Facultad de Ciencias

HDeepRM: Deep Reinforcement Learning para la Gestión de Cargas de Trabajo en Clústeres Heterogéneos

(HDeepRM: Deep Reinforcement Learning for Workload Management in Heterogeneous Clusters)

Trabajo de Fin de Máster
para acceder al

MÁSTER UNIVERSITARIO EN INGENIERÍA INFORMÁTICA

Autor: Adrián Herrera Arcila

Director: Jose Luis Bosque Orero

Marzo - 2019

Contents

List of Figures

List of Tables

Acknowledgements

To my family, specially my parents who have always given me context and perspective as presents.

To my friends, the ones that are always there and those belonging to older stages of my life. The amount of emotional support, knowledge exchange and shared experiences are invaluable.

To every professor I have come across during my education. Thanks for giving me a glimpse of what matters in life when I was little, and also for building the bridges of my path when I grew up.

To anyone pursuing his/her passion for progress, in spite of the day-to-day difficulties the motivation and positivity always make you prevail, and that has a human impact which affects individuals such as myself.

Finally, special thanks to José Luis Bosque for supporting my work since early Bachelor years, including the results of this thesis. He has been an outstanding mentor, offering an ideal blend of guidance and freedom in decision-making.

Abstract

High Performance Computing (HPC) environments offer users computational capability as a service. They are constituted by computing clusters, which are groups of resources available for processing jobs sent by the users. *Heterogeneous* configurations of these clusters allow for providing resources fitted to a wider spectrum of workloads, superior to that of traditional homogeneous approaches. This in turn improves the computational and energetic efficiency of the service.

Scheduling of resources for incoming jobs is undertaken by a *workload manager* following a established *policy*. Classic policies have been developed for homogeneous environments, with literature focusing on improving job selection policies. Nevertheless, in heterogeneous configurations the resource selection is as relevant for optimizing the offered service.

Complexity of scheduling policies grows with the number of resources and degree of heterogeneity in the service. *Deep Reinforcement Learning (DRL)* has been recently evaluated in homogeneous workload management scenarios as an alternative to deal with complex patterns. It introduces an artificial *agent* which estimates via learning the optimal scheduling policy for a given system.

In this thesis, *HDeepRM*, a novel framework for the study of DRL agents in heterogeneous clusters is designed, implemented, tested and distributed. This leverages a state-of-the-art simulator, and offers users a clean interface for developing their own bespoke agents, as well as evaluating them before going into production.

Evaluations have been undertaken to demonstrate the validity of the framework. Two agents based on well-known reinforcement learning algorithms are implemented over HDeepRM, and results show the research potential in this area for the scientific community.

Resumen

Los entornos de *High Performance Computing (HPC)* ofrecen capacidad computacional como servicio a sus usuarios. Están formados por clústeres de cómputo, grupos de recursos que aceptan y procesan trabajos enviados por los usuarios. Las configuraciones *heterogéneas* permiten disponer de recursos adecuados a un espectro de cargas de trabajo superior al de los clústeres homogéneos tradicionales, mejorando la eficiencia computacional y energética del servicio.

La asociación de trabajos con recursos del sistema es llevada a cabo por un *gestor de cargas de trabajo* siguiendo una *política de planificación*. Las políticas clásicas han sido desarrolladas para entornos homogéneos, y la literatura se centra en la selección del trabajo. Sin embargo, en entornos heterogéneos la selección del recurso es de relevancia para la optimización del servicio.

La complejidad de las políticas de planificación crece con el número de recursos y la heterogeneidad del sistema. El *Aprendizaje Profundo por Refuerzo* o *Deep Reinforcement Learning (DRL)* ha sido recientemente objeto de estudio como alternativa para la gestión de cargas de trabajo. En él, se propone un *agente artifcial* que estima mediante aprendizaje la política de planificación óptima para un determinado sistema.

En esta tesis se describe el proceso de creación de HDeepRM, un nuevo marco de trabajo cuyo objetivo es el estudio de agentes basados en DRL para la estimación de políticas de planificación en clústeres heterogéneos. Implementado sobre un simulador actual, HDeepRM permite crear y evaluar nuevos agentes antes de llevarlos a producción.

Se ha llevado a cabo el diseño, implementación, pruebas y empaquetado del software para poder distribuirlo a la comunidad científica. Finalmente, en las evaluaciones se demuestra la validez del marco de trabajo, y se implementan sobre él dos agentes basados en algoritmos de DRL. La comparación de estos con políticas clásicas muestra el potencial de investigación en este área.

Chapter 1

Introduction

In this chapter, a high-level view of the main concepts associated to this thesis' body of work will be provided. Current issues in state-of-the-art High Performance Computing (HPC) *workload managers* will be highlighted, remarking the need for heterogeneity and resource interference support. A brief introduction to *machine learning* along with recent advances in deep learning will be presented, showing the research potential and applicability to workload management. Thesis' objectives will be listed, followed by the pursued methodology, chronologically describing the stages of work. Furthermore, the document structure will be offered for guidance.

1.1 Workload management in HPC data centers

High Performance Computing (HPC) refers to the use of supercomputers and parallel processing for solving complex computational problems. Supercomputers are capable of yielding large values of operations per second when compared to general-purpose computers. They are composed of several clusters, and offered as a service where users can launch instances of their jobs requesting necessary resources for their execution. Computing and storage resources are allocated to the incoming jobs for a given period of time, and multiple jobs from multiple users might coexist within the data center.

The scheduling of incoming jobs to available resources is known as *workload management*. Users' jobs are enqueued in malleable queues, and a *workload manager* is in charge of inspecting them and selecting which jobs are allocated to which resources. The job selection process is known as *job scheduling* [1], and it is undertaken by a *job scheduler*. On the other hand, the *resource management* refers to the operations over the cluster resources, such as allocation, provisioning and monitoring. This is provided by a *resource manager*.

Job and resource selections are based on a *policy pair*, which defines the criteria for both selection processes. In this thesis, *classic* policies are defined as those used in literature and in conventional workload managers. For job selection, typical approaches include *first come first served*, *shortest job first* and *EASY backfilling* [2]. In the case of resource selection, *highest computing capability* or *random* are common choices.

There exists a lack of development in resource selection policies due to traditional clusters offering homogeneous hardware. Nowadays, the majority of supercomputers, including those in the TOP 500 list [3], are based on heterogeneous configurations. This allows for smooth upgrades on cluster configurations, as well as performance gains due to the inclusion of hardware

accelerators and application specific architectures.

Nevertheless, the complexity of workload management also scales with heterogeneity, since the selection of resources becomes relevant for achieving the objectives. For instance, the current Intel Xeon E family of CPUs provides power optimized processors by the suffix of L, such as the *Xeon E5-2650L v4* [4]; this consumes about 40% less power than the *no-L* counterpart and provides 14 cores with respect to 12 from the *no-L*, however it also works at 23% less clock rate, reducing the single-core performance. This means that highly parallel loads which can leverage the 14 cores may use the L processor more efficiently, but sequential loads will perform better in the *no-L* version.

Another example of this complexity happens when deciding between GPU and FPGA based nodes [5]. The former offer higher floating-point performance as well as memory bandwidth, and their GDDR memories are optimized for sequential (streaming) memory accesses, such as those in image processing. The latter are configurable architectures which may be optimized for non-sequential memory accesses, which would make them perform better and consume considerably less energy for applications such as Fast Fourier Transform.

On another note, classic policies are also oblivious to resource interference. Shared resources can create points of conflict for jobs in the same compute node, decreasing individual performance [6]. For instance, the processor's memory controller yields a peak memory bandwidth specified by the manufacturer; if a high number of memory-bandwidth heavy jobs are *consolidated* in the same processor, the controller will be unable to serve all requests simultaneously, and some of the jobs (depending on the priority scheme of the controller) will be delayed execution. This is extendable to shared caches, network and filesystem.

Spreading jobs across multiple processors in the data center reduces resource conflicts, however communication overhead grows, which may impact performance and energy consumption of parallel applications. Moreover, energy efficiency decreases if few cores per processor are used, since the shared system power is only amortized by the active cores. Finally, power consumption due to heat dissipation in multiple areas also augments.

The early study of workload management policies is carried out by the use of *simulation*. Several state-of-the-art workload management simulators have been studied in this thesis, including Alea [7], ScSF [8], Accasim [9] and Batsim [10]. These allow for evaluating user-defined policies over a simulated *platform* where incoming jobs are specified in a *workload trace*. Most of them provide an event-flow interface, where events consist of job arrivals/completions and resource state changes. Metrics are recorded during the simulation for analysis and gathering of insights; these include waiting time, energy consumption or throughput.

In this thesis, HDeepRM is developed to overcome poorly supported heterogeneous platform simulation in current frameworks. It is served as a layer on top of Batsim offering, among other traits, multiple resource types, resource hierarchies, user-defined platforms and resource conflicts.

1.2 Machine learning and neural networks

Learning refers to the ability of acquiring or modifying information in order to improve towards an objective. Humans have this capability embedded in their brains and produced by the connections between neurons, also known as synapses. The conception of this ability in digital systems is known as *machine learning*. It consists of a broad range of techniques [11] allowing

for the system to enhance its capabilities without the need of traditional explicit programming.

In general, there are three distinct types of machine learning:

- *Supervised learning*: consists of learning a function that maps input(s) to output(s) given examples of input-output pairs. An example of this might be a cat/dog image classifier: the input is a picture, and the output is a label indicating whether it is a cat or a dog.
- *Unsupervised learning*: these set of techniques allow the system to learn the underlying structure of unlabelled data inputs. For instance, autoencoders [12] learn a compressed representation of the input by measuring the error between the uncompressed result and the original one, without the need for labels.
- *Reinforcement learning*: refers to scenarios where an *agent* interacts with an *environment* by observing its *state* and undertaking *actions*. Every action *alters* the environment, and the agent is given a *reward* depending on how that alteration contributes to its objective.

Since the initial conception of the *perceptron* [13] in 1958, machine learning has been through several unstable periods. During the 60s and 70s, the prominence of the Von Neumann architecture [14] along with the limitations of perceptrons for learning non-linearities [15] gave birth to the first *Artificial Intelligence (AI)* winter.

Later, in 1986, the introduction of the backpropagation mechanism [16] allowed for the creation of *multi-layered perceptrons (MLPs)*, where multiple layers of artificial neurons were connected to form an *Artificial Neural Network (ANN)* similar in structure to the human brain. In the 90s, other methods such as *Support Vector Machines (SVM)* [17] were found to be more effective, and thus a second AI winter began.

The current popularity trend dates back to 2006, when work by Geoff Hinton and others on iterative training allowed for stacking more layers into the MLP design, making for a *deep* architecture. Since then, *deep learning* has been applied successfully to several complex problems, such as image classification, object recognition or natural language processing (NLP) [18].

Furthermore, deep learning has also permeated the reinforcement learning algorithm class in the form of *deep reinforcement learning* (DRL). Agents are represented by deep ANNs which learn the best actions to take in each state.

The work presented in [19], namely DeepRM, suggests that complex workload management policies bespoke to the underlying platforms may be learned via DRL agents. These would understand which job-resource pairings are optimal for each platform state. They show improvements of up to 50% for some workloads with respect to classic policies such as *shortest job first*, however they only work with small synthetic and homogeneous platforms.

In this thesis, HDeepRM is developed as a novel iteration over DeepRM. It provides a DRL workload management framework for evaluating the ideas proposed in [19] while applying them to heterogeneous platforms and real data center traces. It also provides an interface for user-defined agents as well as newly designed state/observation and action spaces.

1.3 Objectives

The principal goal of this thesis is to allow the study of *deep reinforcement learning* techniques for workload management in heterogeneous environments. There are two main driving premises: (1) the increasing complexity of heterogeneous configurations as new architectures appear in the

market; (2) the recent promising results achieved in [19] along with the room for improvement from the base idea. The following specific objectives are then proposed:

- Development of a novel framework for simulating heterogeneous workload management scenarios: studied state-of-the-art workload management simulators in section 2.2.1 do not support neither heterogeneity nor resource interferences. The new framework should allow both of these traits, while also exposing a user-friendly interface for defining platforms and workloads. Furthermore, it should be fully functional by the end of the thesis.
- Extension of the framework for implementing deep reinforcement learning (DRL) agents. It should model the reinforcement loop shown in figure 1.1. Moreover, it should also offer the possibility of integrating user-defined agents without difficulty.

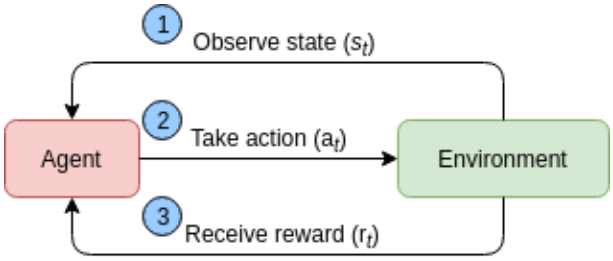


Figure 1.1: Reinforcement learning loop

- Validation of the framework main capabilities. Show the impact of heterogeneity and resource conflicts on performance of classic approaches. Introduce learning as an alternative to estimate optimal hybrid policies which improve the overall objective accomplishment.

1.4 Methodology

In order to achieve the previously explained objectives, a research methodology has been pursued, which is constituted by the following steps:

1. Study of workload management landscape: comprehend concepts behind workload management, understand classic selection policies, evaluate and compare several state-of-the-art simulators.
2. Research of reinforcement learning and deep learning: the reinforcement learning loop, agent and environment interactions, deep neural networks for approximating policies, updating the networks, integration of the concepts within a workload management scenario.
3. Complete understanding of the chosen simulator ecosystem and interfaces: metrics, event flow, callbacks, decision system, visualization tools.
4. Design of HDeepRM from scratch: abstract component interactions, user-focused design for interfaces, extensibility and readability as priorities.
5. Implementation of HDeepRM: integration with the simulator, reinforcement and deep learning libraries comparison, object-oriented approach, tool set development for ease of use, metric system, event-flow debugging through logs.
6. Distribution of the framework: uploading to a public repository, documenting the full project, automation of web documentation, versioning.
7. Evaluation of the solution: validation and verification through exhaustive experimentation, policy comparison, analysis and insights.

1.5 Document structure

In chapter 2, the concepts introduced here will be further explained and specialized towards the thesis body of work. Next, in chapter 3 a complete description of HDeepRM design decisions will be offered, both concerning heterogeneity and deep reinforcement learning capabilities. In chapter 4, implementation of the designed concepts will be explained, along with the integration of tools and algorithms. Experiments are presented in chapter 5, where results showing validity of the framework as well as research potential are reviewed. Finally, the current state of HDeepRM, along with the accomplishment of objectives and future work will be showcased in chapter 6.

Chapter 2

Background

The following chapter is a recommended reading for further delving into concepts introduced in chapter 1. Details about resource heterogeneity, workload management and deep reinforcement learning will be presented. Next, an explanation of tools and algorithms utilized along the journey is provided. A short chronological view of related work from early beginnings to recent achievements will be exhibited for ending the chapter.

2.1 Concepts

In this section, details about the three main concepts in this thesis body of work are presented. These are broad concepts, thus a set of rich references is provided in order to acquire further knowledge; these references are not a requirement, however they are highly recommended readings.

2.1.1 Resource heterogeneity

Through time, computing performance has mainly been achieved by the means of chip scaling. *Moore's law* [20] set a trend for transistor density doubling every 18 to 24 months. Chipmakers adhered to this trend, scaling transistor specs by 0.7x, which in turn provided around 40% performance boost and 50% area reduction for the same amount of power. This worked until 20nm gate voltage lengths, where channel control became inviable. FinFETs [21] were proposed for under 20nm nodes, but their scaling was more difficult, and the cadence increased from 18 months to 2.5 years or longer. Nowadays, 7nm transistors are already in the market [22], however the design and manufacture costs escalate, going from $51.3 million for 28nm to $542.2 million for 5nm [23]. Furthermore, fundamental thermal issues arise at lower scales for high frequency processors; these are conceived in *Dennard's scaling law* [24].

Parallel architectures were proposed as an alternative for tackling specially the thermal constraints. Instead of designing a one-core chip with high clock rate, several cores with lower clock rate are integrated in the same processor, where they share resources (v.gr. memory hierarchy). This requires parallel applications with the ability to leverage the budget of cores. Moreover, there are fundamental limits to how parallelization gains diminish as serial sections of the program stay constant or slightly decrease (due to reductions in clock frequency). This is known as *Amdahl's law* [25].

Current trends point towards *resource heterogeneity*. Nowadays CPUs are complemented by hardware accelerators, devices with specialised architectures targeted at specific workloads. GPUs are used for tasks from pure graphic processing to highly data-parallel jobs, while Deep Learning accelerators leverage MAC[1] based designs for convolution speed-up. Aside from improving performance, resource heterogeneity also improves energy efficiency; examples of this are CPU architectures such as Arm's big.LITTLE [27] or the recent chiplet [28] trend.

This thesis enables evaluation of systems composed of heterogeneous resources, which increase both richness and complexity of workload management.

2.1.2 Workload management

Services provided by High Performance Computing clusters are controlled by a middleware component known as workload manager. Users connect to *frontend nodes*, where utilities for interacting with the workload management system are installed. In order to launch jobs, a script specifying resource requirements for the job as well as the actual job load needs to be created by the user.

When the job is launched via providing this script to the utilities, it is *enqueued* and set for pending execution. The workload manager *schedules* pending jobs over available resources following a *response scheme* and a *policy pair*. The response scheme may be of two kinds: (1) event-based responses: when a job is launched or completed, the scheduling is triggered; (2) periodic responses: the scheduling is done in defined time periods.

On the other hand, *policies* define the *priority scheme* for both jobs and resources. During the scheduling step, the highest priority resources are chosen to be allocated for the highest priority jobs. For instance, in a *shortest job first-highest computing capability* policy pair, the job priority is negative the requested time in the system; furthermore, the resource priority would be the current FLOPs offered for processing.

Processing resources are known as *compute nodes*. In a heterogeneous clusters, three types of compute node heterogeneity may be observable:

- *Specification-based*: nodes are constituted by the same processor micro-architectures, however specs may vary between them. For instance, the clock rate of each processor or the available memory.
- *Micro-architectural*: processors provide the same Instruction Set Architecture (ISA), however the implementations vary between them. For example, Intel Xeon processors differ from AMD EPYC ones, however their ISA (x86-64) is the same, thus a compiled program for that ISA may run in either.
- *Architectural*: different devices, both in terms of architecture and functionality, coexist within the cluster. These may be GPUs, Intel MICs, Google TPUs and other processor architectures, such as IBM POWER ISA.

In HDeepRM, an event-based response scheme is supported by the underlying simulator. Up to 25 classic policy-pairs associated to different priority schemes may be specified for launching experiments. Furthermore, both specification-based and micro-architectural heterogeneity are supported, whereas architectural is left as future work.

[1] MAC stands for multiplier-accumulator. Convolutions comprise over 90% of Deep Neural Networks operations [26], and they are achieved by matrix-multiplications.

2.1.3 Deep reinforcement learning

Reinforcement Learning (RL) constitutes a family of machine learning methods distinguished by one or more *agents* interacting with an *environment*. For each time step t, the agent observes some *state* (s_t) of the environment and is asked to select an *action* (a_t). Applying this action is known as an *alteration*, which makes the environment transition into its next state (s_{t+1}). The agent receives a *reward* (r_t) as a feedback for how positive was to take that action towards the final objective. The agent then observes the new state and the process repeats. This loop (see figure 1.1) is the foundation of RL.

The objective in RL is to maximize the *discounted cumulative expected reward*, given by the equation in 2.1. This is the sum of the expected rewards at each time step, however a discount factor (γ) is applied to each of them. This factor is a *hyperparameter*[2] used to tune how much future rewards lose their value according to how far away in time they are. A γ value close to 0 means the agent will only care about immediate rewards, while a value close to 1 will extend its expectations further in time.

$$G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \; where \; \gamma \in [0, 1) \tag{2.1}$$

As the agent interacts with the environment, it gains *knowledge* of its decisions, and can tell the outcome of selecting certain actions in specific states. There are two main ways of representing this knowledge:

- *Value-based*: value of each state is given by the total amount of reward an agent can expect to accumulate over the future being at that state. This is represented in equation 2.2. The agent will then take the state with the highest value.

$$v_\pi(s) = \mathbb{E}_\pi[r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + ... \mid s_t = s] \tag{2.2}$$

- *Policy-based*: a policy function maps a state to a probability distribution over the action space. In deterministic policies, the action which leads to the highest G_t is given a probability of 1, whereas in stochastic policies, less productive actions are also given a small chance of being selected, thus encouraging the agent to explore. This is pictured in equation 2.3).

$$\pi(a|s) = \mathbb{P}[a_t = a|s_t = s] \tag{2.3}$$

In value-based RL, the value table which holds the actual values for each state can be big in high-dimensional spaces. The policy function, on the other hand, needs to be programmed such as to understand the meaning of being at each state. Both of these issues are tackled by the inclusion of substitute *artificial neural networks* (ANNs) [29], which help estimate values and policy functions respectively.

In order to train these networks, a *loss* function is calculated. This function depends on the RL algorithm implemented, however it is always based on the rewards obtained by the agent in each step of the episode. Higher rewards contribute to a lower loss.

[2] A hyperparameter is a parameter set before the learning process and external to the model, that is, it is not derived via training.

HDeepRM allows for implementing deep reinforcement learning agents based on a simple interface. The user may specify the structure of the ANNs, the loss functions and the reinforcement learning algorithm to be used.

2.2 Tools and algorithms

The majority of tools utilized during the thesis are open-source software. In this section, decisions pertaining the selection of them along with the implemented algorithms are explained.

2.2.1 Batsim ecosystem

In order to create HDeepRM, a workload management *simulator* has been selected. A first idea is to use the SLURM [30] workload manager as a platform for development. It is observed that (1) its *sched plugin* is limited by interfaces with the rest of the system, (2) launch script input options do not cover the thesis proposed range (i.e. memory bandwidth) and (3) its C-based implementation would slow down idea testing.

Main requirements for the selected tool are (1) rich documentation, (2) easy extensibility, (3) presence of analysis tools and (4) possibility of integration with other libraries. Several state-of-the-art simulators are considered however not utilized:

- Alea v4 (2016, Klusácek et al. [7]): developed on top of GridSim [31], implemented in Java, open-source. It uses its own internal representation of machines (nodes), and provides a simple interface for creating management policies in Java. Not selected due to scarce documentation and Java not being a suitable choice for DRL development.
- Scheduler Simulation Framework or ScSF (2017, Rodrigo et al. [8]): implemented as a wrapper around a real SLURM instance. Experiments are defined in a *controller*, which manages *worker* instances spawned in their own virtual machines. Components may be distributed, and physical network latencies have an impact in the simulation. New algorithms are implemented via SLURM plugins. Not selected due to complexity and lack of a real distributed infrastructure to test the components.
- Accasim (2017, Galleguillos et al. [9]): discrete-event-based simulation. An *event manager* processes events within the simulated system, and a *dispatcher* assigns jobs to resources in the system. It defines job arrival, start and completion as main events. Implemented in Python, with new algorithms integrated by extending base classes. Not selected due to poor documentation and lack of analysis tools.

Batsim [10] ecosystem is found to be a suitable choice. It is a simulator framework developed on top of SimGrid [32], which allows for deep customization of components. It consists of three major elements[3]:

- Simulator (*Batsim*): process which runs the workload management simulation. Implemented in C++, it leverages SimGrid's features such as power management via P-states or link bandwidth congestion. It communicates through several events related to job and resource state in the cluster, which are sent to the decision system.
- Decision system: equivalent to the workload manager, it is in charge of mapping jobs to resources in the cluster. Due to being detached from the simulator, it may be implemented

[3]Batsim ecosystem may be found in https://gitlab.inria.fr/batsim.

in any language, with the requirement of establishing a connection to Batsim and handling the emitted events. In this thesis *PyBatsim*, a Python-based decision system, is utilized as a base for developing the framework.

- Analysis toolset (*Evalys*): during the simulation, Batsim gathers several metrics concerning utilization, power consumption, waiting time, etc. *Evalys*, also implemented in Python, provides utilities to analyse *Batsim* outcomes.

HDeepRM leverages Batsim ecosystem heavily. It relies on it for simulation, event handling and result analysis.

2.2.2 PyTorch and Gym

Deep Learning (DL) frameworks have proliferated during last years, specially since the public release of Google's TensorFlow [33] in 2015. In this research, Facebook's PyTorch [34] is selected as a tool for implementing DL algorithms.

PyTorch is an open-source Python framework[4] focused on fitting Python's object-oriented programming and facilitating model debugging. The primary element in DL is the *computational graph*, which defines the order of operations within the model; in TensorFlow, these operations are first defined and later run in what's known as a *session*. This design decision is taken due to efficiency, it is possible to optimize the graph and run it in parallel on a GPU-backed session. This scenario is a good fit for production environments, when the model is already verified to function properly, however it complicates its development, since Python debugging capabilities are limited by the *session* interpreter. PyTorch uses a "define by run" approach, where operations are executed at the definition point, that is, the graph is created dynamically. This enhances user's understanding of his/her model, as well as debugging capabilities.

In this thesis, PyTorch *Modules* are utilized along its functional package to create the agent's *inner models*, which are the artificial neural networks in charge of processing the agent's observations.

On the other hand, reinforcement learning is implemented by the means of OpenAI's Gym library [35]. It provides standardized *environment* and action/observation *spaces* definitions. It is used for encompassing the workload management environment, and describing both possible actions (scheduling decisions) and observable states.

2.2.3 REINFORCE and actor-critic

Two reinforcement learning algorithms are implemented as examples of use on top of HDeepRM:

- *REINFORCE* [36]: REward Increment = Nonnegative Factor times Offset Reinforcement times Characteristic Eligibility class algorithms constitute the fundamentals for policy-based RL algorithms. In policy-based RL algorithms, actions taken by the agent derive from a *parametrized policy*, where each action has a preference. In this thesis, preferences are also parametrized by weights from a deep ANN implemented in PyTorch.

  REINFORCE is a *Monte Carlo* policy-based RL algorithm, meaning that action preferences are updated at the end of each *episode*, which in this context is a complete simulation.

[4] PyTorch is available in https://github.com/pytorch/pytorch.

- *Actor-critic*: this class of algorithms is a hybrid between value-based and policy-based solutions. A *critic* estimates the long term value of the agent being at the current state (value), and an *actor* produces the set of action preferences (policy), from which the agent selects an action. Implementation in this research uses two independent deep ANNs. The *advantage* function shown in equation 2.4 is also utilized to reduce variability of value-based methods. It calculates the difference between the reward in each step and the estimated value from the critic. This measures the improvement of taking an action with respect to the average value of being at that state.

$$A_\pi(s,t) = r_\pi(t) - v_\pi(s,t) \tag{2.4}$$

For further understanding the intrinsics of these two algorithms, please refer to chapter 13 of Sutton and Barto [37].

2.3 Previous work

Several work has been done in workload management over HPC clusters, however the majority has been focused on job scheduling. The following listed contributions are relevant for this thesis.

As of June 2017, six out of the top ten supercomputers in TOP500 were using the SLURM workload manager [30], including top one at that time, Sunway TaihuLight. SLURM schedules jobs based on their priorities; it supports two types: *priority/basic*, which uses job's arrival time as selecting criteria, and *priority/multifactor*, which allows for different properties to be selected as criteria (time enqueued, number of nodes requested, running time, etc.). It also provides two scheduling approaches: *builtin*, which selects based strictly on priority, and *backfill*, which may allow jobs with lower priority to execute if they do not interfere with the highest priority one.

Backfill scheduling calculates interferences based on job properties. When a user submits a job to Slurm, it does so via *sbatch*; this tool takes a description conveyed in a *job script* shown in appendix A, where these properties are specified. One of them is *"–time"*, which specifies the maximum amount of time the job may be running in the system. This is estimated by the user, and it has been shown to be inaccurate with respect to the the actual running time [38], causes being (1) users overestimating to reduce the probability of the job being killed and (2) modality of the estimations.

Accuracy of estimates has been addressed in literature as a key factor for job scheduling performance, and thus has been a research focus. Authors from [39] proposed a modified version of *EASY* backfilling, named *EASY++*, where they introduce, among other contributions, system-generated predictions. They observed high degree of repeatability in per-user job traces, leveraging average runtime of the two most recent jobs from the same user as a predictor. They achieve average job wait time reductions of up to 33% with respect to *EASY*, and average bounded slowdown reductions of up to 47%. Nevertheless, performance-wise, *EASY++* has obtained results similar to shortest job first policies, nevertheless being fairer.

Further studies of improvements over accuracies have been carried out. Authors from [40] propose a machine learning approach, where they fit a L2-regularized polynomial model for predictions. They show this model outperforms *EASY++* on average bounded slowdown by 11%. Moreover, they compare these results with the optimal or *clairvoyant EASY* implementation, where predictions equal running times; their results come close, however in some traces there is

still room for improvement. A number of features are inputted into the model, including user estimate, statistics for the user based on history, time of the day, and occupied resources. In [41], they also consider the job name and submission directory, whereas they use *auto machine learning* frameworks to find the optimal model.

Parallel to accuracy improvements, another popular approach is policy search. Instead of sticking to a single scheduling rule, such as first come first served, shortest job first or *EASY*, the actual policy is inferred from both the incoming jobs and the resource states. Genetic algorithms have been used successfully to optimize job sequencing [42], while RL has also been used [43] via enhanced *Q-learning* [44]. Most recent solutions leverage deep ANNs for learning the optimal scheduling policy; of relevance is DeepRM [19], an architecture developed at MIT based in DRL and *policy gradients*.

To the authors knowledge, there is not as much work done in resource management, specially in resource selection policies. This may be due to historical resource homogeneity. Main focus of this thesis involves the development of a novel framework which (1) enables simulation of heterogeneous platforms, (2) models resource conflicts and interdependence and (3) builds on DeepRM ideas for DRL-based agent development.

Chapter 3

HDeepRM: design decisions

In this chapter, design decisions concerning both the developed heterogeneous framework and the deep reinforcement learning extension are discussed. This chapter will bridge the inception of the ideas with the concepts needed to implement them, whereas their actual implementation is further described in chapter 4. The following contributions are derived from this chapter:

- A novel framework for heterogeneous workload management based on Batsim ecosystem, supporting resource types, interdependence and selection policies.
- A deep reinforcement learning extension over the developed framework with roots in DeepRM architecture. This includes novel action and observation spaces, reward functions and a new agent interface.

3.1 Heterogeneity support

Heterogeneous computing grows relevant as (1) fundamental limits of general purpose architectures are reached, (2) new trends such as machine learning arrive in the mainstream scene and (3) other objectives being power costs or thermal control become relevant in spite of historical performance focus.

One of the current disadvantages of workload management simulators is the lack of explicit heterogeneity support. Computing resources are conceived as individual nodes to be allocated to incoming jobs, however each of these nodes does not have its own characteristics further than its availability. Moreover, these nodes are independent, meaning that there is no notion of resource sharing and interdependence; due to this, there is no emphasis on resource selection policies. In this section, several design choices are presented as a way to provide heterogeneity support over the Batsim environment using HDeepRM.

3.1.1 Consequences of interdependence

Individuality of compute nodes in state-of-the-art designs does not reflect undesired effects of resource interdependence. In the proposed solution, both *node* and *processor* scopes are defined. Cores (compute nodes in state-of-the-art designs) in the same node share the node's memory capacity. Given a few memory-bound jobs $j < n$, where n is the number of cores in the node, if they are scheduled into the same node, memory capacity may be completely utilized before

$n - j$ free cores are allocated, potentially making them unusable.

On the other hand, cores belonging to the same processor scope share the present *memory bandwidth*, meaning that too many memory-bandwidth-bound jobs in the same processor may decrease individual core performance. Similarly, when only one core is processing a job within the same processor, the shared resources power consumption cannot be amortized by idle cores, decreasing overall computing efficiency.

These effects are relevant motivators of resource selection policies. They are reproduced in the proposed framework as a way for the agents to learn rich selection rules. It is important to note that as of now, HDeepRM does not model task dependencies; for instance, a job requesting 4 cores may launch 4 tasks which need communication and synchronization between them. This is scheduled for future work.

3.1.2 Platforms

Batsim is developed on top of SimGrid, and relies on this framework for the actual simulation. SimGrid is capable of simulating high-scale distributed systems. A system is defined in a *platform definition*, which is an XML file with a structure briefly described in section 4.1.1.

This definition allows for specifying generic components of the simulated system. The following list includes the ones utilized in the developed framework:

- *platform*: root element of the platform definition. Every other component is specified as a child of the platform.
- *zone*: group of resources within the system. It is equivalent to a cluster in a data center.
- *host*: any resource with a explicit function within the system. May be computing or storage related. It can be used to represent any level, from core, to processor to node.
- *link*: connection between two or more hosts.
- *route*: path between two hosts. Links are associated to routes via LinkConnections and may belong to multiple of them.
- *zoneRoute*: path between zones in the data center.
- *prop*: generic way of defining a property associated to a host. Commonly used to express power consumption parameters.

Each host has *P-states*, where each P-state is associated to a computational capacity and a power consumption: $p_s = (c, p)$. Hosts may also be specified as multicore via the *core* XML attribute, which indicates the number of cores. Computational capacity is shared among tasks if their number is higher than the number of cores.

One of the premises that this research proposes is to be able to select and control individual cores. The reason for this is to let the workload manager understand the implication of resource conflicts when dispatching same resource-bound tasks on a single processor; for example, if memory bandwidth intensive tasks are consolidated in the same processor, they will over-utilize the channel capacity, and thus individual core performance will reduce. P-state combined with the multicore scheme is a good start, however there is no way of maintaining per-core state, unless each *host* represented a single core [45].

Another relevant premise is to reflect resource hierarchy and thus ownership and interdependence. Only cores from a certain processor may access its memory bandwidth capacity and not

other processors' capacities. Due to SimGrid Document Type Definition (DTD) [46], it is not possible to nest *hosts* inside other *hosts*, and the use of *zones* for hierarchy expression convolutes the XML file heavily, making it harder to comprehend.

Bound to this limitation, the proposed approach is to design a new platform layer which is then automatically transcribed into the XML format for SimGrid. Designed platform architecture in the heterogeneous framework is observed in figure 3.1.

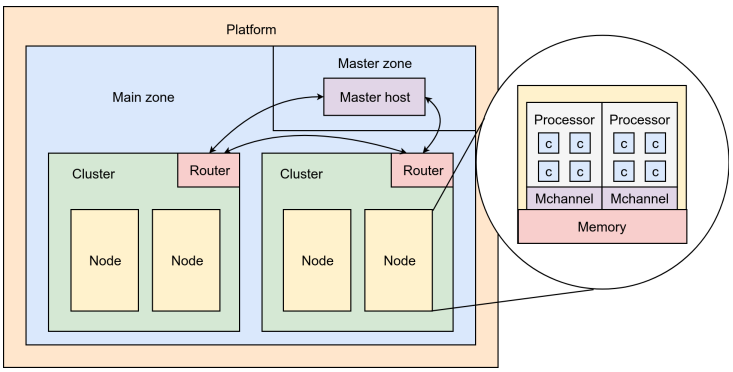


Figure 3.1: Heterogeneous platform design

The following components can be distinguished:

- *Platform*: same as in the XML definition, root element of the system.
- *Main zone*: root zone, owns all the resources in the data center and conveys communications between clusters.
- *Master zone*: separate zone containing the master host.
- *Master host*: special host in charge of executing workload management algorithms. In this thesis, impact of algorithm execution overhead is not considered, however it can be modelled on the basis of this host's computational capacity.
- *Cluster*: range of nodes in a common space. In distributed data centers, such as MetaCentrum [47], clusters from different institutions in different places contribute computational capacity to the platform.
- *Router*: bridge for communicating between clusters and the master host.
- *Node*: computational unit within a cluster. It owns several processors as well as memory capacity. Equivalent to a data center blade with its sockets.
- *Processor*: computational unit within a node. May own one or more cores, which access node's memory through shared memory channels.
- *Core*: minimal computational unit in the system. Computational capacity and power consumption depend on its P-state, which in turn depends on the processor/node congestion. Resource manager selects resources at core-level.
- *Memory and channels*: every node has an amount of memory, and each processor in the node has channels with a given memory bandwidth. Cores share the memory subsystem.

Hierarchy is expressed via several JSON formatted files described in detail in section 4.1.1.

Even though this approach does not leverage *core* attribute provided by SimGrid, it heavily relies on *P-states* for core state. *P-states* are tuples of computational capacity and power consumption. Computational capacity is expressed in *[M/G]FLOPs*, that is, number of floating point operations per second. Power consumption is expressed in Watts. The objective for

learning agents is to understand implications of both consolidating and spreading loads across the computing platform; in order to achieve it, the following four *P-states* are proposed:

- *P0*: job is scheduled on the core, executed at 100% computational capability and consuming 100% power.
- *P1*: job is scheduled on the core, however memory bandwidth for the processor is over-utilized. Job can be executed at 75% computational capability and 100% power.
- *P2*: job is not scheduled on the core, however some other job is scheduled on the same processor. Core is thus consuming 25% of power.
- *P3*: job is not scheduled on the core, and processor is completely unused. Core is consuming 5% power as of idle.

Take into consideration that these numbers are meant to serve as penalties for pivoting the agent actions, they are not empirically obtained. A future approach would be to fine-tune them based on real system benchmarks.

If the agent decides to consolidate the load, it has to consider the memory bandwidth usage, since entering over-utilization (*P1*) would decrease performance of all jobs in the processor. If it spreads the load, unused cores in under-utilized processors would consume more power (*P2*).

It is relevant to distinguish two views of this definition. Batsim, the *simulator*, only sees cores as computing resources, each of them with an associated *P-state*; these are defined in the XML. PyBatsim, the *decision system*, sees the whole hierarchy, understands about nodes, processors, memory and interdependence between them. Benefit from this approach is that decoupling decisions from the simulation allows for it to stay simpler and faster.

3.1.3 Workloads

Workloads constitute incoming jobs into the platform. Batsim provides a specific format for describing workload composition, consisting of a JSON file with a structure briefly described in 4.1.2. Each job has a series of data fields documented in [48].

Batsim introduces the concept of *profiles*. Profiles describe how data is processed and moved around the system for a certain job. There are at least eight types of profiles available[1], however in this thesis two of them are used at most, since network impact is not studied.

The first type is used during testing of the framework, and it is known as *delay*. It is the simplest profile, defining just one data field *delay* with an indication of the number of seconds the job will execute in. When Batsim observes this profile for a job, it schedules it for the amount of time specified in the *delay* field.

The second type, and the one used in the current version of the framework, is the *parallel_homogeneous*. This profile defines two data fields: *cpu* indicates the number of floating point operations to be computed in each core, and *com* the number of bytes sent a received by each pair of distinct machines (as stated earlier, network is not modelled so *com* is not used). Batsim calculates the time a job will be running in a core by dividing *cpu* by the core's computational capability in FLOPS.

One intermediate goal of this thesis is to develop the framework following current standards. In

[1]All types of profiles are available in Batsim's documentation https://batsim.readthedocs.io/en/latest/input-workload.html#profile-types-overview.

this sense, there is a reference workload format known as Standard Workload Format (SWF) [49]. The 18 data fields defined in the standard[2] convey both user request parameters and job execution metrics, that is, they contain both the input and output of the simulation. This is useful for contrasting user request accuracy or evaluating quality of service constraints.

Batsim provides converters from SWF to its own format, however they do not take into account the inclusion of external data fields. A new converter has been developed to be able to add extra data fields to the job profiles:

- *profile.cpu*: real amount of floating point operations associated to the job. SWF has a data field *Requested Time*; this is the user estimated running time, which has been shown to be off the real value. In order to estimate the real running time, a probability distribution of accuracy similar to figure 2 of Tsafrir et al.[39] is used. This estimate is then converted to floating point operations by taking a reference speed; this is based on the average FLOPs per core in the system. The use of a reference speed allows for normalization of job operations. It is important to note that the decision system does not know the real running time of each job when making decisions, it is only seen by the simulator.
- *profile.req_time*: user estimation of running time. Taken from *Requested Time* SWF data field. This is seen by the decision system and used for time related job scheduling policies.
- *profile.req_ops*: amount of requested floating point operations associated to the job. This is used by the decision system to calculate remaining execution. Reason for using operations instead of time is that time may vary with changes in resource *P-states*, whereas operations are absolute.
- *profile.mem*: amount of memory in MB requested for the job. This is taken from *Requested Memory* SWF data field, or if this is not present, from *Used Memory* data field. Jobs can only be allocated if there are enough resources to fulfil the request, so these two fields are equivalent. Original field is in KB, so it is transformed into MB for the framework.
- *profile.mem_bw*: amount of memory bandwidth in GB/s requested for the job. There is no information about memory bandwidth in SWF data fields, so it is decided to generate this field from a random range going from 4 to 24 GB/s, which constitutes from 5% to 75% of offered bandwidths, which are typically 32 to 51.2 GB/s as of spec sheets. Relevance of this field is in teaching the agent implications of consolidating high memory bandwidth jobs in a single processor.

Converter takes an SWF formatted trace, parses it and generates a Batsim-ready JSON trace enhanced with extra data fields.

3.1.4 Components

Conception of the heterogeneous framework begins by identifying the main components, which will then be implemented in the language of choice. It is important to note that Batsim is leveraged for the simulation, so all the effort is applied to designing a decision system. An overview of the design is observable in figure 3.2.

The *Simulator* represents the instance of Batsim. It has three main functions: instantiating the platform, parsing the workload and communicating events to the decision system. Event scheme will be further discussed in subsection 3.1.5.

[2]SWF data field list may be consulted in http://www.cs.huji.ac.il/labs/parallel/workload/swf.html.

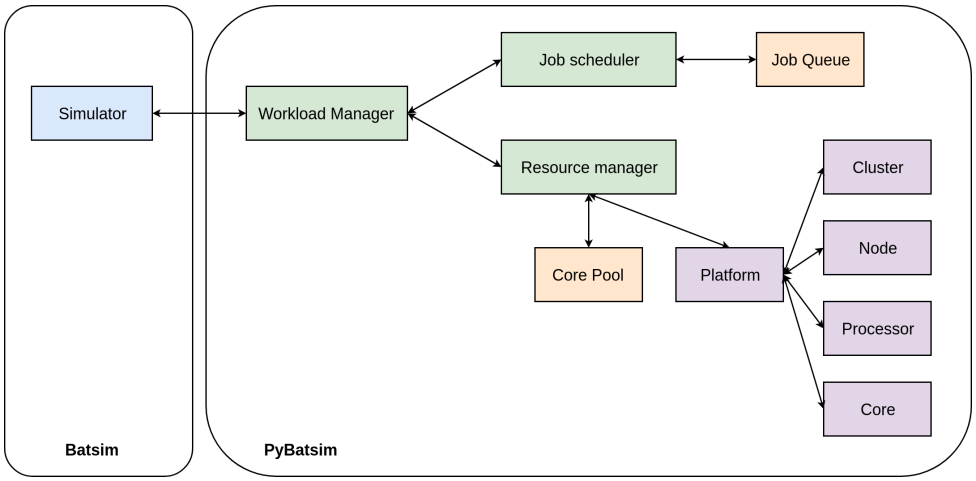


Figure 3.2: Component layout

The *Workload Manager* is the entry point for the decision system. It communicates with the *Simulator* via events. It handles job submissions, job completions, resource releases and changes in P-states. Every decision step, it will call the *Job Scheduler* to pick the next job to schedule and subsequently the *Resource Manager* to allocate resources for that job. Once ready, the mapping between job and resources is communicated to the *Simulator*, followed by changes in resource P-states.

The *Job Scheduler* selects the next job from the *Job Queue* to be scheduled in the platform. Different scheduling policies may be implemented, current framework supports *Random*, *First Arrived*, *Shortest*, *Lowest memory* and *Lowest memory bandwidth*.

New jobs arrive as simulation time advances. The *Job Queue* holds all incoming jobs before being scheduled. *Job Scheduler* has ownership of the queue, and provides different operations over it, like *peeking*, *inserting* or *sorting* jobs.

The *Resource Manager* is in control of the *Core Pool*. When the *Workload Manager* sends a job, the *Resource Manager* checks for availability of resources given the ones requested by the user. When serviceable, those resources' states are updated to reflect their allocation and impact on shared memory, bandwidth and power.

A *Core Pool* contains all cores in the platform, and allows for the *Resource Manager* to select them based on its policy. Current framework supports the following management policies: *Random*, *Highest computing capability (FLOPs)*, *Highest memory available*, *Highest memory bandwidth available* and *Lowest energy consuming*.

The *Platform* represents the hierarchy of resources as a local view to the decision system. It provides relations between each type of resource, and allows for the *Resource Manager* to update shared states between *Cores* in same *Processors*, *Processors* in same *Nodes* and *Nodes* in same *Clusters*. This local view is not exposed to the *Simulator*, which only understands about *hosts* and jobs being served by them.

3.1.5 Simulation flow

In its essence, HDeepRM leverages Batsim's event oriented flow. One of the major contributions from Batsim over SimGrid is the definition of a communication *protocol*. This protocol defines a

request-reply communication style, where the *Simulator* sends a *request* to the *Decision System*, with content regarding what has happened in the platform, and the *Decision System replies* with an action to be taken on the simulation side.

Several events are defined for communication over the protocol[3]. Three types of events may be distinguished relative to the communication direction: *simulator to decision system* events, *decision system to simulator* events and *bidirectional* events. For the current design, events in table 3.1 are of relevance.

Table 3.1: Batsim events utilized in the heterogeneous framework

Simulator - Decision System	
SIMULATION_BEGINS	Platform instantiated and workload ready to be parsed
JOB_SUBMITTED	Job's submission time is reached
JOB_COMPLETED	Job is completed, that is, it has been fully processed
RESOURCE_STATE_CHANGED	Core P-state has been altered in the simulation
SIMULATION_ENDS	Every job has been scheduled and completed
Decision System - Simulator	
EXECUTE_JOB	Job with allocated resources and ready to be processed
SET_RESOURCE_STATE	P-state alteration due to new job in core or local cores
Bidirectional	
NOTIFY	From Simulator when no more jobs are to be submitted

Framework event-based communication can be observed in figure 3.3, which shows the following stages:

1. Before simulation begins, the *Simulator* has to instantiate the platform from the platform definition. This generates all the computing resources (cores) and their default states. It will also parse the workload trace in order to generate the sequence of job arrivals.

2. Once both platform and workload are enabled, the simulation is ready to begin, so the *Simulator* sends a *SIMULATION_BEGINS* request. It then starts looking for the next event in the event loop, which, in the case of the figure, is a job submission. It thus sends a *JOB_SUBMITTED* request. It is important to note that in more complex scenarios, such as the ones presented in chapter 5, several jobs may arrive at the same point in time.

3. The *Decision System* receives this request and triggers the decision process. At the end of this stage, a job will be allocated some resources via both *Job Scheduler* and *Resource Manager*. A reply *EXECUTE_JOB* is sent back indicating the job is ready to be executed.

4. Allocated resources for the executed job have now a different state, and this has to be communicated to the simulator in order to correctly measure times and power consumption. State altering is done inside the *Resource Manager*, however the *Workload Manager* is the one responsible to reply back with a *SET_RESOURCE_STATE* event.

5. Immediately after receiving this reply, the *Simulator* sends another request *RESOURCE_STATE_CHANGED* indicating that the change has been effective inside the simulated platform.

6. Next event in the loop is another job submission, but particularly in this case, it is the last submitted job of the trace. In this situation, a *NOTIFY* request is sent after the *JOB_SUBMITTED* event. The alteration of states is the same as before.

[3]All Batsim's protocol events may be found in https://batsim.readthedocs.io/en/latest/protocol.html#table-of-events

7. When a job completes its processing inside the simulation, the *Simulator* sends a *JOB_COMPLETED* request. In the figure, this only triggers an alteration of the released resources, reason being that there are no more jobs to be scheduled. If there were more jobs waiting, this request would also trigger a scheduling step.
8. When all jobs are completed, the *Simulator* finally send a *SIMULATION_ENDS* request, which closes communications between both subsystems.

The naming *request-reply* refers to the need of the *Decision System* to *handle* the requests and further generate the replies. Handling of events in the developed framework will be explored in subsection 4.1.4.

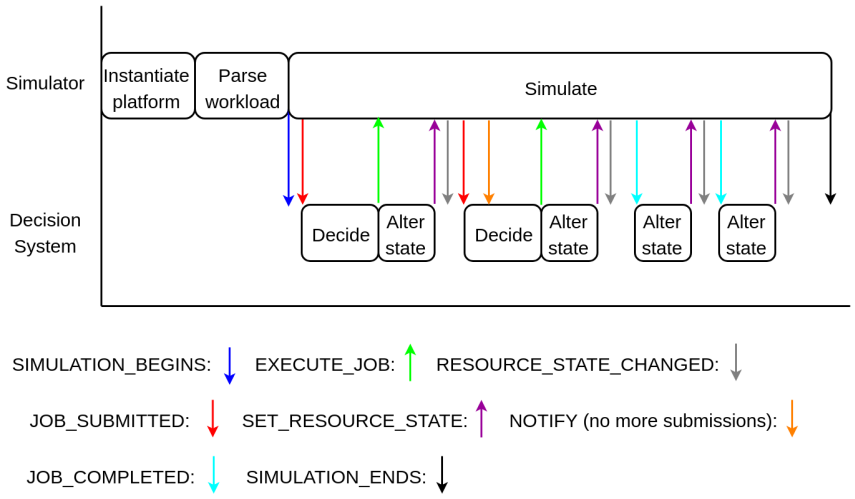


Figure 3.3: Simulation flow. Events are sent between Simulator and Decision System

3.2 Deep reinforcement learning extension

Proposed solution in DeepRM [19] is simple in essence, principally due to being a proof of concept. Currently only two types of resources are taken into account, computing capability and memory capacity, and there are no dependencies between those two, meaning that each core may potentially access the totality of memory. With development of the heterogeneous framework, it is now possible to take into account resource interdependence and also a new resource type, memory bandwidth. In contrast with cores and memory capacity, memory bandwidth is shared between jobs, and might affect individual performance.

The purpose of DeepRM was proving that DRL can improve over other workload management alternatives in literature, and for that they use a simple homogeneous scenario. In this thesis, a novel DRL extension for agent development is designed. This enables researchers to test the feasibility of DeepRM's claims in heterogeneous environments provided by the framework. As a proof of potential, both a REINFORCE and an actor-critic agent are built over the extension.

3.2.1 Extra components

Component overview in figure 3.2 relies on a *Workload Manager* element with a fixed policy established when initiating the framework. This is fine for testing classic policies, however RL problems include new concepts which should be included as an independent extension.

An updated overview of the components is shown in figure 3.4. The following is a brief description of each component function within the system:

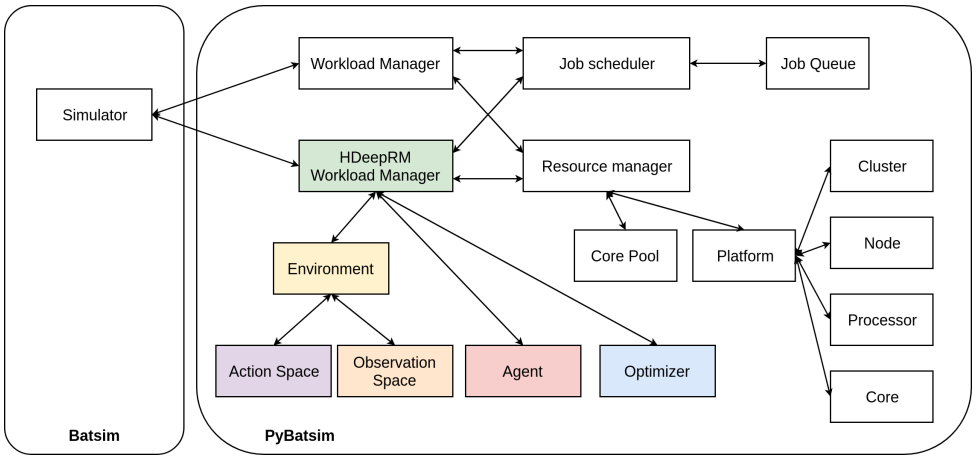


Figure 3.4: Component layout with Deep Reinforcement Learning extension

- *HDeepRM Workload Manager*: entry point when experimenting with Deep Reinforcement Learning algorithms. It has been tuned from the base workload manager adding event flow control between actions taken by the *Agent*, information reported by the *Simulator* and observations made in the *Environment*.
- *Environment*: represents the actual workload management environment. It is important to discern between the observable environment and the actual environment. The first one consists of information received by the *Agent* via *Observations*, whereas the second one includes both the present managers as well as the simulated platform. A useful analogy is that of the agent observing the environment through a dashboard, in which it is possible to take certain *Actions*.
- *Agent*: consists of the model in charge of observing and altering the environment in order to optimize a given objective. As it interacts with the *Environment* through *Observations* and *Actions*, the *Agent* learns a policy, which is a mapping between observations and actions. Note that an observation is associated to a state, and it includes all information perceivable by the *Agent*.
- *Optimizer*: learning process for the *Agent* consists of updating its *preferences*, which are the set of parameters conditioning the probability distribution of actions. In order to do that, a *loss* function is calculated, which measures how good did the agent do. The *Optimizer* takes the output of the loss function and updates the preferences into a higher performing version of the agent.
- *Action Space*: represents the set of all possible actions the *Agent* can choose from. *Actions* are the output of the *Agent*, and they alter the *Environment*.
- *Observation Space*: consists of the set of observations available to the *Agent* as a view of the *Environment* state. *Observations* are the inputs of the agent.

3.2.2 Agents

The entity in charge of taking management decisions is known as the *Agent*. The Reinforcement Learning loop shown in figure 1.1 helps understand the flow of a generic decision. Initially, the agent is proposed an *objective*, and its goal is to optimize it. Objectives evaluated in DeepRM are the following:

- *Average Job Slowdown*: expressed as $\frac{\sum_j^J \frac{C_j}{T_j}}{J}$, where J is the set of jobs in the system, C_j is the time between arrival and completion of the job, and T_j is the running time of the job, with $\frac{C_j}{T_j} \geq 1$. It measures how much of the totality of time is due to the stalling in the job queue. Agent's objective is to minimize this expression.
- *Average Job Completion Time*: expressed as $\frac{C_j}{J}$. Agent's objective is to minimize this value.

Three new objectives are developed for HDeepRM:

- *Average Utilization*: expressed as $\frac{\sum_{t=0}^{T} \frac{AR_t}{TR}}{T}$, where T is the total number of time steps, AR_t is the number of active resources for step t, and TR is the total number of resources in the system. It measures the number of active resources over the simulation time. Agent's objective is to maximize this expression.
- *Makespan*: expressed as $\sum_j^J C_j$. It is the time span from the arrival of the absolute first job until the completion of the absolute last job, essentially the time to complete the whole workload. Agent's objective is to minimize this value.
- *Energy consumption*: expressed as $\sum_r^R r_e$, where r_e is the energy consumption of the resource r over all the simulation time. This is the total amount of energy consumed during the simulation. It is calculated by SimGrid as a function of hosts' power usage. It is expressed in Watts, and agent's objective is to minimize it.

Reward functions, explained in subsection 3.2.4, are tightly coupled with agent's objectives, as they score agent's steps through the simulation.

Going back to decisions, which are the agent's main motive for existence, these are the steps in the developed framework context for a scheduling decision:

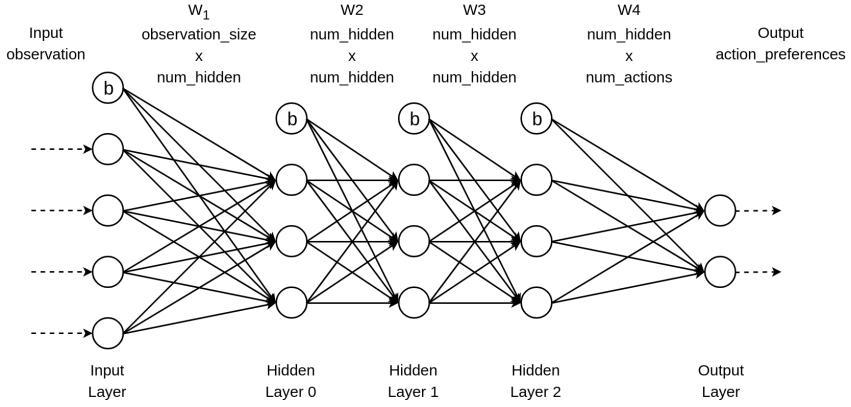
1. Whenever there is a job event, submission or completion, a new *observation* is generated. This gives information to the agent about the workload management environment state, described in detail in subsection 3.2.3.1.
2. With this information, the agent selects an *action*. The way it does it is by *forwarding* the observation through its inner model. This model might be a deep ANN or, in simpler cases, a mapping between observations and actions. Actions are described in detail in section 3.2.3.2.
3. An action produces an environment *alteration*, which consists of the result of allocating requested resources to a selected job. This alteration is not directly observable by the agent, it will rather be evaluated in the next *active point*, which may be another decision or the end of the simulation if there are no more decisions to make.
4. When active point is reached, the agent is *rewarded* in order to provide feedback on the impact of its actions. Furthermore, the agent observes again and the whole cycle repeats.

Each decision is known as a *step*, whereas a full simulation run is an *episode*. This design is based on the idea of *terminal state* (refer to section 3.3 of Sutton and Barto [37] for more details), where the episode ends and the environment is reset to a standard starting state. It is in part reminiscent of DeepMind's approaches to Atari games [50], where the agent movements are considered steps, and the game level is the episode.

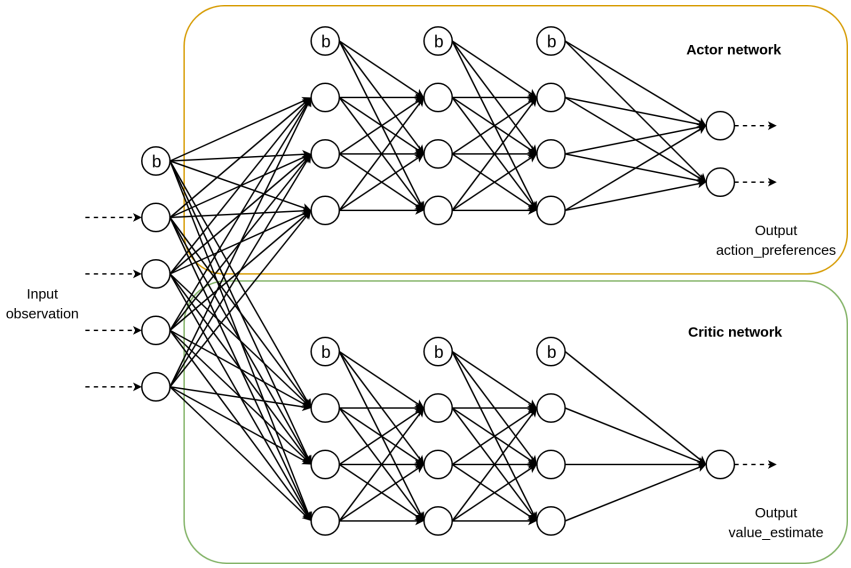
In the original DeepRM paper, REINFORCE is implemented and enhanced with Trust Region Policy Optimization (TRPO) [51]. Agents implemented for validating the HDeepRM framework include REINFORCE without enhancements and actor-critic. REINFORCE serves as the base

algorithm for policy-based RL approaches, while actor-critic is provided as a more sophisticated approach. It is not this thesis objective to develop high performance agents, in fact this is what HDeepRM enables. The user is thus endeavoured to develop new agent architectures adequate to his/her system and workload.

Both agents may be observed in figure 3.5. In the REINFORCE model, only one pipeline of five layers is present. The input layer is fed an observation, which is then forwarded through three hidden layers, and a final output layer, where action preferences are generated. For an initial architecture, every layer is *dense*, such that the layer outputs are of the form $y = activation(dot(W, x) + b)$, where y is the output, x is the input, W is the weight matrix, b is the bias, *dot* is the dot product between input and weights and *activation* is the activation function. A dense architecture is one where each neuron is connected to all neurons in the next layer, thus constituting a *fully connected* network.



(a) REINFORCE architecture design



(b) Actor-critic architecture design

Figure 3.5: Two inner models for agents used in HDeepRM

Activation functions introduce non-linearities in the architecture, and allow the network to learn more complex mappings. *Leaky Rectified Linear Unit* (*LeakyReLU*) is chosen as the activation function; it is an improved version of *ReLU*, which is widely used in literature and has been shown to perform better [52] than others such as *Sigmoid* or *Hyperbolic Tangent*. As observed in equation 3.1, a clipping of negative values is applied element-wise to the output of the dot product plus bias. In contrast to ReLU, where negative values are clipped directly to zero via

$max(0, x)$, in Leaky ReLU the *negative_slope* produces outputs slightly lower than zero, which prevents gradients coming from last layers to be multiplied by zero, losing the error signal in the backpropagation[4] process.

$$LeakyReLU(x) = \begin{cases} x, & \text{if } x \geq 0 \\ negative\_slope * x, & \text{otherwise} \end{cases} \tag{3.1}$$

Softmax, expressed in equation 3.2, is applied to the last layer in order to generate a vector of preferences (probability distribution), one for each action.

$$Softmax(x_i) = \frac{exp(x_i)}{\sum_j exp(x_j)} \tag{3.2}$$

Actor-critic design takes REINFORCE pipeline and duplicates it, thus creating two separate networks. The *actor* network is the one in charge of learning the policy, and it is identical to the REINFORCE network. The *critic* network only shares the input layer with the actor network, and is in charge of estimating a long-term value for being at a particular *state*. This way, the agent can use the value of each step to update its parameters at the end of the episode. *Softmax* is not applied to the output layer, since the value is a scalar, which means that the size of the result will be one.

Learning of the agents is based on *gradient descent*. When an episode is finished, the weights are adjusted based on how they affect the objective, which is the maximization of the expected sum of discounted rewards G_t, as expressed in equation 2.1. This in turn depends on the action probability distribution and, for actor-critic, on the value estimates. Loss derivatives or gradients measure how much does each individual weight contribute to the final loss; they are calculated and backpropagated from the last layers, and serve as factors for updating the weights in the correct direction for maximizing the objective.

Both REINFORCE and the actor network base their learning on a *policy loss*. This is defined in equation 3.3, where a is the selected action, π is the learned policy, θ is the set of weights in the inner model and o is the observation; this is calculated for each decision step, and at the end of the episode, its partial derivative with respect to θ is used for updating.

$$Policy\ loss = -logprob(a|\pi^{\theta}(o)) * r \tag{3.3}$$

The actor network utilizes the *value loss* for learning. It is defined in 3.4, and it is based on the *Huber loss* [53], a more robust alternative to the *Mean Squared Error* [54]. The estimated value is compared to the actual reward in each decision step, and gradient is applied at the end of the episode in order to reduce the difference.

$$Value\ loss = \begin{cases} 0.5(v, r)^2, & \text{if } |v - r| < 1 \\ |v - r| - 0.5, & \text{otherwise} \end{cases} \tag{3.4}$$

[4]Backpropagation refers to the distribution of the error signal from the last to the first layers in the neural network. This is used as a method for weight updating.

The update operation is expressed in equation 3.5, where w_{ij} is the particular weight associated to a pair of neurons i and j, $\frac{dL}{dw_{ij}}$ is the derivative of the loss function with respect to that weight, and α is a hyperparameter known as *learning rate*. This defines how quickly the agent learns from its losses, which is equivalent to how fast the weights are updated. Lower α means the agent will learn slowly, and thus it would need more training steps, however with time it would converge to a local optimum solution; if α is high, the agent will learn fast, nevertheless big variations in weights may not let the agent stabilize in an optimum, meaning that it will bounce around the loss space, even being able to diverge.

$$w_{ij} = w_{ij} - \alpha \frac{dL}{dw_{ij}} \tag{3.5}$$

3.2.3 Environment

The scenario or context which the agent may observe and in which it may exert influence through actions is known as the *Environment*. Interaction between the agent and the environment is framed in what's known as a *Markov Decision Process* [55] or *MDP* (refer to section 3.1 of Sutton and Barto [37]). In MDP terms, the RL loop presented earlier in figure 1.1 is conceived as a *trajectory*, which is a sequence of states, actions and rewards of the form $S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3, ...$

An interesting problem lies in defining the boundary between the agent and the environment. In general, anything that cannot be changed by the agent in an arbitrary way is considered to be part of the environment. A good example of this presented in sec. 3.1 of [37] is a human body, where bones, muscles and sensory organs may not be changed and thus be part of the environment. In the developed framework, muscles may be equivalent to the *Resource Manager* and the *Job Scheduler*, where each would be able to produce an alteration but not be altered per se. In another note, reward functions are also externally defined, thus the agent would not "obtain" a reward, it will rather be "rewarded".

In DeepRM, the interaction is defined as a generalization of MDP, specifically a *Partially Observed Markov Decision Process* or *POMDP*. This means that observations do not totally resemble the environment states, and thus the agent is constraint in the quality and/or quantity of information received. In their case, this is due to the use of a bounded number of jobs M whose characteristics are shown in the observation; these are the first M pending jobs, however, in a given step, the *Job Queue* may have $N > M$ jobs.

Decisions concerning observation and action design in HDeepRM are discussed in the following subsections.

3.2.3.1 Observation space

The agent gathers information from the environment via observations defined in an *Observation Space*. As stated earlier, an observation may be a complete or a partial view of the environment state. In general, observations may be defined in any k-dimensional space, for example data vectors in 1-D or images in 2-D.

In DeepRM, the observation space consists of *cluster images*, like that shown in figure 2 of their paper [19]. In the vertical dimension, time is represented as a series of steps, corresponding to

the small squares. They define a *time horizon*, which is the number of time steps ahead the workload manager may allocate resources to jobs, in this case the value is five. In the horizontal dimension, both contemplated resources, CPU (cores) and memory, are represented. Each small square represents an instance of that resource, that is one unit of CPU and one unit of memory. There is no specific definition of what "unit" means in the paper, however it may be thought of individual cores for CPU units and the minimal requested capacity for memory units. These images are fed into a deep ANN, which constitutes the agent.

When designing HDeepRM, several limitations were found in the original design. Real traces, such as the one discussed in section 5.1, involve thousands of computing resources, not three as exposed in these observations. Requested times are highly varied, from few minutes to several days; given that the time horizon has to be at least the maximum of the requested times, it would grow considerably. Furthermore, traces consist of tens of thousands of jobs, meaning that using small M values would not be representative of the queue status. The observation size would grow in these three directions, making it unfeasible to train a network with such an amount of inputs. An analogy to this is a dashboard with too much information; if the agent cannot understand what is being shown, it will not make the correct decision.

A novel observation space design is then introduced for HDeepRM, which tries to provide a compact and useful packet of information:

1. For each node, the fraction of memory capacity available is shown.
2. For each processor, the fraction of memory bandwidth capacity available is shown.
3. For each core, the current computing capability and power consumption fractions are shown against the maximum possible for that core. Fraction remaining for completing the scheduled job in that core is also shown; this is calculated from the provided user estimation and the current execution span of the job.
4. For each kind of requested resource, being *time*, *cores*, *memory* and *memory bandwidth*, five statistics are shown: the minimum, Q1, median, Q3 and maximum quantities in the whole job queue. This gives the agent a brief idea of the pending jobs distribution for resource request quantities.
5. Finally, a variation ratio of the queue size with respect to the last observation. This ratio falls under the range $[0.0, 1.0]$, with 0.5 meaning no variation between time steps. A new hyperparameter, known as *queue sensitivity* is defined to control the responsiveness of the environment observation to queue variations. When queue sensitivity is high, larger variations are noticeable, however smaller ones are attenuated in the ratio. When it is low, smaller variations are noticeable but larger ones are clipped outside the range, and thus have no impact. For instance, if the sensitivity is 0.2, variations between -20% and 20% are noticed; variations of less than -20% are clipped to 0.0, whereas variations of more than 20% are clipped to 1.0. This hyperparameter needs to be adjusted to the workload trace being simulated.

The reason for all the observation values to be fractions or ratios between 0 and 1 is due to *feature scaling*. When features span in different ranges, artificial neural networks tend to weigh greater values higher whereas smaller values lower. This is comprehended by taking the example of memory availability per node and remaining operations per core; memory per node in the current framework is measured in MB, and a node from the platforms used for experimenting (see section 5.1) has up to one TB of capacity, meaning that the range of availability would be $0 - 1e6$. Jobs may request up to $5.4e5$ seconds of estimated running time, which translated with a reference speed of 1 GFLOPs yields $5.4e14$. This means that remaining operations per core would have higher relevance than available memory per node.

Three types of observations are defined in relation to the amount of information provided. The *normal* type provides all data previously listed, with a number of features equal to $Nnodes + Nprocessors + Ncores * 3 + Nrestype * 5 + 1$; the *small* type skips the per-core information (3rd data item); the minimal type provides only the job distribution (4th item) and variation ratio (5th item). These are meant to be used in varying complexity scenarios.

Data is arranged in a 1-D data vector and sent as input to the developed agents. The three-dimension scaling issue in DeepRM representation is now reduced to a one-dimension scaling, which makes it feasible for larger platforms. Bear in mind that this observation space still constitutes a *POMDP* problem, since the agent does not know the full composition of the job queue. However, instead of showing a fragment of the queue such as in DeepRM, an approximation of the job queue distribution is used as a substitute, which provides a more holistic view of the state.

3.2.3.2 Action space

Agent's way of altering the environment is through actions. The *Action Space* is the set of actions which can be selected by the agent to be applied over the environment. The nature of the space may be discrete, when a set of well-defined actions are present, or continuous, when an action is defined by the assignment of its definition parameters. A simple example in order to comprehend this is a game character and its movements. The agent may be defined as moving *left* or *right*, in which case the action space could be described as *discrete*, and of the form $A_s = \{left, right\}$; in every step of the gameplay, the agent may only take one of those actions. On the other hand, it may also be defined as moving in a certain *direction* and with a given *speed*, which may be represented in radians and metres/second; this means the action space may yield infinite actions depending on these values, and thus could be described as *continuous*, with the form $A_s = \{direction, speed\}$.

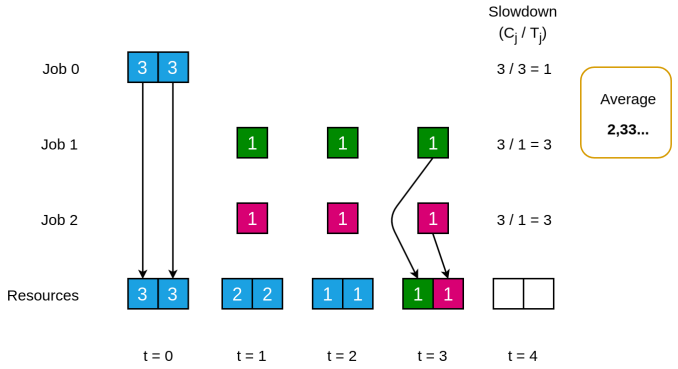
In DeepRM, the designed action space is *discrete*, however they use the concepts of reservations and impossible actions to allow for richness of decision. They define an action as a selection of a job from any of the first M pending jobs, thus the action space is of the form $A_s = \{\emptyset, 1, ..., M\}$. A special *void action* ($\emptyset$) is also incorporated in order for the agent to be able to stall; this is intended to address cases where the result of any other action would result in a worse outcome. An example of this may be observed in figure 3.6.

When a job is selected, resource requirements are checked against the system, and if not enough resources can be allocated within the time horizon, the action is classified as impossible, meaning no job is scheduled and the agent is rewarded negatively. If there are sufficient resources in the current time step, the job is scheduled immediately; if not, but they are within the time horizon, a reservation is set for the earliest time step in which the job requirements are met.

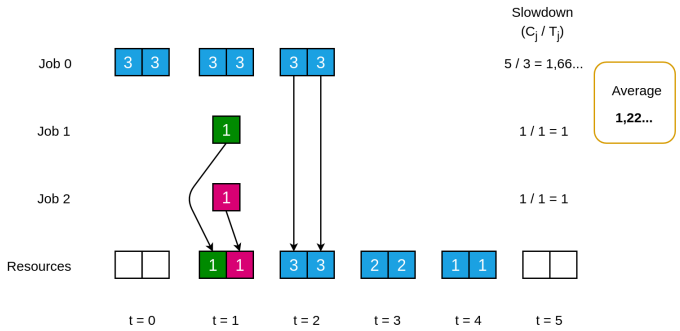
For every time step, several actions might be taken until either a void or an impossible action is selected. This is unconventional, since in the formal definition of an MDP, only one action is taken in each step.

Similar to the observation scaling problem, the action space also grows with the value of M. Moreover, a bigger issue is that of no selection of resources, meaning that all of them are treated as an homogeneous pool, which would not fit in a heterogeneous scenario.

In HDeepRM, limitations from this approach are treated by designing a new action space. Instead of picking from the first M jobs of the queue, the proposed solution is a *policy* space.



(a) Scenario with no void action



(b) Scenario with void action

Figure 3.6: Impact of void action ($\emptyset$) when minimizing average job slowdown

In this space, several classic policies are exposed for the agent to choose from in each time step. A classic policy consists of a pair of $(job\_selection\_policy, resource\_selection\_policy)$, which are shown in table 3.2. In total, there are 36 combinations from classic policies; void action is also incorporated, thus the action space size is 37.

Any subset of policies might be specified by the user to adjust the action space size in their experiments.

When an action is selected, the environment will be altered to use the associated policies in both *Job Scheduler* and *Resource Manager*. These policies are applied until there are no more jobs that can be scheduled in the time step, this being because the queue is empty or due to resource busyness. It has been observed that the average chosen jobs per time step can be as low as two for the evaluated traces, since it is uncommon that multiple jobs arrive at the exact same time step, as well as multiple resources being released in the same step. This means that policy changes are more frequent, and thus provides more value to learning approaches.

3.2.4 Reward functions

A reward function or signal is passed from the environment to the agent every time the agent alters the environment through an action. The reward is designed accordingly to the agent's objective, following the *reward hypothesis*:

That all of what we mean by goals and purposes can be well thought of as the maximization of the expected value of the cumulative sum of a received scalar signal (called reward) [37]

Table 3.2: Classic policies exposed through the Action Space

Job Selection Policies	
Random	Any job may be chosen for scheduling
First arrived	Job with lowest submit time is chosen
Shortest	Job with lowest user requested time is chosen
Smallest	Job with lowest user requested cores is chosen
Lowest memory	Job with lowest user requested memory is chosen
Lowest memory bandwidth	Job with lowest user requested memory bandwidth is chosen
Resource Selection Policies	
Random	Any resource may be chosen for allocation
Highest computing capability	Resource with highest current FLOPS is chosen
Highest core count available	Resource with most available cores is chosen
Highest memory available	Resource associated to the node with most memory is chosen
Highest memory bandwidth available	Resource associated to the processor with most memory bandwidth is chosen
Lowest energy consuming	Resource with lowest current Watts is chosen

As stated in the hypothesis, the reward is a scalar value, and at the end of the episode, the agent will have received as many rewards as steps have occurred. Rewards are discounted by a factor of *gamma*, which is defined to be in the interval $[0, 1)$. When close to 0, only rewards in the immediate future are considered for updating the model; as it approaches the value of 1, rewards from a more distant future will also be considered. Maximizing positive rewards or minimizing negative ones brings the agent to optimize the given objective.

In DeepRM, two reward functions are proposed:

- Minimizing *Average Job Slowdown*: reward is $\sum_j^{AJ} \frac{-1}{T_j}$, where AJ is the set of both scheduled and pending jobs (active jobs) in the system, and T_j is the user requested time or expected duration. This constitutes the negative inverse summation of requested times, with short jobs in the system contributing to a worst reward. If the agent is prioritizing short jobs, slowdowns will also go down, because the working set of jobs will do too.
- Minimizing *Average Job Completion Time*: reward is $-|AJ|$, where $|AJ|$ is the size of the set of active jobs in the system. This is negative the number of unfinished jobs in the system. Low throughput of jobs result in higher completion times.

In HDeepRM, three new reward functions are defined for the three new objectives:

- Maximizing *Average Utilization*: reward is $|AR|$, where $|AR|$ is the size of the set of active (busy) resources in the system.
- Minimizing *Makespan*: reward is $\sum_r^R C_r$, where R is the set of all resources in the system, and C_r is the computational capability being used in resource r. This is the summation of current computational capabilities being used. It supports the idea that as the throughput of operations per time unit in the system increases, all jobs will eventually finish earlier.
- Minimizing *Energy consumption*: reward is $-\sum_r^R P_r$, where R is the set of all resources in the system, and P_r is the power consumption being used in resource r. This is negative the summation of power consumptions in the system. As power grows, reward becomes less valuable.

Chapter 4

HDeepRM: implementation details

In this chapter, the implementation of the developed framework will be explained. Focus will be on showing how tools mentioned in 2.2 have been used to achieve functionality. The content structure has been designed to mirror that of chapter 3, thus it is recommended reading each section while bearing in mind its design counterpart.

4.1 Heterogeneity support

The framework has been implemented following an object-oriented programming [57] approach. Each component has been isolated into its own *class*, and inheritance has been leveraged to increase code reuse and hierarchy semantics.

4.1.1 Platforms

Batsim accepts platforms following SimGrid's XML Document Type Definition (DTD). This states which XML elements are valid and how they may be arranged. A minimal example of a Batsim + SimGrid compliant platform is shown in appendix B.1.

In this platform there is only one *zone*, which can be thought of as a cluster. In this zone, there are two *hosts*: a *master_host*, which is in charge of running the management entities, and a *compute_host*, which will be assigned jobs to process. It is important to note that the *master_host* notation is not standard in SimGrid, however it is required by Batsim. Each of these hosts has a *speed* or computational capability expressed in GFLOPS (Gf).

Moreover, a *link* is defined with a bandwidth of 1 Gigabits per seconds (Gbps) and a latency of 125 microseconds (us). A link has to be attached to a *route* via a *link_ctn*. This *route* communicates a *src* host with a *dst* host, in this case the only two there are in the system. By default, the link bandwidth is *shared* between communication flows, meaning that if both hosts are sending data simultaneously, each of them will receive half of the bandwidth.

In the platform design, cores are represented as hosts. Real platforms utilized in the experiments consist of thousands of computing cores, thus the XML file turns unfeasible to write manually. In HDeepRM, a new platform layer on top of this XML definition is implemented.

An overview of the platform pipeline can be observed in figure 4.1. The following is a description

of each component, as well as representative examples for better understanding.

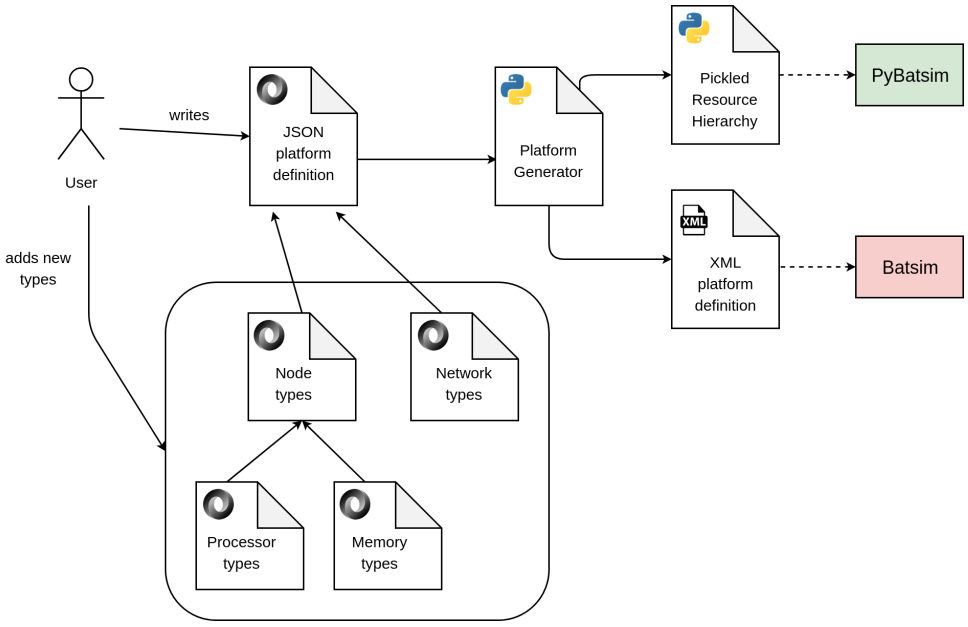


Figure 4.1: Platform pipeline implementation

The user initially writes a JSON platform definition. A minimal example can be observed in appendix B.2. The user describes data center clusters and global links between them. In each cluster, nodes are described along with local links connecting them.

All resource types are specified in separate files belonging to HDeepRM. For instance, if the user needs a new processor type, he/she will have to extend the content of the type files. These are described further in the following paragraphs:

- *node_types.json*: contains node type definitions. An example of a minimal node type is shown in appendix B.3. For each node, processors belonging to it are listed, with type and number. Memory is associated to the node, and it is shared by its multiple processors; in the current framework, Non-Uniform Memory Access (NUMA) effects are not taken into account, thus *capacity* acts as a homogeneous pool for every processor.
- *network_types.json*: comprises network type specifications. In appendix B.4 there is an example for Gigabit Ethernet network technology. Bandwidth is expressed in Gigabits per second (Gbps). Network is currently not modelled in the framework, however if the reader is interested in extending this capability, two clarifications are relevant:
  - The framework links leverage SimGrid's SPLITDUPLEX model; this allows for up and down independent flows, each with the bandwidth specified in the network type. This is useful for modelling full duplex TCP connections.
  - Latency can be expressed in the fields *local_links* and *global_links* of the platform definition. This is because latency depends on the distance between endpoints, and it is not exclusively dependent on the network type.
- *memory_types.json*: contains memory type descriptions. An example for *DDR3-1600* is shown in appendix B.5. Latency expresses the product of memory clock cycle time by the Column Address Strobe (CAS) [58] cycles. In the current framework, memory types are not utilized, since memory access is not modelled; in future work, memory extensions should adhere to this data format.
- *processor_types.json*: processors are referenced from nodes, and their types are contained

in this file. A basic processor definition is shown in appendix B.6. Several fields define a processor:

- *uarch*: microarchitecture of the processor. In the current framework it is purely informational, therefore it is an optional field.
- *id*: unique identification of the processor type.
- *type*: device type, might be CPU, GPU, MIC or others. Current framework only supports specification-based and micro-architectural heterogeneity, future work will also include architectural (see 2.1.2).
- *cores*: number of cores in the processor. This is used by the *Generator* to create the number of hosts related to this processor.
- *clock_rate*: base number of processing cycles per second provided by the processor. It is expressed in Gigahertz (GHz), and used for calculating maximum FLOPs provided. Neither Turbo Boost [59] nor SpeedStep [60] are considered for this thesis.
- *mem_bw*: total memory bandwidth shared between processor cores. This is expressed en Gigabytes per second (GB/s).
- *llc_size*: last-level cache size in Megabytes (MB). It is not used in the current framework, however it is scheduled for an early extension as of future work. It is useful to measure conflicts of jobs with different cache working sets [61] in the same processor.
- *power*: sustained power consumption of the processor in Watts (W). This is estimated from Thermal Design Power (TDP) specifications in product sheets. As stated in [14] sec. 1.5, average power consumption for a given computation is likely to be lower than TDP. In this case, $0.75TDP$ is used as an estimation. In the relative comparison of workload management algorithms, the accuracy of this estimation is not relevant, since all executions are run with the same estimation range.
- *dpflops_per_cycle*: number of double-precision floating point operations achieved in one cycle via vector extensions. This depends on the microarchitecture, and can range from 4 in Intel SSE based systems to 32 in AVX-512 systems with Fused Multiply-Add (FMA)[1] operations. They are used for calculating maximum FLOPs provided by the processor. In this thesis, loads from jobs are considered homogeneous in operation composition. That is, all operations are double-precision floating point, and thus when a core is allocated to a job, this can use the maximum computational capability with the exception of conflicts due to other jobs. In future work, operation profiles will be incorporated into the framework.

The HDeepRM platform definition in JSON is further parsed by a developed *Platform Generator*. This is a Python script which possesses handles for all the JSON type files, and can resolve references between them. This script produces two outputs:

- The XML platform definition compliant with latest versions of SimGrid and Batsim. It structures hosts, links, routes and other components in order for them to be simulated in Batsim. It also sets computing capability and power consumption for each host depending on the processor type. The resulting platform is conceived in the *platform.xml* file.
- The Resource Hierarchy in Python. This is provided to the Resource Manager in PyBatsim for understanding relations between hosts. It is a tree of Python dictionaries, where leaves are Core objects. The Platform root dictionary has a reference to all the Clusters as well as the reference speed for calculating operations from time. Each Cluster dictionary has a reference to its local Nodes. Each Node to its local Processors, and each Processor to its local Core objects. Every data element also has characteristics provided by the JSON type files. In order to serialize it into a data file, the Python *pickle*[2] library is used, and

[1] Executes a multiply and an addition (2 operations) in one cycle.
[2] The *pickle* library documentation is found in https://docs.python.org/3.6/library/pickle.html.

the resulting file is known as *res_hierarchy.pkl*.

At the end of this pipeline, the *platform.xml* is ready to be served to *Batsim*, while the *res_hierarchy.pkl* file can be deserialized by *PyBatsim*. However, *Batsim* still needs the workload definition, which is explained in the following section.

4.1.2 Workloads

Workload management simulators such as those described in section 2.2.1, present different workload and platform specification formats. For the Batsim ecosystem, the workload specification is a JSON based file, and a minimal example may be found in appendix C.1. In it, the number of cores *nb_res* is expressed at the top, followed by incoming jobs and associated profiles. Both job and profile fields have been explained in section 3.1.3.

Nevertheless, the majority of simulators also provide tools to parse and adapt the Standard Workload Format (SWF), which is an effort to transfer workload traces in a standard structure. Several traces from older research have been uploaded in SWF to the Parallel Workloads Archive[3], ranging from 1993 to 2015. The SWF format specifies one job per line, with data fields separated by spaces and comments beginning with ";". An example of a trace with two jobs might be observed in appendix C.2.

It is noticeable that some data fields, such as memory requested (*memReq*), are provided as -1, meaning that they are not available in the trace. For these cases, a data cleaning process is issued before working with the trace; several cleaned traces are already in the archive.

Batsim offers converters located in the "tools" directory[4], however they are limited to compute and time data fields. Moreover, they have not been updated for changes introduced in latest Batsim revision v3.0.0. For these reasons, a new *converter* is implemented as a Python script for translating between SWF and Batsim workload format. It reads SWF file line by line and produces a JSON Batsim-ready structure with profile fields described in section 3.1.3. This file is known as *workload.json*, and it is served along with *platform.xml* to Batsim.

Moreover, due to the design of the observation space explained in section 3.2.3.1, resource request upper limits are needed for calculating percentiles in the job distribution estimation. These are usually specified in the workload manager middleware configuration, however the *converter* will take care of obtaining them by analysing statistics from the workload trace. These limits indicate the maximum amount of resource requested by an individual job:

- *max_time*: upper limit for requested running time. In seconds.
- *max_mem*: maximum for requested memory capacity. In MB.
- *max_mem_bw*: upper limit for requested memory bandwidth. In GB/s.

The limits are pickled in a *job_limits.pkl* file, which is later loaded by PyBatsim in runtime.

[3]The Parallel Workloads Archive is accessible from http://www.cs.huji.ac.il/labs/parallel/workload/.
[4]Converters may be checked online in https://github.com/oar-team/batsim/tree/master/tools.

4.1.3 Components

A new software package named *hdeeprm* is currently available in the Python official repository index, PyPi[5]. Framework components have been divided into different Python *modules* as to decouple functionality. The base module structure for the heterogeneous framework can be observed in figure 4.2.

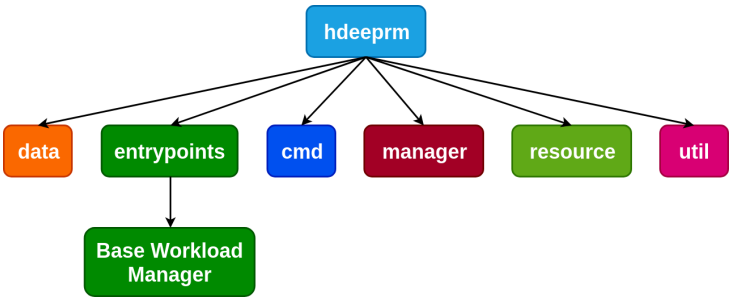


Figure 4.2: HDeepRM base module structure

The *data* directory contains all the JSON resource type files described in 4.1.1. It is used by the *Platform Generator* for establishing relations between resources and generating both the *platform.xml* and the *res_hierarchy.pkl* files.

The *entrypoints* subpackage contains the *BaseWorkloadManager.py* module. PyBatsim needs an entrypoint for instantiating a Batsim proxy, which is a local object to the decision system for accessing and communicating with the Batsim simulation. Entrypoints are defined in their own modules, and the name of the module needs to be the same as the class inside it. For instance, the *BaseWorkloadManager.py* module provides a single class named *BaseWorkloadManager*. All entrypoint classes have to inherit from the PyBatsim *BatsimScheduler* class. This provides event handlers for processing messages received from the simulation.

The *cmd* module provides command line utilities for easily launching HDeepRM experiments and analysing the results. They will be explored in section 4.3.

The *manager* module provides both the *Job Scheduler* and the *Resource Manager* implementations. Any of the policies specified in table 3.2 may be applied by modifying the *sorting_key* parameter. In essence, when the Agent selects an action, it alters the sorting keys of both managers in order to change the selection policies.

The *resource* module contains the *Core* class, which is the leaf resource in the resource hierarchy. Cores provide methods for updating and consulting their *state*, which consists of their current computing capability and power consumption. They are also linked to the job they are serving and the parent resources for hierarchy exploration.

Finally, the *util* module supplies the platform and workload *generators*. These are called during the launch of an experiment to produce the set of files needed for both Batsim and PyBatsim.

In order to better understand the module relations, a class diagram with interactions is shown in figure 4.3. All classes and methods have been documented and are available online at the time of this writing [62]. Source code can be explored for each module, and it is also available on GitHub [63].

[5] *hdeeprm* can be accessed and downloaded from https://pypi.org/project/hdeeprm/.

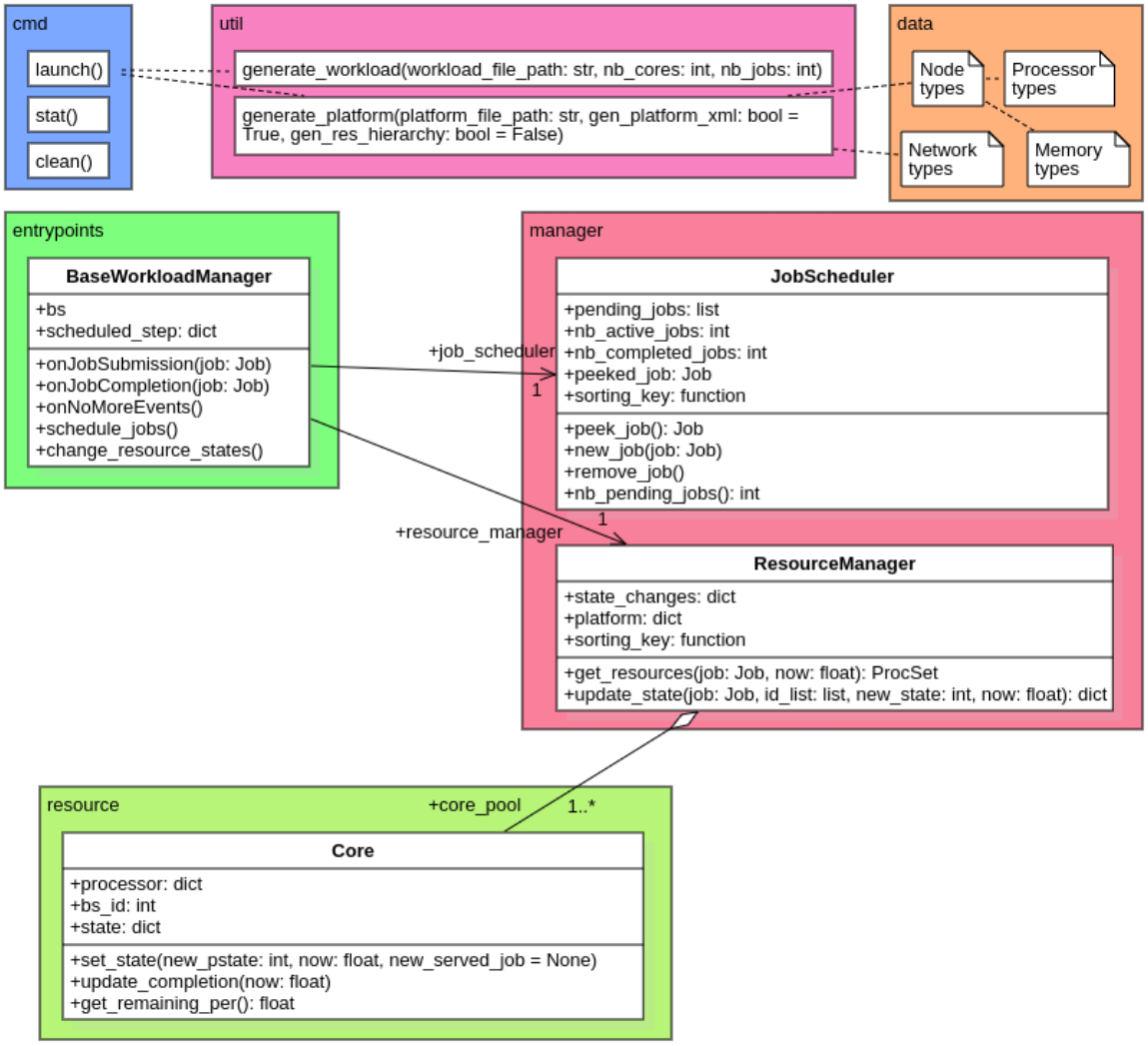


Figure 4.3: HDeepRM base module interactions

4.1.4 Event handling: practical example

In this section a simplified framework execution will be detailed in order to understand the flow of events. The following reduced platform and workload will be used:

- Platform: one cluster, with one node and one dual-core processor. This means there are 2 computing resources in the system within the same processor scope. Each of the cores provides 10^9 FLOPs in P-state 0. The processor's memory controller can handle 24 GB/s of memory bandwidth, while there are 16 GB of memory in the node.
- Workload: one job needing 50^9 total operations. It requires 8 GB of memory and a sustained memory bandwidth of 2 GB/s. Its arrival time is at $t = 0$.

When the launcher is called, all the files (*platform.xml*, *workload.json*, *res_hierarchy.pkl* and *job_limits.pkl*) are generated. Immediately, Batsim and PyBatsim are executed as independent processes, and both establish a connection over the local network. The first message is sent by Batsim at $t = 0$, and it indicates the beginning of the simulation, as shown in appendix D.1.

A *message* consists of JSON content carrying multiple *events*. The message includes a *timestamp*

indicating the simulation time at its emission. The request-reply system induces no latency, meaning that message reception timestamps are the same as emission ones. For each event, the *type* field indicates the kind of event as specified in table 3.1.

The event payload is conceived in the *data* field. Number and characteristics of resources available in the platform, as well as workload references are within this field. In spite of profiles being specified explicitly in the *SIMULATION_BEGINS* event, jobs are generated dynamically by reading the *workload.json* file; this is because the vast amount of them would incur high network traffic and thus slow down the simulation. Another relevant aspect is the ID established automatically by Batsim to the workload: *62b680*. This is used in conjunction with the profile name to do the job-profile mapping.

When PyBatsim receives this message, it stores the platform and workload information locally. There is no special handling for this event type in HDeepRM, however the request-reply protocol mandates a response. By default, PyBatsim sends a message with an empty set of events.

The simulation instance forwards until the next event, which is the arrival of the job at $t = 0$. When this happens, it sends a new message shown in appendix D.2. The *JOB_SUBMITTED* event contains a reference to the job's profile, where further information about requirements is specified. PyBatsim has received all profiles from the *SIMULATION_BEGINS* event, and it can index the job's profile by the provided workload ID and profile name. The *onJobSubmission()* callback is triggered, and the Workload Manager sends the job to the Job Scheduler so it ends up in the Job Queue.

Any job submission or completion triggers the scheduling process, since new jobs or resources are available for the decision. In this case, the Workload Manager decides to associate the job to the first core in the platform. This in turn triggers a change in resource states; since the processor is now active, both cores are consuming power, in particular the first core will be in P0, whereas the second one will be in P2 (refer to section 3.1.2 for more on P-states). This information is sent to Batsim in another message shown in appendix D.3. Notice that both *alloc* and *resources* indicate the core IDs, not the number of them.

Batsim receives the message, alters the P-states and starts executing the job. Based on the *cpu* field in the job's profile and the *speed* field in the core XML specification, it calculates the running time for the job. In this case, the speed provided by the core is 10^9 FLOPs, while the *cpu* operations are 50^9, so this job would run in 5 seconds. Batsim schedules an event at $t = 5$ and forwards to the next event.

Whenever there is a request for changing P-states, Batsim responds with an immediate message confirming the change in the simulation. Observe that the simulation time is still $t = 0$, since there are more messages to be processed. It happens that at this time step, there are no more jobs to be submitted, since the workload was composed of only one. A notification indicating this is concatenated with the state change confirmations in the same response message, shown in appendix D.4.

HDeepRM does handle these events with an empty response, since there are no processing needs. Batsim forwards to the next event, which happens at $t = 5$, and it is the completion of the job. Batsim processes the event, sending a completion message to PyBatsim. It also recognizes that once this job is completed, the simulation is finished: there are no more jobs pending or active. Due to this, it also concatenates the *SIMULATION_ENDS* event, which when received by PyBatsim will shut down the simulation. This is shown in appendix D.5.

A summary of the event flow is described in section 3.1.5. This pattern is also observed in com-

plex traces, however it is harder to identify due to interlacing of distinct job submission/completion flows. When the simulation is not working as expected, understanding how to interpret the flow is necessary for the user to debug his/her deployment; on the other hand, *Evalys*[6], the analysis toolset for the Batsim ecosystem, also utilizes it for visualizations.

4.2 Deep reinforcement learning extension

Deep reinforcement learning functionality has been developed in separate modules and appended to the base implementation. In this section, the final module structure is described, and details about each new component are introduced, encompassing the agent API, environment and reward system.

4.2.1 Extra components

The extended module structure from that shown in figure 4.2 can be observed in figure 4.4, and may also be found in the same GitHub repository [63].

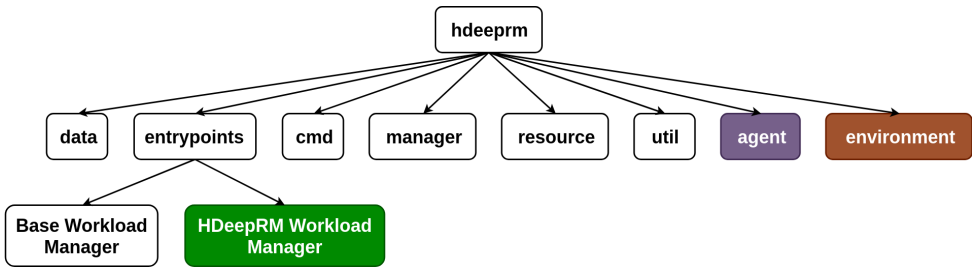


Figure 4.4: HDeepRM extended module structure

A new entrypoint named *HDeepRMWorkloadManager* is introduced to the framework. Extended from *BaseWorkloadManager*, it integrates the event flow shown in section 4.1.4 with the Reinforcement Learning loop.

The *agent* module provides all superclasses in the agent API. Users should utilize these for defining their own agents for the simulation. They will be further detailed in section 4.2.2.

The *environment* module implements the HDeepRM workload management environment. This contains both the action and observation definitions, and it also provides the reward system for the agent. It will be explained in section 4.2.3.

The final relation diagram can be found in figure 4.5. Base modules have been reduced for clarity.

4.2.2 Agents

The main value behind the extension is allowing users to experiment and evaluate their own Deep reinforcement learning agents. HPC services are diverse in both resources offered and job arrival/requirement patterns from users. An agent inner model conditions the latency of decision steps, thus it is relevant to adequate the complexity of this to that of the environment.

[6] *Evalys* is hosted in https://gitlab.inria.fr/batsim/evalys

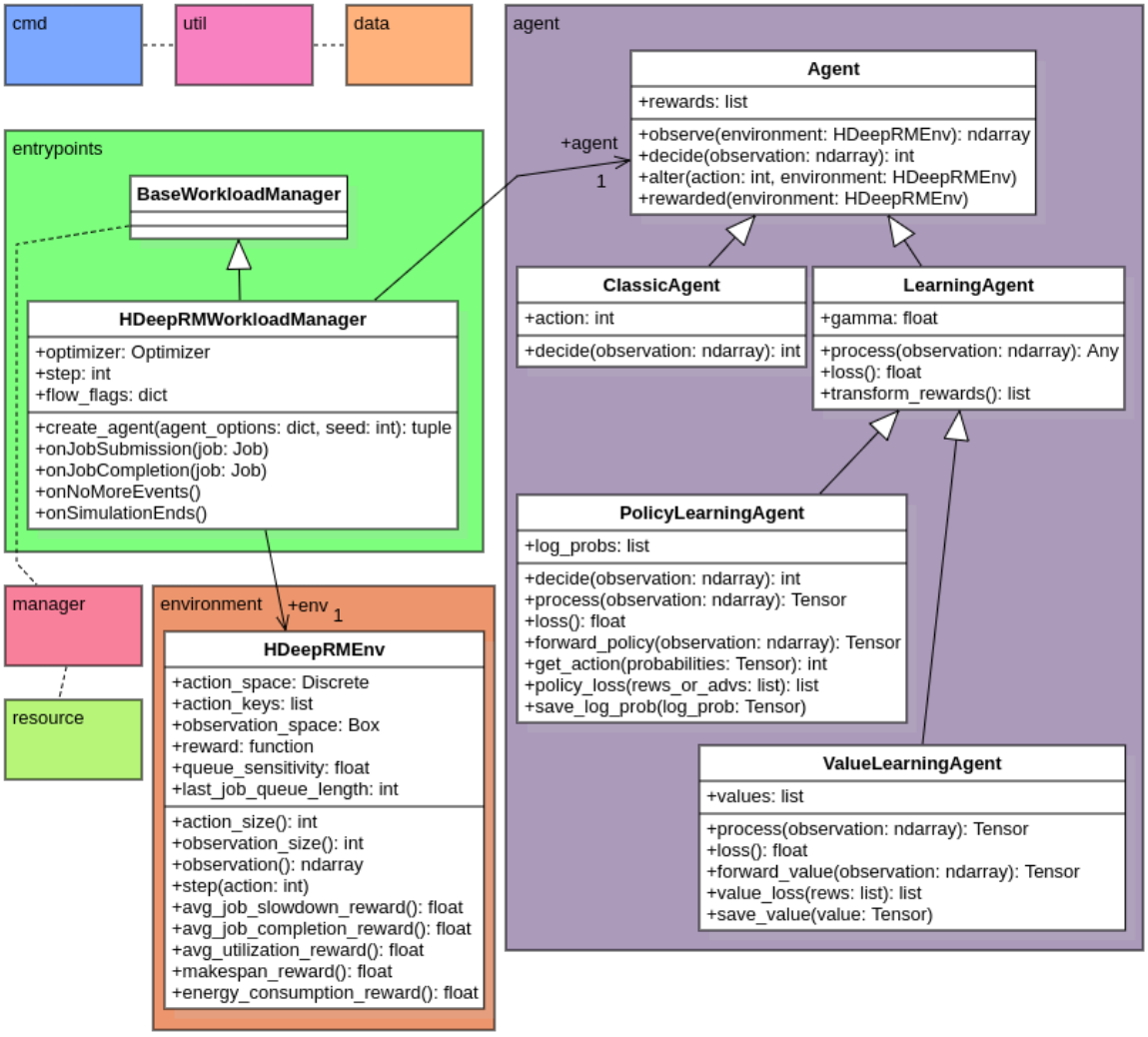


Figure 4.5: HDeepRM extended module interactions

HDeepRM provides a flexible agent API for definition and integration inside the framework. A base *Agent* is defined as the superclass of all agents. This extends the PyTorch *nn.Module*[7] interface for inner model implementation, providing the agent with a set of *parameters*. These are updated each time the agent is trained against a given scenario.

An agent may carry out the following operations:

- *observe*: given the environment, the agent can request an observation in order to obtain information from the current state.
- *decide*: from the observation the agent makes a decision, which consists of selecting the most valuable action for that state.
- *alter*: applies the selected action to the environment. This modifies the Job Scheduler and Resource Manager policies, and triggers the scheduling of jobs over resources.
- *rewarded*: once the action is applied, the agent asks the environment for a reward. Rewards are stored in the agent, and are later utilized for calculating the performance loss.

[7]Module API can be found in https://pytorch.org/docs/stable/nn.html#module.

The *ClassicAgent* offers the possibility of creating agents based on fixed classical selection policy-pairs, supporting combinations from any of the policies in table 3.2. In particular, the agent will always make the same decision based on its policy, thus its inner model is just a mapping to the action ID. The purpose of this interface is for the user to compare developed agents to classical approaches, thus establishing performance baselines.

The *LearningAgent* is the superclass for all trainable agents. During the decision step, learning agents *process* the observation through their inner model, which generally consists of forwarding through a defined artificial neural network (ANN). Layers and activation functions of this ANN are implemented using PyTorch's functional API[8]. In order to update the agent's parameters, a *loss* function is calculated based on the reward history; these are previously discounted and normalized through *transform_rewards*.

A *PolicyLearningAgent* utilizes policy gradients for conceiving a near-optimal mapping of observations to actions. The *process* method output, and thus the inner model output is the probability distribution over all possible actions given an observation. Deciding involves selecting an action from the distribution; there is always a chance of picking lower probability actions, encouraging exploration. The *policy loss* is implemented from the definition in section 3.2.2.

The *ValueLearningAgent* learns an estimation of the expected future reward for being at a given state. It does not decide by itself, instead it is meant to be used as a *critic* in actor-critic approaches. The value loss is also implemented from the definition in section 3.2.2.

When designing and evaluating agents, the user should follow these four rules:

1. Specify the *policy pair* for a fixed-policy approach in the *options file*, which will be introduced in section 4.3. There is no need for extending the *ClassicAgent* class.
2. Extend *PolicyLearningAgent* for designing independent policy gradient agents and *actors* in actor-critic algorithms.
3. Extend *ValueLearningAgent* for defining *critics* in actor-critic algorithms.
4. Extend directly *LearningAgent* for defining other types of agents, such as those based on *Q-learning* [44].

The REINFORCE and actor-critic implementations are available as examples under *agent_examples* in the repository.

4.2.3 Environment

Standardization of the environment was carried out by utilizing OpenAI's Gym library. In this framework, an environment provides an *observation space* and an *action space*. It also offers a principal method *step*, which takes an action to be applied over the current state. When called, *step* minimally returns the new observation and the reward for the effect of the action.

In HDeepRM, *step* has been modularized for easier readability. A new method *observation* offers the current observation of the environment; it may be called multiple times without altering the environment. The *step* method is reduced to only applying the alteration; this decision is based on the semantics, which imply dynamism. Finally, a *reward* method provides the agent with the feedback for its actions; since the reward depends on the optimization objective, this method is mapped to the adequate function from those detailed in 3.2.4.

[8] PyTorch's functional API may be found in https://pytorch.org/docs/stable/nn.html#torch-nn-functional.

4.2.3.1 Observation space

The *observation space* is based on a *Box* space from the Gym library. Boxes are n-dimensional spaces, just like tensors are n-dimensional data structures. They are bounded by maximum and minimum values.

The shape of the box for HDeepRM is a 1-dimension array, with size calculated as expressed in section 3.2.3.1. The lower bound is an array of all zeroes, and the upper bound an array of all ones. The reason for the scaling of observation features to the range $[0, 1]$ was also explained in the mentioned section.

When the agent observes the environment, a new observation is formed compliant with the observation space specification.

4.2.3.2 Action space

For the *action space*, a *Discrete* space from Gym library is utilized. This defines a set of actions of size n. In HDeepRM, the number of actions is 37, including the void one (see section 3.2.3.2).

The agent processes the observation and selects an action through its *decide* method. The environment is passed this action in the *step* method call, where it is asserted to exist within the action space. If it does not, an error is raised.

4.3 Framework usage

In this last section, the usage flow of the framework will be introduced. This is intended for users to understand the main steps in defining and running experiments.

On installing the framework, an integrated experiment launcher is provided. This is immediately accessible in the command line as *hdeeprm-launch*. It accepts several options shown in appendix E.1.

The *options_file* is a JSON formatted file containing the options defining the experiment. The following global options are currently accepted:

- *seed*: random seed for reproducibility.
- *nb_resources*: total number of cores in the simulated platform.
- *nb_jobs*: number of jobs to be generated from the original workload.
- *workload_file_path*: location of the original workload in SWF format.
- *platform_file_path*: location of the original platform defined with HDeepRM JSON syntax.

Options for the PyBatsim decision system are specified in their own entry. Currently, the following are supported:

- *log_level*: logging level[9] for obtaining insights from the simulation.
- Environment options:

[9] Accepted logging levels are those defined by Python, check https://docs.python.org/3/library/logging.html#logging-levels.

  - *objective*: metric for optimization during the simulation run. One of *avg_job_slowdown*, *avg_completion_time*, *avg_utilization*, *makespan* or *enery_consumption*.
  - *actions*: user-defined subset of actions based on those implemented by HDeepRM. Allows for different dimension experiments.
  - *observation*: type of observation, one of *normal*, *small* or *minimal*, as defined in section 3.2.3.1.
  - *queue_sensitivity*: sensitivity of the observation to variations in job queue size.
- Common agent options:
  - *type*: type of agent to be utilized. One of *CLASSIC* or *LEARNING*.
- Learning agent options:
  - *run*: type of run for the learning agent. One of *train* or *test*. When training, the agent's inner model is updated, whereas testing is meant for evaluation purposes.
  - *hidden*: number of units in each hidden layer from the agent's inner model.
  - *lr*: learning rate for updating the agent's inner model.
  - *gamma*: discount factor for rewards.

Appendix F contains an example of an experiment involving a learning agent. Notice that classic agent options can be excluded when utilizing a learning agent and vice versa. For further examples, the reader is encouraged to check the official documentation [62].

For learning agents, there are three optional parameters. The *AGENT* refers to a path to the Python file where the user has defined the actual agent. Agents should be defined in classes extending *LearningAgent* or any of its subclasses. HDeepRM reads this file and it dynamically imports, generates and integrates the agent within the framework.

An example of a minimal learning agent file is shown in appendix G. The constructor parameters are passed from the options file, and should be always defined. The integration with the environment to *observe*, *decide* and get *rewarded* is provided by superclasses.

A custom workload specified in Batsim JSON format can be passed with the *-cw* option. This allows for manually defining workloads, which would be tedious to do in SWF format.

When training agents, HDeepRM is capable of saving the inner model parameters for bootstrapping from pre-trained agents. The *INMODEL* is the path where the pre-trained model is located; if indicated, this will be loaded into the agent previous to the simulation. The *OUTMODEL* is the path where to save the model parameters after the simulation; if not specified, parameters will not be saved.

Notice that all these paths may be provided in their relative form, HDeepRM will take care of finding the absolute location within the system.

When called, *hdeeprm-launch* will carry out the following steps:

1. Check for the existence of the *workload.json*, *platform.xml*, *res_hierarchy.pkl* and *job_limits.pkl* files in the current directory. If they do exist, it will skip regenerations from originals. This may also be utilized as a way of creating a custom workload directly from the Batsim JSON format, without providing a SWF trace.
2. Generate all missing files from paths specified in options.
3. Execute *pybatsim* as a background process passing its options entry. Moreover execute *batsim* as a foreground process for obtaining event flow information during the simulation.

Once the simulation is over, several files are produced as a result. For Batsim output files, the official documentation can serve as a source[10]. As part of the software package, HDeepRM also offers the *hdeeprm-visual* and *hdeeprm-metrics* utilities. These may be utilized for visualising the outcomes and comparing simulation runs metrics, respectively. They also have the ability to save the produced figures via the *-s* option, which indicates the output file name.

hdeeprm-visual utilizes the *evalys* library as part of the Batsim ecosystem, and can be observed in appendix E.2. Currently, these types of visualization are supported:

- *queue_size*: plots the evolution of the Job Queue size over the simulation time.
- *utilization*: show the utilization of cores over the simulation time.
- *lifecycle*: for all jobs, visualise arriving, scheduling and completion event timestamps over time.
- *gantt*: a Gantt chart showing the occupancy of jobs over cores in the system. Jobs are labelled by their job ID inside Batsim.
- *gantt_no_label*: same as *gantt* but without labelling.
- *core_bubbles*, *mem_bubbles* and *mem_bw_overutilization*: show core and memory conflicts over time, as well as spans for bandwidth over-utilization in each processor.
- *losses*, *rewards*, *action_preferences*: provide evolution of losses and rewards, along with the dominance of actions over different simulation episodes.

Examples from these and more visualisations are available in *evalys* official documentation[11].

hdeeprm-metrics is implemented over *matplotlib*[12], and its usage may be observed in appendix E.3.

It takes the *out_schedule.csv* files from the Batsim output of the two simulation runs in order to load, plot and compare the metrics. This, along with *hdeeprm-visual*, is utilized in chapter 5 for plotting purposes.

For the analysis of the workload trace, *hdeeprm-jobstat* may be used. This can calculate different statistics such as mean, median or percentiles over job data fields. Currently, *req_time*, *size*, *mem* and *mem_bw* are supported. Its usage may be observed in appendix E.4.

HDeepRM also generates two output files on top of the Batsim outcome: *insights.log* provides step by step information about several parameters and events such as observation composition, actions taken or memory bandwidth overutilizations; *rewards.log* contains the cumulative sum of all rewards obtained by the agent during the simulation. It is designed to be appended by consecutive simulation results when training an agent.

Finally, a *hdeeprm-clean* utility is provided for fast testing, which erases every file except for the *options file* in the current directory. This should be used with care.

[10]Batsim output files and descriptions are available in https://batsim.readthedocs.io/en/latest/tuto-result-analysis/tuto.html#files-overview.

[11]Evalys documentation available in https://evalys.readthedocs.io/.

[12]Matplotlib is the most popular Python 2D plotting library, documentation can be found in https://matplotlib.org/.

Chapter 5

Evaluation

In this chapter, an exhaustive validation of HDeepRM functionality has been carried out by undertaking several experiments. Real traces from the Parallel Workloads Archive are simulated in conjunction with the heterogeneous platform they were generated from. Learning capabilities are demonstrated and explored in a multiple outcome scenario. Results show (1) completeness and validity of the framework, (2) trade-offs derived from heterogeneous configurations and (3) potential of learning strategies for obtaining near-optimal solutions. All experiment configurations are available under *thesis_experiments* in the official repository [63].

5.1 Platform and workload

Several platforms have been implemented based on traces from the Parallel Workloads Archive (PWA), all of them are available under *platform_examples* in the repository [63]. In these experiments, the Gaia Cluster from the University of Luxemburg is utilized. The platform is available as *platform_examples/gaia.json*. It contains a single heterogeneous cluster composed of 153 nodes, 342 processors and 2280 cores. Configuration of resources is found in table 5.1. Bear in mind that the simulation of this system would not have been possible in plain Batsim, since it does not natively support resource hierarchies and heterogeneous resources.

The workload trace is the second most recent in the PWA, spanning from May 2014 to August 2014. During these three months, 51987 jobs were recorded coming from 6 main users. In the experiments, continuous fragments of the trace involving a lower number of jobs will be used for clarity. Besides, the first 4 jobs from the trace have been cleaned due to being issued considerably earlier than the rest of them. The exclusion of these is not significant for the simulation, and helps for consistency of visualizations. The Gaia SWF may be obtained from PWA's portal [64].

The evaluation has been carried out in a commercial computer, specifically an Acer Aspire E1-571 laptop composed of a 2.6 GHz Intel Core i5-3230M CPU. Nevertheless, a 2x speed-up for Gaia trial runs with 10000 jobs has been observed when using a 3.2 GHz Intel Core i5-650. This means that HDeepRM benefits from having higher single-core performance. Bear in mind that GPUs are not utilized for the simulation. In a future iteration, both the forward passes and the backpropagation to update the model parameters will be offloaded to a GPU.

Table 5.1: UniLu Gaia cluster configuration as in PWA

Node type	NumN	Mem	Processor type	NumP	Cores	ClockRate	MBW
Bullx B500 (0)	60	48	Xeon L5640	2	6	9.04	32
Bullx B505 (0)	2	96	Xeon L5640	2	6	9.04	32
Bullx B505 (1)	10	96	Xeon L5640	2	6	9.04	32
Bullx S6030	1	1024	Xeon E7-4850	16	10	8	51.2
Dell R820	1	1024	Xeon E5-4640	4	8	19.2	51.2
Dell R720	5	64	Xeon E5-2260	2	8	17.6	51.2
Bullx B500 (1)	72	48	Xeon X5670	2	6	11.72	32
Delta D88x-M8-BI	1	3072	Xeon E7-8880 v2	8	15	20	85
SGI UV-2000	1	4096	Xeon E5-4650 v2	16	10	19.2	59.7
NumN	Number of nodes of that type in the cluster						
Mem	Memory capacity in each node (in GB)						
NumP	Number of processors of that type in each node						
ClockRate	Clock frequency for each core in the processor (in GFLOPs)						
MBW	Memory bandwidth in each processor (in GB/s)						

5.2 Primer on simulation analysis

This experiment is designed to demonstrate the validity of HDeepRM for simulating heterogeneous platforms. It also verifies the correct functionality of the *ClassicAgent* interface. Along the Gaia cluster, one more real heterogeneous platform is simulated for demonstrating versatility: the MetaCentrum2 grid (Czech Republic, January 2013 - April 2015 period). More information on this HPC service is available in its the PWA web page [65].

Two classic policy-pairs are compared as to explore metrics: *first-high_gflops* and *shortest-high_gflops*. A total of 1000 jobs from each of the workload traces corresponding to the simulated platforms are used. Moreover, HDeepRM tools such as *hdeeprm-visual* and *hdeeprm-metrics* are utilized for analysing the results based on *evalys* plotting capabilities.

5.2.1 Job life cycles

In order to understand the impact of scheduling policies in job processing, the *lifecycle* visualization may be used. It provides information on when each of the jobs has *arrived*, *started* execution and *completed*. There are three horizontal regions, one representing each event type; the Y-axis for each region represents the number of cores requested by each job. The life cycle for the Gaia cluster is observable in figure 5.1.

It can be seen that the *shortest* job selection policy reduces delays between arrivals and completions for short jobs, with executions being concentrated at the beginning of the simulation. In the *first arrived* job selection policy, both long and short jobs are selected indistinctly, thus there is higher dispersion among life stages. It is noticeable that the makespans (total completion time for the whole workload trace) are practically the same.

Figure 5.2 shows the life cycles for the MetaCentrum2 grid. As opposed to Gaia, jobs arriving in this grid of clusters are in their majority short, with a few very long jobs; this can be further observed via *hdeeprm-jobstat*, which provides a maximum job requested time of 2592000, whereas the 95th percentile is 86400, and even the 99th percentile is 439776, less than 20% of

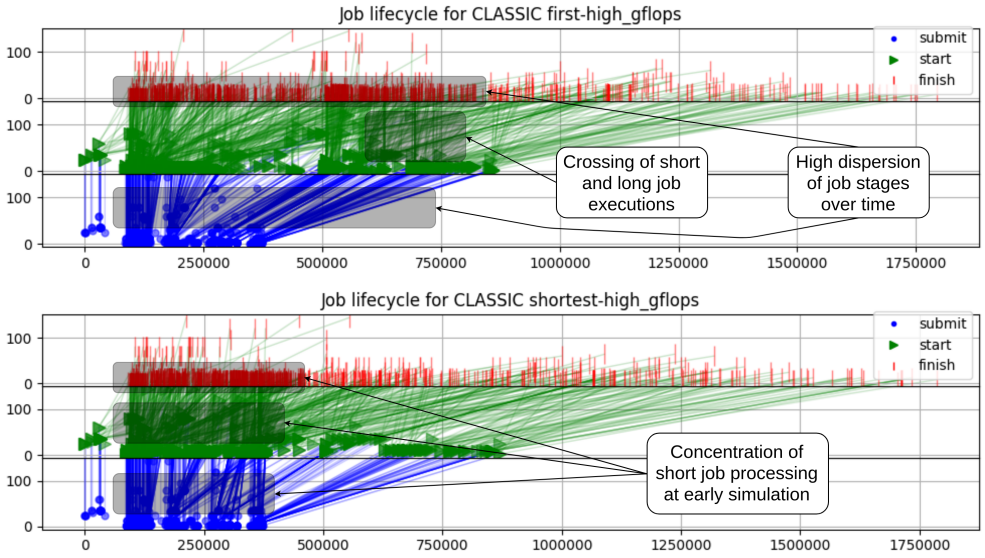


Figure 5.1: Job life cycles for the Gaia cluster

the maximum value.

In this case, the job selection policy does not seem to affect the life stage distribution. In order to understand this, along with the invariance of makespans in the Gaia cluster, other kind of visualizations may be explored, which are explained in the following section.

Figure 5.2: Job life cycles for the MetaCentrum2 grid

5.2.2 Utilization and queue size

For better understanding the reasons behind the two phenomena highlighted in the previous section, the *utilization* and *queue_size* visualizations may be used. In figure 5.3, the utilization of cores within the Gaia cluster is represented. It is observable that during approximately 40% of the simulation, the load peaks at 100%, meaning that all cores are being utilized. The long period of overutilization is an indicator of high queue size; this can be confirmed by looking at figure 5.4.

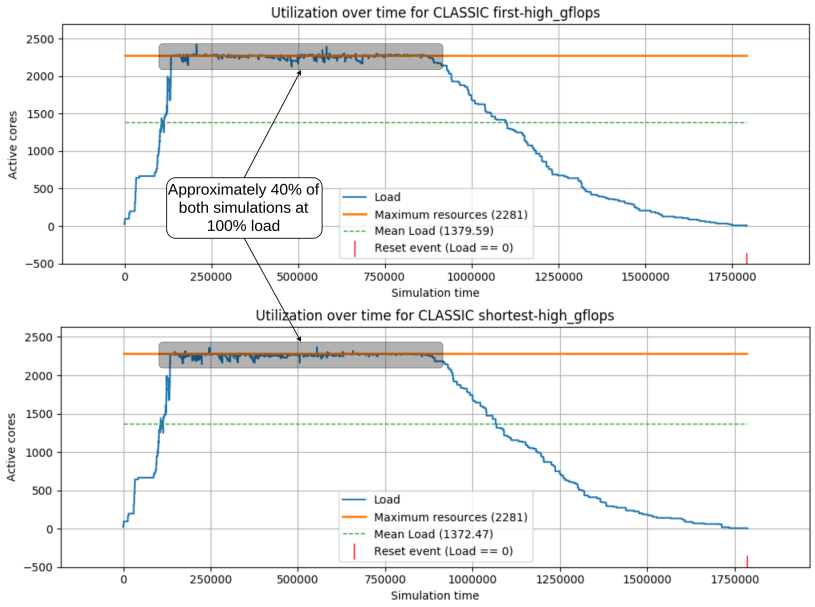


Figure 5.3: Utilization for the Gaia cluster

The queue size is measured in pending core requests in the queue. For *shortest* job, the queue size is considerably smaller than for *first arrived*, reason being that the job processing throughput is higher. Furthermore, steep declines are observable when job arrival rate decreases.

From a theory perspective, makespan is directly conditioned by compute throughput, that is the number of operations per second currently being used within the HPC service. Utilization *bubbles* constitute compute throughput inefficiencies, where one or more cores within the service are unused due to inability of serving the next selected job. This for instance may happen when there are not sufficient cores for a large job or when memory capacity is over-utilized.

Usually, *shortest* selection policies result in lower (better) makespans due to bubbles spanning shorter periods of time. In this case, this does not happen because of (1) the size distribution of jobs and (2) the memory requirements.

At a glance, it seems the majority of jobs request between 0 and 40 cores; via *hdeeprm-jobstat*, it is confirmed that the 95th percentile of job size is 36 cores requested, and the median is just 6 cores. This means that core contention bubbles when the cluster is over-utilized will likely leave less than 1% of the cluster cores unused. Even if these bubbles span longer in the *first arrived* policy, they are not significant for the final makespan.

On the other hand, memory requirements for the first 1000 jobs in the Gaia cluster trace are minimal relative to the service's memory capacity. 95% of the jobs request 2 or less GB, with a median of 11 MB and a maximum request of 9.5 GB. This may be compared with the lowest memory-per-core value in Gaia, which is provided by *Bullx B500* and *Dell R720* nodes as 4 GB per core. When considering all 51987 jobs, statistics vary slightly, with a p95th of 829 MB, median of 72 MB and maximum of 38.5 GB. This is an over-dimensioned cluster for the memory needs of their users. Utilization metrics extracted from the simulation reflect no memory bubbles in any execution, so makespan is not affected by them.

The second phenomenon indicates no variability in life stage distribution for the two policies over the MetaCentrum2 grid. This is an indicator that utilization is low for the service, which is confirmed in figure 5.5. This HPC platform is considerably larger than Gaia with 10002 cores,

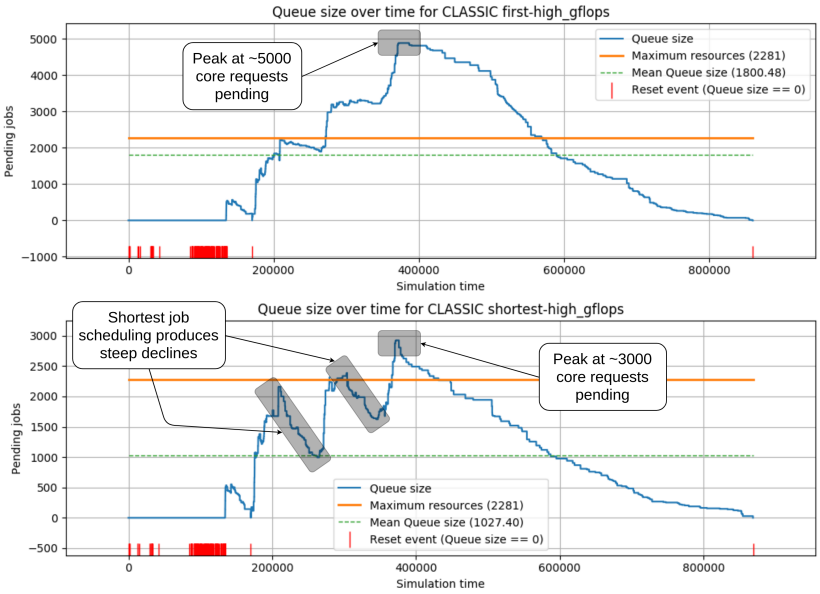


Figure 5.4: Queue size for the Gaia cluster

thus 1000 jobs do not fully occupy them. On the other hand, jobs are also very small across the trace, with a 99th percentile of 64 cores requested.

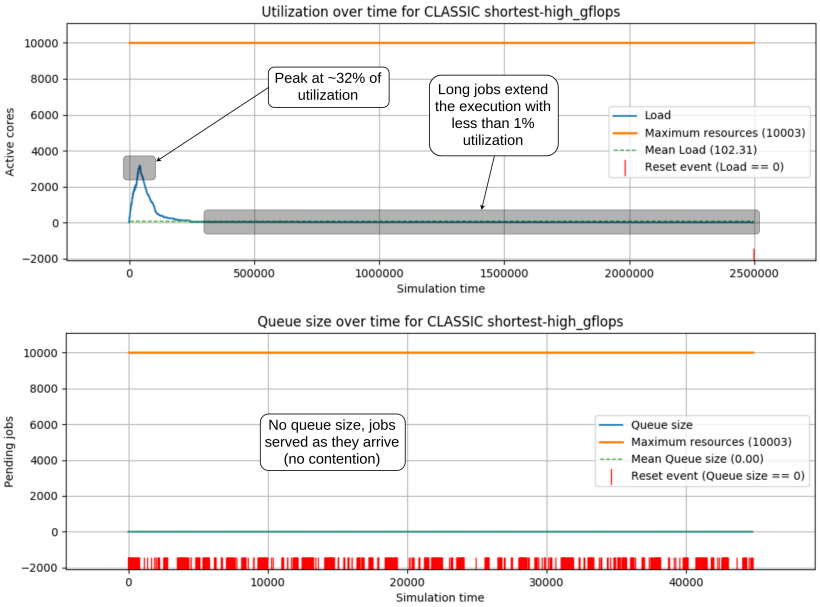


Figure 5.5: Utilization and queue size for the *shortest* policy in MetaCentrum2

5.2.3 Metric comparison between policies

Aside from visualizations, several simulation metrics may be observed in order to understand the effects of each scheduling policy. Both *first arrived* and *shortest* policy metrics are shown for the Gaia cluster in figure 5.6.

Both energy consumed and makespan are practically the same due to reasons discussed in previous sections. The slowdown measures how much of the turnaround time is constituted by

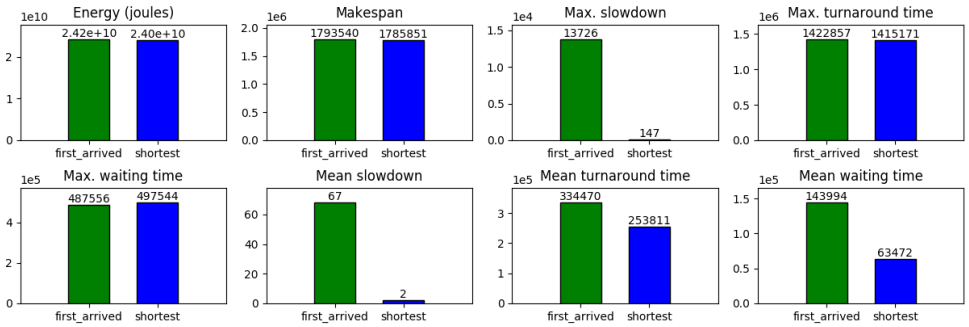


Figure 5.6: Metrics for *first arrived* and *shortest* policies in Gaia

the waiting time, and it is expressed as the ratio between turnaround and running time ($\frac{C_j}{T_j}$). A slowdown of 5 indicates the turnaround time is 5 times the running time, and thus $\frac{4}{5}$ of the turnaround time is due to waiting.

In this case, the maximum slowdown is approximately 93 times greater (worse) for the *first arrived* policy, while the mean slowdown is 33.5 times more. The reason behind this is that in *first arrived*, multiple short jobs are stalled in the queue; since slowdown is expressed as $\frac{C_j}{T_j} = \frac{T_j+W_j}{T_j}$, for small T_j every time unit of W_j increases the slowdown considerably. On the other hand, in *shortest* policy long jobs (high T_j) are stalled in the queue making each time unit of W_j less impactful in the ratio.

This principle also applies to turnaround and waiting time, sine they are all related. It can be concluded that there is an evident trade-off between fairness and response time: in *first arrived* policy, no job is delayed by future arriving jobs, thus acting like a traditional fair queue; in *shortest* policy, short jobs may delay long jobs with the advantage of decreasing response time. From a pragmatic perspective, this last approach improves performance for a considerable amount of user jobs by slightly (relatively) reducing the performance for a few long ones.

5.3 Demonstrating consequences of consolidation and spreading

In this experiment, effects of job consolidation and spreading are highlighted. Only the first 100 jobs from the Gaia workload trace are utilized for better clarity of results. Two different strategies are confronted in order to understand the existent trade-offs. Results show consolidation leading to 43% less energy consumption than spreading, however both makespan and turnaround (completion) time also augment 20% with respect to spreading.

Two classic agents are compared when performing the simulation:

- *shortest-high_gflops*: selects shortest jobs from the queue and cores with the highest peak GFLOPs in the core pool. This agent will pack the jobs in highest computing capability processors, without taking into account memory availability or memory bandwidth utilization.
- *shortest-high_mem_bw*: schedules cores based on the currently available memory bandwidth in the their processor scope. This will start serving cores in processors with the highest memory bandwidth, but as they are allocated, the available memory bandwidth will decrease, and it will start spreading jobs over other processors.

The job to core mapping is observable via Gantt charts in figure 5.7. Notice that the Y-axis

contains all the cores ordered as they are in table 5.1. This means that *Xeon L5640* processors are near the 0, whereas *Xeon E5-4650 v2* and *Xeon E7-8880 v2* are close to 2280. Results show two different strategies are applied depending on the established policy:

- *Consolidation*: the high GFLOPs agent always looks for the highest peak computing capability resource. In Gaia, these are the 120 *Xeon E7-8880 v2* (20 GFLOPs), the 160 *Xeon E5-4650 v2* (19.2 GFLOPs) and the 32 *Xeon E5-4640* (19.2 GFLOPs) cores. The former two range from core number 2000 to 2280, and are observable as active in the top of the chart. The latter is observed in the lower middle of the chart. When these 312 cores are fully occupied, the agent selects from the next highest computing capability cores, corresponding to *Xeon E5-2260* and *Xeon X5670* processors. These are represented in the top-middle of the chart.
- *Spreading*: with the high memory bandwidth agent, the jobs are spread across the cluster, occupying every processor available, but not all of its cores.

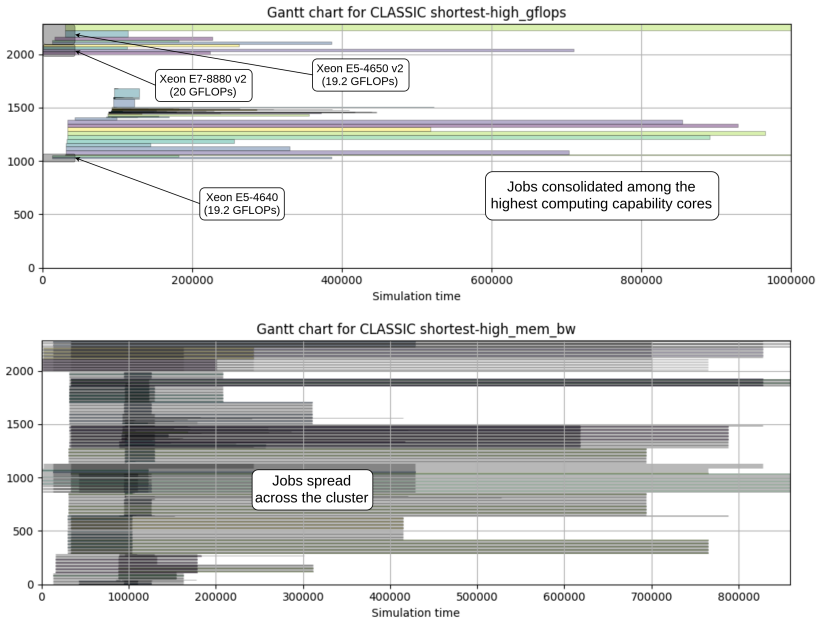


Figure 5.7: Gantt charts with mapping of jobs to cores in Gaia

These two strategies lead to two sets of metrics compared in figure 5.8. The *high_gflops* agent has consolidated the load in a subset of available cores, which has reduced the energy consumption with respect to the *high_mem_bw* agent in approximately 43%. This is due to majority of cores being in P3 (idle, 5% power), whereas with *high_mem_bw* every processor has been active, and thus its unused cores have been in P2 (25% power).

However, consolidation has also brought up the memory bandwidth overutilization problem, where too many jobs demanding access to memory interfere with each other and individual performance decays. This is observable both in the makespan and the turnaround time, which is the completion time of a job. In the experiment, both of them are about 20% worse for *high_gflops*.

Notice that, for some metrics, the Y-axis is expressed in scientific notation, with power being expressed on the top-left corner of each chart. In addition, bear in mind that no waiting time is measured due to all jobs being served at arrival.

Going further, the *mem_bw_overutilization* visualization, which has been developed as part of

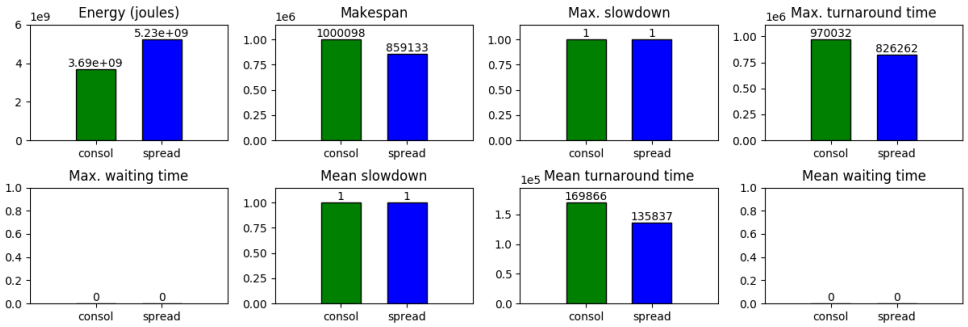


Figure 5.8: Comparison of metrics between high GFLOPs and high memory bandwidth agents

HDeepRM, helps realize metric differences. In figure 5.9, memory bandwidth overutilization spans are shown for both agents; these are the time spans from entering overutilization until going back to normal for each processor. It is observable that for *high_gflops*, overutilization lasts the complete simulation; this is because every time a job completes, another one is scheduled in that core, since it is the highest computing capability available, which keeps overutilization active. On the other hand, the *high_mem_bw* reduces these spans due to scheduling cores where memory bandwidth is high. Because no over-utilizations are found from approximately 60% of the simulation time, the makespan improves considerably.

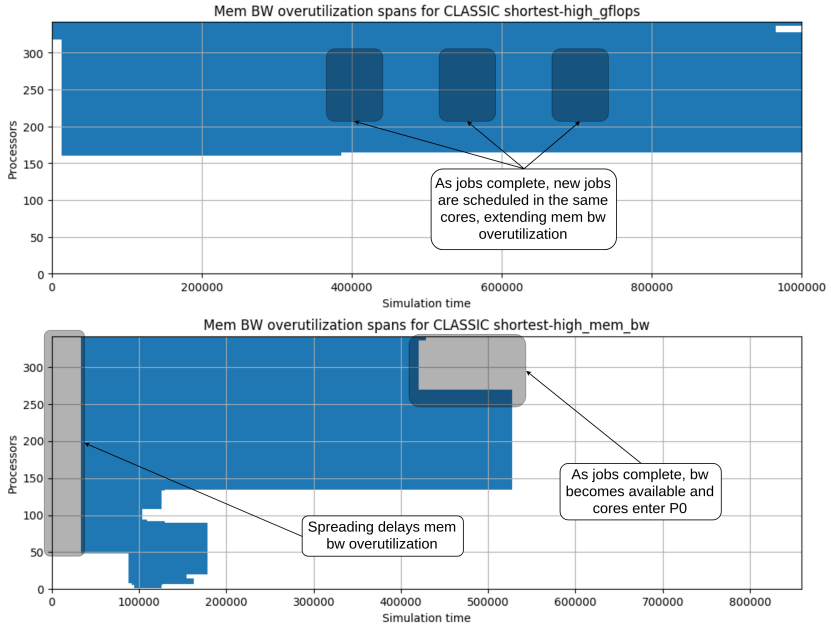


Figure 5.9: Memory bandwidth over-utilization spans for the Gaia cluster

This simple experiment highlights the trade-off when the agent has to decide between consolidating and spreading. In most situations, the value proposition for users is the speed and guarantees of job completions, whereas for providers it is the energy and operation cost. Therefore, a hybrid approach based on *energy efficiency* would lead to a common content.

This hybrid approach is complicated to express algorithmically, since it depends on the previous and current state of the service. A learning agent may be applied to adapt its internal policy to the nuances of the service over time. HDeepRM may be used as a learning framework, which will be shown in the following sections.

5.4 Learning the optimal actions

Previous sections have explored fixed policy scenarios. Each agent may only select a single action for every decision step, which lowers flexibility of scheduling. Learning agents are presented with a subset of actions available for choice, and they have to figure out which of them are optimal towards their objective. In this experiment, a minimal scenario is presented to a policy-learning agent in order to evaluate its ability to adapt. Results show that the agent correctly converges into the optimal actions in approximately 350 simulations.

5.4.1 Scenario

The synthetic scenario consists of a platform with one cluster, one node and two dual-core processors within the node. Both of them provide 32 GB/s memory bandwidth capacity; the former offers 4 GFLOPs per core at 80 watts, whereas the latter is slightly faster and more power hungry, with 4.4 GFLOPs per core and 100 watts.

Two jobs arrive at $t = 0$. They both request one core and 24 GB/s of memory bandwidth for 5 seconds. If they are both scheduled in a single processor (occupying both the cores), the memory bandwidth will become over-utilized.

The agent is based on the REINFORCE implementation presented in 3.2.2. 16 hidden units are used for each of the three hidden layers, with a learning rate of 0.005 and a discount factor of 0.99.

A reduced action space of 5 actions is enabled: *shortest-high_gflops*, *shortest-high_mem_bw*, *shortest-low_power*, *first-high_gflops* and *first-high_mem_bw*. The observation utilized is of the *minimal* type, since the scenario is relatively simple. Queue sensitivity is set to 0.05, however it is not relevant in this experiment since there is only one arrival. Proposed objective is *makespan*.

The agent is trained for 350 simulation episodes of a single step.

5.4.2 Expected results

The idea is for the agent to understand that, even though the second processor is objectively faster, if both jobs are scheduled on it the makespan will increase due to memory bandwidth over-utilization. If each job is scheduled in different processors, there will be no conflicts and makespan will decrease.

The actions that achieve the optimal outcome are *shortest-high_mem_bw* and *first-high_mem_bw*, with all others leading to a sub-optimal resolution. The agent should learn to select among these two actions for this scenario.

5.4.3 Losses

The losses measure how good the agent is deciding in order to achieve its objective. Good (optimal) decisions lower the loss value, whereas bad decisions increase it. In a training scenario, the loss should approach to 0 as the agent comprehends the optimal set of actions to be taken.

The *losses* visualization helps with showing the agent's loss evolution along the 350 simulation episodes. This is shown in figure 5.10.

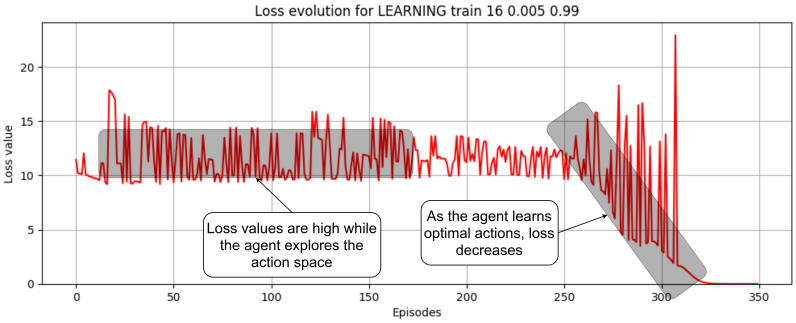


Figure 5.10: Loss evolution for REINFORCE - 350 episodes

At the beginning, the agent is selecting between all 5 available actions, which makes the loss vary within a considerable range. This is because the agent still does not know how good optimal actions are relative to others. From the 200th until the 250th episode, this range decreases significantly. This phenomenon is due to worst actions being less probable, as well as preferences of selected actions becoming similar during that stage, as will be observed in section 5.4.5. As training time passes, the agent understands that some of the actions are always better than others, and thus it sticks to them until the loss converges to zero.

Since this agent is based on stochastic policy gradient, sub-optimal decisions are taken towards the end in spite of their selection probability being lower. They are represented by the high pikes.

5.4.4 Rewards

Rewards are the agent's feedback on its decisions, and they are designed to represent the final objective. Makespan reward can be observed in section 3.2.4. In order to understand the reward evolution over the training period, the *rewards* visualization in figure 5.11 may be explored.

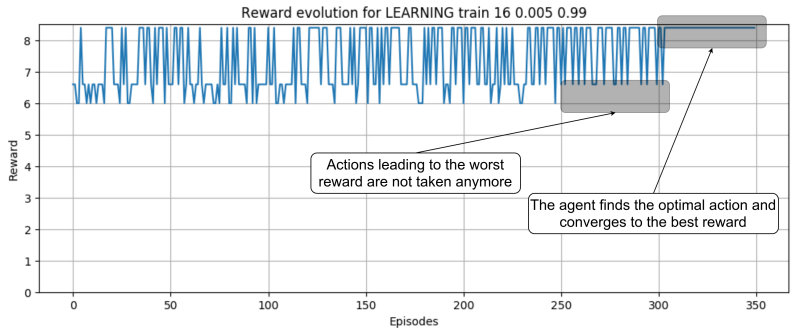


Figure 5.11: Reward evolution for REINFORCE - 350 episodes

Similar to the loss evolution, at the exploratory stage the agent rewards oscillate between the three possible values, 6, 6.6 and 8.4. The first reward corresponds to the *low_power* approach, where the two jobs are consolidated in the slowest processor and over-utilizing its memory bandwidth; the second one happens when they are consolidated in the fastest processor, whereas the latter occurs when they are spread among both.

Towards the end, the worst reward is not received anymore since the *low-power* action has been discarded. Finally, the agent learns that spreading is the best strategy and converges to the best reward.

5.4.5 Action preferences

As the agent explores the action space, *preferences* or probabilities for the actions vary with the inner model updates. The *action_preferences* visualization in figure 5.12 helps realize this progress.

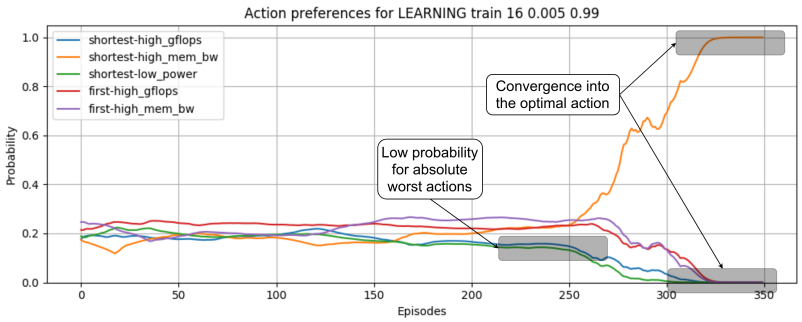


Figure 5.12: Action preferences for REINFORCE - 350 episodes

When the training starts, action preferences are randomly initialized with similar probabilities. This encourages exploration of the space. From the 200th episode, the agent learns that using the *low_power* strategy leads to consistently worse rewards, so its probability lowers. The *shortest-high_gflops* strategy also goes down even though it does not result in the worst reward, reason being it has been selected few times during the simulations (49 with respect to 61 for *first-high_mem_bw* and 124 for *shortest-high_mem_bw*), and the agent does not want to risk it being worse at that stage.

At approximately the 275th step, a divergence of preferences occurs, meaning that the agent has now converged to the optimal action, *shortest-high_mem_bw*. Bear in mind that the *first-high_mem_bw* is also optimal, however the agent does not have the ability to alternate between them; for other seeds, the agent may also converge to the latter.

Chapter 6

Conclusions

This chapter completes the thesis by providing (1) an overview of main achievements, mapped to the initial objectives described in section 1.3, and (2) several directions for future work both in extending HDeepRM and utilizing it for research.

6.1 Achievements

As this thesis concludes, a retrospective on the attainment of initial objectives expressed in section 1.3 is of relevance. Hereunder, specific achievements are mapped to each of the proposed objectives:

- A complete framework for heterogeneous workload management, HDeepRM, has been designed, implemented, tested and distributed. It allows for defining heterogeneous environments composed of distinct resource types, these being processing speed, power consumption, memory and memory bandwidth. It also models resource conflicts by quantifying the amount of resource available in processor and node scopes. A new platform specification has been designed, while several converters have been developed for parsing and instantiating both the platform and the workload trace. In order to simplify experimenting with HDeepRM, these have been abstracted from the user via *hdeeprm-launch* and other command line utilities, allowing future researchers to focus on the experiment design and result analysis. The evaluation in chapter 5 endorses the final functionality and validity of HDeepRM.

  This covers the proposal of the first objective and provides extra value: configurable policies, usage and analysis utilities, distribution for easy installation and documentation.

- A deep reinforcement learning extension has been developed, compliant with the reinforcement loop observed in 1.3. Its modular structure shown in section 4.2.1 provides the framework with extensibility. A specific HDeepRM workload manager is implemented, which merges the reinforcement loop into Batsim event-flow, in a way that it is abstracted from the user. In addition, an agent API is offered for users to define external agents, which are automatically integrated and used in simulation runs. Two algorithms, REINFORCE and actor-critic, are provided as examples for the agent API usage. The base agent methods (*observe*, *device*, *alter*, *rewarded*) have been designed such that they are semantically logical for the user, and improve readability with respect to previous approaches.

  The second objective is covered in its extent, plus additional advantages are provided:

novel environment, observation and action space designs allow for study and modification of the framework, hyperparameter tuning via usage utilities, insights log for the agent performance in simulations.

- Several experiments have been carried out to demonstrate HDeepRM capabilities. In the first one shown in section 5.2, a deep dive into the framework capabilities has provided guidance in which tools to use in order to gain insights of the simulation; job cycles, metrics, as well as utilization and queue size charts have been leveraged, while *hdeeprm-jobstat* has helped with specific details. In the second one, the consolidation and spreading strategies are confronted by means of varying the resource selection policy; it is observed that the former leads to less power consumption but also less performance, thus exposing a trade-off. Finally, in the third experiment a complete learning pipeline is explained, exploring losses, rewards and action preferences; the agent manages to converge to the optimal action for that scenario in approximately 350 episodes.

  The third objective is then achieved, providing the framework with robustness. Nevertheless, there is still more learning validation to do, since the third experiment was an initial minimal approach. The user should also be able to base his/her experiments on the roots here provided.

Another relevant contribution is the use of standardization in the framework. Both the utilization of the Standard Workload Format (SWF) and the environment definitions from OpenAI Gym library grant the framework with forward compatibility and maintainability.

6.2 Future work

During this thesis, several directions of work with high potential for improving HDeepRM have been considered. Due to time constraints, they have not been undertaken, however they are listed hereunder:

- Backfilling algorithm integration: the majority of current workload managers, including SLURM, utilize a version of the *EASY* backfilling algorithm [2]. This is based on the *first come first served* classic policy, however it allows for shorter jobs to be scheduled if they do not delay the first next job to be served. Current HDeepRM iteration does not implement backfill, and it would be relevant to compare it with learned policies.
- Architectural heterogeneity: at this time, HDeepRM supports both specification-based and micro-architectural heterogeneity, as expressed in section 2.1.2. With architectural heterogeneity, different types of devices could be offered from the platforms, including different CPU ISAs, GPUs, MICs and other hardware accelerators. This would better resemble the current state of the data center.
- Memory access modelling: memory hierarchy conflicts are currently modelled by means of quantifying both the capacity in node scope and the bandwidth in processor scope. This is valid for the agent to understand the consequences of interdependence expressed in section 3.1.1, however it does not realistically model these conflicts. Memory access traces could be gathered and added to the workload definition, while HDeepRM should be extended to achieve this level of modelling.
- Network modelling: there is no network simulation in HDeepRM. This works given the assumption that job tasks are independent, thus they do not need communication and synchronization. In reality, there would generally be some interaction between them, and this would require network traces for each job in order to be simulated. SimGrid can model networks natively, so HDeepRM should not need to be modified extensively.

- Last level cache (LLC) conflicts: LLC size definition is already supported in the designed platform specification format. Conflicts between jobs with different working set [61] size should more realistically model the consolidation effects.
- Instruction composition of loads: for simplification purposes, in this thesis jobs instructions are all double-precision floating point operations. This may resemble highly efficient computing tasks, where large amounts of operations are applied to the same data, for instance different convolution filters over an image. However, in reality not all jobs possess these characteristics, thus a trace with instruction composition could be used for better resemblance.
- Job tasks and dependencies: each job is composed of different job *tasks*. Some of them might be executed in parallel, while others require a synchronization point. This is typically seen in MPI [66] workloads, which are supported natively by SimGrid. Directed Acyclic Graphs (DAGs) [67] express dependencies between tasks, and would allow HDeepRM to understand each job workflow.
- User-defined observations, actions and rewards: the observation and action spaces are, at the moment, part of the HDeepRM framework. In a future iteration, they are scheduled to be externalized for the user to modify and integrate his/her own. This would require tuning the framework for more flexibility.

Bibliography

[1] Volker Hamscher et al. "Evaluation of job-scheduling strategies for grid computing". In: *International Workshop on Grid Computing*. Springer. 2000, pp. 191–202.

[2] David A Lifka. "The anl/ibm sp scheduling system". In: *Workshop on Job Scheduling Strategies for Parallel Processing* (1995), pp. 295–303.

[3] TOP500.org. *TOP500 List - November 2018*. URL: https://tinyurl.com/y6va3oxn. (accessed: 06.01.2019).

[4] Intel Corporation. *Intel Xeon Processor E5-2650L v4*. URL: https://tinyurl.com/yygtryta. (accessed: 12.03.2019).

[5] Brahim Betkaoui, David B Thomas, and Wayne Luk. "Comparing performance and energy efficiency of FPGAs and GPUs for high productivity computing". In: *2010 International Conference on Field-Programmable Technology*. IEEE. 2010, pp. 94–101.

[6] Sriram Govindan et al. "Cuanta: quantifying effects of shared on-chip resource interference for consolidated virtual machines". In: *Proceedings of the 2nd ACM Symposium on Cloud Computing*. ACM. 2011, p. 22.

[7] Dalibor Klusáček, Šimon Tóth, and Gabriela Podolníková. "Complex Job Scheduling Simulations with Alea 4". In: *Proceedings of the 9th EAI International Conference on Simulation Tools and Techniques*. ICST (Institute for Computer Sciences, Social-Informatics and ... 2016, pp. 124–129.

[8] Gonzalo P Rodrigo et al. "ScSF: a scheduling simulation framework". In: *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer. 2017, pp. 152–173.

[9] Cristian Galleguillos, Zeynep Kiziltan, and Alessio Netti. "AccaSim: an HPC simulator for workload management". In: *Latin American High Performance Computing Conference*. Springer. 2017, pp. 169–184.

[10] Pierre-François Dutot et al. "Batsim: a realistic language-independent resources and jobs management systems simulator". In: *Job Scheduling Strategies for Parallel Processing*. Springer. 2015, pp. 178–197.

[11] Ayon Dey. "Machine Learning Algorithms : A Review". In: 2016.

[12] Pierre Baldi. "Autoencoders, unsupervised learning, and deep architectures". In: *Proceedings of ICML workshop on unsupervised and transfer learning*. 2012, pp. 37–49.

[13] Frank Rosenblatt. "The perceptron: a probabilistic model for information storage and organization in the brain." In: *Psychological review* 65.6 (1958), p. 386.

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

[15] Marvin Minsky and Seymour A Papert. *Perceptrons: An introduction to computational geometry*. MIT press, 2017.

[16] David E Rumelhart, Geoffrey E Hinton, Ronald J Williams, et al. "Learning representations by back-propagating errors". In: *Cognitive modeling* 5.3 (), p. 1.

[17] Bernhard E Boser, Isabelle M Guyon, and Vladimir N Vapnik. "A training algorithm for optimal margin classifiers". In: *Proceedings of the fifth annual workshop on Computational learning theory*. ACM. 1992, pp. 144–152.

[18] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. "Deep learning". In: *nature* 521.7553 (2015), p. 436.

[19] Hongzi Mao et al. "Resource management with deep reinforcement learning". In: *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*. ACM. 2016, pp. 50–56.

[20] Gordon E Moore. "Cramming more components onto integrated circuits, Reprinted from Electronics, volume 38, number 8, April 19, 1965, pp. 114 ff." In: *IEEE Solid-State Circuits Society Newsletter* 11.3 (2006), pp. 33–35.

[21] Digh Hisamoto et al. "FinFET-a self-aligned double-gate MOSFET scalable to 20 nm". In: *IEEE Transactions on Electron Devices* 47.12 (2000), pp. 2320–2325.

[22] Andrei Frumusanu. *The Apple A12 - First Commercial 7nm Silicon*. URL: https://tinyurl.com/y94xq72b. (accessed: 09.01.2019).

[23] Mark Lapedus. *Big Trouble At 3nm*. URL: https://tinyurl.com/yby9t3wu. (accessed: 09.01.2019).

[24] Robert H Dennard et al. "Design of ion-implanted MOSFET's with very small physical dimensions". In: *IEEE Journal of Solid-State Circuits* 9.5 (1974), pp. 256–268.

[25] Gene M Amdahl. "Validity of the single processor approach to achieving large scale computing capabilities". In: *Proceedings of the April 18-20, 1967, spring joint computer conference* (1967), pp. 483–485.

[26] Vivienne Sze et al. "Hardware for machine learning: Challenges and opportunities". In: *2017 IEEE Custom Integrated Circuits Conference (CICC)* (2017), pp. 1–8.

[27] Violaine Villebonnet et al. ""Big, Medium, Little": Reaching Energy Proportionality with Heterogeneous Computing Scheduler". In: *Parallel Processing Letters* 25.03 (2015), p. 1541006.

[28] Ed Sterling. *Getting Down To Business On Chiplets*. URL: https://tinyurl.com/ya2ssvjb. (accessed: 10.01.2019).

[29] Anil K Jain, Jianchang Mao, and K Moidin Mohiuddin. "Artificial neural networks: A tutorial". In: *Computer* 29.3 (1996), pp. 31–44.

[30] SchedMD LLC. *Main page*. URL: https://www.schedmd.com/. (accessed: 23.01.2019).

[31] Rajkumar Buyya and Manzur Murshed. "Gridsim: A toolkit for the modeling and simulation of distributed resource management and scheduling for grid computing". In: *Concurrency and computation: practice and experience* 14.13-15 (2002), pp. 1175–1220.

[32] Henri Casanova et al. "Versatile, scalable, and accurate simulation of distributed applications and platforms". In: *Journal of Parallel and Distributed Computing* 74.10 (2014), pp. 2899–2917.

[33] Martín Abadi et al. "Tensorflow: a system for large-scale machine learning." In: *OSDI*. Vol. 16. 2016, pp. 265–283.

[34] Adam Paszke et al. "Automatic differentiation in pytorch". In: (2017).

[35] Greg Brockman et al. "Openai gym". In: *arXiv preprint arXiv:1606.01540* (2016).

[36] Ronald J Williams. "Simple statistical gradient-following algorithms for connectionist reinforcement learning". In: *Machine learning* 8.3-4 (1992), pp. 229–256.

[37] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[38] Ahuva W. Mu'alem and Dror G. Feitelson. "Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling". In: *IEEE transactions on parallel and distributed systems* 12.6 (2001), pp. 529–543.

[39] Dan Tsafrir, Yoav Etsion, and Dror G Feitelson. "Backfilling using system-generated predictions rather than user runtime estimates". In: *IEEE Transactions on Parallel and Distributed Systems* 18.6 (2007), pp. 789–803.

[40] Eric Gaussier et al. "Improving backfilling by using machine learning to predict running times". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM. 2015, p. 64.

[41] Mehmet Soysal, Marco Berghoff, and Achim Streit. "Analysis of Job Metadata for Enhanced Wall Time Prediction". In: *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer. 2018, pp. 1–14.

[42] F Pezzella, G Morganti, and G Ciaschetti. "A genetic algorithm for the flexible job-shop scheduling problem". In: *Computers & Operations Research* 35.10 (2008), pp. 3202–3212.

[43] M Emin Aydin and Ercan Öztemel. "Dynamic job-shop scheduling using reinforcement learning agents". In: *Robotics and Autonomous Systems* 33.2-3 (2000), pp. 169–178.

[44] Christopher JCH Watkins and Peter Dayan. "Q-learning". In: *Machine learning* 8.3-4 (1992), pp. 279–292.

[45] SimGrid Team. *Modeling Multicore Machines - Pinning tasks to cores*. URL: https://tinyurl.com/y2f6hmbo. (accessed: 11.02.2019).

[46] SimGrid Team. *SimGrid DTD*. URL: https://tinyurl.com/yyx8gkrp. (accessed: 11.02.2019).

[47] Metacentrum. *Metacentrum - Virtual Organization*. URL: https://tinyurl.com/y4bwcmpn. (accessed: 11.02.2019).

[48] Batsim team. *Job definition*. URL: https://tinyurl.com/y3mhhsj7. (accessed: 18.03.2019).

[49] Steve J Chapin et al. "Benchmarks and standards for the evaluation of parallel job schedulers". In: *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer. 1999, pp. 67–90.

[50] Volodymyr Mnih et al. "Playing atari with deep reinforcement learning". In: *arXiv preprint arXiv:1312.5602* (2013).

[51] John Schulman et al. "Trust region policy optimization". In: *International Conference on Machine Learning*. 2015, pp. 1889–1897.

[52] Vinod Nair and Geoffrey E Hinton. "Rectified linear units improve restricted boltzmann machines". In: *Proceedings of the 27th international conference on machine learning (ICML-10)*. 2010, pp. 807–814.

[53] Wikipedia. *Huber loss*. URL: https://tinyurl.com/y4skyytu. (accessed: 03.03.2019).

[54] Wikipedia. *Mean squared error*. URL: https://tinyurl.com/ps65mcn. (accessed: 03.03.2019).

[55] Oguzhan Alagoz et al. "Markov decision processes: a tool for sequential decision making under uncertainty". In: *Medical Decision Making* 30.4 (2010), pp. 474–483.

[56] Hongzi Mao. *DeepRM Demo*. URL: https://tinyurl.com/y3yyjwds. (accessed: 14.02.2019).

[57] Brad J Cox. "Object-oriented programming: an evolutionary approach". In: (1986).

[58] Wikipedia. *CAS latency*. URL: https://tinyurl.com/ak72lqt. (accessed: 15.02.2019).

[59] Wikipedia. *Intel Turbo Boost*. URL: https://tinyurl.com/ca9p5fe. (accessed: 19.02.2019).

[60] Wikipedia. *SpeedStep*. URL: https://tinyurl.com/ceecf6s. (accessed: 19.02.2019).

[61] Peter J Denning. "The working set model for program behavior". In: *Proceedings of the first ACM symposium on Operating System Principles*. ACM. 1967, pp. 15–1.

[62] Adrián Herrera Arcila. *HDeepRM documentation*. URL: https://tinyurl.com/y4epzmzc. (accessed: 03.03.2019).

[63] Adrián Herrera Arcila. *HDeepRM GitHub repository*. URL: https://tinyurl.com/yy5hjxbs. (accessed: 05.03.2019).

[64] Parallel Workloads Archive. *The University of Luxemburg Gaia Cluster log*. URL: https://tinyurl.com/y56sa2wu. (accessed: 11.03.2019).

[65] Parallel Workloads Archive. *The MetaCentrum 2 log*. URL: https://tinyurl.com/y4lcuwmo. (accessed: 16.03.2019).

[66] Edgar Gabriel et al. "Open MPI: Goals, concept, and design of a next generation MPI implementation". In: *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*. Springer. 2004, pp. 97–104.

[67] Wikipedia. *Directed acyclic graph*. URL: https://tinyurl.com/cs965c8. (accessed: 12.03.2019).

Appendices

Appendix A

Example job script

```
#!/usr/bin/env bash

#SBATCH --nodes=2
#SBATCH --ntasks-per-node=8
#SBATCH --time=00:20:00
#SBATCH --job-name=mandelbrot
#SBATCH --mail-type=ALL
#SBATCH --mail-user=abc.123@gmail.com

# Run the program
srun mandelbrot
```

This job script expresses the following information:

- *nodes*: minimum compute resources required by the job to be allocated. These refer to processors.
- *ntasks-per-node*: number of processes to be spawned in each node. This is useful when dealing with hybrid MPI/OpenMP loads, where each process would be one thread running in the processor cores.
- *time*: limit on total run time of the job. Format is HH:MM:SS. If this limit is surpassed, all tasks belonging to the job are killed.
- *job-name*: name of the job.
- *mail-type*: notify the job's user when an event affects the job. In this case, any event, including start, end and fail, will be notified.
- *mail-user*: user email address.

Appendix B

Platform specification

B.1 Batsim + SimGrid compliant XML platform

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE platform
  SYSTEM 'https://simgrid.org/simgrid/simgrid.dtd'>
<platform version="4.1">
  <zone id="zone0" routing="Full">
    <host id="master_host" speed="1.0Gf"/>
    <host id="compute_host" speed="10.0Gf"/>
    <link id="link0" bandwidth="1.0Gbps" latency="125us"/>
    <route src="master_host" dst="compute_host">
      <link_ctn id="link0"/>
    </route>
  </zone>
</platform>
```

B.2 Minimal HDeepRM JSON platform

```
{"clusters": [
  {"id": "cluster0",
   "nodes": [
     {"type": "Basic", "number": 1}
   ],
   "local_links":
     {"type": "Gigabit Ethernet 1000BASE-T", "latency": "75us"}}
 ],
 "global_links":
   {"type": "Gigabit Ethernet 1000BASE-T", "latency": "125us"}}
```

B.3 Minimal HDeepRM node type

```
{"BasicNode": {
   "id": "basic_node",
   "processors": [
     {"type": "BasicProc", "number": 2}
   ],
   "memory": {"type": "DDR3-1600", "capacity": 32}}}
```

B.4 Minimal HDeepRM network type

```
{"Gigabit Ethernet 1000BASE-T": {"bandwidth": "1.0Gbps"}}
```

B.5 Minimal HDeepRM memory type

```
{"DDR3-1600": {"latency": "103.75ns"}}
```

B.6 Minimal HDeepRM processor type

```
{"BasicProc": {
   "uarch": "basic_arch", "id": "basic_proc", "type": "CPU",
   "cores": 2, "clock_rate": 1, "mem_bw": 32,
   "llc_size": 8, "power": 100, "dpflops_per_cycle": 4}}
```

Appendix C

Workload specification

C.1 Minimal Batsim formatted workload

```
{"nb_res": 8,
 "jobs": [
   {"id": "job0", "subtime": 0, "res": 2, "profile": "A"},
   {"id": "job1", "subtime": 5, "res": 4, "profile": "B"}
 ],
 "profiles": {
   "A": {"type": "parallel_homogeneous", "com": 0, "cpu": 1e10,
         "req_time": 100, "req_ops": 22e11, "mem": 500, "mem_bw": 16},
   "B": {"type": "parallel_homogeneous", "com": 0, "cpu": 5e10,
         "req_time": 300, "req_ops": 66e11, "mem": 2000, "mem_bw": 24}}}
```

C.2 SWF formatted workload

```
id subT   waitT runT   resAlloc cpuUsed memUsed resReq timeReq memReq ...
1  278659 2     268225 4        4023    4864    4      345600  -1
2  339016 1     305581 24       1783    14805   24     432000  -1
```

Appendix D

Events

D.1 Simulation begins


```
{"now": 0.000000,
 "events": [{"timestamp": 0.000000, "type": "SIMULATION_BEGINS",
             "data": {"nb_resources": 2, "nb_compute_resources": 2,
                      "...": "...",
                      "compute_resources": [
                        {"id": 0, "name": "cor_0", "state": "idle",
                         "properties": {"...": "..."}},
                        {"id": 1, "name": "cor_1", "state": "idle",
                         "properties": {"...": "..."}}],
                      "...": "...",
                      "workloads": {"62b680": "/workspace/workload.json"},
                      "profiles": {"62b680": {"A": {"...": "..."}}}}}]}
```


D.2 Job submitted


```
{"now": 0.000000,
 "events": [{"timestamp": 0.000000, "type": "JOB_SUBMITTED",
             "data": {"job_id": "62b680!job0",
                      "job": {"id": "62b680!job0",
                              "subtime": 0,
                              "res": 1,
                              "profile": "A"}}}]}
```


D.3 Job executed and alteration of cores

D.4 State change confirmation and no more submissions

```
{"timestamp": 0.000000,
 "events": [{"timestamp": 0.0, "type": "EXECUTE_JOB",
             "data": {"job_id": "62b680!job0", "alloc": "0"}},
            {"timestamp": 0.0, "type": "SET_RESOURCE_STATE",
             "data": {"resources": "0", "state": "0"}},
            {"timestamp": 0.0, "type": "SET_RESOURCE_STATE",
             "data": {"resources": "1", "state": "2"}}]}

{"now": 0.000000,
 "events": [{"timestamp": 0.000000, "type": "NOTIFY",
             "data": {"type": "no_more_static_job_to_submit"}},
            {"timestamp": 0.000000, "type": "RESOURCE_STATE_CHANGED",
             "data": {"resources": "0", "state": "0"}},
            {"timestamp": 0.000000, "type": "RESOURCE_STATE_CHANGED",
             "data": {"resources": "1", "state": "2"}}]}
```


D.5 Job completed and simulation ends


```
{"now": 5.000000,
 "events": [{"timestamp": 5.000000, "type": "JOB_COMPLETED",
             "data": {"job_id": "62b680!job0",
                      "job_state": "COMPLETED_SUCCESSFULLY",
                      "return_code": 0, "alloc": "0"}},
            {"timestamp": 5.000000, "type": "SIMULATION_ENDS",
             "data": {}}]}
```

Appendix E

Utilities

E.1 HDeepRM launcher

```
hdeeprm-launch -a <AGENT> -cw <CUSTOM_WORKLOAD_PATH> -im <INMODEL>\
               -om <OUTMODEL> <options_file>
```

E.2 HDeepRM visualizations

```
hdeeprm-visual -s SAVE <visualization>
```

E.3 HDeepRM metrics

```
hdeeprm-metrics -s SAVE <results1.csv> <results2.csv>
```

E.4 HDeepRM job statistics

```
hdeeprm-jobstat <workload.json> <job_data_field> <statistic>
```

Appendix F

Options file

```
{
  "seed": 20091995,
  "nb_resources": 64,
  "nb_jobs": 5000,
  "workload_file_path": "/workspace/workloads/my_workload.swf",
  "platform_file_path": "/workspace/platforms/my_platform.json",
  "pybatsim": {
    "log_level": "WARNING",
    "env": {
      "objective": "avg_utilization",
      "actions": {
        "selection": [
          {"shortest": ["high_gflops", "high_mem"]},
          {"first": ["high_mem_bw"]}
        ],
        "void": false
      },
      "observation": "small",
      "queue_sensitivity": 0.03
    },
    "agent": {
      "type": "LEARNING",
      "run": "train",
      "hidden": 128,
      "lr": 0.001,
      "gamma": 0.95
    }
  }
}
```

Appendix G

Example learning agent

```
import torch.nn as nn
import torch.nn.functional as F
from hdeeprm.agent import PolicyLearningAgent

class MyLearningAgent(PolicyLearningAgent):

    def __init__(self, gamma, hidden, action_size, observation_size):
        self.input = nn.Linear(observation_size, hidden)
        self.output = nn.Linear(hidden, action_size)

    def forward_policy(self, observation):
        out_0 = F.leaky_relu(self.input(observation))
        return F.softmax(self.output(out_0), dim=1)
```