Facultad
de
Ciencias

Planificador inteligente consciente de la energía en sistemas HPC

(Intelligent Energy-Aware Task Scheduler for HPC Systems)

Trabajo de Fin de Grado
para acceder al

GRADO EN INGENIERÍA INFORMÁTICA

Autor: Marta López Rauhut

Director: José Luis Bosque Orero

Julio-2023

Contents

List of Figures

List of Tables

Abstract

In recent years, energy consumption has become a limiting factor in the evolution of high-performance computing (HPC) clusters in terms of environmental concern and maintenance cost. As computing power increases and workloads become more intensive owing to the rising popularity of artificial intelligence and machine learning tasks, this problem is not expected to recede in the near future. Consequently, numerous techniques are being researched as a means of tackling energy consumption in HPC systems, including energy-aware job scheduling. Since this is known to be an NP-complete problem, heuristic scheduling algorithms and, more recently, artificial intelligence have been the main focus of the investigation carried out in this area.

Job scheduling approaches to reducing energy consumption have traditionally resorted to time-related metrics as their main optimization target, seeing that execution time and energy consumption are closely related. However, difficulties arise in the face of issues such as memory contention and heterogeneous resources, giving rise to a need for more specific solutions.

This work aims to design an intelligent job scheduler for HPC systems trained using deep reinforcement learning techniques that focuses on reducing energy consumption explicitly, by leveraging the information provided by the power consumption specifications of the resources of a heterogeneous cluster, and by optimizing energy-related metrics directly instead of relying on time measurements.

Keywords: Task scheduling, Deep reinforcement learning, High-performance computing, Heterogeneous clusters, Energy consumption.

Resumen

En los últimos años, el consumo energético se ha convertido en un factor limitante en el desarrollo y evolución de los clusters de computación de alto rendimiento (HPC) en términos de su impacto ambiental y coste de mantenimiento. A medida que la potencia de cómputo de estos sistemas aumenta y la carga de trabajo a la que están sometidos se vuelve más intensa debido a la creciente popularidad de las tareas relacionadas con la inteligencia artificial y el aprendizaje automático, este problema no va a remitir en un futuro cercano. Como consecuencia de esta situación, diversas técnicas están siendo investigadas con el objetivo de reducir el consumo energético en los sistemas HPC, entre las cuales destaca la planificación de tareas consciente de la energía. Puesto que este es un problema NP-completo, las principales vertientes de investigación en esta área se han centrado en algoritmos heurísticos de planificación y, más recientemente, modelos de inteligencia artificial.

El campo de la planificación de tareas en sistemas HPC como medio para reducir el consumo energético tradicionalmente se ha centrado en la optmización de métricas relacionadas con el tiempo, dada la estrecha relación existente entre estas y el consumo energético. Sin embargo, factores como la contención de memoria y la consideración de recursos heterogéneos derivan en una mayor complejidad que requiere el desarrollo de soluciones más específicas.

Este trabajo pretende diseñar un planificador inteligente para sistemas HPC entrenado mediante técnicas de aprendizaje reforzado profundo que se centre explícitamente en reducir el consumo de energía haciendo uso de la información proporcionada por las especificaciones de consumo de potencia de los recursos de un clúster HPC heterogéneo, y optimizando métricas relacionadas con la energía en lugar de con el tiempo.

Palabras clave: Planificación de tareas, Aprendizaje reforzado profundo, Computación de alto rendimiento, Clusters heterogéneos, Consumo energético.

Chapter 1

Introduction

This chapter introduces the motivation for this work as well as some basic concepts required for its comprehension. More specifically, it presents the problem of energy consumption in high-performance computing systems, and how it can be targeted by a variety of strategies that include job scheduling. The complexity of this task is then exposed, followed by the presentation of artificial intelligence techniques as a possible solution. At the end of the chapter, the main objectives pursued in this work are summarized and the structure of the document is outlined.

1.1 Energy Consumption in HPC Systems

High-performance computing systems are constantly evolving, reaching greater heights in computing power, efficiency and scalability [2]. Along the way, energy consumption has always remained a key obstacle both in terms of environmental impact [3] and maintenance cost. As a reference for the latter, the costs of running an HPC cluster for two years can be comparable to the cost of its purchase, owing to this factor [4].

Not only is this a problem in the further development of HPC systems, but also on a larger scale. At the time of writing, data center and data transmission networks account for between 1% and 1.5% of global electricity use and 0.9% of energy-related greenhouse gas emissions, according to the International Energy Agency [5]. This represents only a moderate increase with respect to 2010 thanks to improvements made in energy efficiency, but CO_2 emissions must still be halved by 2030 to achieve the United Nations net zero objective.

Even though these numbers do not refer exclusively to data centers geared towards HPC, the intensity of the tasks assigned to these clusters makes them a major contributor towards energy consumption. In fact, the computational and storage requirements of specialized applications continue to grow as the focus stays on artificial intelligence and data mining, according to the Hyperion Research 2022 report [6]. As for the cause, energy consumption in a cluster can be attributed to several factors. Firstly, the essential components needed for the basic operation of a cluster, such as network cards or switches, need to be powered. Secondly, energy is expended both while preparing and while executing jobs. Thirdly, idle nodes consume energy as well. Finally, the energy consumption of cooling systems is a considerable contributing factor [7].

For these reasons, research is still ongoing in this area. Over the last decade many aspects have been considered in an attempt to reduce energy consumption, for example the im-

pact of parallel programming paradigms of shared memory and message passing [8], the willingness of users to write and submit energy-efficient programs [9], total energy budgets [10] or the influence of fault tolerance mechanisms [11]. These are only a taste of the wide range of alternatives that have been explored, while many more are still being tested. In this work, the objective of reducing energy consumption is tackled through energy-aware scheduling techniques. A broad survey that covers the research carried out in this area in particular during the last 15 years can be found in [12].

1.2 The Job Scheduling Problem

A relevant aspect that influences energy consumption in high-performance computing systems, and efficiency on a broader scope, is *job scheduling*. Job scheduling determines the manner in which free resources are assigned to incoming jobs within a cluster (or viceversa). This task is usually carried out by an independent module with access to the pending job queue and knowledge of the current state of the cluster, commonly known as the *workload manager* or *scheduler*.

The workload manager implements a policy or algorithm that determines which resource is matched to each submitted job. Different objectives to be met through a scheduling policy may be specified, the most relevant being minimizing makespan, job return time or energy consumption. In order to achieve them, several issues have to be taken into account that impose constraints on the way job-resource assignments can be carried out. For example, jobs may have requirements such as requested memory, dependencies with other jobs, a waiting time threshold or affinity to a particular resource. At the same time, the state and architecture of the cluster itself also have to be taken into account while scheduling. This way, the scheduling problem becomes more complex when applied to heterogeneous clusters in which assigning a job to a resource meeting its requirements produces different results depending on the hardware specifications of the chosen candidate. Similarly, network communications and access to shared memory also have to be coordinated in order to prevent contention, and the list of restrictions goes on.

Even without considering these additional difficulties, optimal job scheduling is an NP-complete problem in optimization theory [13], meaning that there is no known algorithm that can solve it in polynomial time. For this reason, several methods have been proposed that can provide approximate solutions, mainly scheduling heuristics and, more recently, artificial intelligence approaches.

1.3 Artificial Intelligence

Soon after the advent of the first computers, researchers in the field started to develop an interest towards the creation of systems that could mimic human intelligence. In other words, a machine capable of thinking or acting like a human, or of thinking or acting rationally. This is what is known as artificial intelligence or AI [14].

Many different subfields exist within AI, such as natural language processing, computer vision or knowledge representation. The field of machine learning is of particular interest for the objectives pursued in this work. Machine learning aims to train a system (or *model*) to be able to reproduce a particular behaviour or make predictions. An example would be image classification, a task in which a model is trained to correctly assign a category to an image based on the pixels that conform it. Machine learning can be further classified into

supervised learning, unsupervised learning and reinforcement learning [15].

- In *supervised learning*, the model is fitted to a collection of labelled data, where each input has an associated tag. Once trained, the model should be capable of assigning the correct tag to data it has not seen before. Image classification fits this category.
- In *unsupervised learning*, the model is expected to find relations or groups (*clusters*) in a collection of unlabelled data, and later be capable of assigning unseen data to one of the extracted categories.
- In *reinforcement learning* [16], the model is embedded in an *agent* that interacts with an *environment*. Based on the state of the environment, the model is used to predict an action. The agent then applies the action to the environment and receives a corresponding reward indicating how beneficial the action was towards the end goal. Through successive iteration, the agent eventually learns which actions are more advantageous in a given situation.

Different variants of models can be trained using machine learning, some of which may be more appropriate than others for a given task. Examples include artificial neural networks, decision trees, support vector machines or bayesian networks. Recent advances have led to *deep learning*, which is characterized by the usage of artificial neural network architectures with multiple layers, devised to progressively extract features from an input at multiple levels. These *deep neural networks* can be leveraged in any of the aforementioned paradigms.

In particular, the scheduling problem applied to an HPC cluster can be suitably modelled as a reinforcement learning task, considering an environment composed by the current state of the cluster and the pending job queue, and identifying the workload manager or scheduler as the agent. This way, a workload manager can be trained to minimize a given metric, such as the total makespan or energy consumption, by noting what kind of job-resource assignments offer the best rewards for the chosen objective.

1.4 Objectives

Many alternatives have been tested to target the issue of energy consumption in the field of high-performance computing, some of which focus on scheduling techniques that allow for a more intelligent use of resources.

This work follows this line of thought and attempts to develop an intelligent energy-aware workload manager for heterogeneous HPC cluster architectures based on deep reinforcement learning, with the specific objective of reducing total energy consumption or EDP (Energy-Delay Product). At the same time, it aims to analyze how this kind of intelligent scheduler behaves when the power consumption specifications of the resources of a cluster are considered as part of the information available for decision-making. This main goal can be subdivided into the following partial objectives:

- *Study of the problem and the applicable AI techniques*: it is known that the scheduling problem can be modelled using a reinforcement learning scenario, but several algorithms exist in this paradigm, each with its own set of properties and drawbacks. Similarly, neural network design is a vast field of research. Both areas shall be studied before proceeding with the development of the intelligent workload manager.

- *Design and implementation of the intelligent scheduler in IRMaSim*: the selected reinforcement learning algorithm shall be implemented in the IRMaSim cluster simulation framework. To this end, the agent and environment components shall also be defined and integrated in the infrastructure of the simulator.
- *Experimental validation of the scheduler*: a first batch of simple experiments shall be executed using the newly designed workload manager to verify that it behaves as expected and that there are no errors in the implementation.
- *Experimental evaluation of the scheduler*: once the implementation has been validated, the developed scheduler shall be tested in more complex scenarios in order to evaluate its performance.

1.5 Document Structure

This document is divided into five chapters, including the current Chapter 1: *Introduction*. The remaining contents are organized as follows:

- *Chapter 2* delves deeper into some of the notions introduced in this first chapter, namely the scheduling techniques employed in modern HPC systems and the theory behind reinforcement learning algorithms and neural networks.
- *Chapter 3* describes the process followed to develop a new intelligent workload manager that specializes in minimizing energy-related metrics, starting from the employed algorithms and techniques and continuing by outlining the implementation.
- *Chapter 4* details the experiments carried out to validate and evaluate the new workload manager, comparing its performance to that of a series of heuristic algorithms.
- *Chapter 5* closes the document by highlighting the main extracted conclusions and proposing future lines of work regarding the project.

Chapter 2

Background

This chapter delves into the fundamental concepts that conform the basis of this work. Specifically, several techniques studied in academia and/or employed by the most commonly used HPC workload managers will be reviewed, and the basic ideas and theory behind reinforcement learning and neural networks will be introduced. The chapter concludes by listing related work concerned with smart job scheduling and energy consumption in HPC systems.

2.1 State-of-the-Art HPC Scheduling

The scheduling problem introduced in Chapter 1 has been exacerbated in recent years owing to several factors including, but not limited to, the steady rise in the processing power of supercomputers, the emergence of new hardware resources and novel hardware structure and the appearance of increasingly diverse workloads that combine compute-intensive and data-intensive applications [17]. Not only workloads have become more varied: the majority of modern HPC cluster systems display an heterogeneous architecture combining resources with different frequencies, number of processing units, bandwidth or memory capacity, in addition to hardware accelerators such as graphical processing units (GPUs) or field-programmable gate arrays (FPGAs) [18].

This scenario creates a need to consider a variety of trade-offs in a dynamic environment where new jobs may be received at any given time [19]. As an example, resources with higher clock rates complete jobs faster, but may consume more energy during execution due to their greater power consumption [20]. Moreover, resource sharing and interconnection give rise to issues such as ensuring resource fairness and minimizing resource contention while also focusing on job performance [17].

A first approach to tackle this complexity involves the use of *heuristic functions* -or *scheduling policies*- that assign priorities to jobs waiting in the queue to the cluster based on their attributes, effectively determining which job shall be scheduled next. These range from simple one-parameter heuristic priority functions such as FCFS (First Come First Served) or SJF (Shortest Job First) to complex metrics combining multiple parameters such as WFP, UNICEF or F1 [21]. Going a step further, machine learning techniques can be used to train the parameters of non-linear scheduling policies and adapt them to the current workload dynamically [22].

Artificial intelligence can also be leveraged to define intelligent schedulers that do not require the specification of a particular heuristic function, but instead rely on current queue

and cluster status to decide on the best job-resource assignments. Such workload managers are therefore more adaptive than those implementing particular scheduling policies. Multiple examples can be found in section 2.4.

Another frequently employed approach is *backfill scheduling* or *backfilling*. Given a priority queue containing the pending jobs, basic backfilling relies on user-defined job runtime estimates to schedule lower-priority jobs, located later in the queue, on already allocated resources if this will not result in a higher-priority, already running job missing its deadline or breaching any other kind of restriction. Common backfilling strategies include *EASY backfilling* and *conservative backfilling*. EASY backfilling is more aggressive and considers only the first job in the queue when evaluating higher-priority jobs that might be delayed, whereas conservative backfilling requires that no job preceding the lower-priority job in the queue is delayed [23].

Backfilling provides good results on production systems, but relies heavily on user-established job time estimates being realistic. It has been found that inadequate user runtime estimates significantly degrade performance due to both underestimation of the actual runtime of the job, which leads to the job being killed, and runtime overestimation, which might prevent the scheduling of jobs in the queue [24].

Another relevant aspect that has an impact on cluster performance is the choice between two scheduling paradigms: *list scheduling* or *pack scheduling*. In list scheduling, pending jobs are first organized in a queue before being dispatched sequentially, whereas in pack scheduling jobs are partitioned into packs. Jobs within each pack are scheduled concurrently, and the next pack cannot start executing before all the jobs in the previous pack have finished running. Pack scheduling is capable of achieving better performance than list scheduling in environments with multiple types of resources, such as I/O, memory or cache [25].

The aforementioned techniques have been implemented in some of the best-known commercial HPC schedulers, including Slurm [26], IBM Spectrum LSF, OpenPBS (an evolution of the NASA Portable Batch System [27]) and the TORQUE workload manager for the Moab HPC Suite platform [17]. In their most basic form, however, commercial workload managers for HPC systems still hold a First-In-First-Out queue of pending jobs that are assigned in submission order to the resource which best matches the requirements specified by each job in terms of processing power, number of nodes, memory, and so on. At the time of writing, only Moab seems to offer a proprietary intelligent scheduling engine.

2.2 Reinforcement Learning

Within the field of machine learning, reinforcement learning is a strategy that involves two main elements: an *agent* and an *environment*. The goal of reinforcement learning is to teach the agent what *action* to take after having observed a particular *state* of the environment so as to maximize a given *reward* value, defined by the pursued objective. The complexity of this model resides in the fact that each action taken by the agent modifies the environment and therefore influences future rewards [16].

More specifically, the agent learns a *policy* that maps states to actions. This policy may be deterministic, always choosing the same action for a given state; or stochastic, assigning a probability to each possible action depending on the current state. Neural networks are often used to approximate the policy in the latter case. Every time the agent chooses an action and applies it to the environment, the state of the environment changes and a *reward* is generated indicating how beneficial the choice of that particular action was towards the

end goal. These rewards are employed by the agent to gradually refine the policy through successive iteration, as shown in figure 2.1.

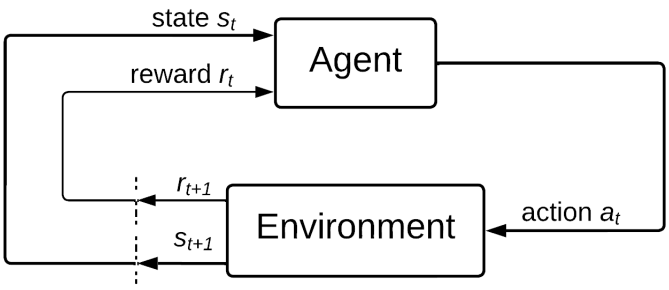

Figure 2.1: Agent-environment interaction in reinforcement learning.

2.2.1 Formalization

The reinforcement learning scenario can be formally modeled as a Markov decision process (MDP). In an MDP, a learner or decision maker (the *agent*) interacts with an *environment*, comprised of anything that is external to the agent. Other elements of an MDP include a set of states $\mathcal{S}$, a set of actions $\mathcal{A}$ and a set of numerical rewards $\mathcal{R} \subset \mathbb{R}$ [16]. In the discrete case, the agent-environment interaction can be dissected into a series of time steps $t = 0, 1, 2, \ldots$ In each time step, the following occurs:

1. The agent receives an observation of the state of the environment $s_t \in \mathcal{S}$.
2. Based on the observed state, the agent selects an action $a_t \in \mathcal{A}$.
3. As a consequence of applying the chosen action, the agent receives a reward $r_{t+1} \in \mathcal{R}$ and the environment advances to state s_{t+1}.

In this manner, a trajectory of the form $s_0, a_0, r_1, s_1, a_1, r_2, s_2, \ldots$ is generated, where s_t and r_t are random variables. The dynamics of how this trajectory evolves are completely characterized by the probabilistic function p (equation 2.1), which determines the probability of transitioning to a certain state s' with a reward of r by taking action a in the current state s. This expression assumes that all the historical trajectory information required for decision-making is contained within the last state, otherwise known as the Markov property.

$$p(s', r \mid s, a) \doteq Pr\{s_t = s', r_t = r \mid s_{t-1} = s, a_{t-1} = a\}, \quad \forall s', s \in \mathcal{S}, r \in \mathcal{R}, a \in \mathcal{A} \quad (2.1)$$

The final goal of the agent is to maximize the *expected return* at each time step. The return G_t is a function of the reward sequence defined as $G_t \doteq r_{t+1} + r_{t+2} + r_{t+3} + \ldots + r_T$, where T is a final time step. At this point it is necessary to make a distinction between *episodic* and *continuing* tasks. Episodic tasks can be broken down into subsequences, or episodes. Episodes terminate when a *terminal state* is reached at time T, a random variable that may vary from one episode to the next. Continuing tasks, on the other hand, cannot be broken down in this way and, logically, $T = \infty$. This poses a problem, since it makes the return infinite. For this reason, G_t is usually defined as the *discounted* return instead (equation 2.2). The parameter γ controls how far-sighted the agent is: a value of $\gamma = 0$ would consider only the immediate reward r_{t+1}, whereas a value of $\gamma = 1$ would weigh the

rewards at all future time steps the same. The expected return can also be formulated recursively as $G_t \doteq r_{t+1} + \gamma G_{t+1}$.

$$G_t \doteq \sum_{k=0}^{\infty} \gamma^k r_{t+k+1},\ 0 \leq \gamma \leq 1 \tag{2.2}$$

Most reinforcement learning algorithms operate by estimating *value functions* that determine the expected return when following the current policy $\pi(a \mid s)$, given either a state (*state-value function*, equation 2.3) or a state-action pair (*action-value function*, equation 2.4). In episodic tasks, the value of a terminal state $v_\pi(s_T)$ is always zero.

$$v_\pi(s) \doteq \mathbb{E}_\pi[G_t \mid s_t = s] \tag{2.3}$$

$$q_\pi(s, a) \doteq \mathbb{E}_\pi[G_t \mid s_t = s, a_t = a] \tag{2.4}$$

The state-value function can also be formulated in a recursive manner thanks to the recursive definition of the expected return. This representation is known as the *Bellman equation* (equation 2.5). A similar expression can be found for $q_\pi(s, a)$.

$$v_\pi(s) \doteq \sum_a \pi(a \mid s) \sum_{s',r} p(s', r \mid s, a)[r + \gamma v_\pi(s')], \quad \forall s \in \mathcal{S} \tag{2.5}$$

Both value functions can also be combined to obtain the *advantage function*, shown in equation 2.6. The advantage of taking action a in state s indicates how much the expected reward can be improved by choosing action a with respect to following the default behaviour of the policy π.

$$A_\pi(s, a) \doteq q_\pi(s, a) - v_\pi(s) \tag{2.6}$$

2.2.2 Policy Gradient Methods

Ideally, an agent should try to find an optimal policy, that is, a policy such that each chosen action maximizes the real final return. This might be possible to achieve for simple problems where the action and state spaces are small enough to solve the Bellman optimality equation without running out of memory or time. Nevertheless, this is seldom the case, since reinforcement learning is usually applied to complex problems where many state-action combinations are possible. In these situations, approximate solution methods are used instead. Policy gradient methods fit in this category.

Policy gradient methods, unlike most other approaches, do not make direct use of the value functions to choose actions. Instead, they define and learn a parametrized policy $\pi(a \mid s, \theta) = Pr\{a_t = a \mid s_t = s, \theta_t = \theta\}$ that is improved iteratively by applying gradient ascent on the estimate of a performance measure $J(\theta)$ (or gradient descent on $-J(\theta)$). This measure is usually an estimate of the expected return [16].

Within policy gradient methods, *actor-critic* methods involve learning both an approximation to the policy (*actor*) as well as to the value function (*critic*). In other words, the value function is parametrized as well. Using the value output by the critic as a baseline for the expected return helps reduce the variance in $J(\theta)$ [28].

2.3 Neural Networks

Neural networks can be considered universal function approximators, which makes them a powerful tool in pattern recognition, classification or regression tasks. They consist in

a series of *layers* of interconnected *neurons*, a structure inspired by the anatomy of the brain. Feed-forward neural networks, also known as multilayer perceptrons (MLPs, figure 2.2), are the simplest kind of neural network models.

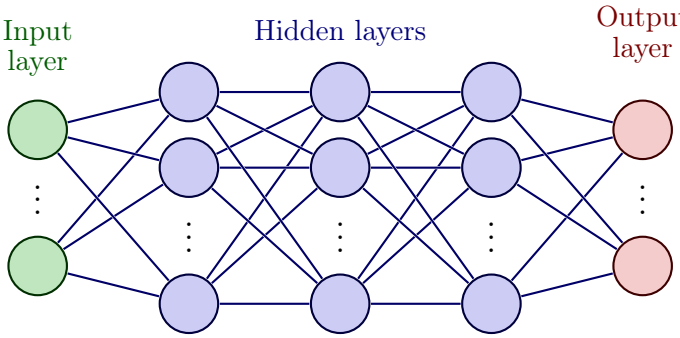

Figure 2.2: A multilayer perceptron (MLP) with three hidden layers.

Each layer of an MLP constructs m linear combinations of its input variables $\mathbf{x}_n$ according to the parameters of each neuron (weights $\mathbf{w}_{m\times n}$ and biases $\mathbf{b}_m$) as follows:

$$a_j = \sum_{i=1}^{n} w_{j,i} x_i + b_j, \quad 1 \leq j \leq m$$

where m is the number of neurons in the current layer and n is the dimension of the input. A differentiable, nonlinear *activation function* h is then applied to an *activation* a_j to give the final output of a neuron as $z_j = h(a_j)$. These outputs can then be used as inputs to a following layer of neurons defined in the same manner [15]. An example of these equations in the context of the graphical representation of an MLP is shown in figure 2.3.

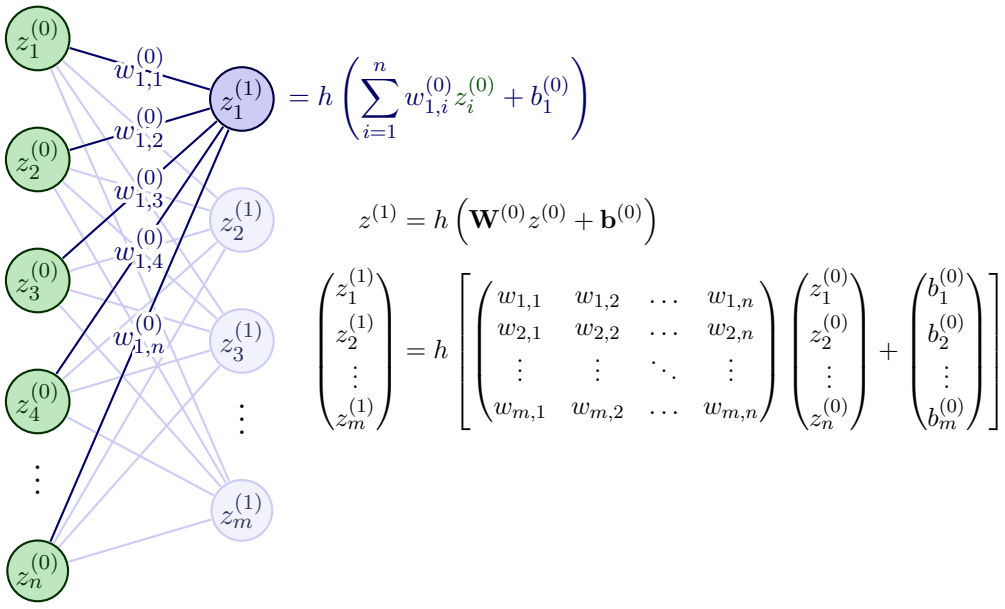

Figure 2.3: Multilayer perceptron activation equations. Superscripts are used to differentiate between layers.

Neural networks have to be trained in order to be capable of approximating a given function. This process involves comparing the output of the network for a series of input vectors $\mathbf{x}_n$ with the corresponding known correct values and adjusting the weights and biases of the network in order to minimize the error E_n, usually making use of the information provided by the gradient of the error function (often called *loss* or *objective* function). This

gradient can be computed efficiently through *backpropagation*, which entails the following steps [15]:

1. *Forward propagation*: an input vector $\mathbf{x}_n$ is propagated through the network to obtain the activations of all the hidden and output units, from here on referred to as a_j and a_k respectively.
2. The errors $\delta_k \equiv \frac{\partial E_n}{\partial a_k}$ are computed for all the output units.
3. *Backpropagation*: the errors of the output layer are backpropagated through the network to obtain the corresponding δ_j error terms for each hidden unit making use of the chain rule for partial derivatives: $\delta_j \equiv \frac{\partial E_n}{\partial a_j} = \sum_k \frac{\partial E_n}{\partial a_k} \frac{\partial a_k}{\partial a_j}$.
4. The required derivatives are evaluated using $\frac{\partial E_n}{\partial w_{ji}} = \frac{\partial E_n}{\partial a_j} \frac{\partial a_j}{\partial w_{ji}} = \delta_j z_i$.

2.4 Related Work

Machine learning approaches have been applied to tackle the scheduling problem in HPC clusters before, especially in academia. Some notable examples of such projects include DeepRM, DeepRM2, Decima, RLScheduler, RLSchert and DRAS.

DeepRM [29] was one of the first initiatives to apply reinforcement learning techniques to the scheduling problem as an alternative to heuristic approaches. It assumes a single large resource pool, abstracting away machine boundaries and resource fragmentation. In other words, it models a cluster as a single collection of resources of different types (CPU, memory and I/O). DeepRM can learn to optimize objectives such as average job slowdown or completion time through the application of a standard policy gradient reinforcement learning algorithm, in particular a variant of REINFORCE. During evaluation, it was shown to best the SJF, Packer and Tetris heuristics.

Years later, DeepRM2 and DeepRM_Off [30], continuous online resource scheduling and offline resource scheduling designs respectively, were proposed as improvements upon DeepRM. The original neural network was replaced with a multi-layer convolutional neural network and other changes were applied to speed up convergence during training.

Decima [31] employs reinforcement learning techniques and neural networks to learn workload-specific scheduling algorithms given a high-level objective such as minimizing average job completion time. As a special characteristic, Decima is capable of scheduling while taking into account job dependencies represented as directed acyclic graphs (DAGs), as is the case in cluster managers such as Hive or Spark. This is a highly complex task for which heuristics cannot possibly provide a satisfactory solution. In fact, upon integrating a Decima prototype with Spark on a 25-node cluster, average job completion time was improved by at least 21%.

RLScheduler [32] is an automated HPC batch job scheduler that uses a deep reinforcement learning approach. It was also brought forward as an alternative to heuristic priority scheduling, which was not deemed flexible enough to deal with varying workloads and changes in scheduling requirements such as maximizing resource utilization, job throughput or minimizing job wait time. RLScheduler manages to achieve these objectives, but it only considers homogeneous cluster architectures, deeming it far too simplistic by present-day standards. For this reason, RLScheduler was later used as a starting point in [33], where it was adapted to heterogeneous architectures.

RLSchert [34] is a job scheduler based on deep reinforcement learning and remaining runtime prediction. On the one hand, it estimates the state of a cluster by means of a dynamic

job remaining runtime predictor that employs a recurrent neural network to encode time series information. On the other hand, it learns the optimal policy in terms of what jobs to schedule, or to kill and reschedule, based on the current status of the system. To learn the policy, it applies imitation learning and a parallel implementation of the Proximal Policy Optimization algorithm so as to speed up training. During evaluation, RLSchert outperformed both static heuristic policies and DeepRM in terms of average slowdown and completion time.

DRAS (Deep Reinforcement Agent for Scheduling) [35] also focuses on improving upon heuristic scheduling strategies in complex HPC environments with dynamic application workloads. DRAS is particularly concerned with preventing resource underutilization and job starvation, which is achieved through *resource reservation*. To this end, it combines a reinforcement learning approach with backfilling and employs a hierarchical neural network for decision-making. This network is divided in two levels: a level 1 network selects jobs to be executed immediately or when enough resources are freed, and a level 2 network identifies backfilling candidates. However, DRAS does not consider heterogeneous cluster architectures.

The aforementioned projects focus on intelligent scheduling, and some of them include the reduction of energy consumption as one of multiple objectives. Other works that specifically tackle energy consumption through job scheduling, combined with other techniques, can also be found in the literature. Some examples are provided below.

ExpREsS [19] is a scheduler for the Apache Spark processing framework. Its main goal is to orchestrate the execution of multiple big data applications in distributed processing systems while satisfying performance requirements and minimizing energy consumption. This is achieved through adaptive dynamic voltage and frequency scaling (DVFS) and accurate models capable of predicting the execution time and power consumption of an application before it starts executing, which is especially needed when scheduling streaming applications for which no historical data is available. More specifically, ExpREsS orchestrates the execution order of applications and automatically determines how many CPU cores should be reserved, and what their running frequencies should be, so as to minimize energy consumption for each particular application.

The HEA-PAS scheduling algorithm [36] minimizes the response time of parallel DAG applications subject to energy constraints in heterogeneous HPC systems. In HEA-PAS, the energy that can be allocated by an application is quantified and distributed among its tasks according to their energy demand rate. This is done in two stages: first, a static energy pre-allocation for each task is carried out before scheduling; then, dynamic energy is allocated dynamically during the operation of the algorithm. The energy allocated to each task is used to find the optimal processor and frequency combination to execute the entire application.

The work presented in [37] explores the use of application signatures instead of full dynamic power profiling to obtain application data for use in conjunction with energy-aware task scheduling in data centers, including HPC systems. An application signature is a reduced version of the original application in terms of execution time. Signatures can therefore be used to estimate the energy consumed by applications without having to monitor and profile their execution until completion, a task which is often not energy-efficient when applied to batches of long-running applications.

Despite the extensive investigation that has been carried out in this area, several aspects remain unexplored. The next chapter introduces the research that has been carried out in this work regarding some of these challenges.

Chapter 3

Development of an Energy-Aware Scheduler

This chapter details the implementation of the new intelligent energy-aware scheduler, which has been named "Energy scheduler". First, an overview of the project is given, followed by a description of the employed machine learning algorithms and techniques. The rest of the chapter is devoted to describing the elements that compose the new scheduler and how they interact with each other.

3.1 Overview

In recent years, energy consumption has become a limiting factor in the evolution of high-performance computing clusters both in terms of environmental impact and maintenance cost. The previous chapters gave several examples of intelligent and heuristic workload managers that are capable of reducing consumption through job scheduling. However, no studies have been found that consider energy-related parameters when optimizing for energy consumption in HPC clusters. Instead, they focus on time-related job and resource attributes such as user-specified job requested time or average resource frequency.

Although this time-focused approach might be enough when considering computationally intensive jobs that do not access memory frequently, since energy consumption is rather straightforward to estimate in these cases, the scheduling of memory-intensive jobs requires a more rigorous analysis. For example, memory contention increases job execution time, but dynamic energy consumption is lower while memory is being accessed. Thus, it is worthwhile to investigate how this situation can be handled using more specific, energy-focused strategies.

The proposed Energy scheduler incorporates the static and dynamic power specifications of each resource as part of the information involved in the decision-making process. Furthermore, at each time step, the workload manager may choose either to schedule a pending job or to wait until resources are freed and a more suitable job-resource assignment becomes available. This option is envisioned to explore how a less greedy strategy that does not make immediate use of free resources can offer better results in terms of energy consumption under certain circumstances. In addition, as an energy-focused tool, the new scheduler allows users to specify maximum energy constraints for jobs.

3.2 Algorithms

This section introduces the basic theory behind the algorithms, functions and models employed in the development of the Energy scheduler. These have been classified by their domain of application: reinforcement learning or neural network design. The presented reinforcement learning strategies are explained in more detail, whereas only the most important characteristics of the introduced neural network design techniques are exposed.

3.2.1 Reinforcement Learning

This subsection introduces and explains Proximal Policy Optimization, the reinforcement learning algorithm chosen to train the scheduler. Afterwards, the advantage estimator that has been used to implement the algorithm is presented.

Proximal Policy Optimization

Vanilla policy gradient methods (introduced in section 2.2.2) suffer from two main deficiencies. On the one hand, they are *sample inefficient*, since a whole trajectory has to be simulated with the same policy in order to carry out a single step of gradient descent and obtain an updated policy. On the other hand, an update step may result in a large *divergence* from the old policy, causing the agent to miss local optima [38].

An algorithm by the name of Trust Region Policy Optimization (TRPO) [39] was first brought forward to remedy these shortcomings, in particular the second issue. TRPO offered a solution by limiting the Kullback-Leibner divergence between the old and new policies to a *trust region*, thus avoiding destructively large policy updates.

Nevertheless, the implementation of the TRPO algorithm was considered complicated and computationally expensive. Hence, in 2017 OpenAI proposed Proximal Policy Optimization (PPO) [1] [38] as a simpler and more sample-efficient family of policy gradient methods. Two main variants of PPO exist: penalty-based PPO, also based on the Kullback-Leibner divergence, and PPO clip. In [1], the latter was found to outperform the former.

The basic PPO algorithm is outlined in Algorithm 1. It considers fixed-length trajectory segments and the use of N independent actors collecting data in parallel to speed up the learning process.

Algorithm 1 PPO, Actor-Critic Style. Extracted from [1].

for iteration=$1, 2, \ldots$ **do**
  for actor=$1, 2, \ldots, N$ **do**
    Run policy $\pi_{\theta_{old}}$ in environment for T time steps
    Compute advantage estimates $\hat{A}_1, \ldots, \hat{A}_T$
  end for
  Optimize surrogate L wrt θ, with K epochs and minibatch size $M \leq NT$
  $\theta_{old} \leftarrow \theta$
end for

In the PPO clip variant, the surrogate objective L is defined as shown in equation 3.1, where ϵ is a hyperparameter controlling the range of acceptable deviation and $\hat{A}_t$ is the advantage estimate at time step t, which may be positive or negative. $R_t(\theta)$ denotes the

probability ratio of the newly updated policy with respect to the old policy (equation 3.2).

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t \left[\min \left(R_t(\theta) \cdot \hat{A}_t,\ \text{clip}(R_t(\theta),\ 1-\epsilon,\ 1+\epsilon) \cdot \hat{A}_t \right) \right] \tag{3.1}$$

$$R_t(\theta) = \frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{old}}(a_t \mid s_t)} \tag{3.2}$$

Figure 3.1 plots L^{CLIP} for a single time step as a function of the ratio[1], for positive advantages on the left and negative advantages on the right. Conceptually, divergence is bounded to a trust region when the new policy performs better, that is, when the probability of choosing a more advantageous action increases or the probability of choosing a less advantageous action decreases; whereas the ratio between policies weighs into the objective function when the performance of the new policy is worse.

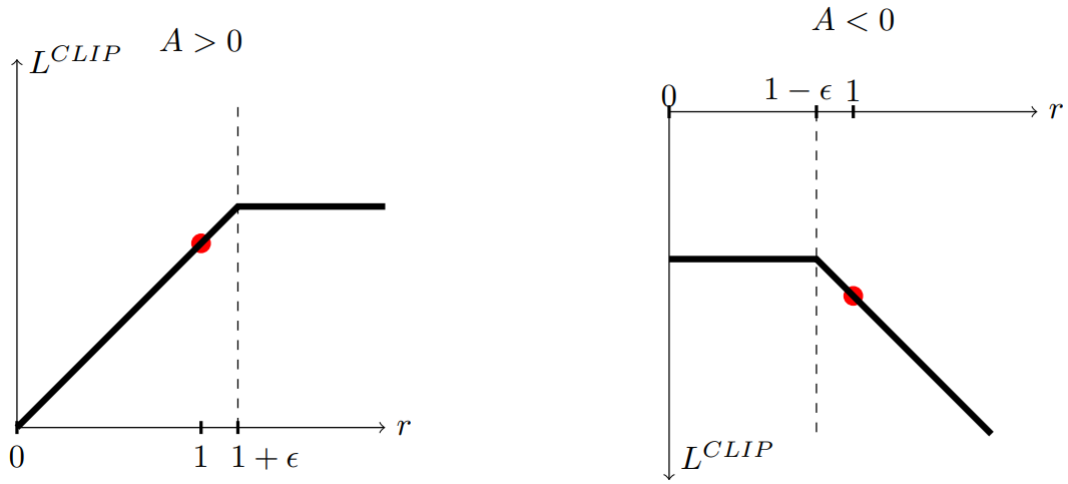

Figure 3.1: Plots showing surrogate function L^{CLIP} for a single time step t as a function of the probability ratio. Extracted from [1].

Generalized Advantage Estimate

Different techniques exist to estimate the advantage function from a series of collected sample trajectories. A frequently chosen alternative is the Generalized Advantage Estimate (GAE), defined in equation 3.3. This estimate can be easily adapted to episodic tasks by setting the final time step T as the upper bound in the summation [40].

$$\hat{A}_t^{GAE(\gamma,\lambda)} \doteq (1-\lambda)\left(\hat{A}_t^{(1)} + \lambda \hat{A}_t^{(2)} + \lambda^2 \hat{A}_t^{(3)} + \dots\right) = \sum_{l=0}^{\infty} (\gamma\lambda)^l \delta_{t+l}^V \tag{3.3}$$

From the bottom up, equation 3.3 originates from the consideration of the advantage of a particular action a_t given an approximate value function V, estimated as the TD residual of V with discount γ [16]. The TD residual expression is shown in equation 3.4, respecting the notation established in section 2.2.1. Next, the TD residuals for successive actions are summed with the same discount factor, yielding $\hat{A}_t^{(k)}$, an estimator of the advantage

[1]The original paper denotes the ratio as r instead of R. In this document the notation r is reserved to represent rewards.

function for a sequence of k actions starting at time step t (equation 3.5). Finally, $\hat{A}_t^{GAE(\gamma,\lambda)}$ is the exponentially-weighted average of these $\hat{A}_t^{(k)}$ estimators up to the specified bound.

$$\delta_t^V = r_{t+1} + \gamma V(s_{t+1}) - V(s_t) \tag{3.4}$$

$$\hat{A}_t^{(k)} \doteq \sum_{l=0}^{k-1} \gamma^l \delta_{t+l}^V = -V(s_t) + r_t + \gamma r_{t+1} + \ldots + \gamma^{k-1} r_{t+k-1} + \gamma^k V(s_{t+k}) \tag{3.5}$$

The γ and λ hyperparameters influence the bias-variance tradeoff in the advantage estimator. Setting λ to 0 is equivalent to considering $\hat{A}_t^{(1)} = \delta_t^V$ exclusively, with high bias and low variance. Setting λ to values close to 1, on the other hand, extends the number of considered terms and leads to low bias but higher variance.

3.2.2 Neural Network Design

This subsection introduces the techniques that have been considered in an attempt to prevent divergence during the training of the actor and critic neural networks embedded in the scheduler, as well as to improve their ability to generalize over different scenarios. In particular, actor and critic have been defined as self-normalizing neural networks and the Adam optimization algorithm has been used for training. Finally, stochastic weight averaging has been implemented as an optional feature to potentially improve test-time results.

Self-normalizing Neural Networks

Standard feed-forward neural networks (FNNs) have not been nearly as successful as convolutional or recurrent neural networks in deep learning. Instead, they have stayed in the shallow end of the spectrum owing to an increased instability in learning upon increasing the number of layers. In FNNs, perturbations introduced by regularization and normalization techniques result in an increased variance in the training error that slows down learning and leads to divergence.

Self-normalizing neural networks (SNNs) [41] modify standard deep FNNs to achieve robustness and reduce variance mainly by using SELU (Scaled Exponential Linear Unit) activation functions, plotted in figure 3.2. These have a normalizing effect that is transitive across layers, pushing neuron activations to zero mean and unit variance. In order to achieve full self-normalization, weights are also initialized with close to zero mean and unit variance. An adapted form of dropout may also be applied during training in order to improve generalization.

Adam

Adam (derived from *adaptive moment estimation*) is an algorithm for first-order gradient-based optimization of stochastic objective functions, characterized by its computational efficiency and low memory requirements [42]. Stochastic objective functions are used, for example, when working with minibatches of data: the objective function is then composed of a sum of subfunctions evaluated at different subsamples. This subsampling can introduce noise, especially when regularization techniques are employed, and hence efficient and effective optimization techniques are required when working with stochastic objectives. In this regard, Adam offers several advantages: it makes the magnitudes of parameter updates

$$selu(x) = \lambda \cdot \begin{cases} x & \text{if } x > 0 \\ \alpha e^x - \alpha & \text{if } x \leq 0 \end{cases}$$

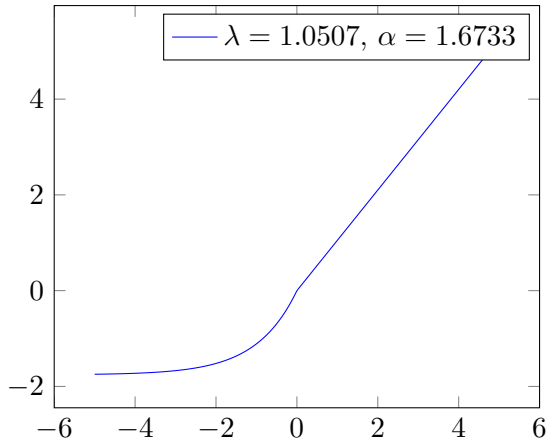

Figure 3.2: *selu*, the SELU activation function.

invariant to rescaling of the gradient, naturally performs a form of step size annealing and can also work with sparse gradients.

The algorithm keeps track of the exponential moving averages of the gradient (m_t) and the squared gradient (v_t) of each of the parameters of the objective function. These moving averages are estimates of the first moment (the mean) and the second raw moment (the uncentered variance) of the gradient, respectively. Since both averages are initially biased towards 0, they have to be bias-corrected, yielding $\hat{m}_t$ and $\hat{v}_t$. These quantities are then used to adjust the step size α in each iteration as $\alpha \cdot \hat{m}_t/(\sqrt{\hat{v}_t} + \epsilon)$, where ϵ is a hyperparameter. The complete specification of the algorithm can be found in [42].

Overall, Adam is an efficient, versatile and scalable algorithm that generally provides good results in deep learning tasks.

Stochastic Weight Averaging

Training deep neural networks with stochastic weight averaging (SWA) improves generalization, enabling models to find broader optima than standard training procedures using some form of stochastic gradient descent (SGD) [43].

SWA consists in averaging the weights of a network taken at different points in time during the training process, making it a simple and computationally inexpensive tool to improve test-time accuracy. The idea behind this technique is that SGD, with cyclical or constant learning rates in particular, leads to high-performing models that move around an optimum in weight space but never actually reach its central points. The choice of constant or cyclic learning rates is envisaged to allow better exploration of the relevant regions of the weight space.

The SWA algorithm does not impose restrictions on when or how often the weights of the network are sampled, making this approach flexible as well as efficient.

3.3 Implementation

This section describes how the proposed energy-aware scheduler has been implemented in Python making use of the IRMaSim and PyTorch frameworks for cluster simulation and machine learning respectively. The scheduler developed in [33] has been taken as a starting

point, although several changes have been made to the structure of the code and the logic of the intelligent agent.

New features have been incorporated as well, in particular the addition of a job maximum energy constraint and an option enabling the scheduler to wait until resources are freed instead of scheduling a new job whenever it is possible. This last option is referred to as the *wait action*. Still, the key idea of assigning a score to each possible job-resource pair remains at the core of the employed model. As in [33], resources are considered at the *compute node* granularity since IRMaSim is not currently capable of simulating parallel applications that involve inter-process communication while running on different nodes. For this reason, as well as to reduce the complexity of the problem, cores and processors are not seen as independent units, but as an aggregation of components belonging to a specific node.

At the time of writing, the source code of the project can be found in GitHub, in a fork of the main IRMaSim repository[2].

3.3.1 Main Components

The development of an intelligent scheduler (or workload manager) in IRMaSim, and of a reinforcement-learning-based scheduler in particular, requires the implementation of three main components: the *workload manager* itself, an *agent* and an *environment*.

- The **workload manager** is employed by the simulator to assign the pending jobs in the queue to available resources according to the choices of the agent.
- The **environment** is responsible for crafting observations based on current cluster and queue status, which are in turn fed to the agent to predict the next job-resource assignment.
- The **agent** is an artificial intelligence model capable of predicting the next job-resource assignment. The parameters of the model are updated during training based on a loss function to improve the quality of future predictions.

In the implementation of the Energy scheduler, these components are interrelated as shown in figure 3.3. Standard arrows indicate an architectural relationship of the type "holds a reference to", whereas dashed arrows represent the flux of information between elements. In the Energy scheduler, the workload manager is only responsible for orchestrating the interaction between the agent and the environment, and it is the environment that takes care of applying the chosen actions to the simulated instance of the cluster.

Before moving on to the specification of each of these elements, it is necessary to delve into the structure of the simulation loop so as to better comprehend how the training of the agent is integrated with the execution of a simulation.

1. Each IRMaSim *simulation* runs a number of trajectories of the specified length, where a trajectory is made up of a sequence of jobs sampled from a workload file, complete with their arrival times and user-specified parameters (requested time, memory, number of cores and so on).
2. Each *trajectory* is itself simulated as a series of time steps delimited by three kinds of events: the arrival of a new job, the completion of a running job or the expiration of a timer set by the workload manager.

[2] `https://github.com/Archiel19/IRMaSim.git`, in branch `EnergyScheduler`.

3. At each *time step*, the agent is invoked to choose an action which is then applied to the simulated instance of the cluster. Then, the corresponding reward and other training data are stored.

The agent is only trained when a complete simulation finishes. For this reason, multiple simulations are required to progressively refine the agent. A flowchart depicting this process is given in figure 3.4.

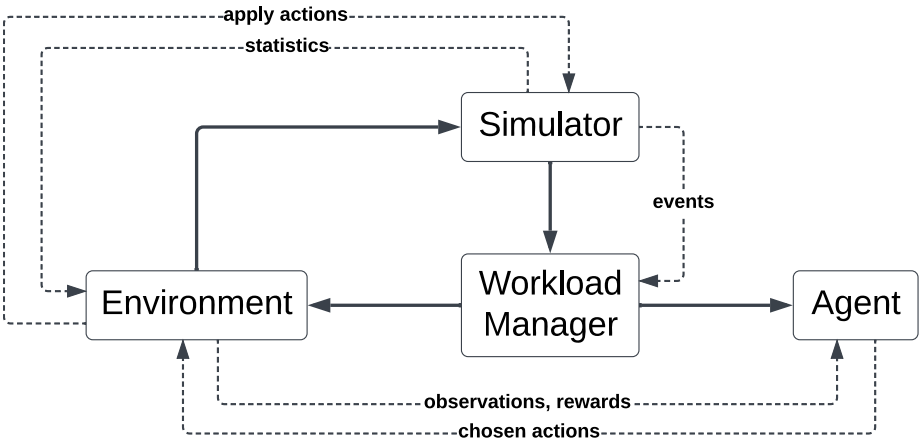

Figure 3.3: Main components and their relations in Energy scheduler.

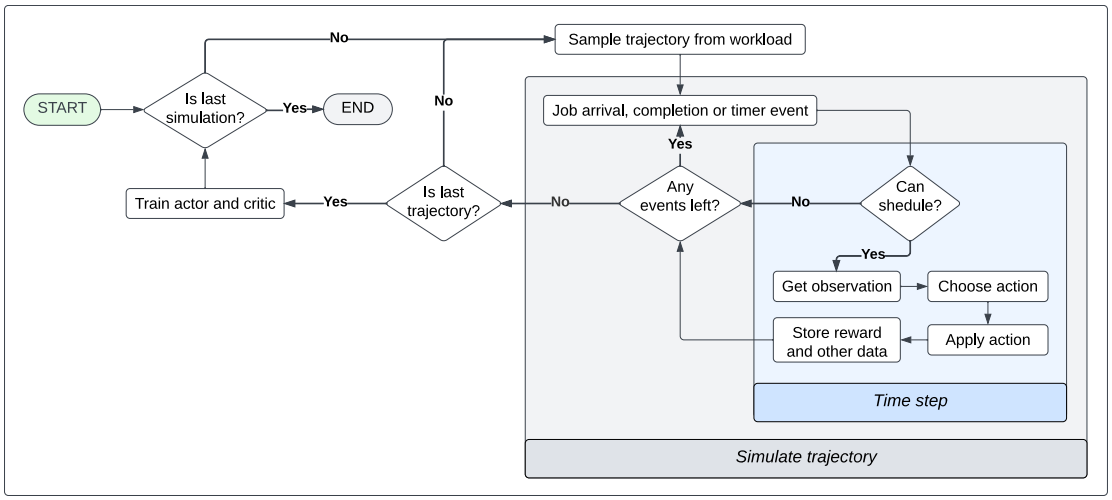

Figure 3.4: Flowchart depicting a high-level view of the main simulation loop using a reinforcement learning agent in IRMaSim.

3.3.2 Workload Manager

The source code of the workload manager component of the Energy scheduler can be found in `EnergyWM.py`, under `irmasim/workload_manager/`.

The `EnergyWM` class implements the IRMaSim `WorkloadManager` interface[3], which defines a series of methods that indicate how the workload manager shall behave after certain events. Namely, `on_job_submission`, `on_job_completion`, `on_end_step`, `on_alarm`,

[3] `WorkloadManager` is actually a class, since interfaces do not exist in Python.

`on_end_trajectory` and `on_end_simulation`. In terms of attributes, `EnergyWM` only contains references to the agent and the environment, as well as a flag indicating whether a new trajectory has just started.

`EnergyWM` is mainly responsible for coordinating calls to methods in the agent and the environment, and it is the environment that congregates most of the functionality related with scheduling jobs on the simulated instance of the cluster. Consequently, the implementation of the methods specified in the interface is quite straightforward:

- The `on_job_submission` and `on_job_completion` methods are used to update a pending job queue in the environment.
- On each call to `on_end_step`, the workload manager first checks if there are any pending jobs available for scheduling by making use of a helper function defined in the environment, `can_schedule`. If at least one job can be scheduled, a new state of the environment is retrieved and fed to the agent, which returns the corresponding action together with its value and log probability. The action is subsequently applied to the environment, successfully scheduling the next job, and the observation, action, value and log probability obtained at the current time step are stored in the agent.

  Normally, at this point the environment would have also provided the agent with a reward, but due to the issues that will be explained in section 3.3.3, this step has to be postponed until the next call to `on_end_step`, before a new action is applied. Appropriate care is taken so as to only start storing rewards after the first time step of a trajectory by making use of the aforementioned flag attribute.
- The same steps are taken when a timer expires, so the `on_alarm` method simply calls `on_end_step`.
- The `on_end_trajectory` event is used to reset the state of the environment and the flag, as well as to store the final reward corresponding to the last action in the trajectory. At the same time, a method by the same name in the agent is called to compute the advantages and expected returns for the trajectory.
- The `on_end_simulation` method calls a `training_step` method in the agent to optimize the parameters of the model based on the collected data and saves the resulting model to a file. When the agent is being tested, this functionality is omitted.

3.3.3 Environment

The environment component of the Energy scheduler is implemented in `EnergyEnvironment.py`, under `irmasim/workload_manager/`.

There was an initial attempt to leverage the OpenAI Gym environment interface and its `step` method, which complies with the standard reinforcement learning sequence shown in section 2.2.1, but problems arose when trying to replicate this structure in the workload manager. When an action is applied and the simulator is given the order to schedule the chosen job, this does not happen until *after* the call to the `on_end_step` method in the workload manager has exited. Therefore, if an energy consumption reward were to be computed in the same call, the result would not be accurate since this value depends on the statistics generated by the simulator, which would not have been updated yet. For this reason the `step` method of the Gym environment interface could not be used. The rest of the methods supplied by the interface were not applicable or relevant, so in the end `EnergyEnvironment` was defined as a standalone class with separate methods to get an observation, apply an action and get the reward for the last applied action.

Once this issue has been clarified, the main attributes of the class shall be introduced. Apart from a reference to the simulator, `EnergyEnvironment` keeps a list of the compute nodes of the cluster, a list of pending jobs and a series of dictionaries used to store precomputed values which are later used to craft observations. The current reward function and the observation and action spaces, as well as variables storing markers of the last measured energy consumption and simulation time, are also defined as attributes of the class. Multiple constants are also specified, the most relevant being `NUM_JOBS`, `NUM_NODES` and `NUM_FEATURES`. These are used to define the observation and action spaces making use of the space utilities provided by the Gym API.

`NUM_JOBS` indicates the number of jobs in the queue that are considered for scheduling at any given time, counted from the front of the queue. This constant is necessary because the actor and critic networks of the agent require a fixed-size input, but the length of the pending job queue is variable. `NUM_NODES` is the number of compute nodes in the cluster, and `NUM_FEATURES` is the number of features considered when assigning a score to each job-node pair, shown in table 3.1.

The main methods of the `EnergyEnvironment` class are `reset`, `add_jobs`, `finish_jobs`, `can_schedule`, `energy_estimate`, `get_obs` and `apply_action`.

- The `reset` methods restores the energy and time marker variables and the pending job queue to their default values, whereas `add_jobs` and `finish_jobs` update the state of the queue when jobs are submitted or completed.
- `can_schedule` checks if any node exists with enough free cores to satisfy the demand of any of the first `NUM_JOBS` jobs in the queue, and performs a similar test regarding the maximum energy constraints on jobs. For this, it uses an estimate of the energy that will be consumed by scheduling a job on a particular node obtained through a call to `energy_estimate`. This last method is detailed in subsection 3.3.3: *Energy Estimation*.
- The `get_obs` method crafts a normalized observation based on the current status of the job queue and the platform as described in subsection 3.3.3: *Observations*.
- The `apply_action` method takes an action encoded as an integer, retrieves the corresponding IDs of the chosen job and node and calls on the simulator to schedule the tasks comprising the job on the free cores of the node.

Observations

Observations are crafted by recording a series of features for each possible pairing between a job and a node, following the definition given in [33]. Thus, the observation space is a box of values bounded between 0.0 and 1.0 of shape (`NUM_NODES` $\times$ `NUM_JOBS`, `NUM_FEATURES`).

In particular, the features shown in table 3.1 are considered. The features that have been added with respect to the ones introduced in [33] are highlighted in italic. The *submit time* of a job serves as an indicator of its position in the queue. The *static power* of a node is the sum of the static power of its cores if it is running at least one task. Otherwise, this value is set to the idle power of the node, which is a percentage of its static power. The *dynamic power* of a node is the sum of the dynamic powers of the cores in said node that are currently running a task. *Availability* is defined as the ratio of free cores to total cores in the node and is meant to provide useful information in the presence of memory contention. This information was already present in [33], but in separate features.

Job	Node	Common
Wait time	Availability	*Energy estimate*
Requested time	*Static power*	
Submit time	*Dynamic power*	
Requested cores	Average clock rate	

Table 3.1: Observation features considered for each job-node pair in Energy scheduler.

However, some job-node combinations are invalid, in particular the ones where the number of free cores in the node is not sufficient to allocate all the tasks in the job or where the energy consumption estimate for the given pairing surpasses the maximum energy constraint specified for the job. In these cases all feature values are set to zero, enabling the agent to mask out the corresponding entries when deciding on an action.

Actions

An action refers to a job-node pairing. This way, the action space can be defined by a discrete space of size (`NUM_NODES` $\times$ `NUM_JOBS` + 1), which shall be referred to as `ACTIONS_SIZE`. The added one corresponds to the *wait action*. This special action corresponds to the choice of waiting until the next time step instead of scheduling a job immediately.

Actions are encoded as an index to a job-node matrix relating all the nodes in the cluster to the first `NUM_JOBS` in the queue, as defined in [33]. A specific action index, corresponding to the position `NUM_JOBS` $\times$ `NUM_NODES`, is reserved for the *wait action*. If this action is chosen, a timer is set in the simulator and the method returns. The use of a timer is required because otherwise the simulation might run out of job submission/completion events before the agent decides to schedule the remaining jobs, leading to the simulation ending even though pending jobs still remain in the queue. In principle, the timer is set for one second into the future, but this value might be more or less appropriate depending on the characteristics of the workload.

Reward Functions

Only two objectives are implemented in the Energy scheduler: energy consumption and EDP (Energy-Delay Product, [44]). The corresponding reward functions are included as private methods of the environment class. They return the *negative* increment in energy consumption, multiplied by the time increment in the case of EDP, so as to *minimize* the chosen metric. The increments are obtained using the previously mentioned marker variables, which are updated each time a reward is generated. It should be noted that the sum of the EDP rewards at all time steps is *not* equivalent to the total EDP of the workload, since this value is impossible to calculate beforehand. Instead, these incremental rewards are envisioned to offer guidance during training.

Special considerations have to be made when the use of the *wait action* is enabled, since a truly intelligent agent may realize that the best scheduling strategy to reduce energy consumption is, in fact, to not schedule any jobs at all. Different ways of including a penalty for this action in the reward function were tested, for example making it depend on the number of jobs currently waiting in the queue, the static power of the platform multiplied by a factor, including a penalty only in the final time step of a trajectory, combinations of all of the above and more. However, no satisfactory solution was found as

the proposed alternatives introduced variance and hindered training.

Energy Estimation

The environment provides a convenient method that takes a job ID and a node ID as input and returns an estimate of how much energy will be consumed during the execution of the job on the specified node. This estimate is computed as the product of the estimated *runtime* and *power consumption* when the job is run on the node, taking into account the current status of the cluster. The resulting formula can be expressed as follows:

$$t_{req} \cdot \frac{f_{min}}{f} \cdot \left(\frac{p_s}{j+1} + n \cdot p_{d_i} \right) \tag{3.6}$$

In equation 3.6, $t_{req} \cdot \frac{f_{min}}{f}$ is the *expected runtime*. The t_{req} user-specified job parameter indicates the expected runtime of executing the job on the *slowest node* in the cluster, running at a frequency f_{min}, so this value has to be adjusted to the frequency of the target node, f.

The parenthesized term corresponds to the *power estimate* for the job-node pair. p_s is the static power consumption of the target node, whereas p_j represents the dynamic power consumption of a single core. The static power is distributed among the j jobs that are currently running on the node, plus one considering the target job. Otherwise, the sum of the individual static energy consumption estimates for all the jobs would be greater than their joint estimated consumption. The dynamic power estimate is obtained by adding the dynamic powers of the n cores needed to allocate the n tasks of the target job.

Data structures had to be added to the `EnergyEnvironment` class to keep track of the number of jobs running on each node at any given time, owing to the fact that this information cannot be extracted from the simulator. To be precise, a dictionary keeping a set of assigned jobs for each node was included and updated on job scheduling and completion. To quickly remove a job from one of these sets, a dictionary storing the node assignment for each job was added as well.

3.3.4 Agent

The agent component of the Energy scheduler can be found under `irmasim/workload_manager/agent/`, in `EnergyActorCritic.py`.

The agent class implements the PPO algorithm in an actor-critic paradigm and therefore contains three key attributes: an actor, a critic and a buffer used to store the collected training data. Both actor and critic are deep neural networks and have been defined as nested classes inside the agent class, whereas the buffer has been implemented as a separate class in the same file.

In the constructor these three attributes, together with the actor and critic Adam optimizers[4], are initialized. The actor and critic models and the states of their optimizers may be loaded from a file or initialized randomly. Afterwards, the infrastructure required to implement stochastic weight averaging is set up. When SWA is enabled, models start to be sampled after 75% of the total training epochs have been completed. At this point, the learning rate starts to decrease linearly for 15% of the remaining epochs and remains constant for the rest. In [43], the authors experimented with constant and cyclic learning

[4]In PyTorch, an optimizer implements an optimization algorithm that is applied to the parameters of a particular model.

rates, but the PyTorch SWA utilities provide linear and cosine annealing only. This addition has only been implemented for the actor network, since it is directly responsible for scheduling.

The most relevant methods of the `EnergyActorCritic` class are `decide`, called in each time step to obtain the next action; and `training_step`, called when a simulation ends in order to update the parameters of the model based on the collected training data.

- The `decide` method receives an observation of the state of the environment and returns the action predicted by the actor, the value predicted by the critic for the observed state and the log probability of the chosen action in the probability distribution over actions output by the actor network.
- The `training_step` method was inspired by the implementation provided in [38]. Actor and critic are trained for a number of epochs with independent loss functions. The loss function corresponding to the actor network includes a term corresponding to the entropy of the policy, which may be set to a non-zero value to encourage exploration. In each epoch, the collected training data is sampled in minibatches of a set size. For each minibatch, the standard PyTorch training structure is applied to obtain the gradients of the loss functions and apply a step of the optimizers to the parameters of the networks.

  Even though actor and critic are trained in the same loop, changes in one of them do not affect the other during the same call to `training_step`, since the training data is collected with a single, constant version of each of them. The changes in the models are only reflected when they are used to sample new data during the next simulation.

Actor

The nested `Actor` class contains a deep neural network with six hidden layers. The input layer has a size of `NUM_FEATURES`, and the hidden layers have decreasing dimensions of 16, 16, 8, 8, 4, and 4 respectively, with a final output of dimension 1. Since the shape of an observation is (`ACTIONS_SIZE`, `NUM_FEATURES`), the resulting shape of the raw output is (`ACTIONS_SIZE`, 1). One additional external dimension is added when working with observation batches. The initial weights are normalized to zero mean and unit variance and SELU is used as the activation function between layers, making the actor a self-normalizing neural network.

After forwarding the observation through the network and flattening it to obtain a tensor of dimension (`ACTIONS_SIZE`), a mask is applied to artificially lower the scores of invalid job-node combinations, preventing the model from selecting them in most scenarios. The masked output is then used to initialize a categorical distribution from which an action is sampled. The `forward` method of the nested actor class returns this action, its log probability and the mean entropy of the distribution.

An issue arises when the use of the *wait action* is enabled: since the features corresponding to this action are set to zero in the observation, they would be masked out by the agent and the probability of the *wait action* would be reduced in the policy. This can be prevented thanks to the established convention of identifying the *wait action* by the last index in the action space. Therefore, unmasking the last action is enough for the agent to consider not scheduling.

The `loss` method of the nested actor class computes the PPO loss function for a minibatch

of collected data.

Critic

The nested `Critic` class contains a neural network with a total of ten hidden layers divided into two phases. The input layer is identical to the input layer of the actor network. The following five hidden layers have decreasing sizes 32, 16, 8, 4 and 1. At this point, the result is squeezed, giving a shape of (`ACTIONS_SIZE`) for a single observation. The five following hidden layers further reduce the dimensionality from (`ACTIONS_SIZE`) to (1) in the sequence 128, 64, 32, 16, 8 and 1 for the output layer, resulting in a single numerical value. SELU is also used as the activation function in this network, and the weights are initialized with zero mean and unit variance as well.

The `forward` method returns the raw output of the forward pass of an observation through the network: the estimated value for the given state of the environment.

The loss function is the standard mean squared error (MSE) comparing the prediction supplied by the network to the expected return values collected during training.

Buffer

The `buffer` attribute of the agent class stores the necessary values after each time step so as to later compute the losses and train the actor and the critic. Specifically, it keeps separate arrays for observations, actions, rewards, values, action log probabilities, advantages and expected returns. The first five are stored after each time step, whereas the last two are computed after each complete trajectory as discounted sums based on the collected data, using GAE in the case of the advantages. To extract the data corresponding to the last trajectory, the buffer keeps an index indicating its first time step, as well as the index of the last stored value.

When a simulation finishes, the content of the buffer can be retrieved through a call to `collect`. This method resets the buffer indices and normalizes the advantages before returning the collected training data.

Special considerations have to be made when the use of the *wait action* is enabled. In this case, the number of time steps in a trajectory cannot be known before the buffer is constructed, since it will not necessarily match the number of jobs to be scheduled. Therefore, the buffer has to be capable of supporting variable trajectory lengths. This can be achieved in two ways: a first option involves increasing the capacity of the buffer up to a fixed upper bound, whereas a second option would be to replace the arrays with lists. The former wastes memory but offers better performance, while the latter comes at a cost in performance but makes a more efficient use of memory.

The buffer implements the first alternative by reserving the originally required array length multiplied by a constant. This choice follows from the fact that memory is not scarce in modern-day systems and that the computational efficiency of NumPy arrays is desirable when computing advantages and expected returns. However, it is difficult to estimate what the upper bound of the size of the buffer should be. In theory, the agent could wait indefinitely, up to an infinite number of time steps. In this case any established bound would make the training process fail due to lack of space if the agent waited more times than expected. Since the actor may decide that not scheduling is the most energy-efficient strategy, this can be an issue in practice, but the same problem would persist if lists were used instead.

Chapter 4

Scheduler Evaluation

This chapter exposes the experiments that have been carried out to validate the correct operation of the newly implemented Energy scheduler and, at the same time, identify possible challenges and areas of improvement. First, two simple examples with synthetic workloads and platforms are employed to comprehend how decision-making takes place and to identify possible shortcomings in the design (section 4.2). Afterwards, the Energy scheduler is put to the test with more complex synthetic examples to analyze its generalization and scaling capabilities (section 4.3). The results obtained by the Energy scheduler in these tests are compared against the reference values generated by a series of heuristic algorithms for two metrics: energy consumption and EDP.

4.1 Methodology

The evaluation process has been supported by the IRMaSim cluster simulation framework, which makes testing possible without the need to deploy the new scheduler in a real HPC environment [45]. IRMaSim provides an infrastructure that is capable of modelling heterogeneous clusters based on multicore architectures according to the hierarchical cluster model shown in figure 4.1, where the resources at each level do not necessarily share the same specifications.

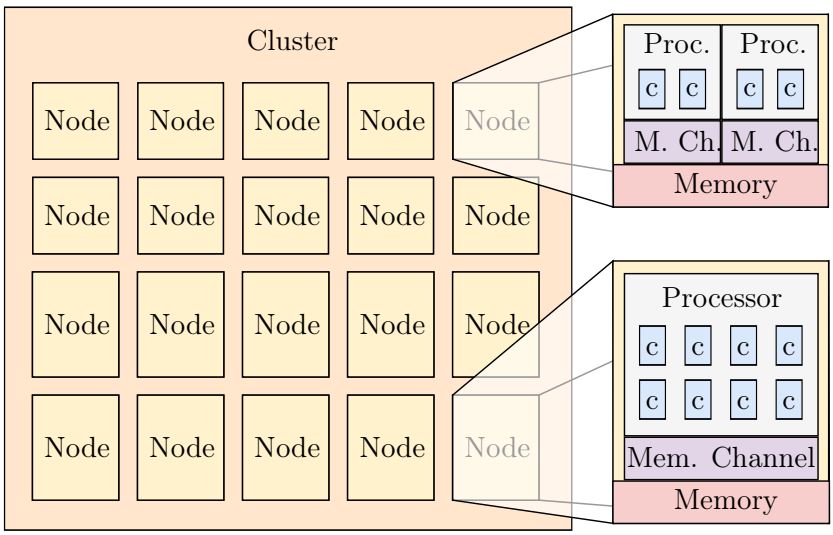

Figure 4.1: Heterogeneous cluster architecture in IRMaSim.

IRMaSim can simulate memory contention at the node level, but at the time of writing it does not, however, implement the interconnection network of a cluster. For this reason, parallel applications that involve inter-process communication while running on different nodes cannot be simulated realistically, as no penalties are introduced for network contention and traffic volume. Consequently, these kinds of jobs have not been considered in this work.

In order to execute a simulation using IRMaSim, separate JSON files have to be created specifying the workload and the target platform (or *cluster*).

Workload definition A workload in IRMaSim is defined as a sequence of jobs, each complete with its own set of user-defined parameters. The most relevant are the following:

- *Submit time (s)*: timestamp signaling the arrival of the job.
- *Requested cores* (or *tasks*): number of cores needed to execute the job.
- *Requested time (s)*: expected execution time of the job, estimated by the user as the runtime when executing the job on the slowest node in the platform.
- *Requested operations*: number of instructions that the job will run.
- *Requested memory (MB)*: memory requested by the job.
- *Memory volume (MB/s)*: parameter indicating the volume of memory instructions in the job.
- *Maximum energy consumption (J)*: maximum amount of energy that may be consumed by the job.

Some of these parameters are meant to be used by the simulator to accurately estimate the progress of the running processes, in particular *submit time*, *requested operations* and *memory volume*; whereas the rest exist to provide useful information for the scheduler exclusively.

In this work, the memory requested by each job is not relevant, but this parameter is required. Therefore, it has been set to a fixed value of 100MB in all cases. The maximum energy constraint has not been used in the experiments, but its implementation has been verified at development time.

Job profiles may be used to specify multiple jobs with the same characteristics.

Platform definition A platform in IRMaSim consists of a series of nodes, each with one or more processors that themselves are composed of multiple cores. Resource characteristics are specified at the processor level. The main modifiable parameters are the following:

- *Clock rate (GHz)*: average working frequency of the processor.
- *Cores*: number of cores in the processor.
- *Dynamic power (W)*: dynamic power consumed by a single core while executing a job. In a real processor, dynamic power is expressed as $P_d = 0.5 \cdot C_L \cdot V^2 \cdot F$.
- *Static power (W)*: static power consumed by the hardware of the processor that is shared by all of its cores. In a real processor, $P_s = V \cdot I$.
- *Minimum power (%)*: when the processor is idle, its power consumption is equivalent to its static power multiplied by this factor.

The static and dynamic power consumption parameters used in the experiments have been estimated based on a regression model trained on data collected from real Intel processors.

The following paragraphs detail the pursued scheduling objectives, as well as the heuristic algorithms that have been used as points of comparison to evaluate the performance of the Energy scheduler.

Scheduling objectives Two objectives are considered in the proposed experiments: minimizing either energy consumption or EDP.

- *Energy consumption* measures the sum of the energy consumed by each of the individual nodes in the platform over the time period required to finish executing the workload.
- *EDP* multiplies the energy consumption metric described above by the total execution time of the workload.

Heuristic algorithms To find the best-performing heuristic for each of the proposed experiments, another intelligent scheduler developed in IRMaSim by the name of "Policy scheduler" has been used [45]. This workload manager schedules jobs in an intelligent manner according to the selected objective by choosing the best *job selection policy* and the best *resource selection policy* for the target platform and workload. The available policies are given in table 4.1.

Job Selection Policies	
random	Any job
first	Oldest job in the job queue
shortest	Job with the lowest expected execution time
smallest	Job with the least requested cores
low_mem	Job with the lowest requested memory
low_mem_ops	Job with the lowest memory access volume
Resource Selection Policies	
random	Any resource
high_gflops	Resource with the highest peak compute capability
high_cores	Resource with the most currently available cores
high_mem	Resource with the highest currently available memory
high_mem_bw	Resource with the highest currently available memory bandwidth
low_power	Resource with the lowest power consumption

Table 4.1: Job and resource scheduling policies in the Policy workload manager.

The Policy scheduler is capable of distributing the tasks of a single job among cores in separate nodes, but since network communications cannot currently be simulated realistically in IRMaSim, and since the Energy scheduler does not consider this possibility, the Policy scheduler was modified accordingly in order to provide a fair comparison during testing.

Four main heuristics implemented in the Policy scheduler have been applied to the designed experiments to serve as reference points for the performance of the Energy scheduler, namely *first - high_gflops*, *first - high_cores*, *first - high_mem_bw* and *random - random*. In all cases, the most fitting heuristic has also been retrieved using the Policy scheduler and used for comparison.

All the experiments have been executed in a workstation equipped with an AMD Ryzen 5 3600 6-core processor with 12 threads and 32MB of L3 cache, running at a variable

frequency between 3.6 and 4.2 GHz. This system is also provided with 16 GB of RAM memory, an NVIDIA GeForce RTX 3060 12GB graphics card and Ubuntu 20.04.6 LTS as a native operating system.

4.2 Experimental Validation

This section explores the capabilities of the Energy scheduler regarding scenarios of interest that consider memory contention and heterogeneous platforms where not all nodes consume the same amount of power. With this objective in mind, two basic, synthetic experiments have been tested. A first experiment considers a workload with two types of jobs and a platform with two nodes running at the same frequency, but with a different number of cores. A second experiment adds more heterogeneity in terms of job profiles and nodes.

4.2.1 Experiment 1

This first synthetic experiment has been devised to analyze whether the new scheduler is capable of taking appropriate decisions when the main issue penalizing energy consumption (or EDP) is an increased execution time derived from memory contention. For this reason, nodes with the same power specifications but with a different number of cores, and hence different bandwidths, are used.

In addition, this same scenario is also employed to test the impact of allowing the workload manager to wait instead of scheduling a pending job whenever enough resources are available. This is achieved by enabling the usage of the implemented *wait action*.

Configuration

The complete configuration for the execution of this first experiment is gathered in tables 4.2, 4.3, 4.4 and 4.5.

Node	Node 0	Node 1
Cores	4	8
Clock rate (GHz)	2.5	2.5
Dynamic power (W)	2.3	2.3
Static power (W)	24.38	24.38
Min. power (%)	5	5

Table 4.2: Experiment 1 - Platform.

Profile	A	B
Req. cores	4	2
Req. ops. ($\times 10^{10}$)	1.25	6.25
Req. time (s)	5.5	25
Mem. vol. (MB/s)	1	10^4

Table 4.3: Experiment 1 - Job profiles.

Time (s)	0.00	0.05	0.10	0.15	0.20	0.25
Job ID	J0	J1	J2	J3	J4	J5
Profile	A	A	B	A	B	A

Table 4.4: Experiment 1 - Workload.

The employed platform, described in table 4.2, consists of two nodes with a single processor each, both running at the same frequency. The regression model used to estimate the power consumption specifications of the resources strongly depends on processor frequency, so the resulting values for static and dynamic power consumption are the same. This way, the only difference between the nodes is the number of cores of their processor and, therefore, their maximum bandwidth.

Simulations	100	Seed	40
Trajectories	3	ϵ	0.1
Trajectory length	6	γ	0.99
Minibatch size	18	λ	0.95
Epochs	50	Actor lr	0.001
SWA	No	Critic lr	0.001

Table 4.5: Experiment 1 - Training hyperparameters.

Two types of jobs, defined in table 4.3, are considered. The B profile models a heavy job that accesses memory frequently and is almost five times more computationally intensive than profile A, which is also lighter in terms of memory accesses. However, each A job requires four cores to allocate its four tasks, whereas a B job only requires two. The platform is faced with a workload of six jobs that arrive in fifty-millisecond intervals, following the sequence shown in table 4.4.

Table 4.5 shows the hyperparameters that have been used to train the agent. This simple experiment attempts to overfit a particular example in order to explore the capabilities of the scheduler, so the three sampled trajectories correspond to the same sequence of six jobs. For this same reason, SWA has been disabled. By scheduling the same trajectory more than once, more data is made available to evaluate the loss of the current policy, since different runs may produce different schedules from which useful information can be extracted. The minibatch size has been set to average over the data collected during all three trajectories.

The PPO clipping factor ϵ has been made small in order to prevent excessive policy fluctuation during training, considering that even slight changes in the policy can have a significant impact on the reward. The remaining hyperparameters have been modified through trial and error until arriving at a configuration that seemed to work well in the general case.

Energy Consumption Objective

The Energy scheduler has first been trained for the energy consumption objective without enabling the use of the *wait action*. Figure 4.2 illustrates the evolution of the rewards and the policy and value losses during training.

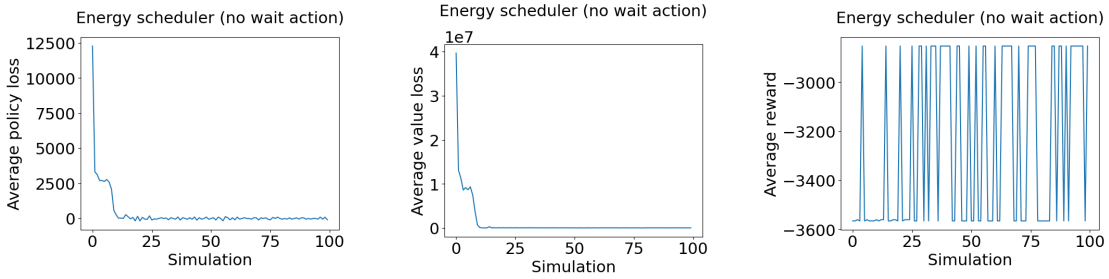

Figure 4.2: Experiment 1 - Loss and reward plots for the energy consumption objective, *wait action* disabled.

The loss plots in figure 4.2 show that convergence can be achieved quite smoothly for both actor and critic, although this is not always the case: several executions were run before selecting the one that gave rise to the most promising model. Regarding the reward plot, no clear progress can be appreciated past the first twenty-five simulations. Instead, the

actor oscillates between two scheduling alternatives and does not settle on a single solution, even though one of them offers a considerable reduction in energy consumption.

The two scheduling configurations obtained by the resulting policy are shown in figures 4.3 (configuration 1) and 4.4 (configuration 2). In the figures, jobs are segregated by their assigned node and appear in the order in which they have been scheduled, from top to bottom. The waiting time of each job has been highlighted by reducing the opacity of the corresponding color.

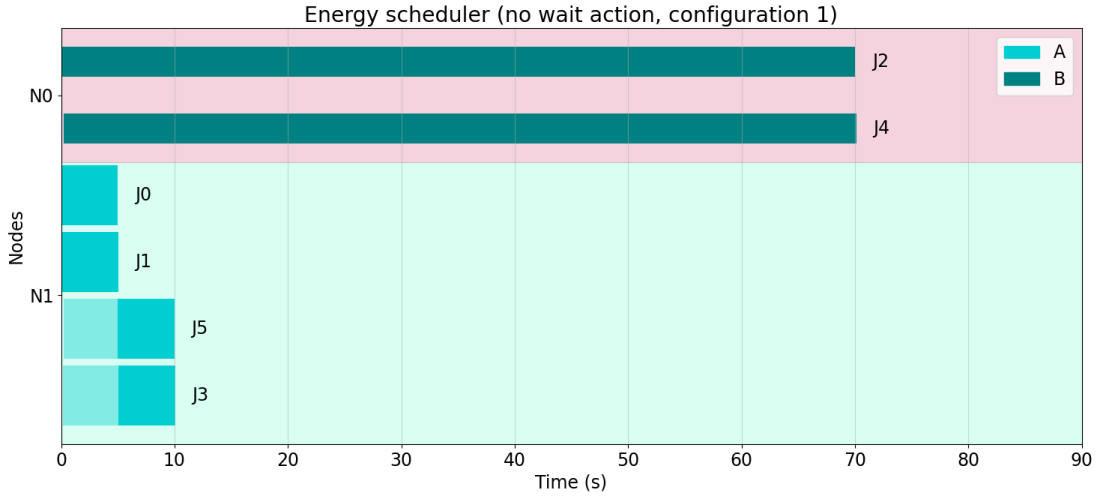

Figure 4.3: Experiment 1 - Configuration 1 obtained by the Energy scheduler for the energy consumption objective, *wait action* disabled.

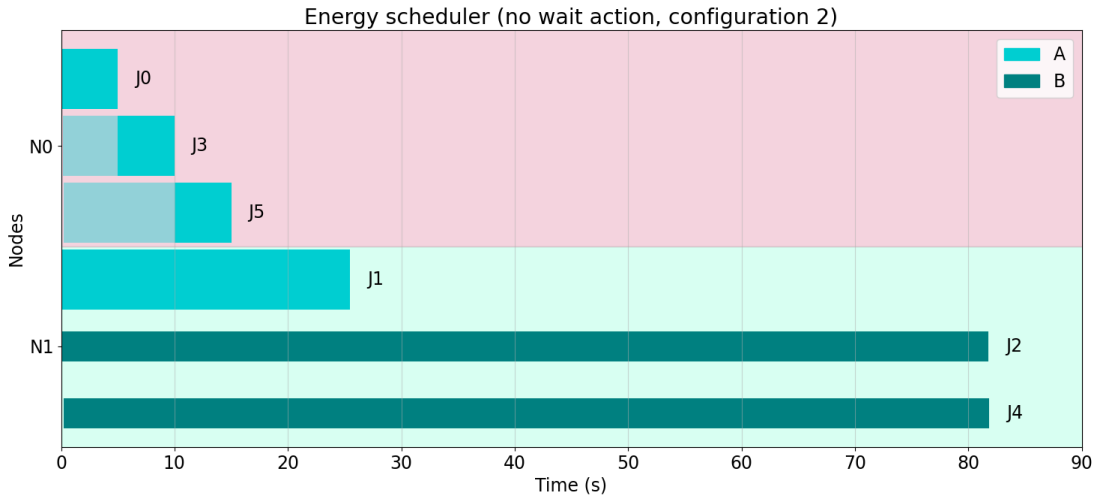

Figure 4.4: Experiment 1 - Configuration 2 obtained by the Energy scheduler for the energy consumption objective, *wait action* disabled.

Configuration 1 results in an energy consumption of approximately 2853.83 J, whereas configuration 2 consumes around 3564.94 J. Comparing both figures brings to light that the increased memory contention, however low, introduced in N1 by J1 in configuration 2 is enough to increase the total execution time by more than ten seconds with respect to configuration 1. Conversely, executing the two B profile jobs on the same node without additional contention in configuration 1 reduces execution time and hence energy consumption, both static and dynamic, on N0. The static energy consumption in N1 is also reduced

in this configuration, since the four scheduled jobs finish early and the node only consumes a very low fraction of its static power while idle.

As to how the intelligent scheduler is capable of reaching these conclusions, it is hard to provide an answer when the employed models are deep neural networks. Still, it is safe to say that the scheduler has based its decisions on the features considered in table 3.1.

Different hypotheses can be elaborated in an attempt to explain why the scheduler cannot seem to converge to a single configuration for such a simple example. A relevant factor might be that the agent has no access to any information regarding the memory volume of a job, a key feature that could have made it possible to obtain a stable policy in this case, although not with certainty. Another plausible explanation could be that the provided training data was too scarce or not representative enough.

For reference, the schedule found by the best heuristic, which assigns the job with the lowest memory access volume to the node with the highest currently available memory bandwidth, is shown in figure 4.5 and results in a a slightly higher energy consumption of 3575.07 J.

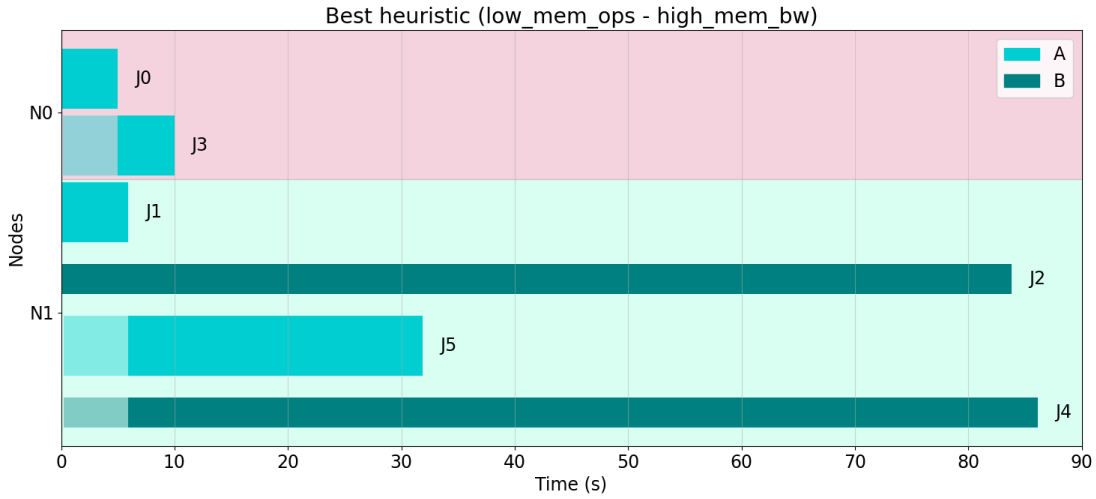

Figure 4.5: Experiment 1 - Configuration obtained by the best heuristic (*low_mem_ops - high_mem_bw*) for the energy consumption objective.

To understand the resulting configuration, it should be noted that the Policy scheduler measures available bandwidth as the sum of the requested bandwidths of the jobs currently running on the target node. Due to a tie, J0 was assigned to N0 instead of N1. Afterwards, although J4 could have been scheduled at the same time as J1 and J2 on N1 (as in figure 4.4), the scheduling policy prevented this from happening since J3, with a lower memory access volume, was still in the queue and no resources were available to satisfy its demand of four cores. Consequently, the rest of the jobs in the queue were stalled until N0 was freed and J3 was scheduled, increasing the total execution time and energy consumption.

This shows that even if the chosen heuristic was conceptually correct, a more intelligent approach can be more appropriate.

Enabling the Wait Action

The solutions found by the Energy scheduler in the previous section were restricted by the need to schedule an incoming job whenever possible. To unlock new scheduling combina-

tions, the same experiment has been tested enabling the use of the *wait action*. When this option is enabled, the scheduler may wait even when it is possible to schedule a new job. Otherwise, the scheduler is forced to wait only when no resources are available.

Since the use of the *wait action* may increment the trajectory length, the minibatch size was increased to sixty-four in order to capture the three simulated trajectories at once in most cases. At the same time, the clipping factor ϵ was set to 0.2 in order to allow for greater changes in the policy and discourage stagnation. The plots of the resulting actor and critic losses, as well as the evolution of the rewards during training, is shown in figure 4.6.

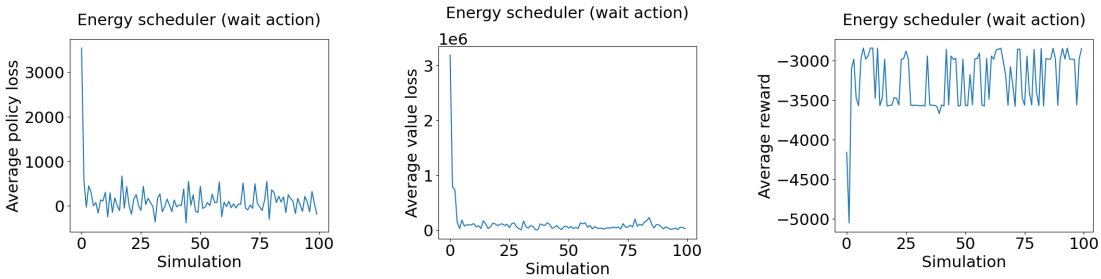

Figure 4.6: Experiment 1 - Loss and reward plots for the energy consumption objective, *wait action* enabled.

The policy and value loss plots in 4.6 do not seem to have converged as smoothly as in the previous subsection at a first glance, but it should be noted that the scale of the y-axis is relatively small when compared to the plots in figure 4.2. In contrast, more variability can be observed in the reward plot. This is logical, since introducing the option of not scheduling at every time step greatly increases the scheduling configuration space. Again, the rewards do not seem to increase steadily during training, fluctuating within a particular range instead. This might be related to the factors mentioned in the previous subsection.

One of the best schedules obtained during testing when employing the *wait action* is shown in figure 4.7, with an energy consumption of 2737.42 J.

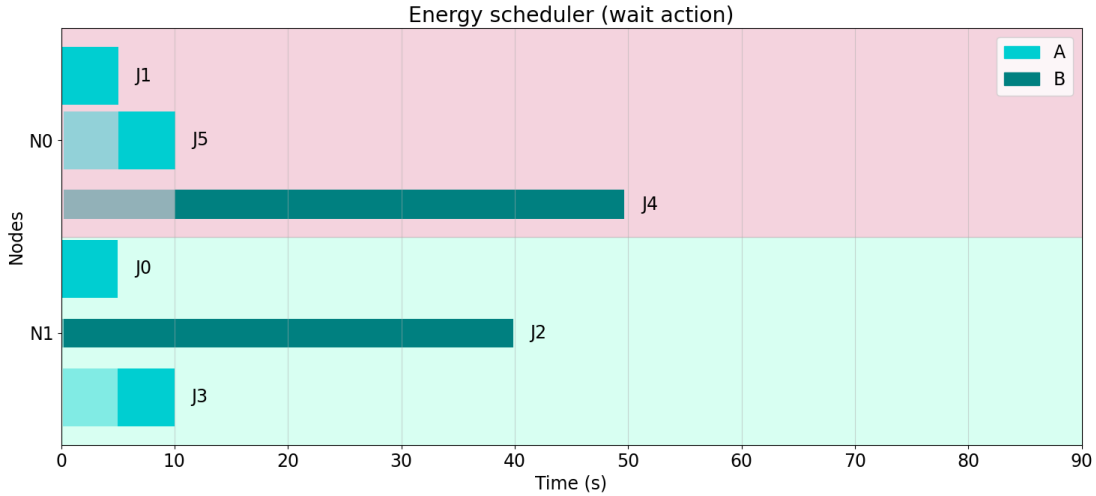

Figure 4.7: Experiment 1 - Best configuration obtained by the Energy scheduler for the energy consumption objective, *wait action* enabled.

In this case the scheduler is capable of distributing the two heavy tasks in different nodes, thus reducing memory contention. With the *wait action* disabled, J0, J1, J2 and J4 would

be scheduled in this order, while J3 and J5 would be forced to wait due to a lack of available cores. Then, they would be scheduled immediately after one of the lighter jobs had finished. This is precisely what happens in figures 4.3 and 4.4. In figure 4.7, however, the six jobs are instead scheduled in the order J0, J1, J2, J3, J5, J4 by waiting during the arrivals of J2, J3 and J4. In particular, J2 is scheduled when J5 arrives, whereas J3 and J4 are scheduled as the running jobs finish executing. This makes it possible for the scheduler not to immediately schedule J4 on N1 even though enough cores are available to allocate its two tasks, and instead wait until N0 is freed to execute both heavy jobs on different nodes.

Although the difference in energy consumption with respect to the best schedule found when disabling the *wait action* is not significant, execution time is reduced from 70 s to around 50 s, which has a noticeable effect on EDP. Still, the scheduler waits more than necessary in some cases, introducing slight delays. This could be mitigated by adjusting the timer.

EDP Objective

The Energy scheduler has also been trained for the EDP objective. No changes in the policy were discovered with respect to optimizing the energy consumption objective when the *wait action* was disabled, or when training the Policy scheduler to obtain the most appropriate heuristic.

However, the previous subsection showed that enabling the use of the *wait action* unlocks configurations capable of drastically reducing EDP. In theory, training the agent to optimize the EDP objective should help it converge to one such policy. Nevertheless, this was not the case upon evaluating this scenario, as the agent tended to stagnate on worse policies or fluctuate considerably during training.

An example of the evolution of the loss and reward plots during a training execution with $\epsilon = 0.1$ can be found in figure 4.8. The rewards oscillate between the values corresponding to the two policies exposed in figures 4.3 and 4.4 even though much lower EDP values can be achieved, as indicated by the peaks in the reward slightly before simulation thirty. Increasing ϵ to 0.2 prevents this stagnation and lets the agent explore better policies, but makes convergence harder to achieve. This is illustrated in the loss and reward plots in figure 4.9, which are representative of a standard training execution using this setting. Even though the scale of the y-axis is smaller, the value of the losses does not decrease towards the end as in figure 4.8.

It should be noted that the EDP reward is *not* equal to the total EDP but rather to the sum of the EDP calculated at discrete time steps. Consequently, intermediate rewards may not be providing the agent with enough guidance. This could also be a factor involved in the instability experienced during training.

Comparison

The energy consumption and EDP statistics collected throughout one hundred test iterations with and without the use of the *wait action* have been compared to the results obtained by the heuristics mentioned in section 4.1 for the same platform and workload. Strictly speaking, testing the proposed scheduler on the workload used for training only serves as an indicator of how the model has overfitted the training set, and not as validation. But this is precisely the goal of this initial experiment.

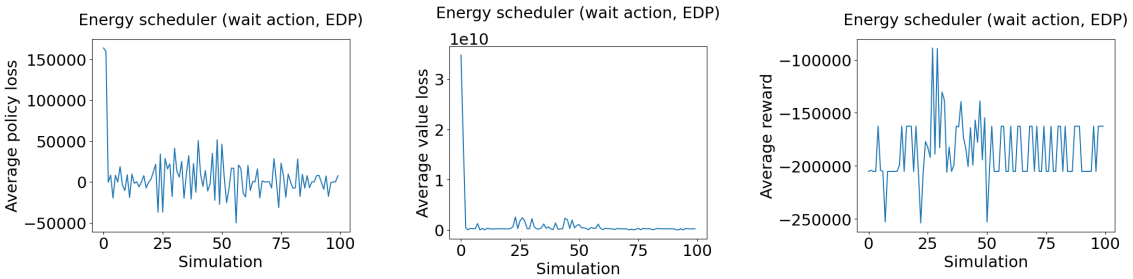

Figure 4.8: Experiment 1 - Loss and reward plots for the EDP objective, *wait action* enabled, $\epsilon = 0.1$.

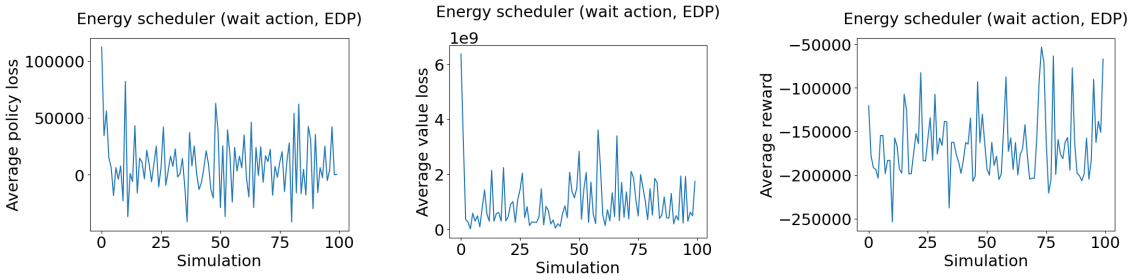

Figure 4.9: Experiment 1 - Loss and reward plots for the EDP objective, *wait action* enabled, $\epsilon = 0.2$.

The resulting energy consumption and EDP statistics are shown in figures 4.10 and 4.11, where the mean and standard deviation are plotted, as well as the minimum and maximum values (white and black triangles) obtained during the testing of the Energy scheduler. The bars plotted for the *wait action* alternatives correspond to the values obtained after training with $\epsilon = 0.2$, for the energy consumption and EDP objectives respectively. Table 4.6 gives the percentage reductions in both metrics with respect to the best heuristic for the minimum, maximum and average values obtained by the Energy scheduler in the three analyzed cases.

In the worst case scenario, the Energy scheduler obtains an energy consumption similar to that of the best heuristic, but is capable of providing a reduction of between 9% and 14% on average, and 20% in the best case scenario. When the *wait action* is disabled, the scheduler fluctuates between the two schedules introduced in subsection 4.2.1: *Energy Consumption Objective*, which explains the deviation on the corresponding bar in figure 4.10. Enabling the *wait action* does not make much of a difference regarding this metric.

When comparing EDP on the other hand, the best schedule obtained when the *wait action* is enabled more than halves the EDP obtained by the heuristics, as opposed to a 35% reduction in the best case when this option is disabled. On average, both configurations reach a reduction of between 20% and 30% in EDP.

In general, the Energy scheduler does not consistently offer better schedules for this objective but instead fluctuates considerably. The standard deviation is higher than for energy consumption since EDP combines this metric with execution time, which also varies between executions. And, in the case of the *wait action*, this is further aggravated by the introduction of delays that do not follow a specific pattern. Figure 4.11 also evidences that optimizing for the EDP objective instead of for energy consumption when the *wait action* is enabled has not led to better results in this particular scenario.

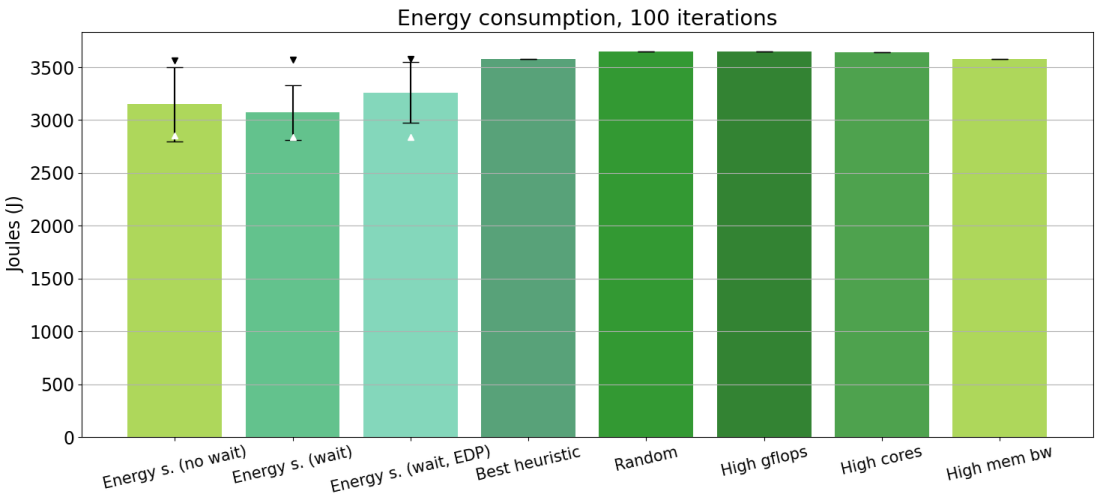

Figure 4.10: Experiment 1 - Energy consumption comparison.

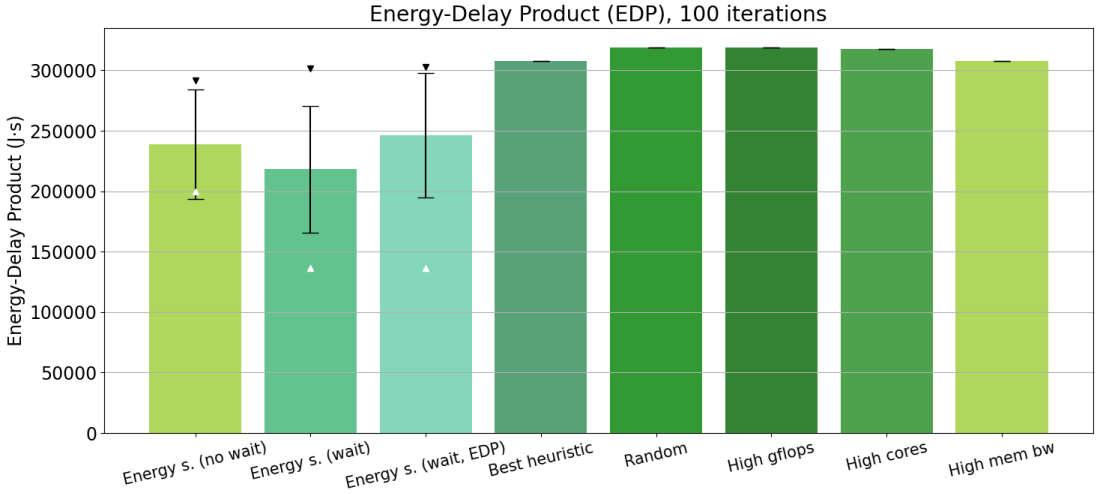

Figure 4.11: Experiment 1 - EDP comparison.

Percentage reduction w.r.t. Best heuristic	*Energy consumption*			*EDP*		
	Min	*Avg*	*Max*	*Min*	*Avg*	*Max*
No wait action	20.20	11.84	0.28	35.04	22.54	5.26
Wait action, Energy consumption objective	20.58	14.05	0.03	55.58	29.13	2.20
Wait action, EDP objective	20.58	8.80	-0.09	55.58	20.01	1.77

Table 4.6: Experiment 1 - Percentage reductions in energy consumption and EDP obtained by the Energy scheduler with respect to the best heuristic.

Discussion

This first experiment showcases the complexity of the scheduling problem even for a small example. Although the Energy scheduler has been capable of finding some good solutions, especially when enabling the use of the *wait action*, difficulties have been encountered upon training and consistent results at test time have been hard to achieve. This could be explained by the fact that the scheduler has no way of directly estimating the memory access volume of a job.

Still, a different conclusion might be reached after training the agent in a more complex scenario, with more training data and different hyperparameters. At the same time, this approach has been shown to be more suitable than simple heuristic scheduling despite these issues, indicating that the implemented scheduler behaves as intended.

4.2.2 Experiment 2

A second, more complex experiment has been devised that considers resources with distinct compute capabilities and static and dynamic power consumption specifications. The target workload has also become slightly more varied, including three job profiles with increasing intensities in terms of memory access and requested time.

In this synthetic scenario, the cluster is first filled with light jobs that occupy all the available cores. Afterwards, a series of heavier jobs are submitted. The scheduler should be capable of predicting which of the light jobs will finish first according to the average frequency of their assigned node, their submit time and the level of memory contention, enabling it to plan ahead so that the most suitable nodes are available when the heavy jobs arrive. At the same time, it should discover what these most suitable nodes are in order to minimize total energy consumption or EDP.

Configuration

The complete specification of the platform, workload and training hyperparameters employed in this experiment can be found in tables 4.7, 4.8, 4.9 and 4.10.

Node	Node 0	Node 1	Node 2	Node 3
Cores	8	16	32	48
Clock rate (GHz)	4.2	3.8	3.4	3.0
Static power (W)	68.81	56.33	45.09	35.11
Dynamic power (W)	6.49	5.32	4.26	3.31
Min. power (%)	39.59	39.59	39.59	39.59

Table 4.7: Experiment 2 - Platform.

Profile	A	B	C
Req. cores	8	8	4
Req. ops ($\times 10^{10}$)	1.375	6.25	12.5
Req. time (s)	4.6	20.8	41.7
Mem. vol. (MB/s)	1	10^3	10^6

Table 4.8: Experiment 2 - Job profiles.

Time (s)	0.00-0.65	0.70-0.75	0.80-0.85
Job ID	J0-J13	J14-J15	J16-J17
Profile	A	C	B

Table 4.9: Experiment 2 - Workload.

Simulations	100	Seed	1234
Trajectories	5	ϵ	0.1
Trajectory length	18	γ	0.99
Minibatch size	45	λ	0.95
Epochs	50	Actor lr	0.001
SWA	No	Critic lr	0.001

Table 4.10: Experiment 2 - Training hyperparameters.

The proposed platform, shown in table 4.7, consists of a total of four nodes, each with an increasing number of cores from 8 up to 48. As the number of cores increases, their clock

rate decreases and so does their static and dynamic power consumption. This way, larger nodes may allocate more jobs and have a greater bandwidth but execute tasks more slowly, and vice versa.

Table 4.8 describes the three job profiles considered in this experiment, ordered from least to most intensive in terms of both requested time and memory access. The number of requested cores per job has been increased with respect to Experiment 1 according to the capacity of the new platform. Again, the most intensive profile requires less cores. These kinds of jobs are submitted in the order given in table 4.9, in fifty-millisecond intervals. The platform is first saturated with A jobs before one more A job and two intensive C jobs arrive. Finally, another two B jobs of intermediate intensity have to be scheduled.

The hyperparameters used to train the agent for this example can be found in table 4.10. The complete sequence of eighteen jobs is scheduled five times using the same policy and baseline state values before using the collected data to update the actor and critic networks. The minibatch size has been set to cover half of the total data at a time so that not much information is lost by averaging over the whole dataset, while at the same time achieving reasonable training times. The actor and critic learning rates shown in the table are appropriate for this example, although others were found to provide similar results.

Energy Consumption Objective

The Energy scheduler has first been trained with the objective of minimizing energy consumption. The evolution of the rewards and the average actor and critic losses during one of the training executions is plotted in figure 4.12.

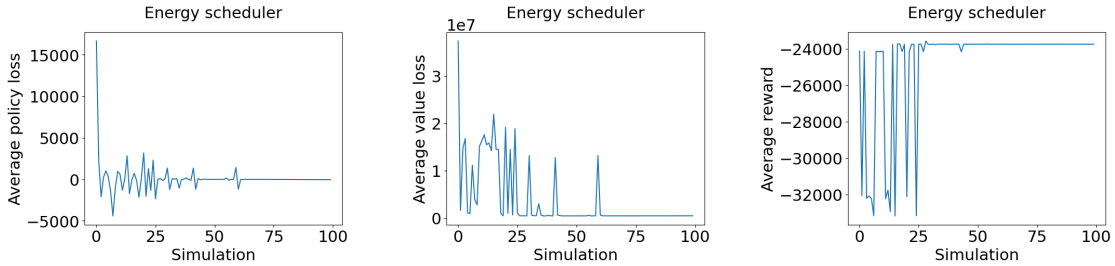

Figure 4.12: Experiment 2 - Loss and rewards plots for the energy consumption objective.

The loss plots reflect some peaks that are shared between actor and critic. This shows that, when the value predicted by the critic is not accurate, the policy chosen by the actor is affected since it cannot correctly predict the advantage of the selected actions. Nevertheless, in this execution the actor managed to recover quickly thanks to the properties of the PPO algorithm. At the same time, it is clear that the fluctuations in the policy have affected the rewards, especially during the first quarter of the execution.

After training, the Energy scheduler manages to define a policy that is almost deterministic with an energy consumption of around 23743 J, shown in figure 4.14, . This is not, in fact, the optimal configuration for this problem, as will be exposed later. It should also be noted that no better policy was found after several attempts at training the agent with these settings. Still, this value is not far from the lowest achievable energy consumption, especially when compared to the results obtained by the most appropriate heuristic.

This heuristic assigns the job with the least requested cores to the node with the highest available memory bandwidth, resulting in the configuration shown in figure 4.13 and an energy consumption of around 32090 J. In figure 4.13, the jobs up to J12 are scheduled on

the nodes in order, filling all the cores in N0 up to N3. While these jobs execute, the rest arrive and are forced to wait. J0 finishes first, freeing N0, and the C profile jobs in the queue (J14 and J15, with only two requested cores each) are immediately scheduled. The rest of the A jobs finish in submit order as they execute the same number of instructions and smaller nodes have a higher clock rate. This way, J13, J16 and J17 are scheduled as N1 and N2 are freed since the heuristic is not capable of waiting until more suitable resources become available.

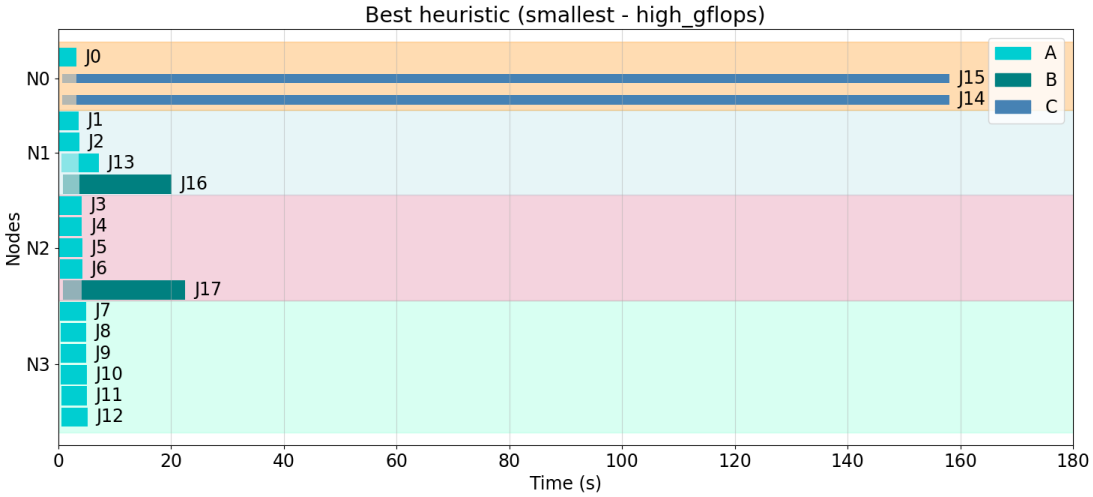

Figure 4.13: Experiment 2 - Configuration obtained by the best heuristic (*smallest - high_mem_bw*) for the energy consumption objective.

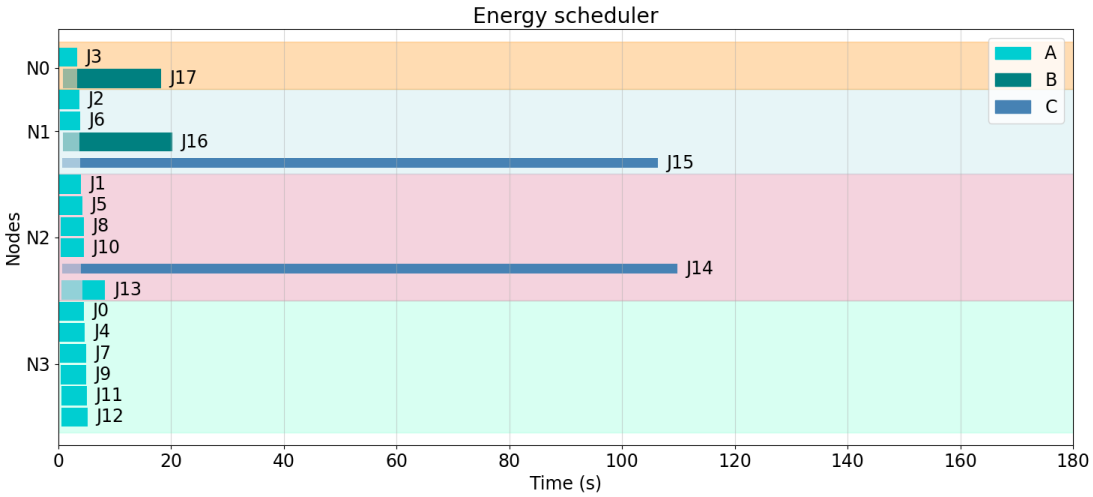

Figure 4.14: Experiment 2 - Configuration obtained by the Energy scheduler for the energy consumption objective.

But consumption can be reduced without needing to wait. In this situation, the Energy scheduler (figure 4.14) assigns the medium-intensity B jobs to the fastest nodes and distributes the high-intensity C jobs in two different nodes, also the fastest available, so as to reduce memory contention. Scheduling J15 and J16 on the same node when both make considerable use of memory does not seem optimal, however. J13 on N1 would have been a better choice due to its low memory access requirements and shorter requested time, which would have reduced the execution time of J15.

Since the *wait action* has not been enabled and the jobs assigned to N3 never finish in time to allocate one of the B or C jobs, it remains to be tested whether this allocation scheme would be more advantageous. Two reasons explaining why this could be the case are the following:

- In general, nodes with a higher bandwidth are more appropriate for scheduling jobs with a high memory access volume, as long as the node is not overcrowded.
- In this particular scenario only four heavier tasks have to be distributed among four nodes, so memory contention could be palliated by avoiding assignments to the same node.

Consequently, the experiment was repeated enabling the use of the *wait action*, but no better configuration was found throughout several executions. It is speculated that the delays introduced by waiting might increase static energy consumption more than it is reduced by redistributing the heavy jobs.

Removing Energy-Related Features

The same experiment has been carried out after removing the energy estimate and power consumption specifications of the resources from the features considered by the agent for decision-making (table 3.1). No significant difference was observed and the agent showed convergence patterns similar to the original version throughout several executions, but a better scheduling configuration was found.

The corresponding plots of the evolution of the losses and rewards in this execution are shown in figure 4.15. The reward plot clearly reflects how the agent converges to a policy that results in an energy consumption of around 23000 J, slightly lower than the one obtained when considering the energy-related features in the observation. Figure 4.17, in the next subsection, is representative of the schedules obtained by this policy.

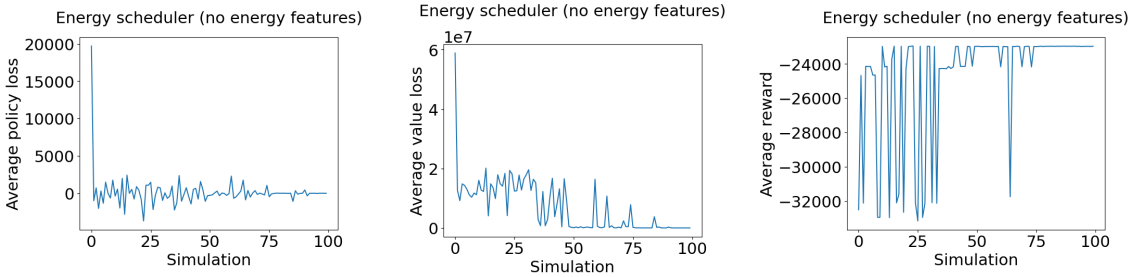

Figure 4.15: Experiment 2 - Loss and rewards plots for the energy consumption objective without considering energy-related features.

However, a different conclusion might have been reached had the power specifications of the nodes not depended as heavily on their frequency: static and dynamic power are formally defined as $P_s = V \cdot I$ and $P_d = 0.5 \cdot C_L \cdot V^2 \cdot F$ respectively, but in this case the approximation introduced in section 4.1 was used. Still, it is true that frequency has a considerable impact on power consumption, since $V \propto F$. It remains to be seen whether there would be any considerable difference if the power estimates were to be modelled in a more realistic manner.

EDP Objective

Similar results were observed when optimizing the EDP objective. After multiple executions, the agent managed to converge to a rather stable policy. The evolution of the actor and critic losses and the rewards during the corresponding training execution can be found in figure 4.16.

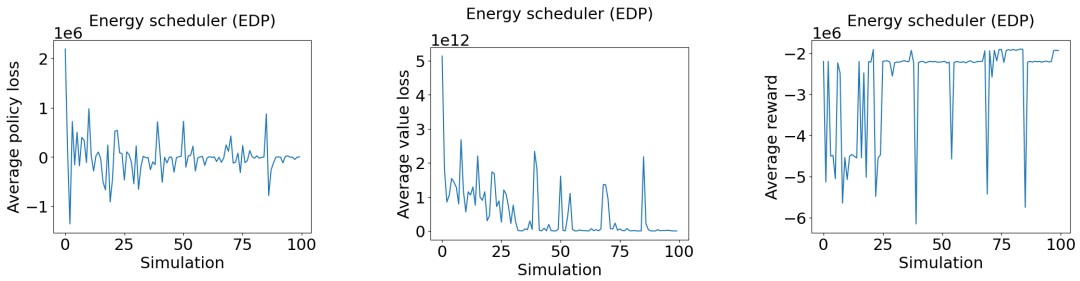

Figure 4.16: Experiment 2 - Loss and reward plots for the EDP objective.

It should be noted that the hyperparameters were altered slightly in an attempt to ease convergence after observing the patterns found in previous executions. In particular, the resulting plots were obtained when reducing the actor learning rate to 0.0005 and increasing the number of trajectories to ten. It is theorized that providing the Energy scheduler with more training data and reducing the step size is appropriate when the magnitude of the loss function is greater. Nevertheless, the aforementioned plots illustrate that, overall, convergence is harder to achieve for EDP than for energy consumption owing to an increase in variance, regardless of these modifications. At the same time, it is not certain that the chosen hyperparameters have in fact aided with convergence, since randomness plays an important role in this process as well.

The evolution of the rewards shows that a better policy is attained only in the last few simulations. The fact that the policy is not maintained throughout a greater number of simulations does not imply that the result is not stable, as will be shown when comparing the EDP obtained at test time in figure 4.19.

A representative configuration obtained by the resulting policy is shown in figure 4.17. This schedule results in an energy consumption of around 23000 J and an EDP of approximately 2260000 J·s, which are the same as the values obtained when minimizing energy consumption without considering energy-related features in the previous subsection. The best heuristic remains the same when changing the objective from energy consumption to EDP, resulting in an EDP of around 5071504 J·s.

Figure 4.17 shows that in these cases the Energy scheduler reduces time and energy consumption by assigning the heaviest jobs to the two fastest nodes, one to each of them so as to avoid memory contention. The two B jobs have been allocated to the same node, the fastest available, but only occupying half of the available cores. This configuration was mostly forced because no other nodes were available at the time, but it is nevertheless a better choice than assigning J13 instead of J16 to N1, as was the case in the configuration found when optimizing the energy consumption objective.

Comparison

The trained models for the energy consumption and EDP objectives, as well as the simplified model that does not consider energy-related features in the observation, have been

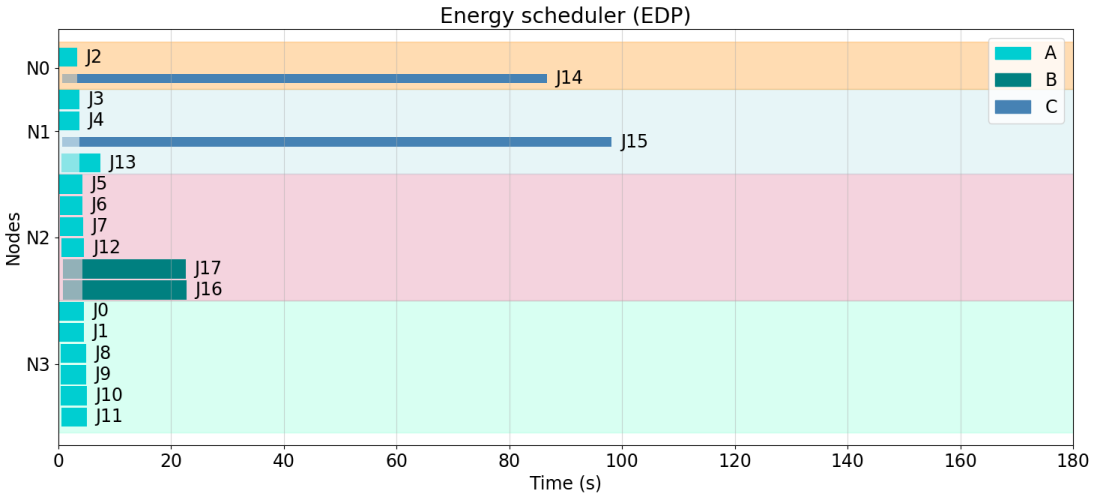

Figure 4.17: Experiment 2 - Configuration obtained by the Energy scheduler for the EDP objective.

tested for one hundred simulations with the same workload and platform used for training. In practice, a different test workload should be used, but these validation experiments are meant to analyze the capability of the agent to learn and investigate its behaviour when overfitting a simple, human-comprehensible example. Figures 4.18 and 4.19 plot the resulting energy consumption and EDP respectively, including the corresponding values for the best heuristic and the four others introduced in section 4.1. Table 4.11 summarizes the percentage reductions in both metrics for the minimum, maximum and average values obtained in all of the presented cases, as compared to the results provided by the best heuristic.

A considerable reduction in energy consumption (around 27%), and EDP especially (around 53%), is achieved by employing the Energy scheduler. The policy obtained when optimizing for energy consumption with energy-related features performs slightly worse than the other two, and the negligible standard deviation points to the policy being almost deterministic. This in turn seems to be an indicator of strong overfitting. The policies obtained in the two other cases have overfitted as well, but not as drastically.

There is no significant difference between the results obtained by these last two policies, other than a slight increase in standard deviation when no energy-related features are considered. This increase persevered upon incrementing the number of test simulations and was consistent across several executions, but repeating the tests with another model could lead to a different conclusion. This scenario has not been tested since the models trained with this setting seldom reach the solution shown in figure 4.17.

Percentage reduction w.r.t. Best heuristic	*Energy consumption*			*EDP*		
	Min	*Avg*	*Max*	*Min*	*Avg*	*Max*
Energy consumption objective	26.02	26.01	26.00	48.60	48.59	48.57
EDP objective	28.46	28.25	24.71	55.62	55.24	49.22
No energy-related features	28.45	28.10	23.22	55.61	54.96	42.28

Table 4.11: Experiment 2 - Percentage reductions in energy consumption or EDP obtained by the Energy scheduler with respect to the best heuristic.

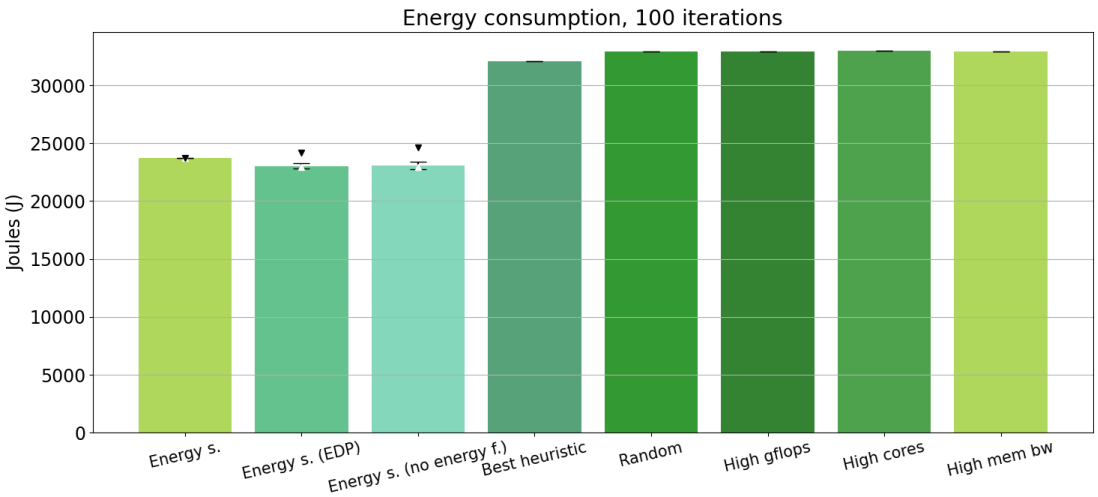

Figure 4.18: Experiment 2 - Energy consumption comparison.

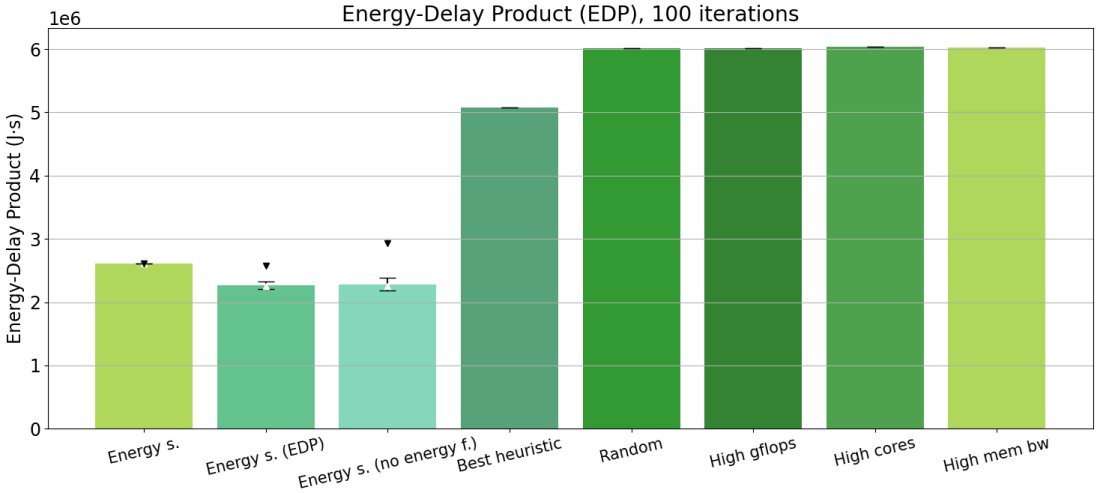

Figure 4.19: Experiment 2 - EDP comparison.

Discussion

The Energy scheduler struggled to find the closest to optimal configuration for this problem, which was only achieved when factoring execution time into the equation by optimizing for the EDP objective (even though both EDP and *power consumption* were lower for this solution), or when removing the power specifications and energy estimate from the features used for decision-making. A hypothesis is that, when optimizing energy consumption exclusively, the agent is led to give priority to B jobs since they present lower energy estimates for the currently available nodes. When optimizing EDP, on the other hand, the agent focuses on minimizing execution time together with energy consumption, which leads to a better global solution as both metrics are correlated.

At the same time, this experiment has showcased the potential of employing machine learning techniques to reduce energy consumption and EDP through job scheduling, as both metrics have improved considerably when comparing the results offered by the Energy scheduler to the configurations produced by a variety of heuristic algorithms, including the most suitable one for this particular problem. Nevertheless, the true value of such

an approach resides in producing an agent capable of generalizing well for a variety of workloads, and not just overfitting a simple example.

4.3 Experimental Results

This section tests the Energy scheduler against larger synthetic workloads and platforms with a greater capacity. In particular, the abilities of the scheduler to scale and generalize are evaluated in two separate examples. In both cases, SWA has been enabled in hopes of obtaining more robust models upon increasing the scale of the scheduling problem.

4.3.1 Experiment 3

This experiment analyzes the behaviour of the Energy scheduler after scaling Experiment 2 by a factor of ten. In this case the generalization capacity of the scheduler will not be evaluated, since it will be trained and tested on the same, predictable workload. Instead, this experiment attempts to analyze the convergence patterns and behaviour of the Energy scheduler for a larger, more complex instance of the scheduling problem.

Configuration

The configuration for this experiment is the same as the one shown in tables 4.7, 4.8 and 4.9 for Experiment 2 in the previous section, but with ten copies of each node for the platform and ten copies of each job in the workload, resulting in the trace shown in table 4.12. These modifications scale the problem up to a total of 1040 cores and 180 jobs. The interval between job arrivals has been reduced to ten milliseconds so as to increase the load on the cluster, but this measure does not have much of an effect since scaling the workload and platform by the same factor in reality increases the capacity of the platform more than it increments the load, as each node contributes with multiple cores.

The training hyperparameters have been adapted as shown in table 4.13. In each simulation, five random trajectories of up to ninety jobs are sampled from the workload and scheduled with the same policy before training the actor and critic networks based on the collected data.

Time	0.00-1.39	1.40-1.59	1.60-1.79
Job ID	J0-J139	J140-J159	J160-J179
Profile	A	C	B

Table 4.12: Experiment 3 - Workload.

Simulations	100	Seed	1024
Trajectories	5	ϵ	0.1
Trajectory length	90	γ	0.99
Minibatch size	32	λ	0.95
Epochs	50	Actor lr	0.001
SWA	Yes	Critic lr	0.001

Table 4.13: Experiment 3 - Training hyperparameters.

Energy Consumption Objective

The scheduler has first been trained to minimize energy consumption. Not as many training executions as in the previous experiments have been carried out, as each of them may easily last for eight hours on the employed workstation, depending on the chosen trajectory and

minibatch hyperparameters. A variety of sampling and training strategies have been tried regardless.

The plots shown in figure 4.20 represent the evolution of the loss and reward functions throughout the most promising training execution, using the hyperparameters presented in table 4.13.

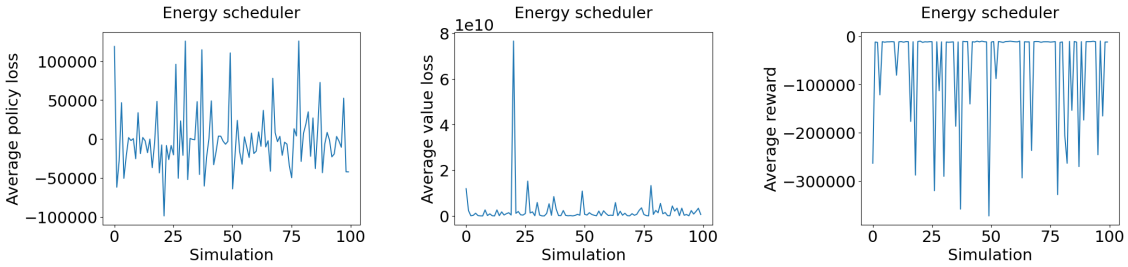

Figure 4.20: Experiment 3 - Loss and reward plots for the energy consumption objective.

The policy loss plot does not appear to converge, as no clear progress can be observed. Despite this fact, the scale of the y-axis is considerably small when compared to the plots obtained during other training executions (not shown in this document) and thus the resulting model is the one that has been chosen for evaluation. The value loss plot follows a similar pattern, although in this case the peak at simulation nineteen makes the rest seem less significant. The dips in the reward can be explained by the fact that only fragments of the complete workload are being scheduled in each simulation, and these may or may not have contained heavy jobs.

As expected, scaling the problem has made the training process considerably more difficult and time-consuming. Subsampling the workload and simulating shorter trajectories reduces training time, but introduces variance as the sampled workload fragments may not be well-balanced. Stochastic weight averaging may have also affected the evolution of the policy loss, despite the reduced number of epochs during which models are retrieved.

EDP Objective

Afterwards, the Energy scheduler has been trained for the EDP objective, obtaining the loss and reward plots shown in figure 4.21. These are similar to the ones presented in the previous subsection, so the same reasoning applies.

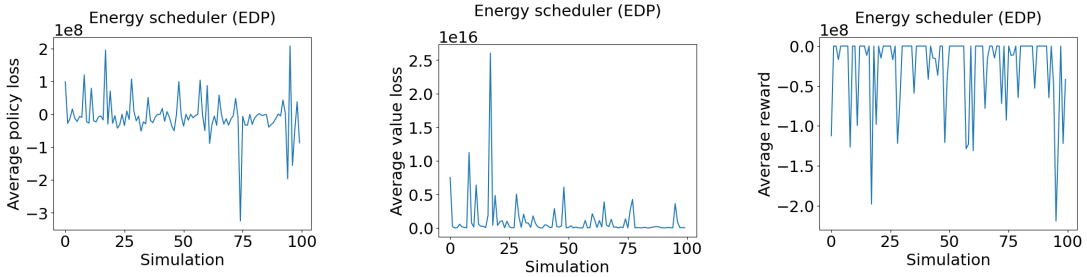

Figure 4.21: Experiment 3 - Loss and reward plots for the EDP objective.

In this case, the model seems to have converged around ten simulations from the end of the execution before finally diverging again. An early stopping mechanism could be implemented to stop training after the loss has stayed stable over a specified number of iterations. However, adding this feature would not be straightforward as each simulation

is an independent execution of the simulator, meaning that data would have to be shared over a file.

Comparison

Figures 4.22 and 4.23 show the resulting energy consumption and EDP when applying the trained scheduler, for both objectives, to the complete workload throughout one hundred test simulations. These values are compared to the results obtained by the heuristics mentioned in section 4.1. Table 4.14 shows the percentage reductions with respect to the best heuristic (*smallest - high_mem*) in both metrics for the minimum, maximum and average values obtained by the Energy scheduler during testing. As the Energy scheduler is not the best-performing solution in this case, it has also been compared against the baseline offered by the uninformed random heuristic in table 4.15. The results obtained by the best heuristic have also been included in this table for reference.

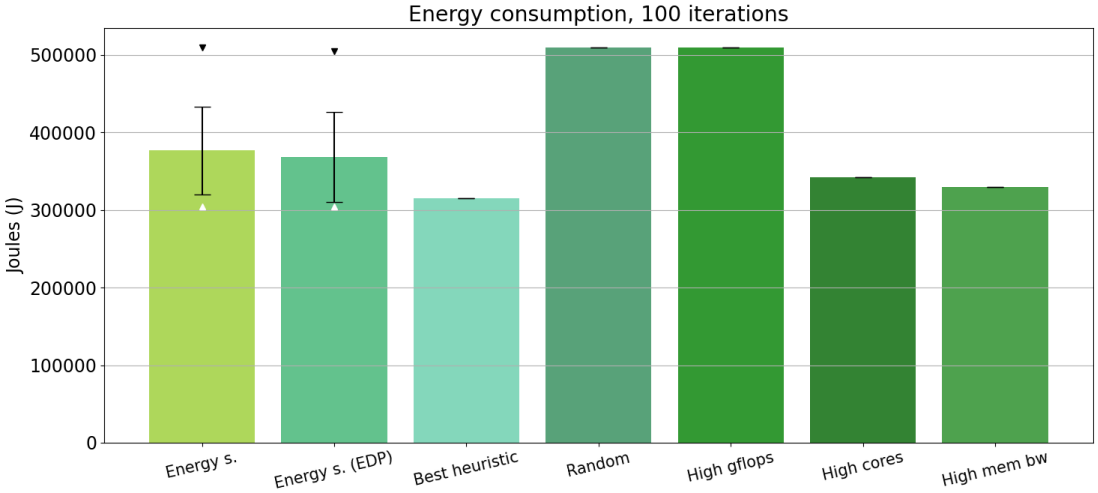

Figure 4.22: Experiment 3 - Energy consumption comparison.

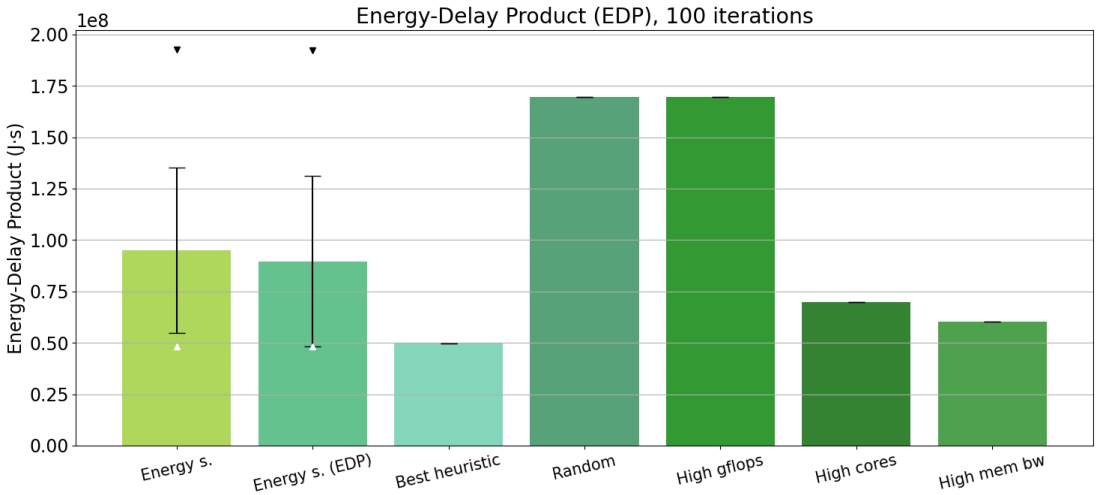

Figure 4.23: Experiment 3 - EDP comparison.

The models obtained for both objectives behave in a similar manner regardless of the evaluated metric. The slight improvements offered by the agent trained to minimize EDP

Percentage reduction w.r.t. Best heuristic	*Energy consumption*			*EDP*		
	Min	*Avg*	*Max*	*Min*	*Avg*	*Max*
Energy consumption objective	3.37	-19.52	-61.58	3.21	-90.49	-286.22
EDP objective	3.46	-16.87	-59.99	3.33	-79.63	-285.67

Table 4.14: Experiment 3 - Percentage reductions in energy consumption or EDP obtained by the Energy scheduler with respect to the best heuristic.

Percentage reduction w.r.t. Random heuristic	*Energy consumption*			*EDP*		
	Min	*Avg*	*Max*	*Min*	*Avg*	*Max*
Energy consumption objective	40.18	26.01	-0.03	71.52	43.96	-13.62
EDP objective	40.23	27.65	0.96	71.56	47.16	-13.46
Best heuristic	38.09			70.58		

Table 4.15: Experiment 3 - Percentage reductions in energy consumption or EDP obtained by the Energy scheduler and the best heuristic with respect to the random heuristic.

cannot be clearly attributed to the chosen objective, although it would be a reasonable hypothesis since execution time and energy consumption seem to be correlated in this problem. This can be deduced by comparing figures 4.22 and 4.23, especially the bars corresponding to the heuristics, and by noticing that the height differences are accentuated proportionally when multiplying energy by execution time.

Regarding energy consumption (figure 4.22), the Energy scheduler is not capable of consistently reducing this metric and instead fluctuates in a range between the best and worst values obtained by several heuristics, even though it is capable of beating the best heuristic by a small margin in the minima observed at test time. On average, the two tested models perform about 18% worse than the most appropriate heuristic and around 27% better than a random, uninformed scheduling policy for this particular scenario, but at times they fail and make no difference with respect to the latter.

These maxima are even more pronounced when comparing EDP in figure 4.23. The value obtained in these cases is 13% greater than the one produced by an uninformed policy. However, it should be taken into account that the tested models are probably suboptimal, as they did not completely converge during training. Even under these circumstances, the Energy scheduler manages to reduce EDP by around 46% on average with respect to an in appropriate heuristic, and by 3% with respect to the best heuristic in the observed minima.

Overall, the Energy scheduler does not stand out when compared to the other heuristics shown in figures 4.22 and 4.23, but instead occupies a middle ground. This indicates that the agent has not overfitted the proposed problem. Similarly, the instability in the results can be attributed to a failure to reach convergence during training.

Discussion

Despite the promising results found in Experiment 2, training the Energy scheduler for a scaled-up version of this example has resulted in a dissimilar conclusion. The agent has not been capable of consistently producing schedules that reduce energy consumption, although on average it has offered better performance than an inappropriate heuristic. Three out of five heuristics, however, have beaten the Energy scheduler.

Nevertheless, it should be taken into account that the load on the cluster has been reduced with respect to Experiment 2, and at the same time the number of jobs and resources

has increased. In such a situation, an intelligent scheduler has trouble investigating the solution space since there are many more available configurations to choose from, whereas the most appropriate heuristics thrive as their policies can be applied with few restrictions. In addition, this particular problem could be solved by following a simple heuristic strategy, but this would most often not be the case in a real context.

Overall, it can be said that the new scheduler has been capable of offering reasonable performance on a larger, more complex version of the scheduling problem, but adjustments should be made to the training hyperparameters, or even the actor and critic models, so as to reach convergence in a more stable manner.

4.3.2 Experiment 4

This last experiment has been designed with the objective of determining how well the Energy scheduler is capable of adapting to different workloads. To this end, the scheduler has been trained on a generic workload and then tested on both a memory-intensive and a computation-intensive workload. For brevity, only the results obtained for the energy consumption objective are presented, since the behaviour of the scheduler for the EDP objective was found to be similar.

Seeing the increase in training time derived from increasing the size of the platform and workload in Experiment 3, this experiment has been reduced to a modest scale so as to better focus on analyzing the ability of the scheduler to generalize.

Configuration

The target platform is composed of thirty instances of Node 0 and four instances of Node 3, following the descriptions given in table 4.7. In this way, multiple jobs may be assigned to the same 48-core node or distributed among several faster, 8-core nodes.

Two workloads containing one hundred jobs each have been defined using the job profiles introduced in table 4.16: a memory-intensive workload using A1, B1 and C1; and a computation-intensive workload using A2, B2 and C2. In each workload, jobs arrive in one-second intervals in a random order. A separate mixed workload of four hundred jobs combining all six profiles has been used to train the scheduler, using the hyperparameters gathered in table 4.17.

Profile	A1	B1	C1	A2	B2	C2
Req. cores	8	8	8	8	8	8
Req. ops ($\times 10^{10}$)	2.75	6.25	12.5	5.5	8.25	12.5
Req. time (s)	9.2	20.8	41.7	18.4	27.5	41.7
Mem. vol. (MB/s)	10^5	10^6	10^8	1	1	1

Table 4.16: Experiment 4 - Job profiles.

Simulations	100	Seed	4321
Trajectories	3	ϵ	0.1
Trajectory length	50	γ	0.99
Minibatch size	25	λ	0.95
Epochs	50	Actor lr	0.001
SWA	Yes	Critic lr	0.001

Table 4.17: Experiment 4 - Training hyperparameters.

Energy Consumption Objective

The Energy scheduler has only been trained to optimize the energy consumption objective. The evolution of the rewards and the policy and value losses during the most promising training execution are shown in figure 4.24.

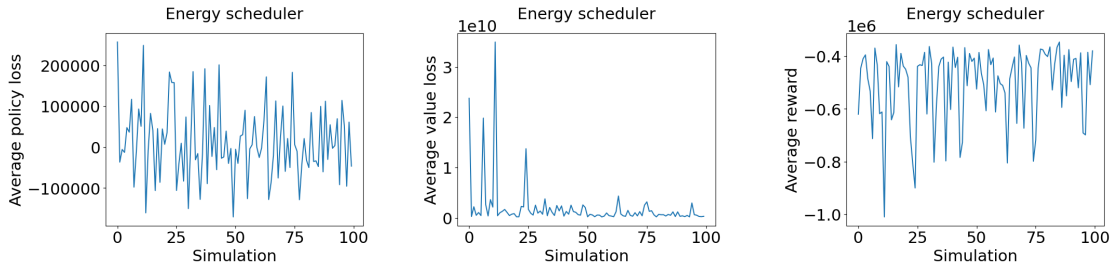

Figure 4.24: Experiment 4 - Loss and reward plots for the energy consumption objective.

Only the critic seems to have converged to an adequate degree, according to the value loss plot. The policy loss and the rewards, on the other hand, show a slight improvement over time, but still oscillate considerably at the end of the training execution.

As in Experiment 3, it remains to be tested whether increasing the number of simulations, or altering the training hyperparameters, would enable the agent to converge to a more stable policy. These alterations would lead to an increase in training time and thus have not been evaluated in this work.

Comparison

The results provided by the Energy scheduler for the memory-intensive and computation-intensive workloads over one hundred test iterations have been compared to the energy consumption produced by three of the heuristics introduced in section 4.1: *first - high_ gflops*, *first - high_ cores* and *first - high_ mem_ bw*. Conceptually, the first heuristic should be appropriate for computation-intensive workloads, since these kinds of resources should be capable of completing tasks faster. The second heuristic tended to perform well overall in similar experiments (not shown in this document), so it has been included as well. Finally, the third heuristic should be able to reduce energy consumption in the presence of memory contention, assigning jobs to the nodes with the highest currently available bandwidth.

Figure 4.25 illustrates the differences in energy consumption for each of the two workloads, and table 4.18 summarizes the percentage reductions obtained by the Energy scheduler with respect to the three tested heuristics. The minimum, average and maximum values observed at test time have been considered.

Percentage reduction in energy consumption	*High_ gflops*			*High_ cores*			*High_ mem_ bw*		
	Min	*Avg*	*Max*	*Min*	*Avg*	*Max*	*Min*	*Avg*	*Max*
Memory workload	12.66	-1.31	-12.87	15.32	1.77	-9.43	21.98	9.50	-0.83
Computation workload	23.36	4.02	-0.35	2.28	-22.37	-27.95	23.36	4.02	-0.35

Table 4.18: Experiment 4 - Percentage reductions in energy consumption obtained by the Energy scheduler with respect to the three tested heuristics for both workloads.

Unexpectedly, *high_ gflops* is the best-performing heuristic for the memory-intensive workload. *High_ mem_ bw*, on the contrary, has achieved the worst performance in this case. A possible explanation could be the caveat mentioned in section 4.2.1: this heuristic es-

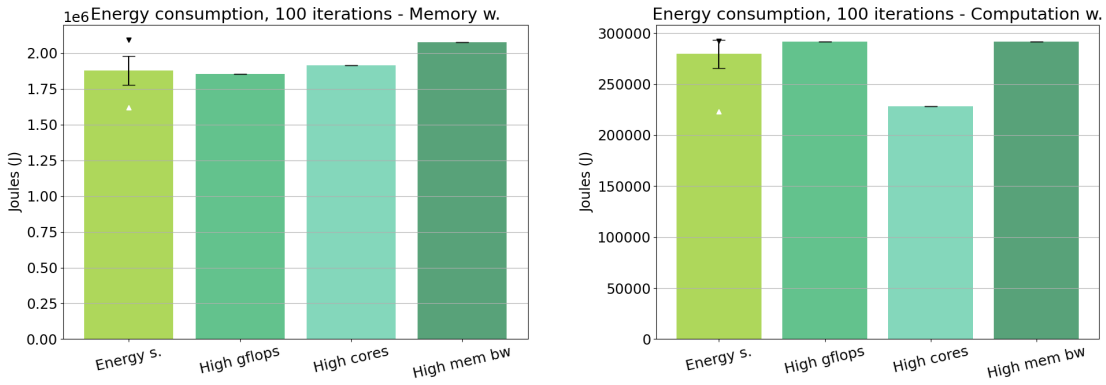

Figure 4.25: Experiment 4 - Energy consumption comparison for the memory-intensive (left) and computation-intensive (right) workloads.

timates the currently available bandwidth as the sum of the requested bandwidths of the jobs currently running on the target node. *High_ cores* offers an intermediate solution.

When compared to these three heuristics for the same workload, the Energy scheduler is closest to *high_ gflops* on average, as can be seen in table 4.18. Occasionally, it produces considerably better scheduling configurations that result in a reduction of around 12% with respect to this heuristic, but also others that perform even slightly worse (around 1%) than *high_ mem_ bw*.

As for the computation-intensive workload, in this case *high_ gflops* and *high_ mem_ bw* produce similar configurations with the same energy consumption, and it is *high_ cores* that manages to offer a considerable reduction with respect to these heuristics, of around 22%. Employing the Energy scheduler, on the other hand, provides a reduction of only around 4% on average with respect to the worst heuristics, even if it occasionally manages to reduce consumption in 2% with respect to *high_ cores*. In the worst case scenario, it behaves similarly to the other two heuristics.

Discussion

For this particular problem, the *high_ cores* heuristic would have been the best choice due to its well-rounded performance. The results obtained by the Energy scheduler, on the other hand, have not been very impressive. Nevertheless, it should be taken into account that the tested model had not converged completely, was trained on trajectories of half the test-time length, and still was able to produce reasonable results for both workloads on average.

Even if the Energy scheduler was not the best-performing workload manager in all scenarios, its behaviour did not vary as much as that of the *high_ gflops* and *high_ cores* heuristics across different workloads. For example, *high_ gflops* offered the best solution for the memory-intensive workload, but did not perform well on the computation-intensive workload. On the contrary, the Energy scheduler was capable of generalizing and adapting to both workloads to a certain extent.

The final conclusion is not very different from the one reached in Experiment 3: ideally, the training hyperparameters, or the actor and critic models, should be adjusted in order to improve convergence and hopefully bring the results obtained by the Energy scheduler closer to the observed minima, while at the same time avoiding overfitting.

Chapter 5

Conclusion and Future Work

This chapter provides a summary of the project and closes this document by exposing the key points of the conclusions extracted during the evaluation process. Finally, it suggests a series of possible improvements and additions as candidates for future investigation in this line of work.

5.1 Conclusion

This work has achieved all the proposed objectives, finally resulting in the development of an intelligent scheduler for heterogeneous HPC systems that focuses on reducing energy consumption or EDP. As a novelty, the implemented workload manager makes use of the power consumption specifications of the resources of a cluster for decision-making.

This new scheduler leverages the Proximal Policy Optimization algorithm and the capabilities offered by self-normalizing neural networks and stochastic weight averaging. As an additional, optional feature, the scheduler may choose to wait until resources are freed instead of scheduling a job as soon as it is possible. Discovering, investigating and understanding these techniques has been one of the main challenges of this work.

The resulting scheduler has been validated and evaluated using the IRMaSim cluster simulation framework with a series of synthetic experiments that explore various aspects of the implementation. In all cases, the new scheduler has been compared against the results provided by multiple heuristic algorithms. The main findings can be summarized as follows:

- *No difference was reported* when comparing the developed workload manager with an analogous version that did not consider power consumption features for decision-making. This has been attributed to the fact that the employed power consumption model was too simple and strongly depended on processor frequency, a parameter already employed by most schedulers.
- *Allowing the scheduler to wait* unlocks more advantageous scheduling configurations, but makes convergence harder to achieve. When the training hyperparameters are adjusted to solve this issue, the scheduling agent tends to stagnate on suboptimal solutions. Moreover, a robust mechanism that prevents the scheduler from waiting indefinitely has yet to be found.
- *Certain heuristic algorithms* are capable of reducing energy consumption, even in the presence of memory contention, but relying on a single heuristic is most often not

appropriate in a realistic scenario where the characteristics of submitted jobs vary over time. On the other hand, an intelligent workload manager may not offer the best solution for each particular case, but it provides a more robust and adaptable approach to job scheduling. However, these models can be hard to train.

Despite the various difficulties faced during the development and evaluation of the newly developed intelligent workload manager, the observed results show potential regarding the possibilities of energy-focused job scheduling based on deep reinforcement learning. Much work is still left to do in this area, but resolving the discovered issues could be a first step towards developing a robust and efficient workload manager capable of reducing energy consumption and EDP in real HPC systems.

5.2 Future Work

The previous section signaled various areas of improvement in the proposed implementation. Nevertheless, the development of solutions to the identified issues is outside of the scope of this project and shall therefore be left as possible future work for the Computer Architecture and Technology group of the University of Cantabria. The following research paths are proposed:

- The level of detail of the employed *energy consumption model* should be increased. Firstly, a more precise resource power consumption estimate that does not depend exclusively on processor frequency could be defined so as to better leverage the capabilities of an energy-oriented scheduler. Secondly, the IRMaSim simulator should be adapted to consider factors such as the reduction in consumption that takes place during the execution of memory access instructions or Dynamic Voltage and Frequency Scaling (DVFS), which reduces the running frequency of a processor when possible so as to consume less energy.
- *Job definitions* could also be defined in a more complex manner in order to better represent the characteristics of real applications. This would include defining a pattern indicating peaks, flats or dips in energy consumption during the execution of a job owing to the type of instructions it contains, especially accesses to memory, I/O operations or network communications. Similar patterns could help predict memory contention.
- The potential of *allowing the scheduler to wait* has been demonstrated, but several issues arise during training. Measures should be taken so as to stabilize the learning process and prevent the agent from waiting indefinitely, for example including a reasonable penalty every time the agent chooses to wait instead of scheduling a new job. Limiting the amount of total wait time in relation to the characteristics of the workload would be another possible solution.
- The *difficulties experienced during training* for larger examples may have been a consequence of the complexity of the employed deep neural networks. The intelligent scheduler might therefore benefit from simplifying the actor and critic networks or replacing them by a different type of model, such as a bayesian network.
- Due to execution time constraints, and because it has been hard to find real cluster traces containing energy consumption information, this work has only evaluated the proposed implementation by means of synthetic examples with particular features. *Further testing with realistic workloads* would be required to determine how the scheduler would operate in a real environment.

Bibliography

[1] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. arXiv 1707.06347, 2017.

[2] José Luis Bosque and L. P. Perez. Theoretical scalability analysis for heterogeneous clusters. In *4th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2004), April 19-22, 2004, Chicago, Illinois, USA*, pages 285–292. IEEE Computer Society, 2004.

[3] Mueen Uddin and Azizah Abdul Rahman. Energy efficiency and low carbon enabler green it framework for data centers considering green metrics. *Renewable and Sustainable Energy Reviews*, 16(6):4078 – 4094, 2012.

[4] M. Witkowski, A. Oleksiak, T. Piontek, and J. Węglarz. Practical power consumption estimation for real life hpc applications. *Future Generation Computer Systems*, 29(1):208–217, 2013. Including Special section: AIRCC-NetCoM 2009 and Special section: Clouds and Service-Oriented Architectures.

[5] International Energy Agency. Data centres and data transmission networks. `https://www.iea.org/reports/data-centres-and-data-transmission-networks`, 2022. [Accessed 02 May 2023].

[6] John Russell. Hyperion paints a positive picture of the hpc market. *HPCwire*, 11 2022. `https://www.hpcwire.com/2022/11/08/hyperion-paints-a-positive-picture-of-the-hpc-market/` [Accessed 02 May 2023].

[7] Carlos de Alfonso, Miguel Caballer, Fernando Alvarruiz, and Germán Moltó. An economic and energy-aware analysis of the viability of outsourcing cluster computing to a cloud. *Future Generation Computer Systems*, 29(3):704–712, 2013. Special Section: Recent Developments in High Performance Computing and Security.

[8] Javier Balladini, Remo Suppi, Dolores Rexachs, and Emilio Luque. Impact of parallel programming models and cpus clock frequency on energy consumption of hpc systems. In *2011 9th IEEE/ACS International Conference on Computer Systems and Applications (AICCSA)*, pages 16–21, 2011.

[9] Yiannis Georgiou, David Glesser, Krzysztof Rzadca, and Denis Trystram. A scheduler-level incentive mechanism for energy efficiency in hpc. In *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 617–626, 2015.

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

[11] Marina Morán, Javier Balladini, Dolores Rexachs, and Enzo Rucci. Towards management of energy consumption in hpc systems with fault tolerance. In *2020 IEEE Congreso Bienal de Argentina (ARGENCON)*, pages 1–8, 2020.

[12] Bartłomiej Kocot, Paweł Czarnul, and Jerzy Proficz. Energy-aware scheduling for high-performance computing systems: A survey. *ENERGIES*, 16(2):890–, 2023.

[13] J.D. Ullman. Np-complete scheduling problems. *Journal of Computer and System Sciences*, 10(3):384–393, 1975.

[14] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach, Third Edition*. Pearson, Boston, 2020.

[15] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer, New York, 2006.

[16] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. A Bradford Book, Cambridge, MA, USA, 2018.

[17] Yuping Fan. Job scheduling in high performance computing. arXiv 2109.09269, 2021.

[18] Esteban Stafford and José Luis Bosque. Improving utilization of heterogeneous clusters. *Journal of Supercomputing*, 76(11):8787–8800, 2020.

[19] Stathis Maroulis, Nikos Zacheilas, and Vana Kalogeraki. A holistic energy-efficient real-time scheduler for mixed stream and batch processing workloads. *IEEE Transactions on Parallel and Distributed Systems*, 30(12):2624–2635, 2019.

[20] José Luis Bosque, Pablo Toharia, Oscar David Robles, and Luis Pastor. A load index and load balancing algorithm for heterogeneous clusters. *J. Supercomput.*, 65(3):1104–1113, 2013.

[21] Wei Tang, Zhiling Lan, Narayan Desai, and Daniel Buettner. Fault-aware, utility-based job scheduling on blue, gene/p systems. In *2009 IEEE International Conference on Cluster Computing and Workshops*, pages 1–10, 2009.

[22] Danilo Carastan-Santos and Raphael Y. de Camargo. Obtaining dynamic scheduling policies with simulation and machine learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '17, New York, NY, USA, 2017. Association for Computing Machinery.

[23] Ahuva W. Mu'alem and Dror G. Feitelson. Utilization, predictability, workloads, and user runtime estimates in scheduling the ibm sp2 with backfilling. *IEEE transactions on parallel and distributed systems*, 12(6):529–543, 2001.

[24] Dan Tsafrir, Yoav Etsion, and Dror G. Feitelson. Backfilling using system-generated predictions rather than user runtime estimates. *IEEE Transactions on Parallel and Distributed Systems*, 18(6):789–803, 2007.

[25] Hongyang Sun, Redouane Elghazi, Ana Gainaru, Guillaume Aupy, and Padma Raghavan. Scheduling parallel tasks under multiple resources: List scheduling vs. pack scheduling. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 194–203, 2018.

[26] Andy B. Yoo, Morris A. Jette, and Mark Grondona. Slurm: Simple linux utility for resource management. In Dror Feitelson, Larry Rudolph, and Uwe Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing*, pages 44–60, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[27] Robert L. Henderson. Job scheduling under the portable batch system. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, pages 279–294, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg.

[28] Vijay Konda and John Tsitsiklis. Actor-critic algorithms. In S. Solla, T. Leen, and K. Müller, editors, *Advances in Neural Information Processing Systems*, volume 12. MIT Press, 1999.

[29] Hongzi Mao, Mohammad Alizadeh, Ishai Menache, and Srikanth Kandula. Resource management with deep reinforcement learning. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*, HotNets '16, page 50–56, New York, NY, USA, 2016. Association for Computing Machinery.

[30] Yufei Ye, Xiaoqin Ren, Jin Wang, Lingxiao Xu, Wenxia Guo, Wenqiang Huang, and Wenhong Tian. A new approach for resource scheduling with deep reinforcement learning. arXiv 1806.08122, 2018.

[31] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. Learning scheduling algorithms for data processing clusters. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, page 270–288, New York, NY, USA, 2019. Association for Computing Machinery.

[32] Di Zhang, Dong Dai, Youbiao He, Forrest Sheng Bao, and Bing Xie. Rlscheduler: An automated hpc batch job scheduler using reinforcement learning. arXiv 1910.08925, 2020.

[33] Jaime Fomperosa, Mario Ibáñez, Esteban Stafford, and José Luis Bosque. Task scheduler for heterogeneous data centres based on deep reinforcement learning. In Roman Wyrzykowski, Jack J. Dongarra, Ewa Deelman, and Konrad Karczewski, editors, *Parallel Processing and Applied Mathematics - 14th International Conference, PPAM 2022, Gdansk, Poland, September 11-14, 2022, Revised Selected Papers, Part I*, volume 13826 of *Lecture Notes in Computer Science*, pages 237–248. Springer, 2022.

[34] Qiqi Wang, Hongjie Zhang, Cheng Qu, Yu Shen, Xiaohui Liu, and Jing Li. Rlschert: An hpc job scheduler using deep reinforcement learning and remaining time prediction. *Applied Sciences*, 11(20), 2021.

[35] Yuping Fan, Boyang Li, Dustin Favorite, Naunidh Singh, Taylor Childers, Paul Rich, William Allcock, Michael E. Papka, and Zhiling Lan. Dras: Deep reinforcement learning for cluster scheduling in high performance computing. *IEEE Transactions on Parallel and Distributed Systems*, 33(12):4903–4917, 2022.

[36] Jiwu Peng, Kenli Li, Jianguo Chen, and Keqin Li. Hea-pas: A hybrid energy allocation strategy for parallel applications scheduling on heterogeneous computing systems. *Journal of Systems Architecture*, 122:102329, 2022.

[37] Juan Carlos Salinas-Hilburg, Marina Zapater, José M. Moya, and José L. Ayala. Energy-aware task scheduling in data centers using an application signature. *Computers & Electrical Engineering*, 97:107630, 2022.

[38] Daniel Bick. Towards Delivering a Coherent Self-Contained Explanation of Proximal Policy Optimization. Master's thesis, Rijksuniversiteit Groningen, the Netherlands, 2021.

[39] John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, and Pieter Abbeel. Trust region policy optimization. *CoRR*, abs/1502.05477, 2015.

[40] John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation. arXiv 1506.02438, 2018.

[41] Günter Klambauer, Thomas Unterthiner, Andreas Mayr, and Sepp Hochreiter. Self-normalizing neural networks. *Advances in Neural Information Processing Systems (NIPS)*, 2017.

[42] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. arXiv 1412.6980, 2017.

[43] Pavel Izmailov, Dmitrii Podoprikhin, Timur Garipov, Dmitry Vetrov, and Andrew Gordon Wilson. Averaging weights leads to wider optima and better generalization. arXiv 1803.05407, 2019.

[44] Emilio Castillo, Lluc Alvarez, Miquel Moretó, Marc Casas, Enrique Vallejo, José Luis Bosque, Ramón Beivide, and Mateo Valero. Architectural support for task dependence management with flexible software scheduling. In *IEEE International Symposium on High Performance Computer Architecture, HPCA 2018, Vienna, Austria, February 24-28, 2018*, pages 283–295. IEEE Computer Society, 2018.

[45] A. Herrera, M. Ibáñez, E. Stafford, and J.L. Bosque. A simulator for intelligent workload managers in heterogeneous clusters. In *2021 IEEE/ACM 21st Int. Sym. on Cluster, Cloud and Internet Computing (CCGrid)*, pages 196–205, 2021.