler, Jan Karel Lenstra, Alexander HG Rinnooy Kan, and David B Shmoys. Sequencing and scheduling: Algorithms and complexity. *Handbooks in operations research and management science*, 4:445–522, 1993. 19

[87] Charles Lefurgy, Xiaorui Wang, and Malcolm Ware. Power capping: a prelude to power shifting. *Cluster Computing*, 11(2):183–195, 2008. 14

[88] 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

[89] Jan Karel Lenstra, AHG Rinnooy Kan, and Peter Brucker. Complexity of machine scheduling problems. In *Annals of discrete mathematics*, volume 1, pages 343–362. Elsevier, 1977. 35

[90] Yunbo Li, Anne-Cécile Orgerie, and Jean-Marc Menaud. Balancing the use of batteries and opportunistic scheduling policies for maximizing renewable energy consumption in a cloud data center. In *PDP 2017-25th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, 2017. 15

[91] Pierre Lopez and François Roubellat. *Production scheduling*. John Wiley & Sons, 2013. 18

[92] 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. 68

[93] Jens Malmodin and Dag Lundén. The energy and carbon footprint of the global ict and e&m sectors 2010–2015. *Sustainability*, 10(9):3027, 2018. 14

[94] Toni Mastelic, Ariel Oleksiak, Holger Claussen, Ivona Brandic, Jean-Marc Pierson, and Athanasios V Vasilakos. Cloud computing: survey on energy efficiency. *ACM Computing Surveys (CSUR)*, 47(2):33, 2015. 9, 22

[95] Teena Mathew, K Chandra Sekaran, and John Jose. Study and analysis of various task scheduling algorithms in the cloud computing environment. In *Advances in Computing, Communications and Informatics (ICACCI, 2014 International Conference on*, pages 658–664. IEEE, 2014. 18

[96] Andreas Merkel and Frank Bellosa. Balancing power consumption in multiprocessor systems. In *ACM SIGOPS Operating Systems Review*, volume 40, pages 403–414. ACM, 2006. 22

[97] Lei Miao, Yong Qi, Di Hou, Chang-li Wu, and Yue-hua Dai. Energy saving task scheduling for heterogeneous cmp system based on multi-objective fuzzy genetic algorithm. In *Systems, Man and Cybernetics, 2009. SMC 2009. IEEE International Conference on*, pages 3923–3927. IEEE, 2009. 23

[98] Ramesh Mishra, Namrata Rastogi, Dakai Zhu, Daniel Mossé, and Rami Melhem. Energy aware scheduling for distributed real-time systems. In *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*, pages 9–pp. IEEE, 2003. 23

[99] Katsuto Nakajima, S. Louis Hakimi, and Jan Karel Lenstra. Complexity results for scheduling tasks in fixed intervals on two types of machines. *SIAM Journal on Computing*, 11(3):512–520, 1982. 35

[100] Benedetto Nastasi, Livio de Santoli, Angelo Albo, Daniele Bruschi, and Gianluigi Lo Basso. Res (renewable energy sources) availability assessments for eco-fuels production at local scale: Carbon avoidance costs associated to a hybrid biomass/h 2 ng-based energy scenario. *Energy Procedia*, 81:1069–1076, 2015. 15

[101] Margaux Nattaf, Christian Artigues, and Pierre Lopez. A hybrid exact method for a scheduling problem with a continuous resource and energy constraints. *Constraints*, 20(3):304–324, 2015. 20

[102] 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. 9

[103] 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. 9

[104] Pavlos Petoumenos, Lev Mukhanov, Zheng Wang, Hugh Leather, and Dimitrios S Nikolopoulos. Power capping: What works, what does not. In *2015 IEEE 21st International Conference on Parallel and Distributed Systems (ICPADS)*, pages 525–534. IEEE, 2015. 14

[105] Michael Pinedo. *Scheduling*. Springer, 2012. 17

[106] Michael L. Pinedo. *Scheduling: Theory, Algorithms, and Systems*. Springer Publishing Company, Incorporated, 3rd edition, 2008. 17

[107] Eduardo Pinheiro, Ricardo Bianchini, Enrique V Carrera, and Taliver Heath. Load balancing and unbalancing for power and performance in cluster-based systems. 2001. 22

[108] Eduardo Pinheiro, Ricardo Bianchini, Enrique V Carrera, and Taliver Heath. Dynamic cluster reconfiguration for power and performance. In *Compilers and operating systems for low power*, pages 75–93. Springer, 2003. 22

[109] Gang Quan and Xiaobo Hu. Energy efficient fixed-priority scheduling for real-time systems on variable voltage processors. In *Proceedings of the 38th annual Design Automation Conference*, pages 828–833. ACM, 2001. 23

[110] 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. 13

[111] 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. 14

[112] Sherief Reda, Ryan Cochran, and Ayse K Coskun. Adaptive power capping for servers with multithreaded workloads. *IEEE Micro*, 5(32):64–75, 2012. 14

[113] V Dinesh Reddy, Brian Setz, G Subrahmanya VRK Rao, GR Gangadharan, and Marco Aiello. Metrics for sustainable data centers. *IEEE Transactions on Sustainable Computing*, 2(3):290–303, 2017. 120

[114] Ankur Sahai. Renewable energy prediction using weather forecasts for optimal scheduling in hpc systems. *arXiv preprint arXiv:1402.6552*, 2014. 15

[115] Abusayeed Saifullah, Jing Li, Kunal Agrawal, Chenyang Lu, and Christopher Gill. Multi-core real-time scheduling for generalized parallel task models. *Real-Time Systems*, 49(4):404–435, 2013. 18

[116] Subhash Saini, Robert T Hood, Johnny Chang, and John Baron. Performance Evaluation of an Intel Haswell-and Ivy Bridge-Based Supercomputer Using Scientific and Engineering Applications. 2016. 12

[117] Panneerselvam Senthilkumar and Sockalingam Narayanan. Literature review of single machine scheduling problem with uniform parallel machines. 2010. 18

[118] Lui Sha, Tarek Abdelzaher, Karl-Erik Årzén, Anton Cervin, Theodore Baker, Alan Burns, Giorgio Buttazzo, Marco Caccamo, John Lehoczky, and Aloysius K Mok. Real time scheduling theory: A historical perspective. *Real-time systems*, 28(2-3):101–155, 2004. 18

[119] Navin Sharma, Sean Barker, David Irwin, and Prashant Shenoy. Blink: managing server clusters on intermittent power. In *ACM SIGARCH Computer Architecture News*, volume 39, pages 185–198. ACM, 2011. 15

[120] Sushant Sharma, Chung-Hsing Hsu, and Wu-chun Feng. Making a case for a green500 list. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, pages 8–pp. IEEE, 2006. 4

[121] Arman Shehabi, Sarah Smith, Dale Sartor, Richard Brown, Magnus Herrlin, Jonathan Koomey, Eric Masanet, Nathaniel Horner, Inês Azevedo, and William Lintner. United states data center energy usage report. 2016. 11

[122] 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. 21

[123] Alina Sîrbu and Ozalp Babaoglu. Predicting system-level power for a hybrid supercomputer. In *High Performance Computing & Simulation (HPCS), 2016 International Conference on*, pages 826–833. IEEE, 2016. 11

[124] Roman Słowinski. Multiobjective network scheduling with efficient use of renewable and nonrenewable resources. *European Journal of Operational Research*, 7(3):265–273, 1981. 19

[125] Shekhar Srikantaiah, Aman Kansal, and Feng Zhao. Energy aware consolidation for cloud computing. In *Proceedings of the 2008 conference on Power aware computing and systems*, volume 10, pages 1–5. San Diego, California, 2008. 21, 22

[126] Christopher Stewart and Kai Shen. Some joules are more precious than others: Managing renewable energy in the datacenter. In *Proceedings of the workshop on power aware computing and systems*, pages 15–19. IEEE, 2009. 23, 29

[127] Balaji Subramaniam and Wu-chun Feng. Towards energy-proportional computing for enterprise-class server workloads. In *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering*, pages 15–26. ACM, 2013. 11, 12, 21

[128] Andrei Tchernykh, Luz Lozano, Pascal Bouvry, Johnatan E Pecero, Uwe Schwiegelshohn, and Sergio Nesmachnow. Energy-aware online scheduling: Ensuring quality of service for iaas clouds. In *High Performance Computing & Simulation (HPCS), 2014 International Conference on*, pages 911–918. IEEE, 2014. 21

[129] Silvana Teodoro, Andriele Busatto do Carmo, Daniel Couto Adornes, and Luiz Gustavo Fernandes. A comparative study of energy-aware scheduling algorithms for computational grids. *Journal of Systems and Software*, 117:153–165, 2016. 21

[130] Haluk Topcuoglu, Salim Hariri, and Min-you Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE transactions on parallel and distributed systems*, 13(3):260–274, 2002. 18

[131] 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. 12, 22

[132] Rico Walter. *Analyzing Various Aspects of Scheduling Independent Jobs on Identical Machines*. Logos Verlag Berlin GmbH, 2010. 18

[133] Lee Wang, Howard Jay Siegel, Vwani P Roychowdhury, and Anthony A Maciejewski. Task matching and scheduling in heterogeneous computing environments using a genetic-algorithm-based approach. *Journal of parallel and distributed computing*, 47(1):8–22, 1997. 18

[134] 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. 12, 23

[135] Lizhe Wang, Gregor Von Laszewski, Jay Dayal, and Fugang Wang. Towards energy aware scheduling for precedence constrained parallel tasks in a cluster with DVFS. In *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, pages 368–377. IEEE Computer Society, 2010. 12, 23

[136] Rui Wang, Robin C Purshouse, and Peter J Fleming. Preference-inspired coevolutionary algorithms for many-objective optimization. *IEEE Transactions on Evolutionary Computation*, 17(4):474–494, 2013. 25

[137] J Weglarz. On certain models of resource allocation problems. *Kybernetes*, 9(1):61–66, 1980. 19

[138] Jan Węglarz. Project scheduling with continuously-divisible, doubly constrained resources. *Management Science*, 27(9):1040–1053, 1981. 20

[139] Mark Weiser, Brent Welch, Alan Demers, and Scott Shenker. Scheduling for reduced cpu energy. In *Mobile Computing*, pages 449–471. Springer, 1994. 21

[140] 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. 12, 23

[141] Yuming Xu, Kenli Li, Jingtong Hu, and Keqin Li. A genetic algorithm for task scheduling on heterogeneous computing systems using multiple priority queues. *Information Sciences*, 270:255–287, 2014. 18

[142] Chuan-Yue Yang, Jian-Jia Chen, Tei-Wei Kuo, and Lothar Thiele. An approximation scheme for energy-efficient scheduling of real-time tasks in heterogeneous multiprocessor systems. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 694–699. European Design and Automation Association, 2009. 21

[143] 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. 24

[144] Yanyong Zhang, Hubertus Franke, Jose Moreira, and Anand Sivasubramaniam. An integrated approach to parallel scheduling using gang-scheduling, backfilling, and migration. *IEEE Transactions on Parallel and Distributed Systems*, 14(3):236–247, 2003. 18