Author/s:
Wang, Lingfei

Title:
Job Scheduling in High Performance Computing Clusters with Deep Reinforcement Learning

Date:
2025

Persistent Link:
https://hdl.handle.net/11343/360878

Job Scheduling in High Performance Computing Clusters with Deep Reinforcement Learning

Lingfei Wang

Submitted in total fulfilment of the requirements of the degree of

Doctor of Philosophy

School of Computing and Information Systems
THE UNIVERSITY OF MELBOURNE

March 2025

ORCID: 0009-0003-8293-4287

Dedicated to my family, for your endless love, strength, and belief in me.

And to everyone who walked this journey with me — your support, encouragement, and presence made all the difference.

Job Scheduling in High Performance Computing Clusters with Deep Reinforcement Learning

Lingfei Wang
Principal Supervisor: Dr. Maria Rodriguez Read
Co-supervisor: A/Prof. Nir Lipovetzky

Abstract

High Performance Computing (HPC) systems are designed to execute large-scale computational workloads by leveraging parallel processing across multiple compute nodes. These systems support a wide range of applications, including scientific simulations, engineering design, and artificial intelligence, where tasks usually require substantial computational power and execution time. Efficient job scheduling in HPC environments is essential to ensure timely job completion and effective resource utilization. In HPC scheduling, two important components are the job queue and the computing cluster. The job queue holds jobs submitted by users, each describing the resources it needs — such as CPU cores, GPUs, memory, and expected runtime — before it can run. The computing cluster is the pool of available hardware resources that can be assigned to these jobs. Since resources are limited and jobs arrive over time with varying demands, the scheduler must continuously decide which jobs to run and how to allocate resources to them. This process, known as job scheduling, involves both selecting jobs from the queue and assigning suitable resources across the cluster. These decisions must respect system constraints, such as resource availability and job placement requirements, and aim to utilize resources well while maintaining job waiting time as low as possible.

Traditional HPC schedulers often rely on simple heuristics like First-Come-First-Served (FCFS) or Shortest Job First (SJF), which are easy to implement but can lead to poor resource utilization and job starvation under dynamic workloads. Meta-heuristic algorithms have been used to improve decision quality by exploring a broader solution space, but they are computationally costly and require careful tuning. Supervised machine learning methods have also been applied to predict job priorities, offering better scheduling decisions than fixed heuristics. However, they depend on labeled datasets and struggle to adapt to real-time system changes. In contrast, Deep Reinforcement Learning (DRL) enables a scheduler to learn scheduling policies through interaction with the system, using performance feedback to improve decisions over time. It does not rely on labeled data or predefined heuristics, and instead learns to optimize long-term scheduling outcomes by exploring and evaluating different actions in varied system states. This makes DRL well-suited for handling dynamic workloads, complex resource constraints, and changing system conditions in HPC environments. However, applying DRL to HPC scheduling

introduces several key challenges. First, designing an effective job selector is difficult due to the unbounded and dynamic nature of the job queue. As jobs continuously arrive and depart, the number of scheduling actions and the information required to represent each job can vary significantly over time. This makes it challenging to define a consistent and scalable state representation and to construct a manageable action space for the agent. Second, job selection and resource allocation are often handled separately, which can result in poor coordination and inefficient scheduling. Addressing this requires a unified DRL framework that jointly considers both decisions. Third, scheduling objectives — such as minimizing job waiting time and maximizing resource utilization — can conflict and shift depending on workload intensity and system state, requiring the scheduler to dynamically adjust its priorities. Fourth, as HPC systems are upgraded, modified, or newly developed, changes in hardware architecture can alter the structure and semantics of the system state used by DRL-based schedulers. These changes affect how job and resource features are represented and interpreted, making it difficult to directly reuse DRL models trained in previous environments without adaptation.

In particular, this thesis makes the following contributions to DRL-based HPC scheduling:

- Develops a DRL-based job selector designed to handle unbounded job queues and support efficient backfilling.
- Presents a hierarchical DRL scheduler that jointly manages job selection and resource allocation in HPC environments.
- Introduces a dynamic controller that adjusts scheduling objectives based on real-time system conditions.
- Proposes a transfer learning framework that enables DRL schedulers to adapt efficiently to evolving HPC architectures.

Declaration

I declare that this thesis and the work presented in it are my own. I confirm that:

- the thesis comprises only my original work towards the Doctor of Philosophy except where indicated in the preface;
- due acknowledgement has been made in the text to all other material used; and
- the thesis is less than 100,000 words in length, exclusive of tables, maps, bibliographies and appendices.

Lingfei Wang, March 2025

Preface

This thesis research has been carried out at the School of Computing and Information, University of Melbourne, Australia. The main contributions of the thesis are discussed in Chapters 3 to 6 and are based on the following publications.

- Article 1

  Lingfei Wang, Aaron Harwood, and Maria A. Rodriguez. A Deep Reinforcement Learning Scheduler with Back-filling for High Performance Computing. In *Proceedings of 2021 IEEE Asia-Pacific Conference on Computer Science and Data Engineering (CSDE)*, pp. 1-6. IEEE, 2021.

- Article 2

  Lingfei Wang, Aaron Harwood, and Maria A. Rodriguez. Deep Back-Filling: a Split Window Technique for Deep Online Cluster Job Scheduling. In *Proceedings of 2023 IEEE International Conference on High Performance Computing & Communications*, pp. 772-779. IEEE, 2023.

- Article 3

  Lingfei Wang, Maria A. Rodriguez, and Nir Lipovetzky. Optimizing HPC Scheduling: A Hierarchical Reinforcement Learning Approach for Intelligent Job Selection and Allocation. *The Journal of Supercomputing* 81, no. 8 (2025): 918.

- Article 4 (under review)

  Lingfei Wang, Maria A. Rodriguez, and Nir Lipovetzky. MetaPilot: A DRL-Based Controller for Balancing User-Centric and System-Centric Objectives in HPC Scheduling. Submitted to *Future Generation Computer Systems*.

- Article 5

  Lingfei Wang, Maria A. Rodriguez, and Nir Lipovetzky. Accelerating RL-Based Scheduler Adaptation with Transfer Learning in Evolving HPC Architectures. In *Proceedings of 2025 IEEE 18th International Conference on Cloud Computing (CLOUD)*, pp. 1-11. IEEE, 2025.

- Article 6

  Lingfei Wang, Maria A. Rodriguez, and Nir Lipovetzky. HPCsim: A High-Level Simulation and Workload Data Schema Framework for HPC Workload Management Research. In *Proceedings of 2025 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, IEEE, 2025.

Funding.

Lingfei Wang was supported by the *China Scholarship Council - University of Melbourne PhD Scholarship* from the China Scholarship Council and the University of Melbourne. The support was in terms of both *stipend* and *100% fee remission*. Conference travels were in part funded by the *Faculty of Engineering and Information Technology* and in part by the *School of Computing and Information Systems*, both at the University of Melbourne.

Acknowledgements

This journey is something I could never have achieved alone. Completing a PhD is as much a personal milestone as it is a collective effort, and I am deeply grateful to the many people who have supported me along the way.

I would like to express my heartfelt gratitude to my principal supervisor, Dr. Maria Rodriguez Read, for her unwavering support, thoughtful guidance, and encouragement throughout the course of this research. Your mentorship has been foundational to both my academic growth and personal development. I am sincerely thankful to my co-supervisor, Associate Professor Nir Lipovetzky, for his insights, constructive feedback, and the generous sharing of his expertise. Your perspective has been invaluable in shaping the direction and depth of this work. Additionally, thanks to Dr. Aaron Harwood, whose early guidance helped lay the foundations of this project and set me on the right path during the initial stages of my PhD. I would also like to thank Professor Lars Kulik, my program chair, for his continuous support and for carefully monitoring my progress to help keep this journey on track.

I gratefully acknowledge the financial support provided by the University of Melbourne and the China Scholarship Council, without which this research would not have been possible. I would also like to thank the Faculty of Engineering and Information Technology and the School of Computing and Information Systems for providing a supportive academic environment, excellent resources, and the opportunity to grow as a researcher.

To my colleagues and friends in Melbourne Connect, with whom I spent most of my time — thank you for the countless discussions, shared frustrations, and moments of laughter that made the long days more enjoyable and the journey less lonely. Your support and company meant a great deal to me. A special thanks also to the building itself — Melbourne Connect — which became a quiet companion during the many nights I worked late. Its welcoming spaces and calm atmosphere made it not just a place to work, but a place to think, reflect, and keep moving forward.

To my family, thank you for being my unwavering source of strength. To my grandparents and parents, your love, wisdom, and encouragement have guided me throughout my life, and I am forever grateful for the values you instilled in me. To my extended family, thank you for your support and belief in me from near and far. I am truly fortunate to be surrounded by such a big, caring family that stood by me every step of the way.

Lingfei Wang
Melbourne/Australia, March 2025

Contents

List of Figures

List of Tables

Chapter 1

Introduction

High Performance Computing (HPC) refers to the use of large-scale, parallel computing systems to solve complex computational problems that exceed the capabilities of traditional computing architectures. These systems consist of a large number of interconnected compute nodes that collectively execute parallel workloads, significantly reducing execution time for tasks in domains such as scientific simulations, engineering design, financial modeling, and artificial intelligence. The demand for HPC has grown rapidly due to increasing computational complexity and the explosion of data generated by modern applications [2]. HPC systems represent a significant financial investment, with top-tier systems such as Frontier requiring extensive funding for development and deployment [3]. Beyond hardware expenses, operational costs — including system upgrades, software development, and maintenance — add to the financial obligations of the HPC providers. Efficient resource management is, therefore, critical to maximizing computational throughput and ensuring cost-effectiveness.

HPC job scheduling is a dynamic and complex decision-making process that aims to efficiently manage computational resources while ensuring the timely execution of diverse workloads. It can be fundamentally modeled as a long-running, multi-level, multi-resource scheduling problem, where jobs dynamically arrive and request multiple types of resources, including nodes, CPU cores, memory, GPUs, and network bandwidth. At the node level, a job must be assigned to one or more nodes, and each node must accommodate the job's specific resource demands. Since multiple jobs can run on the same node simultaneously, the scheduler must ensure that the total allocated resources do not exceed the

node's available capacity. This creates a multi-resource scheduling problem at the node level, where jobs must be packed efficiently without causing fragmentation or leaving idle resources. At the cluster level, jobs may request multiple nodes to execute in parallel. The scheduler must determine which set of nodes should be allocated to each job, taking into account resource availability, inter-node communication overhead, and system-wide efficiency. Poor job placement decisions at this level can lead to an unbalanced resource distribution across the cluster, reducing overall utilization. This creates a multi-level scheduling problem, where cluster-level decisions influence node-level resource packing, and efficient node utilization is necessary for maintaining system-wide performance. The complexity of HPC scheduling requires novel scheduling strategies to optimize resource usage while ensuring timely job execution.

HPC scheduling is a multi-stage process, as illustrated in Fig. 1.1. The workflow begins with users submitting jobs to a job queue, where they await scheduling decisions. A scheduler, composed of a job selector and a resource allocator, manages the scheduling process. The job selector determines which job to execute based on system policies. Once a job is selected, the resource allocator assigns available compute resources — such as nodes, CPU cores, GPUs, memory, and network bandwidth — ensuring that multi-resource constraints are met. After allocation, the compute cluster executes the scheduled jobs, and their performance is continuously monitored. The results contribute to performance evaluation, providing feedback on system efficiency, job completion times, and resource utilization. This feedback loop informs future scheduling decisions. This dynamic, multi-resource allocation problem is inherently NP-hard [4], requiring advanced approaches to enhance overall performance.

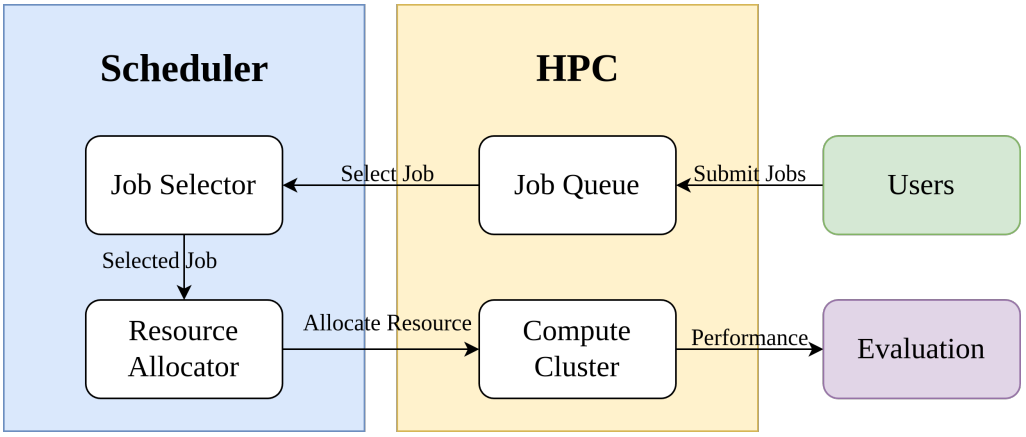

FIGURE 1.1: HPC scheduling process.

Traditional HPC scheduling relies on rule-based and heuristic-based approaches to determine job selection and resource allocation. Rule-based job selection methods, such as First-Come-First-Serve (FCFS) and Shortest Job First (SJF), follow predefined policies aimed at simplicity and predictability but often lead to inefficiencies such as job starvation, resource fragmentation, and reduced overall throughput. Heuristic-based job selection [5–8] enhances rule-based methods by incorporating domain knowledge into predefined job priority rules. These heuristics assign job priorities using static scoring functions that typically include job attributes such as estimated runtime, requested resources, queue waiting time, or current system load. Other techniques like backfilling allow smaller jobs to be scheduled earlier if they do not delay larger jobs, improving system throughput. Once a job is selected, the resource allocator determines how available compute resources are assigned. For example, First-Available allocation prioritizes decision-making speed by assigning jobs to the first set of free nodes that meet resource requirements but may cause resource fragmentation. Best-Fit allocation minimizes resource waste by selecting nodes with the least available resources that can still satisfy the job's requirements, though it increases scheduling overhead. Topology-Aware allocation optimizes performance by considering interconnect topology and reducing communication overhead. These heuristic-based methods can perform effectively under stable or predictable workloads. However, they rely on fixed policies that do not adapt to workload fluctuations, evolving hardware configurations, or changing system constraints.

To improve heuristic-based scheduling, meta-heuristic algorithms — such as Genetic Algorithms (GA) [9], Simulated Annealing (SA) [10], and Ant Colony Optimization (ACO) [11] — have been explored in HPC scheduling [12–15]. These approaches offer greater flexibility than heuristics by searching a broader solution space and iteratively improving scheduling decisions according to defined performance metrics or optimization objectives. Despite their improved scheduling decisions, meta-heuristic methods can be computationally expensive and require careful tuning which may be impractical. Beyond meta-heuristic methods, machine learning-based approaches [16–19] have been explored to improve job selection by learning patterns from historical scheduling data. While machine learning offers adaptability beyond static heuristics, these approaches often rely on extensive labeled datasets, struggle to generalize across different workloads, and cannot respond to real-time system variations dynamically. Additionally, traditional machine learning-based schedulers require frequent retraining to adapt to evolving HPC conditions,

limiting their practicality in highly dynamic environments.

Reinforcement learning (RL) has emerged as a promising approach for HPC job scheduling, offering adaptability beyond traditional heuristics and other machine learning-based methods. Unlike supervised learning, which relies on labeled datasets, RL enables a scheduler to learn job scheduling strategies through interaction with the environment. RL-based schedulers formulate scheduling as a sequential decision-making problem, where an agent receives state observations — such as queue length, resource availability, and job characteristics — and selects scheduling actions to maximize a long-term reward. By continuously refining its policy through trial and error, an RL-based scheduler can dynamically adjust to changes in workload characteristics and system conditions. Recent studies [20–29] have integrated Deep Reinforcement Learning (DRL) to enhance scalability and decision-making, allowing schedulers to handle large-scale HPC workloads more effectively. However, applying RL to HPC scheduling presents challenges that necessitate further research in order to design practical RL-based scheduling frameworks that balance learning efficiency and scheduling performance.

This research focuses on leveraging DRL to develop effective solutions for HPC scheduling problems. Although, DRL enables adaptive decision-making that dynamically responds to workload fluctuations and system constraints, applying DRL to HPC scheduling introduces several challenges. The unbounded state space resulting from the ever-changing job queue and system status complicates learning stability and convergence. Delayed and sparse rewards make it difficult for RL agents to associate scheduling actions with long-term performance outcomes. Additionally, balancing user-centric and system-centric objectives — such as minimizing job waiting time while maximizing resource utilization — requires designing effective reward functions that reflect scheduling trade-offs. Beyond these fundamental challenges, DRL-based scheduling must also coordinate job selection and resource allocation efficiently. Poor coordination can lead to suboptimal scheduling decisions, where jobs are selected without considering resource availability, or resources are allocated inefficiently, leading to fragmentation and reduced utilization. Furthermore, adapting DRL-based schedulers to evolving HPC architectures remains a significant challenge. The ability to generalize across diverse environments and quickly adapt to hardware changes is essential for ensuring long-term applicability. This research systematically investigates these challenges and proposes scalable DRL-based scheduling frameworks to enhance adaptability, efficiency, and performance in HPC environments.

1.1 Challenges in RL-based HPC Scheduling

Job Selector Design. One of the fundamental challenges in developing an RL-based job selector for HPC scheduling is handling the unbounded nature of the job queue. In RL, the agent makes decisions by observing the system state and selecting an action from a defined set. In HPC scheduling, the state typically includes features such as the current job queue, the availability of compute resources, and the status of running jobs, while the action corresponds to selecting a job from the queue for scheduling. In many RL applications, both the state and action spaces are fixed in size. However, in HPC systems, the job queue is dynamic and unbounded — new jobs can arrive at any time, and there is no fixed upper limit on the number of pending jobs. This makes it challenging to construct a stable and compact representation of the state, since the scheduler must encode an arbitrary number of jobs, each with multiple attributes. Likewise, the action space grows with the queue length, as each additional job becomes a new decision candidate. This variability introduces significant complexity into the learning process and requires special design choices to ensure the RL agent can operate effectively.

To manage this complexity, existing RL-based job schedulers typically address the unbounded job queue challenge by considering only a fixed window of jobs at the front of the queue [20, 24–26, 28]. However, since jobs outside the window are not visible to the scheduler, a separate backfilling process is often needed for them to fill resource gaps and improve utilization, ensuring that small, runnable jobs are not unnecessarily delayed. This approach, while simplifying the state and action space, introduces severe limitations in decision-making. By restricting visibility to only a subset of jobs at the head of the queue, the scheduler operates under partial observability, limiting the RL agent's ability to make fully informed decisions. Over time, this may lead to large or more resource-intensive jobs accumulating at the front of the queue, as they are repeatedly postponed while smaller jobs that fit available resources are prioritized. Consequently, job starvation occurs as large jobs experience excessive delays and are unable to be scheduled until a rare opportunity arises when sufficient resources become available simultaneously. Additionally, inefficient resource utilization emerges because resources are allocated in a fragmented manner, favoring short jobs instead of making strategic scheduling decisions that optimize long-term system efficiency.

Resource Allocation Design. Designing an effective resource allocator for HPC scheduling presents significant challenges due to the complex, multi-resource nature of compute nodes and the dynamic variability of job demands. Also, job selection and resource allocation must work in coordination to achieve efficient scheduling decisions. Traditional approaches, as well as some recent RL-based works [20, 24–26, 28], often handle these two components separately where the job selector determines which job(s) to schedule next, and then the resource allocator assigns available nodes to those jobs based on predefined allocation strategies. However, this separation can lead to suboptimal scheduling decisions, as the job selector may not be fully aware of resource availability constraints, and the allocator may not account for the long-term impact of job selection choices. The challenge lies in ensuring that job selection and resource allocation are jointly optimized rather than conflicting in their objectives. For example, consider a scenario where the job selector prioritizes minimizing average waiting time by selecting short jobs first. Meanwhile, the resource allocator aims to minimize the number of idle nodes by distributing jobs across the cluster to balance resource usage. While each component operates with a reasonable objective, their decisions can conflict. Selecting short jobs may be effective for reducing queue delays, but if these small jobs are spread across multiple nodes to balance the load, it can lead to inefficient packing. This fragmentation leaves insufficient contiguous resources for larger jobs, increasing their waiting time and reducing overall system throughput. Such conflicts highlight the need for coordinated scheduling decisions. A joint optimization approach considers both job characteristics and resource availability, allowing small jobs to be compactly scheduled onto fewer nodes while preserving space for larger, resource-intensive jobs. By aligning job selection with resource allocation, the scheduler can respond more effectively to dynamic workload conditions and achieve better overall performance.

Scheduling Objectives Shifting. In HPC scheduling, some system-centric objectives include maximizing CPU and memory utilization, increasing system throughput, and reducing resource fragmentation. These objectives ensure that HPC resources are efficiently utilized to maximize computational output. User-centric objectives, on the other hand, emphasize minimizing job waiting time and job turnaround time, ensuring that users receive timely access to compute resources. However, these two types of objectives are inherently conflicting, and balancing them effectively across diverse workload scenarios and system states remains a central challenge in HPC scheduling. One major challenge is

that fixed-objective schedulers cannot adjust their optimization focus in response to diverse workload scenarios, where the primary scheduling objective may shift. For instance, during periods of high system load, when many jobs are competing for limited resources, the scheduler should prioritize maximizing resource utilization to ensure that available CPUs and memory are used as efficiently as possible. In contrast, during low system load, when resources are abundant and jobs arrive infrequently, utilization is no longer a concern; instead, minimizing job waiting time becomes more important to provide users with a better experience. Since high-load and low-load conditions fluctuate frequently in HPC systems [30, 31], fixed-objective schedulers, with static weights assigned to each objective, lack the flexibility to adapt their priorities accordingly.

A fundamental need to dynamically balance these objectives is that the scheduler must determine when to prioritize utilization over waiting time and vice versa. This requires capturing real-time system conditions, such as job queue characteristics, resource availability, and workload distribution. Learning a dynamic policy that can adjust scheduling decision goals based on system states remains an open research problem. Addressing this challenge requires developing adaptive scheduling strategies that shift priorities based on workload patterns, ensuring that HPC systems remain both efficient and responsive to user demands.

Adaptation to Evolving HPC Architectures. HPC architectures are continuously evolving, driven by advances in processor technologies, interconnects, memory hierarchies, accelerators, and increasing computational demands. These architectural changes have a direct impact on job scheduling, as resource availability, performance characteristics, and workload behavior shift over time. RL-based schedulers, which rely on historical training data to optimize scheduling decisions, face significant challenges when deployed in a modified or entirely new cluster environment. One of the biggest challenges in adapting RL-based schedulers to evolving HPC architectures is that architectural changes lead to shifts in the state and action spaces. When hardware configurations are modified, such as the addition of new processor types, and variations in memory capacity and bandwidth, the features that define the system state and the available scheduling actions may no longer align with those in the original training environment. This disrupts the learned policies of the RL agent, as the previously optimized mappings between states and actions may no longer be valid. Consequently, an RL scheduler trained on an older system may struggle to generalize to the new architecture. Developing methods to efficiently adapt RL

models to these evolving state and action spaces while preserving transferable knowledge remains a key research challenge in HPC scheduling.

1.2 Research Questions

Building upon the challenges identified in the previous section 1.1, this thesis addresses four key research questions. The detailed research questions are:

Q1: How can a reinforcement learning-based job selection strategy be developed to optimize scheduling in HPC environments, addressing the challenge of representing an unbounded and dynamic job queue, integrating backfilling techniques, and ensuring stable and efficient model training?
(Addressing Challenge: Job Selector Design)

Q2: How can a reinforcement learning framework be designed to jointly optimize job selection and resource allocation, and how does this integration impact overall scheduling performance across different HPC systems?
(Addressing Challenge: Resource Allocation Design)

Q3: How can a deep reinforcement learning agent dynamically adjust scheduling decisions to balance user-centric and system-centric objectives in HPC environments, based on real-time system state and workload characteristics?
(Addressing Challenge: Scheduling Objectives Shifting)

Q4: How can reinforcement learning-based HPC schedulers be adapted to evolving cluster architectures using transfer learning, by addressing changes in system state representation, identifying transferable components, and ensuring efficient adaptation in terms of scheduling performance and model convergence?
(Addressing Challenge: Adaptation to Evolving HPC Architectures)

1.3 Research Contributions

This thesis addresses critical gaps in the current literature on reinforcement learning-based HPC scheduling. The major contributions of this research are summarized as follows:

- Introduces innovative strategies to enhance reinforcement learning-based job selection in HPC environments, focusing on overcoming the challenges posed by unbounded job queues and improving scheduling efficiency.
  - Proposes a Split Window Technique to handle the unbounded job queue problem.
  - Develops Schedule Cycling to structure scheduling decisions and training cycles based on workload events.
  - Integrates job selection and backfilling into a unified RL framework, eliminating the need for separate heuristic-based backfilling mechanisms.
- Develops HeraSched, a novel hierarchical reinforcement learning (HRL)-based scheduler for intelligent job selection and resource allocation in heterogeneous HPC environments.
  - Introduces the HRL-based approach for joint job selection and node-level allocation.
  - Implements a heterogeneous-aware resource allocation mechanism.
  - Designs distinct reward mechanisms for job selection and resource allocation.
  - Evaluate HeraSched with different HPC systems with both CPU and GPU clusters, confirming its adaptability in various HPC environments.
- Proposes MetaPilot, a DRL-based controller that dynamically selects scheduling objectives based on real-time system conditions, rather than relying on fixed or mixed-objective heuristics.
  - Introduces a scheduling framework that selects between utilization-based and waiting-time-based objectives in response to real-time system conditions.
  - Designs a state representation that captures key workload and resource status features.

  - Implements a reward function that adapts to varying system states, enabling MetaPilot to make objective-aware scheduling decisions.
  - Shows that MetaPilot achieves better waiting time and resource utilization compared to fixed-objective baselines in the evaluated scenarios.

- Develops a transfer learning-based adaptation framework for RL-based HPC schedulers, enabling efficient migration to evolving cluster architectures.
  - Introduces Separate Feature Extraction to isolate state changes in job and cluster dynamics, preventing unnecessary retraining of unaffected model components.
  - Proposes Selective Transfer Learning to retrain only the most impacted model parts, reducing adaptation overhead while maintaining scheduling performance.
  - Evaluations demonstrate significantly reduced retraining time and improved adaptability compared to schedulers trained from scratch.

- Comprehensive Empirical Evaluation on Real HPC Workloads. Existing research on RL-based HPC scheduling often lacks evaluation on modern HPC clusters, particularly those with diverse CPU and GPU architectures using recent trace data. Many studies still rely on outdated traces from the Parallel Workloads Archive [30], such as the SDSC-SP2 (1998), HPC2N (2002) and PIK-IPLEX (2009), which do not reflect the complexity of current HPC environments. To address this gap, we conducted extensive experiments using real-world HPC configurations and job traces collected from operating HPC in recent years. Our evaluation spans multiple HPC partitions, including both CPU- and GPU-based clusters, ensuring that the proposed methods are rigorously tested under realistic and diverse workload conditions. This empirical validation demonstrates the effectiveness and adaptability of our approach in practical HPC scheduling scenarios.

Limitations. This thesis presents a set of RL-based scheduling techniques evaluated using real-world trace data and system configurations from operational HPC systems. However, all experiments were conducted in simulation rather than directly on live production clusters. This choice was necessary due to the high cost and potential disruption associated with running large-scale scheduling experiments on operational HPC systems.

While the simulation environment was carefully designed to reflect real system behavior — including accurate job submissions, resource availability, and scheduler constraints — it cannot fully capture real-world variability such as hardware faults, user interaction patterns, or system-level contention. As a result, although the findings demonstrate promising adaptability, efficiency, and performance improvements, the real-world effectiveness of the proposed methods remains to be validated through deployment in real production environments.

1.4 Thesis Organization

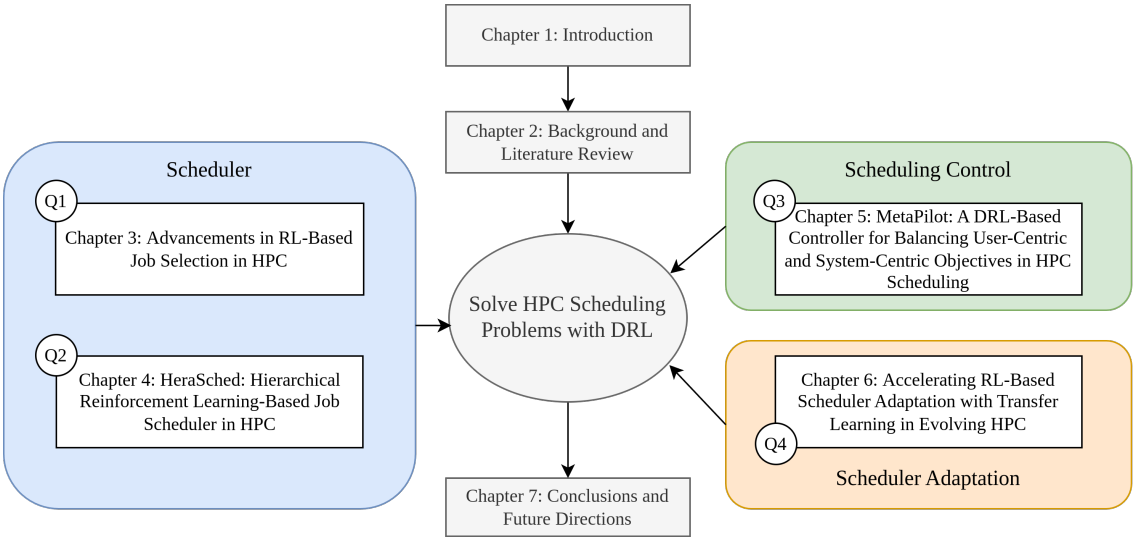

FIGURE 1.2: Organization of the Thesis.

This thesis explores the application of DRL in HPC scheduling, addressing challenges and research questions summarized in previous sections. The research is structured into seven chapters, as illustrated in Fig. 1.2.

- **Chapter 2: Background and Literature Review.** This chapter reviews existing HPC scheduling approaches, including rule-based, heuristic, meta-heuristic, and machine learning-based techniques. It also discusses existing applications of reinforcement learning in scheduling and identifies key gaps in the literature.

- **Chapter 3: Advancements in RL-Based Job Selection in HPC.** This chapter presents an RL-based job selector and introduces the Split Window Technique to address the unbounded state space issue, schedule cycling to enhance training

efficiency, and an integrated backfilling mechanism within the RL framework. The chapter is derived from the following publications:

  - **Lingfei Wang**, Aaron Harwood, and Maria A. Rodriguez. A Deep Reinforcement Learning Scheduler with Back-filling for High Performance Computing. In *Proceedings of 2021 IEEE Asia-Pacific Conference on Computer Science and Data Engineering (CSDE)*, pp. 1-6. IEEE, 2021.
  - **Lingfei Wang**, Aaron Harwood, and Maria A. Rodriguez. Deep Back-Filling: a Split Window Technique for Deep Online Cluster Job Scheduling. In *Proceedings of 2023 IEEE International Conference on High Performance Computing & Communications*, pp. 772-779. IEEE, 2023.

- **Chapter 4: Hierarchical Reinforcement Learning-Based Job Scheduler in HPC.** This chapter presents HeraSched, an HRL-based scheduling framework that integrates job selection and resource allocation into a unified decision-making process. Designed for heterogeneous HPC clusters, HeraSched accounts for variations in memory on different nodes, improving scheduling efficiency for diverse workloads. The chapter is derived from the following paper:

  - **Lingfei Wang**, Maria A. Rodriguez, and Nir Lipovetzky. Optimizing HPC Scheduling: A Hierarchical Reinforcement Learning Approach for Intelligent Job Selection and Allocation. *The Journal of Supercomputing* 81, no. 8 (2025): 918.

- **Chapter 5: MetaPilot – A DRL-Based Controller for Balancing User-Centric and System-Centric Objectives in HPC Scheduling.** This chapter introduces MetaPilot, a high-level scheduling controller that dynamically selects between user-centric and system-centric objectives based on real-time system conditions. MetaPilot continuously learns from system conditions, enabling adaptive decision-making to optimize HPC performance. The chapter is derived from the following paper (under review):

  - **Lingfei Wang**, Maria A. Rodriguez, and Nir Lipovetzky. MetaPilot: A DRL-Based Controller for Balancing User-Centric and System-Centric Objectives in HPC Scheduling. Submitted to *Future Generation Computer Systems*.

- **Chapter 6: Accelerating RL-Based Scheduler Adaptation with Transfer Learning in Evolving HPC.** This chapter investigates the challenge of adapting RL-based schedulers to evolving HPC architectures. A Transfer Learning framework is proposed, incorporating Separate Feature Extraction to handle environment-specific changes and Selective Transfer Learning to enable efficient adaptation with minimal retraining. The chapter is derived from the following paper:

  – **Lingfei Wang**, Maria A. Rodriguez, and Nir Lipovetzky. Accelerating RL-Based Scheduler Adaptation with Transfer Learning in Evolving HPC Architectures. In *Proceedings of 2025 IEEE 18th International Conference on Cloud Computing (CLOUD)*, pp. 1-11. IEEE, 2025.

- **Chapter 7: Conclusions and Future Directions.** This chapter summarizes the key findings of the thesis, discusses the broader impact of DRL-based scheduling in HPC, and outlines potential future research directions.

Chapter 2

Background and Literature Review

Reinforcement learning (RL) has gained traction in High Performance Computing (HPC) job scheduling, offering adaptive decision-making in dynamic environments. This section reviews fundamental RL concepts, deep RL, hierarchical RL, and transfer learning, emphasizing their relevance to HPC scheduling. Traditional heuristic-based schedulers, such as First-Come-First-Serve (FCFS) and backfilling, remain widely used but face limitations in fairness and efficiency. Meta-heuristic and machine learning-augmented schedulers improve optimization but struggle with real-time adaptability. Reinforcement learning-based approaches offer a promising alternative, dynamically selecting jobs and allocating resources. The review also explores resource allocation strategies, including first-fit, best-fit, and topology-aware methods, highlighting their role in modern heterogeneous HPC systems. This study consolidates recent advances and challenges, providing a foundation for intelligent job scheduling research.

2.1 Introduction

HPC scheduling is a critical component in ensuring efficient resource utilization and timely job execution in large-scale computing clusters. The complexity of modern HPC systems, characterized by heterogeneous resources, dynamic workloads, and diverse user requirements, necessitates advanced scheduling strategies that go beyond traditional static

heuristic-based approaches. This chapter provides a structured overview of HPC job scheduling, establishing the necessary background, defining key problems, and reviewing existing methodologies.

This chapter begins with a problem definition, outlining the core objectives of HPC scheduling, including job selection and resource allocation. Next, it introduces the evaluation metrics commonly used to assess scheduling performance, such as resource utilization, job waiting time, turnaround time, and fairness. Following this, the chapter presents a background on reinforcement learning. It introduces fundamental RL concepts, including environment, Markov decision processes, value functions, and policy optimization, before discussing advancements in deep reinforcement learning, hierarchical reinforcement learning, and transfer learning in the context of reinforcement learning. The final section offers a comprehensive survey on HPC scheduling, examining the evolution of scheduling techniques. It covers traditional heuristic-based schedulers, widely adopted real-world HPC schedulers, and meta-heuristic-based approaches that integrate optimization strategies. Additionally, it explores machine learning-improved schedulers and reinforcement learning-based schedulers, highlighting their advantages and limitations. Through this literature review, we identify key gaps in existing work and establish the motivation for our proposed RL-based HPC scheduling framework.

2.2 HPC Job Scheduling

This section provides an in-depth exploration of the HPC scheduling problem, including its fundamental challenges, system components, and key considerations in job selection and resource allocation. It introduces the formal problem definition, discusses the dynamic nature of job queues and resource availability, and establishes the notations used throughout the research.

2.2.1 Problem Definition

HPC job scheduling involves dynamically assigning computing resources to user-submitted jobs while optimizing the scheduling objectives, such as job waiting time, resource utilization, etc. An HPC cluster consists of multiple interconnected compute nodes that execute user-submitted jobs. Compute nodes are connected through a set of network switches,

represented as $S = \{s_1, s_2, \ldots, s_K\}$, where each switch s_k interconnects a subset of nodes. Each compute node is equipped with a specific set of resources, such as CPUs, memory, and GPUs. Nodes are connected through a network topology with switches. Each compute node n_i is uniquely identified by a system ID nID_i, where $i \in \{1, 2, \ldots, N\}$. Compute nodes in an HPC cluster are equipped with different resources to accommodate diverse computational needs. In general, jobs can be categorized based on their primary processing requirements: CPU-intensive jobs primarily rely on CPUs, while GPU-accelerated jobs leverage GPUs for computations. To support various workloads, compute nodes are provisioned with different hardware configurations, including varying numbers of CPU cores, memory capacities, and specialized accelerators such as GPUs or other hardware accelerators. While resource configurations can be highly heterogeneous, in practice, clusters are often organized into partitions to optimize scheduling and resource allocation. The most common partitions include CPU partitions, which primarily consist of nodes optimized for multi-threaded CPU workloads, and GPU partitions, which include nodes equipped with GPUs to handle deep learning, simulations, and other parallelizable tasks. In this research, we define a compute node n_i with the following resources:

- c_i - Number of CPU cores available.
- m_i - Amount of available memory (in GB).
- g_i - Number of GPUs available (if any).

A node can execute multiple jobs simultaneously, subject to resource constraints. The set of jobs currently running on node n_i at time t is denoted as:

$$\mathcal{R}_i(t) = \{J_j \mid J_j \text{ is running on } n_i \text{ at time } t\}. \tag{2.1}$$

In an HPC system, jobs arrive dynamically as users submit computational tasks to the cluster. Unlike batch processing systems, where all jobs are predefined before execution begins, HPC workloads evolve continuously, requiring the scheduler to make real-time decisions based on the current system state. Each submitted job enters a centralized *job queue* $\mathcal{Q}(t)$, where it waits for execution according to system policies. The job queue is dynamic, expanding as new jobs arrive and shrinking as jobs are scheduled and completed. The number of jobs in the queue at time t is denoted as $L(t) = |\mathcal{Q}(t)|$ which fluctuates as job submissions and completions occur.

Each job J_j is characterized by a set of attributes that define its resource requirements and execution constraints. A job is formally represented as:

$$J_j = \langle ID_j, w_j, e_j, l_j, h_j, q_j, b_j, d_j, r_j(t), A_j \rangle. \tag{2.2}$$

The system assigns a unique job identifier ID_j upon submission, ensuring that each job can be tracked throughout its lifecycle. The submission time w_j records the timestamp when the job enters the queue, reflecting the dynamic nature of job arrivals. A job j has a user-provided requested runtime, e_j, which denotes an upper bound on the job's execution time. The actual runtime of a job may be shorter, depending on system conditions and job efficiency. Jobs can request multiple compute nodes to enable parallel execution. The number of compute nodes requested is denoted as l_j, ensuring that applications requiring distributed computation can execute across multiple machines. Each node assigned to a job must satisfy its resource demands, including the number of CPU cores per node h_j, the amount of memory per node q_j, and the number of GPUs per node b_j ($b_j = 0$ for CPU jobs). These resource specifications vary significantly based on workload characteristics; for example, CPU-bound tasks primarily demand computing cores, while deep learning applications require GPUs with high memory. The scheduler guarantees that the job is allocated exactly l_j nodes meeting these requirements before execution begins.

Additionally, the scheduled start time of a job, denoted as d_j, defines when the system assigns resources and begins execution. As the job runs, its current runtime $r_j(t)$ tracks the elapsed execution time at any given moment, ensuring that it does not exceed e_j. The set of allocated nodes A_j represents the physical machines assigned to execute the job.

Additionally, job workloads in HPC clusters exhibit *high heterogeneity*, where different jobs have vastly different computational demands [32]. Some jobs are CPU-intensive, performing complex mathematical computations, while others require GPU acceleration for high-throughput parallel processing. Memory-intensive jobs, such as large-scale data analytics, demand significant memory per node. The execution time of a job is also highly variable, ranging from short-lived tasks to long-running simulations that may persist for hours or even days.

A key aspect of HPC scheduling is determining whether a job is executable at a given time. A job is considered feasible at time t if the cluster has sufficient available resources to satisfy its requirements. This feasibility condition is formally expressed using a binary

indicator $k_j(t)$, where:

$$k_j(t) = \begin{cases} 1, & \text{if } \exists A_j \subseteq \mathcal{N}, |A_j| = l_j, \text{ such that resources satisfy } J_j, \\ 0, & \text{otherwise.} \end{cases} \tag{2.3}$$

The scheduling process in an HPC cluster is event-driven, with scheduling decisions triggered by two key events: the arrival of a new job and the completion of a running job. When a new job is submitted, it enters the pending job queue $\mathcal{Q}(t)$, updating the scheduling state. Similarly, when a running job completes, the resources it occupied are released back into the system, allowing the scheduler to reassign them to pending jobs. At any scheduling decision point, denoted by time t, the scheduler may schedule multiple jobs simultaneously, depending on available resources and job feasibility.

The scheduling process consists of two core steps: job selection and resource allocation. Job selection determines which job(s) from the queue should be scheduled next. The selected job must satisfy feasibility constraints, ensuring that sufficient resources are available for execution. Once a job is selected, the scheduler proceeds to *resource allocation*, where it assigns compute nodes and system resources to the job. This allocation must satisfy the job's requested number of nodes l_j, as well as its per-node requirements for CPU cores h_j, memory q_j, and GPUs b_j. If sufficient resources are not available at time t, the job remains in the queue until a future scheduling opportunity. This process continues iteratively as jobs arrive and complete, maintaining a dynamic balance between job execution and resource availability. Efficient scheduling ensures that HPC workloads are processed with minimal wait times while optimizing overall system performance. These scheduling objectives, including the evaluation metrics used to assess scheduling performance, are discussed in detail in Section 2.2.3.

We adopt the general model above: each job specifies a node count and per-node requirements (cores, memory, and GPUs where applicable), and allocation is subject to node heterogeneity and network topology. Exception: Chapter 3 intentionally isolates the job-selection problem under a simplified cluster model to analyze learning dynamics. In Chapter 3, jobs declare a total CPU-core request (no explicit node counts), nodes are treated as homogeneous; memory, GPUs, and topology are not modeled in that chapter. Chapters 4 to 6 reinstate the full heterogeneous model introduced here and evaluate selection and allocation jointly.

TABLE 2.1: Notations for HPC Scheduling

Cluster and Network Notations	
N	Total number of compute nodes in the cluster
S	Total number of network switches
s_k	Switch indexed by k, where $k \in \{1, \ldots, S\}$
n_i	Compute node indexed by i, where $i \in \{1, \ldots, N\}$
nID_i	System identifier for node n_i
$F(t)$	Number of free nodes in the cluster at time t (nodes running no jobs)
Node-Level Resource Notations	
c_i	Total number of CPU cores available on node n_i
$c_i^{\text{occ}}(t)$	Number of CPU cores occupied on node n_i at time t
m_i	Total memory available on node n_i (e.g., in GB)
$m_i^{\text{occ}}(t)$	Amount of memory occupied on node n_i at time t
g_i	Total number of GPUs available on node n_i
$g_i^{\text{occ}}(t)$	Number of GPUs occupied on node n_i at time t
$\mathcal{R}_i(t)$	Set of jobs running on node n_i at time t
Job Queue Notations	
$\mathcal{Q}(t)$	Set of jobs in the pending queue at time t
$L(t)$	Number of jobs in the queue at time t
Job-Level Notations	
J_j	Job indexed by j, where $J_j \in \mathcal{Q}(t)$
ID_j	Unique job identifier assigned at submission
w_j	Job submission time (timestamp)
e_j	Requested runtime of job J_j (upper bound)
l_j	Number of compute nodes requested by job J_j
h_j	Number of CPU cores per node requested by job J_j
q_j	Amount of memory per node requested by job J_j
b_j	Number of GPUs per node requested by job J_j
d_j	Scheduled start time of job J_j
$r_j(t)$	Current runtime progress of job J_j at time t, where $r_j(t) \leq e_j$
r_j	Total runtime of job J_j after job J_j completion, where $r_j \leq e_j$
A_j	Set of allocated nodes for job J_j
$k_j(t)$	Feasibility indicator: $k_j(t) = 1$ if job can run at time t, else 0

2.2.2 Notation and Terminology

All notations used in this research are summarized in Table 2.1, providing a formal definition of system parameters, job attributes, and scheduling variables.

2.2.3 Evaluation Metrics

The effectiveness of an HPC job scheduling strategy is determined by how well it balances competing priorities, such as optimizing resource usage, ensuring timely job execution, and maintaining fairness among users and workloads. The selection of evaluation metrics is crucial, as different metrics emphasize different aspects of scheduling performance, and optimizing one often comes at the expense of another. For example, prioritizing short job wait times may lead to inefficiencies in resource utilization, while maximizing system throughput could result in job starvation.

Scheduling objectives can generally be categorized into system-centric and user-centric goals. System-centric metrics, such as resource utilization and makespan, focus on maximizing computational efficiency at the cluster level. User-centric metrics, such as waiting time and slowdown, aim to improve user experience by minimizing delays and ensuring fair resource allocation. Additionally, fairness metrics play a crucial role in multi-user environments, preventing individual users or workloads from monopolizing computational resources. This section discusses key scheduling objectives, evaluates the advantages and limitations of commonly used metrics, and presents their formal mathematical definitions.

Resource utilization measures how effectively computing resources are used over time. It is often employed as a system-centric metric, helping administrators ensure that HPC clusters operate efficiently. Utilization is particularly relevant in batch-processing systems, where high utilization often correlates with improved throughput. A major advantage of focusing on utilization is that it aligns with system efficiency goals — higher utilization typically implies better use of available computing power. However, utilization alone does not necessarily reflect scheduling quality. In cases where job submission rates are low, even an optimal scheduling policy will result in low utilization, making it an ineffective discriminator of scheduling performance. Conversely, in high-load scenarios where many jobs are queued and the system remains consistently busy, different scheduling policies may still produce similar utilization levels over a long period [33]. Similarly, in non-steady-state workloads, where job arrivals fluctuate significantly, utilization fails to capture scheduling effectiveness.

Another limitation is that utilization is often a system administrator's priority, rather than a user-centric metric. A naive way to maximize utilization is to maintain a large queue of waiting jobs and schedule those that fit best at any given moment. This strategy frequently favors small jobs, as they can easily fill available resource gaps. However, this approach can lead to significant delays for larger jobs and even potential job starvation, neither of which impact the utilization metric directly.

The general formula for resource utilization $\mathcal{U}$ can be extended to account for different resource types:

$$\mathcal{U}_r = \frac{\sum_{j \in J} r_j \cdot \sum_{n_i \in A_j} R_i}{T \cdot N} \tag{2.4}$$

where R_i represents the specific resource type (CPU cores, GPUs, memory, or nodes), T is the observation period, and N is the total number of available resources of that type.

Average Waiting Time and Turnaround Time. Average waiting time measures how long jobs remain in the queue before execution. It is widely used in user-centric scheduling policies, where minimizing waiting times improves user experience. Minimizing waiting time improves system responsiveness, particularly for short jobs. However, using this objective assigns equal importance to all jobs, regardless of their computational demands. In practice, this often favors small jobs, which are more numerous but contribute little to the total workload. As a result, schedules that optimize waiting time may appear inefficient in terms of overall job packing. Additionally, focusing solely on waiting time does not guarantee an improvement in overall scheduling quality. Some studies also examine maximum waiting time, aiming to bound the worst-case delay that a user may experience.

The average waiting time (AWT) is defined as:

$$\text{AWT} = \frac{1}{|J|} \sum_{j \in J} (d_j - w_j) \tag{2.5}$$

And the maximum waiting time (MWT) is defined as:

$$\text{MWT} = \max_{j \in J} (d_j - w_j) \tag{2.6}$$

where w_j is the submission time of job J_j.

Closely related to waiting time is turnaround time, which includes both the waiting time and execution time, providing a complete measure of how long a job spends in the system. The average turnaround time (ATT) includes execution time:

$$\text{ATT} = \frac{1}{|J|} \sum_{j \in J} (d_j - w_j + r_j) \tag{2.7}$$

Average Bounded Slowdown (ABS). The slowdown of a job quantifies the ratio of the time it spends in the system relative to its execution time. It is widely used in scheduling research as an evaluation metric, measuring that all jobs, regardless of length, experience proportional delays. However, one major issue in HPC workloads is that very short jobs may experience arbitrarily high slowdowns even with negligible wait times. For example, a job that executes for one second but waits for ten minutes has a slowdown of 600, despite an insignificant absolute delay. To mitigate this issue, the bounded slowdown is used instead, introducing a threshold τ to prevent extreme values:

$$Slowdown = \frac{\max(d_j - w_j, \tau)}{\max(r_j, \tau)} \tag{2.8}$$

where τ is a constant to prevent extremely small jobs from skewing the metric. The average bounded slowdown (ABS) is then computed as:

$$\text{ABS} = \frac{1}{|J|} \sum_{j \in J} Slowdown_j \tag{2.9}$$

In the context of HPC scheduling, ABS incorporates the runtime of a job rather than just its queue delay. However, while ABS differentiates between short and long jobs based on their execution times, it does not consider the resource requested for a job. The amount of the resource a job required is also a key factor in scheduling decisions. Two jobs with the same runtime but vastly different resource requirements may have significantly different impacts on system performance, yet ABS treats them equivalently. This limitation means that ABS may favor small, resource-light jobs at the expense of large, resource-heavy jobs, as the latter inherently experience higher slowdowns. While this prioritization can improve waiting times for small jobs, it may reduce scheduling efficiency by leading to fragmented resource allocation. Large jobs may be unfairly penalized under a policy that strictly minimizes ABS. Thus, while ABS serves as an effective fairness measure in

general computing contexts, its applicability in HPC scheduling is more nuanced, requiring additional considerations such as job resource requests and system-wide scheduling objectives.

Makespan is a system-wide metric that measures the total time required to complete a given set of jobs. It is particularly relevant in the bag of task scheduling scenarios, where all jobs are known in advance. In such contexts, minimizing makespan ensures that the system completes the workload as quickly as possible, thereby maximizing throughput.

A major advantage of makespan is its direct correlation with system efficiency; a shorter makespan generally indicates better resource utilization and faster job completion. However, in real-world HPC workloads, jobs do not arrive as a fixed batch; instead, they are continuously submitted by users over time. As a result, minimizing makespan for a given subset of jobs does not necessarily optimize scheduling performance.

Another limitation of makespan is that it focuses only on overall completion time, without considering individual job performance. In interactive or continuously running HPC systems, some jobs may experience extreme delays while others complete quickly, leading to unfair resource allocation. Furthermore, optimizing makespan does not inherently improve scheduling fairness, as a scheduler minimizing makespan may prioritize large parallel jobs at the expense of smaller jobs, potentially leading to job starvation.

In mathematical terms, makespan is defined as:

$$Makespan = \max_{j \in J}(d_j + r_j) \tag{2.10}$$

where d_j is the start time of job J_j and r_j is its execution time.

Fairness ensures that no job or user monopolizes system resources at the expense of others. Unlike the previous metrics, fairness does not have a single standard mathematical definition. Instead, fairness can be evaluated at multiple levels. *Job fairness*, ensuring jobs are scheduled in a way that avoids excessive waiting times. *User fairness*, balancing resource allocation among different users. *Workload fairness* ensuring fair resource distribution across different workloads.

Jain's Fairness Index [34] is a widely used metric for measuring fairness in resource allocation. Given a set of resource allocations $x_1, x_2, \ldots, x_n$, it is defined as:

$$\mathcal{J}(x_1, x_2, \ldots, x_n) = \frac{(\sum_{i=1}^{n} x_i)^2}{n \sum_{i=1}^{n} x_i^2}. \tag{2.11}$$

This index yields a value in the range $[1/n, 1]$, where 1 indicates perfect fairness, meaning all users receive equal resource shares, and lower values indicate imbalances. However, Jain's Fairness Index is not well-suited for HPC scheduling. HPC scheduling involves jobs with heterogeneous resource demands, including varying CPU, memory, and GPU requirements. Jain's index assumes equal importance across all users, ignoring job-specific constraints such as job size, execution time, and priority policies. Also, Jain's index only considers a single-dimensional fairness measure, failing to capture fairness in multi-resource environments.

Instead of relying on Jain's Fairness Index, practical HPC scheduling systems enforce fairness through policy-based mechanisms. SLURM [35] employs a fair-share scheduling policy that dynamically adjusts job priorities based on historical resource usage. The fair-share factor influences job priority by considering the proportion of computing resources allocated to a user or account and their past resource consumption. This approach ensures that users who have consumed fewer resources in the past are given higher priority, promoting equitable resource distribution. The Maui Scheduler [36] incorporates a fair-share mechanism that allows historical resource utilization to influence job scheduling decisions. Administrators can set system utilization targets for users, groups, accounts, classes, and quality of service levels. Maui adjusts job priorities to meet these targets, ensuring that resource distribution aligns with organizational policies and usage goals. Borg [32] is Google's cluster management system that handles a vast number of jobs across numerous clusters. It employs workload-aware scheduling to balance different types of jobs, such as batch and interactive workloads. This ensures that long-running batch jobs do not monopolize resources, allowing interactive jobs to access the necessary resources promptly.

2.3 Reinforcement Learning

This section provides a structured overview of reinforcement learning and its extensions. It begins with an introduction to *Fundamental Reinforcement Learning*, covering key concepts such as states, actions, policies, and value functions. *Deep Reinforcement Learning* extends traditional reinforcement learning by leveraging deep neural networks to approximate value functions and policies, enabling learning in high-dimensional and continuous action spaces. *Hierarchical Reinforcement Learning* introduces structured decision-making by decomposing tasks into multiple levels of abstraction, improving sample efficiency and long-term planning. Finally, *Transfer Learning in Reinforcement Learning* explores methods for leveraging prior knowledge to accelerate learning in new tasks, enhancing generalization and reducing sample complexity. Together, these subsections provide a comprehensive foundation for understanding RL methodologies and their relevance to real-world applications.

2.3.1 Fundamental Reinforcement Learning

Learning methods, such as those used in supervised and unsupervised learning contexts, differ in how they process data. **Supervised Learning** is a method where a model is trained to map inputs to outputs by using labeled data [37]. The model compares its predictions to the ground truth provided in the dataset and calculates the prediction error. This error is then used to adjust the model iteratively, aiming to minimize the error and improve the accuracy of its predictions. In **Unsupervised Learning**, a model is trained on unlabeled data to identify underlying patterns or structures within the dataset [37]. Unlike supervised learning, there are no predefined labels or correct outputs to guide the process. Instead, the model learns to group similar data points, reduce dimensionality, or uncover hidden relationships by optimizing a specific objective, such as minimizing intra-cluster distances in clustering or reconstructing data in dimensionality reduction techniques. **Reinforcement Learning** (RL) differs fundamentally from these approaches. It is an adaptive control framework designed to solve problems where feedback is delayed, partial, or dependent on a sequence of actions. In RL, an agent learns by interacting with an environment, where each action taken by the agent affects the future state of the system. The agent receives feedback in the form of rewards, which guide it toward achieving long-term objectives. This dynamic learning process enables

RL to address complex decision-making tasks that traditional learning methods cannot effectively solve [38].

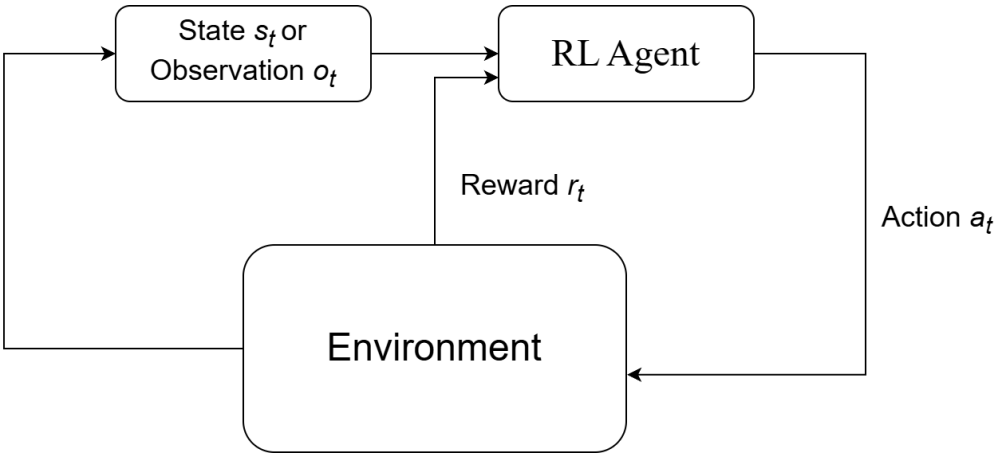

FIGURE 2.1: Agent-environment interaction loop in reinforcement learning.

Figure 2.1 illustrates the general interaction between an RL agent and its environment in a high-level overview. At each time step t, the environment provides the agent with a state s_t or observation o_t, which represents the current situation of the environment. Based on this information, the RL agent selects and executes an action a_t to the environment. The environment responds by transitioning to a new state s_{t+1} and providing the agent with a reward r_t, which indicates the immediate outcome of the agent's action. This interaction loop repeats over time, generating episodes or trajectories that vary in length until a terminal state is encountered. The primary objective of the agent is to determine a policy that maximizes the cumulative rewards collected throughout these interactions. A fundamental concept in RL is learning through trial and error. The agent explores its set of possible actions and observes the resulting outcomes. Larger rewards encourage actions that lead to higher returns, while lower rewards discourage less favorable choices, guiding the agent toward optimal behavior. Through repeated interactions, the agent gradually learns which actions are most effective in specific scenarios.

2.3.1.1 Key Concepts in RL

To formalize and implement RL effectively, several key concepts must be defined and understood, including states and observations, action spaces, policies, and trajectories. These concepts provide the foundation for RL algorithms and their applications to solve complex decision-making tasks.

States and Observations. In reinforcement learning, the state represents the current configuration or condition of the environment, encapsulating all the information necessary for decision-making. The collection of all possible states is referred to as the *state space*, which can be finite or infinite depending on the problem. For example, in a grid-world environment, the state space consists of all grid positions, while in continuous domains like robotic control, the state space may include infinitely many configurations. However, in many real-world scenarios, the agent does not have access to the full state of the environment. Instead, it receives observations, which are partial or noisy representations of the true state. This distinction is critical in reinforcement learning: in a fully observable environment, the agent can make decisions based directly on the state (s_t), while in partially observable environments, the agent must infer the underlying state from observations (o_t). The ability to interpret states or observations is fundamental for the agent to make effective decisions.

Action Spaces. The action space defines the set of all possible actions an agent can take in the environment. Actions can be discrete, such as selecting one of a finite number of choices (e.g., moving left or right), or continuous, where the action space consists of a range of values (e.g., controlling the speed of a car). The structure and size of the action space significantly impact the complexity of the learning process. In high-dimensional or continuous action spaces, advanced techniques such as policy approximation [38, 39] or action sampling [40, 41] are often required to efficiently explore and optimize actions.

Policy. A policy is a strategy that defines the agent's behavior by mapping states (or observations) to actions. It can be represented as a deterministic function ($\pi(s) = a$) or a probabilistic distribution over actions ($\pi(a|s)$). Policies are central to reinforcement learning, as the agent aims to learn an optimal policy (π^*) that maximizes cumulative rewards. In practice, policies can be parameterized by models, which are updated iteratively during the learning process.

Trajectories. A trajectory is a sequence of states, actions, and rewards observed by the agent during its interaction with the environment. It is commonly represented as $\tau = (s_0, a_0, r_0, s_1, a_1, r_1, \dots)$, where s_t is the state at time t, a_t is the action taken in state s_t, and r_t is the reward received after taking action a_t. Trajectories may have varying lengths depending on the task. For episodic tasks, the trajectory terminates upon reaching a terminal state, while for continuing tasks, it can extend indefinitely. Trajectories are

essential in reinforcement learning as they capture the agent's experience and provide the data necessary for evaluating and improving its policy.

2.3.1.2 Markov Decision Process

The Markov property is a fundamental concept in reinforcement learning, describing the memoryless nature of the environment's dynamics. An environment satisfies the Markov property if the future state depends only on the current state and action, and not on any prior states or actions. Formally, this can be expressed as:

$$P(s_{t+1} \mid s_t, a_t, s_{t-1}, a_{t-1}, \ldots, s_0, a_0) = P(s_{t+1} \mid s_t, a_t), \tag{2.12}$$

where s_t represents the state at time t, a_t is the action taken at time t, and $P(s_{t+1} \mid s_t, a_t)$ is the transition probability to the next state. Reinforcement learning problems are typically modeled as a Markov Decision Process (MDP), which assumes that the Markov property holds. This simplifies the learning process by reducing the dependency of decisions to the current state and action, allowing the agent to learn optimal policies based on these local dynamics. When the environment is partially observable, such that the agent does not have access to the true state s_t, a Partially Observable Markov Decision Process (POMDP) may be used, where the agent must infer the hidden state from observations.

Return and Optimization Problem. The return (G_t) is the cumulative reward the agent receives over time, starting from a specific time step t. It is defined as:

$$G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \tag{2.13}$$

where $\gamma \in [0, 1]$ is the discount factor that balances immediate and future rewards. Different tasks may use variations of the return. For example, in finite-horizon problems, the return is computed over a fixed number of steps, while in infinite-horizon tasks, it considers the long-term behavior. Other formulations, such as average reward or episodic reward, are used depending on the problem setting. The goal in reinforcement learning is to find an optimal policy π^* that maximizes the expected return from any given state. This is formalized as:

$$\pi^* = \arg\max_{\pi} \mathbb{E}_{\pi} \left[G_t | s_t \right] \tag{2.14}$$

This optimization problem can be approached using different methods, such as value-based methods (e.g., Q-learning [42]), policy-based methods (e.g., REINFORCE [43]), or actor-critic methods [44], which combine the two. These approaches will be introduced in detail in later sections. The choice of optimization technique depends on the nature of the environment, the complexity of the action space, and computational constraints.

Value Functions. Value functions measure the expected return for a given state or state-action pair and are essential for evaluating the quality of decisions in reinforcement learning. These functions provide the foundation for assessing and improving policies. The *state-value function*, denoted as $v_\pi(s)$, represents the expected return when the agent starts in state s and follows policy π. It is defined as:

$$v_\pi(s) = \mathbb{E}_\pi \left[G_t \mid S_t = s \right] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s \right], \tag{2.15}$$

where G_t is the return, $\gamma \in [0, 1]$ is the discount factor that controls the weight of future rewards, and $\mathbb{E}_\pi$ denotes the expectation assuming the agent follows policy π.

The *action-value function*, denoted as $q_\pi(s, a)$, defines the expected return starting in state s, taking action a, and then continuing to follow policy π. It is expressed as:

$$q_\pi(s, a) = \mathbb{E}_\pi \left[G_t \mid S_t = s, A_t = a \right] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s, A_t = a \right]. \tag{2.16}$$

Optimal value functions are derived when using the best possible policy. The *optimal state-value function* $v_*(s)$ and the *optimal action-value function* $q_*(s, a)$ are defined as:

$$v_*(s) = \max_\pi v_\pi(s), \quad q_*(s, a) = \max_\pi q_\pi(s, a). \tag{2.17}$$

These optimal functions determine the maximum achievable return for any state or state-action pair, serving as a guide for identifying the best possible policy π^*.

Bellman Equations. All four of the above value functions satisfy recursive relationships known as Bellman equations, which express their self-consistent properties. The Bellman equations define recursive relationships between value functions, capturing how the value of a state (or state-action pair) depends on future states and rewards. They serve as the foundation for many reinforcement learning algorithms, as they enable the

estimation of optimal policies by iteratively updating value functions. The Bellman equations express the expected return of a state or action in terms of immediate rewards and the expected return of subsequent states, allowing for systematic policy evaluation and improvement.

The Bellman equation for the *state-value function* $v_\pi(s)$ is:

$$v_\pi(s) = \sum_{a \in \mathcal{A}} \pi(a \mid s) \sum_{s' \in \mathcal{S}} P(s' \mid s, a) \left[R(s, a, s') + \gamma v_\pi(s') \right], \tag{2.18}$$

where $P(s' \mid s, a)$ is the transition probability from state s to s' given action a, and $R(s, a, s')$ is the expected reward. The Bellman equation for the *action-value function* $q_\pi(s, a)$ is:

$$q_\pi(s, a) = \sum_{s' \in \mathcal{S}} P(s' \mid s, a) \left[R(s, a, s') + \gamma \sum_{a' \in \mathcal{A}} \pi(a' \mid s') q_\pi(s', a') \right]. \tag{2.19}$$

This equation describes the expected return of selecting action a in state s, considering both immediate rewards and the expected future return from subsequent actions.

For optimal policies, the Bellman optimality equations provide a way to compute the best possible value functions. The *optimal state-value function* $v_*(s)$:

$$v_*(s) = \max_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}} P(s' \mid s, a) \left[R(s, a, s') + \gamma v_*(s') \right]. \tag{2.20}$$

This equation determines the best possible value for a state by selecting the action that maximizes the expected return.

Similarly, the *optimal action-value function* $q_*(s, a)$:

$$q_*(s, a) = \sum_{s' \in \mathcal{S}} P(s' \mid s, a) \left[R(s, a, s') + \gamma \max_{a' \in \mathcal{A}} q_*(s', a') \right]. \tag{2.21}$$

Here, the agent maximizes over possible actions in future states, allowing for optimal decision-making.

These equations form the basis for value iteration and policy iteration, key techniques used in reinforcement learning to compute optimal policies by iteratively updating value estimates.

2.3.2 Deep Reinforcement Learning

Deep Reinforcement Learning (DRL) extends traditional reinforcement learning by integrating deep neural networks as function approximators. This allows DRL to handle environments with high-dimensional state and action spaces that are computationally infeasible for classical RL algorithms. Neural networks approximate key functions in RL, such as value functions, policies, and transition models, enabling the agent to generalize effectively across complex state spaces.

The use of neural networks for function approximation is a cornerstone of DRL, addressing challenges associated with representing and learning from high-dimensional data. In traditional reinforcement learning, functions such as $v_\pi(s)$, $q_\pi(s, a)$, or $\pi(a|s)$ are often tabulated or expressed using simple parametric models. However, these approaches become impractical when the state or action space is large or continuous. Neural networks overcome these limitations by providing scalable representations of complex functions, capturing nonlinear relationships between inputs and outputs, generalizing to unseen states through learned data representations, and enabling end-to-end optimization with gradient-based algorithms, such as stochastic gradient descent.

RL methods, including DRL, are generally categorized into two main types: model-free and model-based methods. These categories differ based on whether the approach involves learning a model of the environment's dynamics or directly interacting with the environment to optimize behavior. **Model-free methods** rely solely on direct interactions with the environment to learn optimal policies or value functions. They do not explicitly construct or utilize a model of the environment's transition dynamics or reward function. Model-free methods are particularly effective in environments where the dynamics are unknown, complex, or computationally expensive to model. **Model-based methods** aim to explicitly learn a model of the environment, which typically consists of two key components: the transition dynamics, which describe the probability of transitioning to a new state given the current state and action; and the reward function, which predicts the reward received when taking action in state. These learned models are leveraged for two primary purposes: planning, where simulated trajectories are used to evaluate and optimize actions without directly interacting with the environment; and policy learning, where synthetic data generated from the model augments real-world experiences, improving sample efficiency.

While model-based methods are powerful due to their sample efficiency and planning capabilities, their applicability is often constrained by the need for an accurate environment model. In HPC scheduling, the system dynamics are complex and influenced by unpredictable job arrivals, varying resource demands, and heterogeneous hardware configurations, making it difficult to construct a precise and computationally efficient model. Given these challenges, model-free methods are more suitable, as they do not require explicit modeling of the environment and can instead learn directly from interactions with the system. This flexibility allows model-free RL to generalize across different workload conditions and adapt to dynamic scheduling scenarios. Based on the function approximation objectives, model-free methods can be categorized into value-based approaches and policy gradient approaches, which will be discussed in the following sections.

2.3.2.1 Value-based DRL

Value-based DRL focuses on approximating value functions, which estimate the expected cumulative rewards associated with states or state-action pairs. These value functions form the foundation for deriving optimal policies, where the agent selects actions that maximize the value function. To understand how value-based methods operate, it is important to first examine how value functions are estimated, particularly through Monte Carlo and Temporal Difference methods.

In reinforcement learning, value functions such as the state-value function $v_\pi(s)$ and the action-value function $q_\pi(s, a)$ are used to estimate the expected return. However, the challenge lies in how these estimates are computed and updated. Monte Carlo and Temporal Difference (TD) serve as the foundation for approximating these functions.

Monte Carlo methods estimate the value of states or actions by averaging the returns observed from complete episodes. For example, the action-value function can be updated as:

$$q_\pi(s, a) \approx \mathbb{E}_\pi \left[G_t \mid S_t = s, A_t = a \right], \tag{2.22}$$

where $G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$ is the total return. While Monte Carlo methods are simple and intuitive, they require the agent to wait until the end of an episode to calculate returns. This makes them computationally expensive and impractical for environments with long episodes or where intermediate feedback is essential.

TD methods, in contrast, update value estimates incrementally using bootstrapping. Instead of waiting for the entire episode to complete, TD methods use an estimate of the value of the next state to update the current value. The TD update for the action-value function is:

$$q_\pi(s,a) \leftarrow q_\pi(s,a) + \alpha \left[r + \gamma q_\pi(s',a') - q_\pi(s,a) \right], \tag{2.23}$$

where α is the learning rate, r is the immediate reward, and $q_\pi(s',a')$ is the estimated value of the next state-action pair. This approach is more sample-efficient and updates values after each step, making it particularly useful in dynamic environments. Value-based DRL methods adopt the principles of TD learning. By using a neural network to approximate the Q-function, these methods extend TD ideas to high-dimensional state and action spaces, enabling the agent to make decisions in more complex environments.

Deep Q-Networks (DQN) [45] are a seminal value-based DRL algorithm that applies neural networks to approximate the Q-function. In DQN, a neural network parameterized by θ predicts the Q-values $q_\pi(s,a)$, Equation 2.16, for all possible actions in a given state. The goal is to iteratively improve the neural network's parameters θ so that the predicted Q-values closely align with the true Q-values as defined by the Bellman equation (Equation 2.19). DQN minimizes the loss function based on the temporal difference (TD) error with the equation:

$$\mathcal{L}(\theta) = \mathbb{E}_{(s,a,r,s')\sim\mathcal{D}} \left[(y - Q(s,a;\theta))^2 \right], \tag{2.24}$$

$$\mathcal{L}(\theta) = \mathbb{E}_{(s,a,r,s')\sim\mathcal{D}} \left[\left(r + \gamma \max_{a'} Q_{\text{target}}(s',a';\theta^-) - Q(s,a;\theta) \right)^2 \right]. \tag{2.25}$$

$Q(s,a;\theta)$ represents the predicted Q-value from the current network. The current network is the primary network being trained. $Q_{\text{target}}(s',a';\theta^-)$ represents the Q-value computed using a separate target network with parameters θ^-. The target network is a separate copy of the current network that is used to compute the target Q-value during training. The target network typically copies the parameters of the current network every fixed number of steps. The use of a target network addresses a key challenge in Q-learning: instability during training. This instability arises because the target value $y = r + \gamma \max_{a'} Q(s',a';\theta)$ depends on the predictions of the current network, which are constantly changing as the network is updated. This creates a *moving target* problem, where the network is attempting to predict values based on its own rapidly fluctuating predictions. Such instability can

lead to divergence or oscillations in the learning process. The target network provides a stable reference for computing the target value y by keeping its parameters (θ^-) fixed for a certain number of updates. After this period, the parameters of the target network are updated to match the current network.

DQN demonstrated its effectiveness by achieving human-level performance on a wide range of Atari games, directly from raw pixel input. This marked a significant milestone in reinforcement learning, showing that deep neural networks can successfully approximate value functions in high-dimensional state spaces. DQN's innovations, such as experience replay and target networks, became foundational components of many subsequent DRL algorithms. Despite its success, DQN has several limitations: the use of $\max_{a'} Q(s', a')$ in the target calculation can lead to overestimation of Q-values; DQN is not well-suited for environments with continuous action spaces, as it requires discretization.

In addition to foundational value-based approaches like DQN, several advancements have been proposed to address its limitations and improve performance. Double DQN [46] mitigates overestimation bias by decoupling action selection and evaluation, while Dueling DQN [47] separates the estimation of state-value and action-advantage functions to enhance learning efficiency. Rainbow DQN [48] integrates multiple improvements, including Double DQN, prioritized experience replay, and distributional RL, into a unified framework. Average-DQN [49] improves stability and performance by maintaining an average of past target network parameters instead of a single target network. More recent works include Never Give Up (NGU) [50], which focuses on directed exploration strategies, and Agent57 [51], the first agent to surpass human performance on all Atari 57 games by combining multiple exploration and learning mechanisms. Lastly, Human-level Atari 200x Faster [52] introduces techniques for dramatically improving the efficiency of DQN-based approaches, enabling human-level performance in Atari games with significantly fewer resources.

2.3.2.2 Policy Gradient Methods

Policy gradient methods are a class of reinforcement learning algorithms that directly optimize the policy, parameterized as $\pi_\theta(a|s)$, where θ represents the parameters of the neural network modeling the policy. Unlike value-based methods, which derive policies

indirectly by learning value functions, policy gradient methods focus on learning and improving the policy itself. This direct optimization makes policy gradient methods particularly effective in environments with continuous or high-dimensional action spaces, where value-based approaches often struggle.

The primary goal of policy gradient methods is to find the optimal policy $\pi_\theta(a|s)$ that maximizes the expected cumulative reward. Formally, the objective is defined as:

$$J(\theta) = \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{\infty}\gamma^t r(s_t, a_t)\right], \tag{2.26}$$

where $\tau = (s_0, a_0, r_0, s_1, \dots)$ is a trajectory sampled from the policy π_θ; $\gamma \in [0, 1]$ is the discount factor that balances the importance of immediate and future rewards; $r(s_t, a_t)$ is the reward received after taking action a_t in state s_t. The agent learns to adjust the policy parameters θ to maximize $J(\theta)$, which represents the expected cumulative reward over all possible trajectories.

The key challenge in policy optimization is determining how to adjust θ in a way that improves $J(\theta)$. **Policy Gradient Theorem** provides a solution by expressing the gradient of $J(\theta)$ as:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{\infty}\nabla_\theta \log \pi_\theta(a_t|s_t) G_t\right], \tag{2.27}$$

where $G_t = \sum_{k=0}^{\infty}\gamma^k r_{t+k+1}$ is the return, representing the cumulative discounted reward starting from time step t. This equation shows that the gradient of $J(\theta)$ depends on two components. $\nabla_\theta \log \pi_\theta(a_t|s_t)$, this term encourages the policy to increase the probability of actions that lead to higher rewards. G_t, this term quantifies the total reward received after taking action a_t in state s_t. By iteratively adjusting θ in the direction of $\nabla_\theta J(\theta)$, the policy improves over time.

The REINFORCE algorithm [43] is one of the simplest implementations of policy gradient methods. It follows the mathematical foundation described above and uses sampled trajectories to estimate the gradient of $J(\theta)$. The parameters θ are updated using stochastic gradient ascent:

$$\theta \leftarrow \theta + \alpha\nabla_\theta J(\theta), \tag{2.28}$$

where α is the learning rate. The gradient is estimated using the following formula:

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \sum_{t=0}^{\infty} \nabla_\theta \log \pi_\theta(a_t|s_t) G_t, \tag{2.29}$$

where N is the number of trajectories sampled. Intuitively, this update rule increases the probability of actions that result in higher returns.

However, REINFORCE suffers from high variance in its gradient estimates, which can lead to slow and unstable training. Additionally, it does not make use of baseline functions (such as value functions) to reduce variance, which motivates the development of more advanced methods like actor-critic.

Actor-critic [53] methods. The *actor* is responsible for parameterizing and optimizing the policy $\pi_\theta(a|s)$, which selects actions based on the current state. The *critic*, on the other hand, approximates a value function (e.g., the state-value function $V(s)$ or the action-value function $Q(s, a)$) to provide feedback to the actor. This feedback is used to evaluate the quality of the actor's decisions and guide policy updates. The reason for introducing a critic is to reduce the high variance inherent in policy gradient estimates. In basic policy gradient methods like REINFORCE, the return G_t is used directly to compute the policy gradient ($G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}$). Since G_t represents the cumulative reward over a trajectory, it is highly variable, especially in environments with sparse or delayed rewards. This variability leads to noisy gradient estimates, which result in unstable and slow convergence. By introducing a critic, the high variance of G_t is mitigated. The critic approximates a baseline value function $V(s_t)$ or $Q(s_t, a_t)$, which provides a less noisy estimate of expected returns. This baseline helps isolate the improvement due to a specific action, making updates more focused and stable.

The critic is used to compute the **Advantage Function**, which measures how much better or worse an action a_t is compared to the expected value at state s_t. The Advantage Function is defined as:

$$A(s_t, a_t) = Q(s_t, a_t) - V(s_t), \tag{2.30}$$

where $Q(s_t, a_t)$ represents the expected return for taking action a_t in state s_t; $V(s_t)$ represents the expected return for being in state s_t.

Using the advantage function, the policy gradient theorem becomes:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau\sim\pi_\theta}\left[\nabla_\theta \log \pi_\theta(a_t|s_t) A(s_t, a_t)\right]. \tag{2.31}$$

In the advantage function, the value function $V(s_t)$ serves as a baseline, removing the average expected return from the cumulative reward. This reduces variance because only the improvement over the baseline (i.e., the advantage) is used for policy updates. For instance, if an action is no better or worse than expected, its contribution to the gradient will be zero. Instead of waiting for the complete return G_t to be observed, the critic uses bootstrapping to estimate the return. For example, $Q(s_t, a_t)$ can be computed as:

$$Q(s_t, a_t) \approx r_t + \gamma V(s_{t+1}), \tag{2.32}$$

where $V(s_{t+1})$ is the critic's estimate of the value of the next state. This allows the agent to update its policy after every step, rather than waiting for the end of an episode. By using the advantage function, actor-critic methods reduce variance in policy updates and improve training stability. To further reduce variance and improve training stability, Generalized Advantage Estimation (GAE) [54] is often used to compute smoother and more robust advantage estimates.

However, in basic on-policy actor-critic methods, instability can arise if the actor and critic update at significantly different rates. For instance, if the policy (actor) changes much faster than the value function (critic), the critic's value estimates may no longer align with the updated policy. This mismatch can result in unstable training, where the policy oscillates or fails to converge effectively. To address this, Trust Region Policy Optimization (TRPO) [55] introduces a constraint on the policy updates to prevent the actor's policy from changing too much between updates. TRPO ensures that the updated policy π_θ stays close to the old policy $\pi_{\theta_{\text{old}}}$ by using a constraint based on the Kullback-Leibler (KL) divergence $D_{\text{KL}}(\pi_{\theta_{\text{old}}} \,||\, \pi_\theta) \leq \delta$, where δ is a small positive hyperparameter. By limiting how far the policy can move in distribution space, TRPO stabilizes the updates and ensures monotonic improvement in the policy's performance. The optimization problem is formulated as:

$$\text{maximize } \mathbb{E}_{s,a\sim\pi_{\theta_{\text{old}}}}\left[\frac{\pi_\theta(a|s)}{\pi_{\theta_{\text{old}}}(a|s)} A_{\pi_{\theta_{\text{old}}}}(s,a)\right], \quad \text{subject to } D_{\text{KL}}(\pi_{\theta_{\text{old}}} \,||\, \pi_\theta) \leq \delta. \tag{2.33}$$

$\pi_{\theta_{\text{old}}}$ is the policy used to collect trajectories; π_θ is the updated policy; $A_{\pi_{\theta_{\text{old}}}}(s,a)$ is the advantage function, indicating how much better an action a is compared to the baseline value $V(s)$.

While TRPO effectively stabilizes training, it is computationally expensive because it involves calculating second-order derivatives (via the Fisher Information Matrix) to solve a constrained optimization problem. This makes TRPO challenging to apply in practice, especially for large-scale problems. To simplify this process, **Proximal Policy Optimization (PPO)** [40] introduces a more efficient approach to achieving the same goal. Instead of explicitly enforcing a trust region constraint using second-order optimization, PPO uses a clipped surrogate objective function to limit policy updates indirectly. The PPO objective is defined as:

$$L^{\text{CLIP}}(\theta) = \mathbb{E}_{s,a}\left[\min\left(r_t(\theta)A_{\pi_{\text{old}}}(s,a), \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)A_{\pi_{\text{old}}}(s,a)\right)\right], \tag{2.34}$$

where: $r_t(\theta) = \frac{\pi_\theta(a|s)}{\pi_{\theta_{\text{old}}}(a|s)}$ is the probability ratio between the new and old policies; ϵ is a hyperparameter defining the allowable range of policy updates.

The clipping mechanism ensures that the probability ratio $r_t(\theta)$ stays within the range $[1-\epsilon, 1+\epsilon]$, preventing excessively large updates to the policy. This approach avoids the computational overhead of TRPO while still maintaining stable and efficient training. By striking a balance between stability and computational efficiency, PPO has become one of the most widely used reinforcement learning algorithms. It has demonstrated strong performance across a variety of tasks, including continuous control and complex environments.

Beyond PPO, several other methods have been developed to address specific challenges. Soft Actor-Critic (SAC) [41] introduces entropy regularization into policy optimization, encouraging the agent to explore more robustly while maintaining sample efficiency, making it particularly effective in stochastic environments. Deep Deterministic Policy Gradient (DDPG) [39] extends DPG to high-dimensional continuous action spaces by combining deterministic policy gradients with experience replay, but suffers from issues like overestimation bias, which are addressed in Twin Delayed Deep Deterministic Policy Gradient (TD3) [56]. Hindsight Experience Replay (HER) [57] tackles sparse reward settings by allowing agents to relabel failed trajectories as successful by redefining

the goal, enabling more effective learning. Q-Prop [58] improves the stability and sample efficiency of policy gradient updates by incorporating value function estimates into the gradient. Similarly, Normalized Advantage Function (NAF) [59] simplifies continuous action learning by parameterizing the advantage function quadratically, reducing computational complexity. Stochastic Value Gradient (SVG) [60] leverages value gradients to update policies, enabling effective training in environments with complex dynamics.

2.3.3 Hierarchical Reinforcement learning

RL suffers from serious scaling issues when applied to complex, long-horizon tasks. Traditional RL methods often struggle in environments where decisions need to be made at multiple levels of abstraction, particularly when solving high-dimensional tasks or when rewards are sparse and delayed. Standard RL approaches require extensive exploration of the entire state-action space, making learning prohibitively slow and inefficient for large-scale problems. Hierarchical Reinforcement Learning (HRL) introduces a structured approach to decision-making by decomposing tasks into multiple levels of abstraction. Instead of directly mapping states to actions, HRL introduces high-level policies that set abstract goals or subgoals, which are then executed by low-level policies. This decomposition allows for better credit assignment, improved sample efficiency, and faster learning in complex environments [61].

MDP planning and learning algorithms can be naturally extended to accommodate hierarchical structures. In HRL, the traditional transition probability $p(s'|s,a)$ in standard MDPs is generalized to a semi-Markov decision process (SMDP), where transitions may occur over variable time steps τ. The new transition probability in an SMDP is defined as $p(s',\tau|s,a)$, where τ represents the number of steps taken to transition from state s to state s' when executing action a. This formulation allows HRL to operate at different temporal resolutions, enabling the learning of temporally extended actions or options.

Figure 2.2 illustrates an HRL process, demonstrating how a high-level policy decomposes tasks into intermediate subgoals, which are then executed through low-level primitive actions.

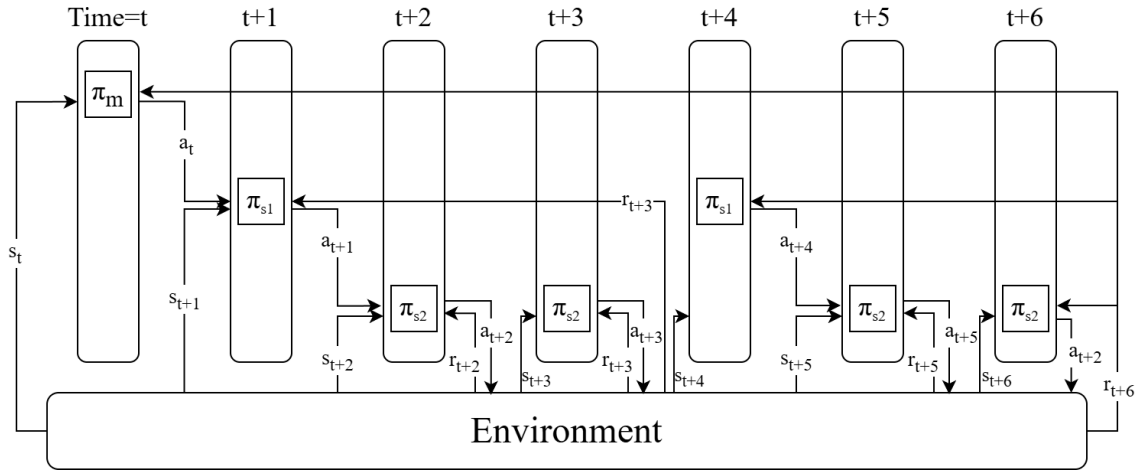

FIGURE 2.2: The temporal process of Hierarchical Reinforcement Learning.

- **High-Level Policy (π_m) – Manager:** Receives the current state s_t from the environment and selects a high-level subgoal a_t, which remains unchanged for multiple timesteps.
- **Intermediate-Level Policy (π_{s1}) – Subgoal Generator:** Receives the subgoal from π_m and generates intermediate subgoals $a_{t+1}, a_{t+4}, \ldots$ that are passed to the low-level policy.
- **Low-Level Policy (π_{s2}) – Action Executor:** Executes primitive actions $a_{t+2}, a_{t+3}, \ldots$ based on the subgoal from π_{s1}, interacting directly with the environment.
- **Environment Interaction:** The environment receives the executed actions and returns new states and rewards. Both the high-level and low-level policies receive feedback in the form of state transitions and rewards, though at different temporal scales.
- **Hierarchical Temporal Structure:** As shown in the diagram, the high-level policy makes decisions less frequently, setting goals for the low-level policies, which operate at finer time scales to execute the detailed steps required to achieve these goals.

Consider the following example. In a warehouse automation setting, a robot must retrieve an item from a shelf and transport it to a designated location, a task effectively handled using HRL. The **high-level policy** (π_m) acts as the task manager, processing the environment's state and generating a broad objective, such as *"Move to the item's location*

and pick it up." This goal is then passed to the **intermediate-level policy** (π_{s1}), which decomposes it into smaller, sequential subgoals, such as *"Navigate to the correct aisle"* followed by *"Move to the precise pickup position."* Each of these intermediate subgoals requires multiple fine-grained actions to complete. The **low-level policy** (π_{s2}) then executes primitive actions, such as controlling motor movements to adjust position, align with the target, and grasp the item. Throughout the process, the environment updates states and provides rewards, allowing each policy to adjust accordingly. Once the object is secured, π_m assigns a new high-level subgoal, such as *"Deliver the item to location C,"* and the cycle repeats, with π_{s1} and π_{s2} handling navigation and execution. This structured decision-making approach enables efficient learning, modular subpolicy reuse, and scalable control, making HRL well-suited for complex, long-horizon tasks in robotics and real-world automation.

2.3.3.1 Foundational Methods in HRL

HRL builds on several foundational methods that introduced key principles such as temporal abstraction, hierarchical credit assignment, structured decision-making, and task decomposition. These methods include Feudal Learning, the Options Framework, Hierarchical Abstract Machines, and MAXQ Decomposition, each offering a unique perspective on hierarchical decision-making.

Feudal Learning, proposed by Dayan and Hinton [62], introduces a manager-worker hierarchy, where a high-level manager assigns goals, and a low-level worker learns how to execute them. Unlike standard reinforcement learning, where actions are selected directly based on rewards, Feudal Learning disentangles strategic planning from execution, allowing higher-level decisions to remain abstract while lower-level policies handle execution details.

A key feature of Feudal Learning is its hierarchical credit assignment, where the manager is only rewarded based on long-term goal achievement, while the worker is rewarded for executing the subgoals set by the manager. This decentralized reward structure enables better exploration and structured learning, particularly in environments with delayed rewards and long-term dependencies. Additionally, by allowing managers to set goals in latent spaces rather than direct action spaces, Feudal Learning improves scalability in complex decision-making tasks.

Options Framework, introduced by Sutton et al. [63], formalizes the concept of temporally extended actions by defining options as macro-actions that consist of a policy, an initiation set, and a termination condition. An option is an abstract decision-making unit that spans multiple time steps, making it an abstraction over primitive actions. The initiation set specifies the states where an option can be executed, while the termination function defines when an option ends.

Unlike Feudal Learning, where hierarchical control emerges from goal setting, the Options Framework provides a structured way to define and execute sub-policies that operate at different time scales. By explicitly modeling when and how sub-policies should be activated, options allow reinforcement learning agents to improve sample efficiency, exploration, and task reusability. The introduction of options also extends Markov Decision Processes (MDPs) to Semi-Markov Decision Processes (SMDPs), enabling agents to handle complex decision-making over varying time horizons.

Hierarchical Abstract Machines (HAM) [64] introduce a structured approach to policy representation using finite state machines. Unlike Feudal Learning and the Options Framework, which focus on subgoal discovery and execution, HAM explicitly structures decision-making into a predefined hierarchy of abstract states, where each state corresponds to a sub-policy.

HAMs constrain exploration by enforcing hierarchical constraints on policy execution, allowing for structured decision-making in environments with well-defined substructures. This makes HAM particularly useful in domains where expert knowledge or domain-specific rules can inform the design of hierarchical policies. The ability to combine hand-crafted rules with learned sub-policies makes HAM a flexible framework for structured RL problems.

MAXQ Decomposition framework, introduced by Dietterich [65], proposes a method to decompose a complex MDP into a hierarchy of subtasks. Each subtask has its own value function, which contributes to the overall task, allowing hierarchical policies to be learned in a structured manner. Unlike the Options Framework and Feudal Learning, MAXQ explicitly models the value function decomposition, ensuring that each subtask is optimized independently while still contributing to the overall objective.

A key benefit of MAXQ is its ability to enable modular learning, where different subtasks can be trained separately and later combined to solve complex problems. This decomposition leads to faster convergence and improved generalization, as subtasks can be reused across different problems. Unlike HAM, which imposes predefined policy structures, MAXQ provides a flexible framework that allows the learning-based decomposition of tasks without requiring prior domain knowledge.

2.3.3.2 Recent Advances in HRL

Beyond the foundational HRL methods, recent research has introduced new frameworks to improve sample efficiency, goal abstraction, temporal coherence, and hierarchical credit assignment. This section highlights several key advancements in HRL.

FeUdal Networks (FuN) [66] build upon Feudal RL by introducing a latent state space for goal representation. Instead of setting subgoals in raw state space, FuN encodes goals in a latent space, allowing hierarchical policies to learn abstract and transferable representations. The manager selects a direction in this latent space, while the worker learns to achieve that direction through primitive actions. FuN enhances long-term credit assignment by decoupling high-level strategic planning from low-level execution and allows agents to develop meaningful behavioral primitives that generalize across tasks.

HIRO [67] improves sample efficiency in HRL by introducing hindsight goal relabeling and eliminating the need for goal representation learning. Unlike previous approaches that define abstract goal spaces, HIRO uses the raw state observations as goals, simplifying the training process and improving stability. The hierarchical policy consists of a high-level goal generator and a low-level policy that learns to achieve goals within the environment. HIRO's off-policy correction mechanism ensures that past experience remains useful, making it particularly effective in long-horizon continuous control tasks.

Hierarchical Actor-Critic (HAC) [68] extends the hierarchical structure of HIRO by employing an actor-critic mechanism at multiple levels. HAC is specifically designed to handle sparse reward environments by using hindsight experience replay (HER) across different hierarchy levels. Unlike HIRO, which does not explicitly incorporate hindsight in multi-level learning, HAC allows each level to independently relabel subgoal failures

as successful learning opportunities. This enhances sample efficiency and accelerates the training of hierarchical policies.

Hierarchical Deep Q-Networks (h-DQN) [69] introduce a two-level Q-learning framework that integrates hierarchical goal selection with reinforcement learning. The higher-level policy learns over intrinsic goals, guiding the lower-level policy, which executes primitive actions to achieve them. This architecture improves exploration in sparse reward environments by allowing the agent to autonomously define subgoals. h-DQN has been particularly effective in discrete-action domains like *Montezuma's Revenge*, where hierarchical goal-setting improves sample efficiency.

Meta Learning Shared Hierarchies (MLSH) [70] introduces a meta-learning framework for training hierarchical policies that generalize across multiple tasks. The agent alternates between meta-learning a high-level controller and training reusable sub-policies. By leveraging shared representations, MLSH enables faster adaptation to new environments without requiring manual task decomposition. Unlike traditional HRL methods that rely on predefined hierarchies, MLSH allows agents to dynamically discover useful sub-policies during training.

Strategic Attentive Writer STRAW [71] introduces a deep recurrent neural network that enables agents to commit to sequences of actions over extended time horizons. The key innovation is the ability to partition an internal representation into contiguous subsequences, allowing the agent to follow a plan until re-planning is necessary. STRAW facilitates structured exploration and long-term planning, leading to strong improvements in ATARI games like *Ms. Pacman* and *Frostbite* by learning temporally extended macro-actions.

Hierarchical Deep Reinforcement Learning with Network Distillation (H-DRLN) [72] proposes a lifelong learning framework that enables agents to retain and transfer knowledge across multiple tasks. The architecture incorporates a Deep Skill Network, where learned skills are distilled into a reusable knowledge base. This allows hierarchical agents to efficiently build upon previous experience rather than learning from scratch. Applied in Minecraft, H-DRLN demonstrated the ability to learn and reuse sub-policies effectively, highlighting its potential for long-term reinforcement learning.

Abstract Markov Decision Processes (AMDP) [73] introduces a hierarchical approach to decision-making by defining abstract MDPs, where states represent high-level subproblems rather than raw environmental observations. Unlike MAXQ, which propagates values bottom-up from primitive actions, AMDP models each subtask's transition and reward functions locally, allowing for top-down planning. This significantly improves the efficiency of multi-level reasoning by reducing unnecessary backup operations across hierarchy levels.

Iterative Hierarchical Optimization for Misspecified Problems (IMHOP) [74] is designed to address policy representation challenges in high-dimensional MDPs, particularly in cases where function approximation cannot express an optimal policy. IMHOP operates as a meta-algorithm that iteratively refines hierarchical policies, forcing different components of the hierarchy to specialize in different regions of the state space. This structure improves robustness in environments where standard HRL approaches struggle with inaccurate representations.

Learning Goal Embeddings via Self-Play for Hierarchical Reinforcement Learning (HSP) [75] introduces asymmetric self-play as a pre-training phase for hierarchical agents. During self-play, an agent sets its own tasks via goal embeddings and attempts to solve them before full hierarchical training. This method improves exploration in environments with sparse rewards by ensuring the hierarchical policy is trained on meaningful subgoals. In AntGather, HSP demonstrated improved performance by learning to generate goal-directed behaviors without explicit supervision.

Learning Representations in Model-Free Hierarchical Reinforcement Learning [76] explores the role of representation learning in hierarchical RL by focusing on how different hierarchical levels extract useful subtask features. Instead of relying on predefined state features, the approach learns hierarchical feature representations that generalize across tasks. By structuring the learning process hierarchically, agents can develop robust sub-policies that improve long-term adaptation.

In summary, recent advancements in HRL have introduced more efficient goal abstraction, improved sample efficiency, and structured hierarchical learning. Methods like FuN, HIRO, HAC, h-DQN, and MLSH focus on learning hierarchical subgoals, while

techniques like STRAW, MPH, and AMDP emphasize macro-action planning and structured representations. These advancements continue to push HRL towards more scalable, generalizable, and data-efficient solutions for complex decision-making tasks.

2.3.4 Transfer Learning in Reinforcement Learning

RL algorithms typically require substantial interactions with the environment to learn optimal policies, making them computationally expensive and sample inefficient. **Transfer Learning (TL)** aims to alleviate these challenges by leveraging prior knowledge from one task to accelerate learning in new, related tasks. TL in RL can be categorized based on what type of knowledge is transferred from a source task to a target task. The primary types include instance-based transfer, feature-based transfer, policy transfer, value function transfer, model transfer, and reward shaping. Each type has distinct advantages and limitations, making them suitable for different problem settings [77–79].

Instance-based Transfer focuses on reusing experience data collected from a previously learned task to improve learning efficiency in a new task. In reinforcement learning, past experience is typically stored in replay buffers and used for training, even in tasks where direct policy transfer is not feasible. This approach is particularly effective in off-policy [38] learning methods such as Q-learning and actor-critic methods [45].

A common implementation of instance-based transfer is experience replay, which enables agents to store and sample past transitions to stabilize learning [80]. By replaying stored experiences, the agent reduces the risk of catastrophic forgetting and improves sample efficiency. Prioritized experience replay extends this concept by assigning higher importance to experiences with higher learning potential, ensuring that informative transitions are replayed more frequently [81]. Another instance-based approach is successor representation learning, where an agent encodes task-agnostic state representations that allow knowledge to be efficiently transferred between similar tasks [82]. Successor features provide a structured way to generalize learned representations across different reward functions, improving the adaptability of value-based methods. The primary challenge in instance-based transfer is dataset bias. Stored experiences may not align well with the new task, leading to inefficient learning. Additionally, excessive reliance on replay buffers may slow down adaptation to novel scenarios, especially in dynamic environments.

Feature-based Transfer aims to learn shared representations that capture common structures across different tasks. Instead of transferring raw data or policies, the focus is on extracting meaningful state features that remain useful in a variety of environments. This method is particularly valuable when tasks share underlying dynamics but differ in state distributions or reward functions [83].

One approach to feature-based transfer is unsupervised pre-train, where a neural network is initially trained on an auxiliary task before being fine-tuned for the target task [79]. Techniques such as autoencoders and self-supervised learning are commonly used to extract informative state representations that can later be applied to reinforcement learning [84]. Multi-task learning is another approach where an agent jointly trains across multiple tasks, allowing it to develop representations that generalize well to unseen environments [85].

Reward shaping is a specific form of transfer learning where prior knowledge is encoded into the reward function to guide the learning process [86]. This is particularly useful in sparse-reward environments, where an additional shaping function $F(s, a, s')$ modifies the original reward:

$$R'(s, a, s') = R(s, a, s') + F(s, a, s') \tag{2.35}$$

For example, in intrinsic motivation-based reinforcement learning, shaping rewards can be derived from curiosity-driven exploration [87] or predictive models of environmental dynamics [88]. The main drawback of reward shaping is the risk of reward exploitation, where agents optimize for the auxiliary reward rather than solving the intended task.

Policy Transfer involves reusing a previously trained policy to accelerate learning in a new environment. This approach is particularly effective when tasks share similar state-action spaces and reward structures, allowing the agent to adapt its behavior without relearning from scratch. Policy transfer can be implemented through imitation learning, behavioral cloning, or direct fine-tuning of a pre-trained policy [77].

One method of policy transfer is teacher-student learning, where a new policy is trained using demonstrations from an existing policy [89]. This can be achieved using supervised learning techniques such as policy distillation, where the student network learns to mimic the teacher's actions while optimizing for the target task [90]. Another strategy

is zero-shot transfer, where an agent directly applies its pre-trained policy in an unseen environment without additional training [91]. Despite its advantages, policy transfer is sensitive to task mismatch. Differences in task dynamics or action spaces can cause the transferred policy to perform poorly, requiring additional adaptation mechanisms such as reward shaping or fine-tuning [92].

Value Function Transfer leverages previously learned value approximations to bootstrap training in a new task. Instead of transferring entire policies, this method focuses on reusing value estimates to guide learning in the target environment. A key technique in value function transfer is successor features, where an agent learns value representations that generalize across multiple reward functions. This enables faster adaptation when transitioning to a new environment with a different reward structure [82]. By decoupling state representations from task-specific rewards, successor features improve transfer efficiency in reinforcement learning.

A major limitation of value function transfer is its dependency on reward distributions. Since value estimates are directly influenced by the rewards observed in the source task, transferring them to an environment with significantly different rewards may require additional adjustments [45].

Model Transfer focuses on reusing learned environment dynamics to improve sample efficiency in a new task. This approach is particularly beneficial in model-based reinforcement learning, where agents develop predictive models of state transitions and reward functions.

One example of model transfer is learning reusable world models, where the agent constructs a transition model that can be applied across multiple tasks. This is commonly used in robotics, where a model trained in simulation is later transferred to a real-world robot with minimal fine-tuning [93]. Another strategy is domain adaptation, where the agent adjusts its learned dynamics model to account for differences between the source and target environments [94]. A significant challenge in model transfer is ensuring that the learned dynamics remain valid in the target task. Even small discrepancies between source and target environments can lead to model inaccuracies, reducing the effectiveness of transfer.

To conclude, the different types of transfer learning in reinforcement learning each address specific challenges in generalization and sample efficiency. Instance-based transfer reuses past experiences, feature-based transfer extracts shared representations, policy transfer adapts pre-trained policies, value function transfer leverages learned Q-values, model transfer applies predictive models, and reward shaping modifies incentives for learning. Selecting the appropriate transfer method should depend on the nature of the tasks, the availability of prior knowledge, and the degree of similarity between source and target environments [79].

2.4 Survey on HPC Scheduling

This section provides a comprehensive survey of HPC scheduling approaches, categorizing existing methodologies based on their design principles and optimization strategies. *Heuristic-Based Schedulers* rely on predefined rules to select jobs, prioritizing simplicity and low computational overhead. *Real-World HPC Schedulers* examine widely deployed systems such as SLURM, Maui, and Moab, analyzing how they implement job prioritization policies. *Meta-Heuristic-Based Schedulers* introduce optimization techniques such as genetic algorithms and simulated annealing to explore improved scheduling solutions beyond traditional heuristics. *Machine Learning-Improved Schedulers* enhance job selection by leveraging models trained on historical workload data. *Reinforcement Learning-Based Schedulers* further refine this approach by dynamically learning optimal scheduling policies through continuous interaction with the system. Finally, *Resource Allocation in HPC Scheduling* discusses techniques for effectively distributing computational resources across jobs, considering workload heterogeneity and system constraints.

2.4.1 Heuristic-Based Schedulers

Heuristic algorithms have been widely used to deal with scheduling problems [95]. Heuristic-based scheduling algorithms prioritize jobs based on predefined rules, making them computationally efficient and easy to implement. These algorithms assign each job a ranking score based on attributes such as submission time, execution time, or priority weights. The scheduler selects the job with the highest priority score at each decision step. While heuristic methods do not guarantee globally optimal solutions, they are widely used in

HPC environments due to their simplicity and effectiveness. This section introduces common heuristic scheduling policies, highlighting their ranking functions and implications.

First-Come, First-Served (FCFS) schedules jobs in the order they arrive, ensuring that the oldest job in the queue is always executed next. The ranking function for FCFS is given by:

$$\text{Priority}(J_j) = -w_j, \tag{2.36}$$

where w_j is the job submission time. The negative sign ensures that earlier-submitted jobs (lower w_j) receive higher priority. While FCFS is fair in the sense that jobs are scheduled in order of arrival, it suffers from the *convoy effect*, where short jobs can be delayed significantly behind long-running jobs.

Last-Come, First-Served (LCFS) prioritizes the most recently submitted job, opposite to FCFS. The ranking function is:

$$\text{Priority}(J_j) = w_j. \tag{2.37}$$

This scheduling policy is advantageous in scenarios where newly arrived jobs need immediate execution, but it can lead to starvation of older jobs if new jobs keep arriving continuously.

Shortest Job First (SJF) prioritizes jobs based on their estimated execution time, aiming to minimize overall waiting time. The ranking function is:

$$\text{Priority}(J_j) = -e_j, \tag{2.38}$$

where e_j is the estimated runtime of job J_j. By selecting the job with the smallest execution time, SJF reduces the average waiting time. However, it can lead to starvation of long jobs, as shorter jobs will always be preferred.

In contrast to SJF, **Longest Job First (LJF)** prioritizes jobs with the longest execution time:

$$\text{Priority}(J_j) = e_j. \tag{2.39}$$

LJF is rarely used in practice but can be beneficial in batch processing scenarios where large computational tasks need to be prioritized.

Highest Response Ratio Next (HRRN) is a dynamic scheduling heuristic that balances fairness between short and long jobs by considering both waiting time and execution time. The ranking function is:

$$\text{Priority}(J_j) = \frac{w_j + e_j}{e_j}. \tag{2.40}$$

This equation ensures that as a job waits longer in the queue, its priority increases, eventually overtaking shorter jobs. HRRN mitigates starvation issues found in SJF while still favoring shorter jobs under normal conditions.

Weighted Fair Queuing (WFQ) is a priority-based scheduling mechanism that assigns each job a weight ω_j, typically based on factors such as user-defined priorities, historical resource usage, or job type. The scheduling priority is computed as:

$$\text{Priority}(J_j) = \frac{\omega_j}{\sum_{k=1}^{n} \omega_k}. \tag{2.41}$$

This approach ensures that jobs receive a fair share of system resources proportional to their assigned weight. WFQ dynamically adjusts job priorities based on these weight assignments, allowing for flexible and controlled resource distribution. However, the effectiveness of WFQ depends on the accuracy of weight assignments. Improperly assigned weights can lead to resource monopolization by certain jobs or users, reducing overall system fairness. Additionally, in HPC environments where jobs request heterogeneous resources such as CPUs, memory, and GPUs, assigning a single weight value may not adequately capture the complexity of resource demands.

In practice, HPC scheduling systems often combine these heuristics with additional optimizations, such as backfilling [96], to achieve better performance across diverse workloads. **Backfilling** is an optimization technique designed to improve the utilization of computational resources by allowing smaller jobs to execute ahead of larger, higher-priority jobs, as long as they do not interfere with the expected start time of high-priority jobs. The fundamental condition for backfilling is that the *next job in the queue cannot be started immediately due to insufficient resources.* Backfilling aims to reduce the overall waiting time and improve system throughput without compromising the priority order of jobs. It was introduced to address the limitations of simple heuristic-based scheduling

algorithms like FCFS, which can lead to inefficient resource usage and increased waiting times for small jobs.

EASY Backfilling, the most commonly used backfilling approach, works by allowing smaller jobs to execute in idle slots, as long as they do not delay the start of higher-priority jobs. If a job with higher priority (i.e., the job with the earliest expected start time) is scheduled, a smaller job can run if it fits in the time window without delaying the larger job's execution. The general approach for EASY backfilling can be described as three steps. Firstly, identify the next job to be scheduled, denoted as J_{next}, from the queue based on priority. Then, if J_{next} cannot start immediately due to insufficient resources, search for smaller jobs J_j that fit within the idle resources. Next, schedule J_j only if it does not delay J_{next}'s expected start time. **Conservative Backfilling** follows the same principle as EASY backfilling but enforces a stricter constraint: *no job in the queue should be delayed by backfilling.* Unlike EASY backfilling, which may allow smaller jobs to run ahead of larger jobs, Conservative backfilling ensures that jobs only run if they do not delay others. This approach prioritizes fairness and prevents job starvation, but it may result in lower overall utilization. The following pseudocode illustrates the decision-making process of backfilling.

Algorithm 1 Backfilling Decision Process

Input: Job queue $\mathcal{Q}(t)$, system resources

Output: Scheduled job or deferred job

```
Identify the next job J_next in the queue  if J_next can start immediately then
    Schedule J_next
end
else
    foreach job J_j in Q(t) do
        if J_j fits in the available resources and does not delay J_next then
            Schedule J_j
        end
        else if Using Conservative Backfilling then
            if J_j does not delay any job in Q(t) then
                Schedule J_j
            end
        end
    end
end
```

Researchers have also focused on enhancing heuristic-based schedulers by addressing various critical aspects beyond simple job prioritization. Tang et al. [5] proposed a series of heuristic-based scheduling algorithms called **WFP and UNICEF**. These algorithms are seen as effective heuristic-based schedulers and are often used in the literature as benchmarks [16, 24, 27]. The scheduling process follows two parts, Utility-based Job Selection, and Fault-Aware Job Allocation. Utility-based Job Selection presents a series of utility functions to rank jobs' priorities, such as WFPs and UNICEF, based on the calculations of job waiting time, requested number of nodes, and requested time. The Fault-Aware Job Allocation calculates the allocation costs for the eligible jobs based on partition failure probability, the location of the partition, and the current allocation status of the machine. Its experiments validate the different utility functions with simulated Cobalt environment on Blue Gene/P trace data. While these heuristic-based schedulers are relatively simple to implement and understand, they may not always provide the most efficient solutions, particularly when dealing with complex and dynamic workloads.

Ghodsi et al. [6] introduce **Dominant Resource Fairness (DRF)**, a multi-resource fair allocation mechanism designed to ensure fairness in environments where jobs request different types of resources (e.g., CPU, memory, GPUs). Traditional fair-sharing approaches, such as max-min fairness, focus on equalizing a single resource type but fail to account for jobs with varying multi-resource demands. DRF extends the max-min fairness principle by allocating resources based on a job's dominant resource, which is the resource that accounts for the highest fraction of its requested allocation. This ensures that jobs with high demands for a particular resource do not starve others that require different resource combinations. Experimental evaluations demonstrate that DRF significantly improves multi-resource fairness while maintaining high cluster utilization.

Bridi et al. [7] present a constraint programming (CP)-based scheduling approach for heterogeneous HPC systems, addressing the challenges of efficiently managing diverse computing resources. Traditional heuristic-based schedulers often struggle with resource heterogeneity, leading to suboptimal job placements and resource underutilization. To overcome these limitations, the authors propose a CP-based scheduler that formulates job scheduling as a constraint satisfaction problem (CSP). Their method explicitly models hardware constraints, job dependencies, and execution requirements, allowing the scheduler to find globally optimized schedules that balance makespan, resource utilization, and energy efficiency.

Rodrigo et al. [8] propose a workflow-aware heuristic scheduling framework that optimizes HPC job execution with interdependencies. Their approach accounts for task dependencies and resource availability, using priority-based heuristics to schedule dependent tasks more efficiently. This results in improved makespan and system throughput.

Chadha et al. [97] extend SLURM with an adaptive scheduling mechanism that dynamically modifies scheduling decisions based on real-time system conditions. By monitoring resource availability, job priorities, and system workload levels, the scheduler adapts its heuristic-based policies, reducing job wait times and improving scheduling efficiency.

Carretero et al. [98] propose an I/O-aware job scheduling approach that integrates backfilling with priority-based heuristics to reduce I/O contention in HPC systems. Traditional backfilling focuses on CPU and memory constraints but ignores I/O bottlenecks, leading to performance degradation. Their scheduler dynamically considers I/O bandwidth utilization, ensuring efficient scheduling of I/O-intensive jobs.

Byun et al. [99] propose a node-based scheduling approach to optimize short-running jobs. Traditional per-task scheduling incurs high scheduling overhead for workloads with short execution times. Their approach allocates entire nodes to groups of short jobs, reducing scheduling latency and improving system throughput.

Nichols et al. [100] present a heuristic-based job scheduling algorithm that incorporates real-time resource utilization metrics to mitigate performance variability. Traditional schedulers often overlook resource contention, leading to degraded job performance. Their method proactively schedules jobs based on real-time CPU, memory, and I/O usage, ensuring minimal resource interference and improved runtime predictability.

2.4.2 Real-World HPC Schedulers

While heuristic-based scheduling policies provide simple and effective job selection strategies, real-world HPC workload managers integrate multiple scheduling techniques, including priority-based heuristics, backfilling, and fair-share scheduling, to improve system efficiency and fairness. We introduce commonly used workload managers and their scheduling algorithms, highlighting their implementation of scheduling heuristics in large-scale computing environments.

SLURM [35] is one of the most widely used HPC workload managers, known for its multi-priority scheduling combined with backfilling. Jobs are assigned a priority score, and the scheduler selects the highest-priority job that can be executed given the available resources. The priority function in SLURM is determined by a weighted combination of factors, expressed as:

$$\text{Priority}(J_j) = \text{Age}(J_j) + \text{Job Size}(J_j) + \text{Fair-Share}(J_j) + \text{QoS}(J_j). \tag{2.42}$$

The job age factor ensures that older jobs receive higher priority, while job size can be weighted to either favor large or small jobs based on system configuration. The fair-share component dynamically adjusts priorities based on previous resource consumption, ensuring equitable access across users. The quality-of-service setting allows administrators to assign priority levels based on job criticality. SLURM also implements backfilling using the EASY Backfilling approach, where smaller jobs are scheduled ahead of waiting jobs as long as they do not delay the highest-priority job.

PBS (Portable Batch System) and its derivative **Torque** [101] primarily use an FCFS queue for job scheduling, where jobs are selected based on their submission time. To improve resource utilization, PBS supports EASY Backfilling. Additionally, PBS allows priority-based scheduling, where job selection is determined by a configurable priority function incorporating factors such as job size, user-defined weights, and fair-share policies. While PBS/Torque is widely used for its simplicity and extensibility, its default FCFS-based scheduling is limited in efficiently handling diverse workloads and optimizing resource utilization. As a result, many deployments customize the scheduling policy by integrating priority-based ranking or external schedulers such as Maui [36] or Moab [102] to enhance fairness and efficiency.

Maui [36] Scheduler is an open-source, advanced job scheduler designed to enhance the capabilities of resource managers like PBS/Torque. It offers a comprehensive suite of features aimed at optimizing resource utilization and ensuring fair access across users. One of its key functionalities is fair-share scheduling, where job priorities are dynamically adjusted based on historical resource usage, preventing any single user or group from monopolizing resources. Additionally, Maui employs a sophisticated job prioritization mechanism that considers various factors such as job size, user-defined weights, and system objectives, allowing for a flexible prioritization strategy tailored to specific organizational needs. To improve system utilization, Maui implements both EASY and conservative backfilling strategies. Furthermore, the scheduler supports advanced preemption policies, enabling higher-priority jobs to interrupt lower-priority ones, ensuring that critical tasks receive the necessary resources promptly.

Building upon the foundation laid by Maui, **Moab** [102, 103] Scheduler offers enhanced features suitable for large-scale, complex computing environments. While retaining all functionalities of Maui, Moab introduces high availability through synchronized head nodes for immediate failover, ensuring continuous operation and minimizing downtime in case of failures. It integrates with provisioning managers to dynamically allocate and configure resources, adapting to changing workload demands and optimizing resource utilization. Furthermore, Moab supports workload management across multiple clusters, enabling unified policy enforcement and resource sharing in distributed computing environments.

Google's **Borg** [31, 32] is a large-scale cluster management system designed to optimize resource allocation across thousands of machines, handling both batch jobs and long-running services. Operating in highly dynamic environments, Borg employs job priority scheduling, resource overcommitment, and process-level isolation to efficiently manage workloads. Jobs are scheduled based on multiple factors, including workload type, resource requirements, and predefined priority levels, with higher-priority jobs capable of preempting lower-priority ones to ensure critical services maintain performance. The system follows a multi-resource optimization approach, aiming to maximize system efficiency while balancing job placement constraints, which can be expressed as:

$$\max \sum_{J_j} \text{Utility}(J_j, \text{Allocated Resources}), \tag{2.43}$$

where the utility function accounts for job importance, resource efficiency, and system constraints. To enhance resource utilization, Borg employs bin-packing heuristics that efficiently pack jobs while ensuring process isolation. Although Borg does not explicitly implement traditional backfilling heuristics, it leverages task evictions and preemptions to optimize scheduling decisions. Additionally, workload-aware scheduling ensures a balance between batch processing and interactive services, preventing long-running batch jobs from monopolizing system resources. By integrating these techniques into a multi-layered scheduling framework, Borg achieves high cluster utilization, low job latency, and scalable performance, making it a foundational system for large-scale distributed computing at Google.

2.4.3 Meta-Heuristic-Based Schedulers

Meta-heuristic-based schedulers leverage optimization techniques inspired by natural and evolutionary processes to improve scheduling efficiency in HPC environments. Unlike traditional heuristic-based methods, which rely on predefined rules, meta-heuristic approaches explore a broader solution space through stochastic search, enabling more effective handling of complex scheduling problems with multiple constraints and objectives.

Vasile et al. [12] introduce a hybrid scheduling algorithm that combines priority-based

heuristics with genetic algorithms (GA). Their method dynamically assigns jobs to resources based on real-time system constraints, improving task execution efficiency and system throughput. Results show that combining heuristics with evolutionary optimization significantly enhances job scheduling performance in heterogeneous HPC environments. Rekha and Dakshayini [13] adopted a GA to optimize the task allocation over cloud resources and achieved better makespan compared to a greedy algorithm and simple task placement strategy. The results demonstrated that the GA-based task allocation approach outperformed the other algorithms in terms of makespan, throughput, and resource utilization. The study concluded that the proposed method provides an efficient and effective way to allocate tasks in cloud computing environments, ultimately leading to improved overall system performance. Zhao [14] proposed an approach based on Particle Swarm Optimization (PSO) to maximize resource utilization and reduce the completion time of the jobs. The performance of the proposed cost-aware PSO-based scheduling algorithm is compared to other well-known scheduling methods, such as Min-Min and Max-Min algorithms, through simulation experiments. The results of the experiments demonstrate that the proposed cost-aware PSO-based scheduling algorithm outperforms other algorithms in terms of cost minimization while maintaining acceptable execution times and resource utilization levels. Li and Wu [15] proposed a modified algorithm based on Ant Colony Optimization (ACO) by adding a cost-aware fitness function to minimize the turnaround time and maximize resource utilization. The experiments show that their approach outperformed the standard ACO, and PSO algorithms. Meta-heuristic-based schedulers offer more flexibility than heuristic-based schedulers, as they can explore a wider range of solutions and adapt to complex problem domains. However, meta-heuristic-based schedulers tend to require more computational resources due to their iterative nature and the need to search for optimal solutions over a larger solution space. This increased complexity can also make them more challenging to implement and tune effectively, as they often involve multiple parameters and settings that must be adjusted to achieve the best performance.

Meta-heuristic-based schedulers utilize optimization algorithms such as GA, PSO, and ACO to explore large search spaces and improve scheduling efficiency. These methods are particularly effective for bag-of-tasks workloads, where all jobs are known in advance, allowing the scheduler to optimize resource allocation globally. By iterating through multiple candidate schedules and refining solutions over time, meta-heuristic approaches can often achieve near-optimal scheduling decisions. However, their applicability in highly

dynamic HPC environments is significantly limited due to inherent computational overhead and response time constraints. One of the primary drawbacks of meta-heuristic scheduling is its long optimization time. Unlike heuristic-based schedulers, which can make near-instantaneous scheduling decisions, meta-heuristic approaches require multiple iterations to converge to a good solution, making them impractical for real-time job scheduling. Additionally, these methods demand extensive parameter tuning, where hyperparameters such as mutation rates or cooling schedules must be carefully adjusted for optimal performance. In dynamic HPC systems, where job arrivals, execution times, and resource availability fluctuate continuously, such tuning is not only impractical but also ineffective in responding to changing conditions. While meta-heuristic methods may still be valuable for offline scheduling or periodic workload optimization, they are generally unsuitable for environments that require low-latency and adaptive scheduling strategies.

2.4.4 Machine Learning-Improved Schedulers

Machine learning-improved heuristic schedulers are scheduling approaches that combine traditional heuristic algorithms with machine learning techniques to optimize job scheduling in HPC systems. These schedulers leverage historical data and machine learning models to enhance the performance of heuristic-based algorithms, making better-informed decisions about job scheduling and resource allocation. One such example is the **F1-4**, proposed by Carastan et al. [16]. This work obtains 4 job priority functions used to rank the jobs' urgency which are calculated by adding the requested number of processors, the job's submission time, and the requested execution time with different weights. The weights are obtained by training nonlinear regression models to minimize the average bounded job slowdown. Then, they use both simulated and real workloads to evaluate the functions compared with commonly used heuristics in a homogeneous system.

Tanash et al. [17] explore the use of supervised machine learning techniques to predict job resource requirements in HPC systems. The primary goal is to improve resource utilization by making better-informed scheduling decisions. By training these models on historical data, they can accurately predict job resource requirements, such as CPU usage, memory usage, and I/O requirements, for incoming jobs. The results of the experiments demonstrate that the machine learning-based prediction method outperforms traditional

heuristics and other state-of-the-art techniques in terms of prediction accuracy. Furthermore, the study shows that incorporating these predictions into the scheduling process leads to improved system performance, better resource utilization, and reduced job waiting times.

Nemirovsky et al. [18] investigate the application of machine learning techniques to predict application performance and improve scheduling decisions on heterogeneous CPU architectures. Using predictions, the authors develop a scheduling algorithm that assigns applications to the most suitable processor in a heterogeneous computing environment. The scheduling algorithm aims to minimize the overall execution time of the applications while considering the predicted performance on different processors and the current workload of the system. The results of the experiments demonstrate that the machine learning-based approach achieves higher prediction accuracy and better scheduling decisions compared to traditional heuristics and other techniques.

Gaussier et al. [19] improve back-filling by using machine learning to predict running times. The authors propose a machine learning-based method to estimate job running times using historical job data and job characteristics. They incorporate these predictions into the back-filling scheduling algorithm, which aims to minimize job waiting times while considering the predicted running times. Comparing the performance of the machine learning-improved back-filling against traditional heuristics, the results show that the machine learning-based method leads to reduced job waiting times, and more efficient resource utilization. Machine learning-improved heuristic schedulers have some drawbacks, including limited adaptability due to reliance on historical data, inability to learn entirely new or optimal scheduling policies, potential overfitting to training data, and the need for manual feature engineering. These limitations can restrict their overall effectiveness in dynamic HPC environments and complicate the process of finding optimal scheduling solutions.

Despite their advantages, machine learning-improved schedulers face several limitations. First, they heavily depend on the quality and availability of historical data. In HPC environments with highly dynamic workloads, training data may become outdated quickly, leading to inaccurate predictions that degrade scheduling performance. Additionally, these methods require manual feature selection and model tuning, making them sensitive to workload variations and difficult to generalize across different HPC clusters.

Moreover, the computational overhead associated with training and updating machine learning models can be significant, particularly in large-scale systems with real-time scheduling constraints. Due to these challenges, learning-based scheduling approaches have increasingly shifted toward RL methods, which offer greater adaptability and autonomous policy learning.

2.4.5 Reinforcement Learning-Based Schedulers

RL-based schedulers represent a paradigm shift in HPC scheduling by formulating the scheduling problem as a sequential decision-making process. Unlike heuristic-based or supervised learning-improved approaches that rely on static rules or historical data, RL-based schedulers continuously learn from interactions with the system, adapting to changing workloads, resource availability, and scheduling constraints. By leveraging reward-driven optimization, these schedulers dynamically refine scheduling policies to improve overall system performance. Mao et al. [20] present the first HPC Resource Manager with DRL named **DeepRM**. The state of DeepRM is distinct images with the x-axis as the requested number of CPU and the requested memory, and the y-axis as the requested time for each resource. The actions are selecting a job for each step from the first M jobs in the backlog. The objective of DeepRM is to minimize the average slowdown of jobs. They use randomly generated jobs and action space as 10 to evaluate the scheduling on a small-scale cluster. The study highlights the potential of DRL-based methods to improve resource management in HPC.

Qin et al. [21] propose a region-based reinforcement learning (**RRL**) approach for machine learning model serving scheduling. Unlike traditional scheduling, their method partitions the scheduling space into regions and applies reinforcement learning within each region, enabling more efficient decision-making. By dynamically adjusting scheduling policies based on workload patterns, RRL improves resource utilization and reduces serving latency. Experiments demonstrate that the approach outperforms conventional heuristic-based scheduling, making it a promising solution for large-scale model serving in HPC environments.

CuSH is proposed by Domeniconi et al. [22], which is the first DRL-based approach that has the action to choose one of two job allocation policies. CuSH has two DRL-based agents. One is a job selector which can select one job from the five first jobs in

the queue. The other is a policy selector that selects an allocation policy for the selected job (depth-first or breadth-first). The reward function counts the current average waiting time at each step and the average normalized turnaround time at the terminal state. The experiments are conducted on a simulated 128-node cluster with each node containing 2 CPUs and 4 GPUs.

Mao et al. [23] present **Decima**, a DRL-based scheduler that can schedule interdependent jobs. The goal of Decima is to minimize the average completion time of interdependent jobs. The dependencies of jobs are formulated as Directed Acyclic Graphs (DAGs). The state is the combination of the DAGs (the number of tasks remaining in the stage), the average task duration, the number of executors currently working on the node, the number of available executors, and whether available executors are local to the job. The actions are picking a stage of the jobs and then calculating how many executors can be allocated to the stage of the job.

RLScheduler, proposed by Zhang et al. [24], utilizes PPO [40] to build a batch job scheduler for HPC systems. The results demonstrate that the DRL-based scheduler outperforms heuristic-based schedulers for some workloads when considering performance in terms of job slowdown or job waiting time. However, it only shows a limited improvement compared to the best-performing heuristic. RLScheduler doesn't support back-filling actions; it relies on a simple back-filling heuristic to fill the system's holes with possible jobs. Also, the Agent optimizes a single scheduling objective, either average job slowdown or average waiting time. Based on PPO, Wang et al. [25] designed a job scheduler named **RLSchert**. The key improvement of RLSchert is that it enables a prediction model to estimate the remaining runtime of jobs by feeding a recurrent neural network-based model with the features derived from HPC jobs. The experimental results showed that their scheduler outperformed DeepRM [20] and commonly used heuristics in terms of average slowdown and resource utilization. However, inaccurate runtime predictions can lead to jobs being killed by the scheduler, resulting in resource wastage and reduced user satisfaction.

DRAS, proposed by Fan et al. [26], uses two DRL agents, one to perform job selection and another one to make back-filling decisions. The two agents observe the job queue with a window size of 50. The authors evaluated DRAS with two reinforcement learning algorithms, DQN and PG. The results suggested that DRAS with PG achieves an overall

best performance when compared to DRAS with DQN, Decima [23], and other heuristics, such as FCFS, and BinPacking. For DRAS, more than 83% of the jobs are back-filled, which means the back-filling agent is used more frequently. During the training process, DRAS used the same workload in every episode, which can lead to overfitting the scheduler to the training workload.

Zhang et al. [27] present **SchedInspector** that builds an inspector based on reinforcement learning. The inspector takes into consideration the cluster's and queue's state to determine whether to execute or ignore the scheduling decision made by a heuristic algorithm. A scheduling decision may be ignored with the aim of making better scheduling decisions in the future, which can improve the scheduling objective. However, the performance of this method largely depends on the base heuristic. Further, it relies on a separate heuristic-based back-filling algorithm.

Li et al. [28] propose a Multi-Resource Scheduling for HPC called **MRSch**. The implementation is similar to DRAS but it considers burst buffers as a resource for HPC systems. However, the burst buffers are not a critical resource of the HPC system. They have to randomly generate burst buffer usage in the simulation to evaluate the performance.

Narantuya et al. [104] build their scheduler based on DQN in the **GIST AI-X** system. However, the title of the paper mentioned Multi-Agent. However, it means training different scheduler agents for different partitions of clusters. As partitions do not share jobs and resources, and agents do not communicate with others, "Multi-Agent" becomes multiple independent agents.

Li et al. [105] develop **GARLSched** which is an HPC job scheduler trained by Generative adversarial deep reinforcement learning. It introduces a Generative adversarial deep reinforcement learning to task scheduling in HPC. The model consists of three parts, which are expert pooling, discriminator, and DRL scheduler. The main idea is to use experts' prior knowledge to guide the learning of the scheduler agent. The discriminator is used to classify the given actions made by experts or agents and accelerate the learning of the agent toward the experts. However, the expert pooling is using scheduling heuristics which have been outperformed by a number of recent works. The results show that GARLSched cannot largely outperform the best heuristics.

De et al. [106] introduce **semi-MDP** in DRL-based HPC job scheduling. The state space consists of the number of processors allocated, the number of processors free up to a time horizon size of H, a window of size W from the job queue, and statistics of the cluster. The action is to select a job from the window of size W. They describe their MDP as semi-MDP because the scheduler only makes the scheduling decision when job arrivals or completion as they claim that time-based MDP is MDP and the event-based MDP is semi-MDP. In fact, there is no difference in their DRL algorithm.

Kolker-Hicks et al. [29] propose an RL-based backfilling strategy for HPC batch job scheduling, named **RLBackfilling**. Traditional backfilling heuristics improve utilization but lack adaptability. Their approach models backfilling as a sequential decision-making problem, where an RL agent dynamically selects jobs to backfill based on queue states and available resources. Using deep reinforcement learning, the model optimizes job selection over time. Experiments show that the RL-based strategy reduces job waiting times and improves system throughput compared to traditional backfilling methods.

Upon reviewing existing works for DRL-based HPC schedulers, we identify two main challenges that remain unsolved. The first challenge is that existing DRL-based HPC schedulers face significant limitations in their evaluation methodologies, hindering their applicability to modern HPC environments. Firstly, many studies rely on fully simulated settings where job arrivals, resource availability, and system behaviors are artificially generated, providing a controlled yet oversimplified environment that fails to capture real-world complexities [20–23, 25, 28, 104]. Second, while some research attempts to bridge this gap by using historical job traces from real HPC systems, such as SWF [30], these traces often come from older systems with homogeneous architectures, lacking representation of modern HPC clusters that incorporate heterogeneous computing resources, including GPUs [24, 27, 96, 105, 106]. This reliance on outdated traces limits the generalizability of DRL-based schedulers to current and future HPC systems. Third, existing approaches predominantly focus on CPU-based scheduling, overlooking the challenges posed by other resources, such as memory, GPUs [26]. Scheduling policies optimized for homogeneous CPU clusters may not effectively scale to modern HPC workloads that demand intelligent resource allocation across diverse hardware components. These limitations highlight the need for more comprehensive evaluations that account for real-time system interactions, up-to-date workload characteristics, and the heterogeneous nature of modern HPC clusters.

The second challenge focuses on modifying and improving DRL algorithms and architectures to better suit the HPC scheduling domain. This may involve developing new DRL algorithms, incorporating domain-specific knowledge into the learning process, and designing novel neural network architectures to capture the unique characteristics of HPC systems. For example, theoretically, the job queue length is unbounded for the HPC system which means the system can receive arbitrary numbers of jobs. However, the input of a DRL-based scheduler is pre-defined as the number of neurons is fixed. Adjusting DRL algorithms and architectures may also include addressing scalability issues to accommodate large-scale HPC systems, ensuring robustness to dynamic workloads, and improving convergence and stability during training.

2.4.6 Resource Allocation in HPC Scheduling

In traditional HPC scheduling research, the focus has been on job selection, as efficient job ordering was seen as the primary lever for optimizing system performance. Many early scheduling policies aimed to prioritize jobs based on fairness, responsiveness, or system utilization, assuming that a fixed resource allocation strategy would be sufficient to achieve overall efficiency. However, as HPC systems become increasingly heterogeneous, with a mix of CPUs, GPUs, FPGAs, and high-bandwidth memory architectures, resource allocation of jobs has emerged as a critical factor affecting performance. In this section, we introduce some commonly used resource allocation heuristics, including first-fit, best-fit, and topology-aware allocation. Additionally, we summarize recent research efforts that explore resource-aware scheduling, highlighting key advancements and limitations in the field.

First-fit allocation is a simple and widely used heuristic where jobs are assigned to the first available set of resources that meet their requirements. Given a sorted list of nodes, the scheduler iterates through the list and assigns the job to the first node (or set of nodes) that can accommodate it. This approach is fast and computationally efficient, making it suitable for large-scale HPC clusters. However, first-fit allocation can lead to resource fragmentation, where smaller jobs fill up available nodes inefficiently, leaving scattered unused resources that cannot accommodate larger jobs. Despite its simplicity, first-fit remains a baseline allocation strategy in many practical HPC workload managers due to its low scheduling overhead.

Best-fit allocation improves upon first-fit by selecting the most tightly packed set of resources that can accommodate a job. Instead of simply choosing the first available nodes, best-fit searches for nodes that minimize resource wastage by filling the remaining capacity as efficiently as possible. This heuristic reduces fragmentation and improves resource utilization but comes with higher computational overhead, as the scheduler must evaluate multiple placement options before making a decision. Best-fit allocation is particularly useful in environments where minimizing idle resources is a priority, though it may lead to increased scheduling delays when job arrival rates are high.

Topology-aware allocation is a strategy in HPC that aims to optimize job placement by considering the network topology of the system. A common approach involves minimizing the number of switches utilized during job execution to reduce communication latency and contention. This is typically achieved by assigning jobs to nodes connected under the same lowest-level switch in the network hierarchy, thereby enhancing data locality. The Slurm workload manager, for instance, implements such topology-aware placement algorithms. It identifies the lowest-level switch capable of accommodating a job's resource requirements and allocates nodes under that switch using a best-fit strategy. This method not only reduces communication overhead but also improves overall system performance by leveraging network proximity.

In the realm of topology-aware scheduling, several studies have focused on optimizing task placement to minimize communication overhead in HPC systems. Agarwal et al. [107] introduced a topology-aware task mapping algorithm designed to reduce communication contention on large parallel machines by considering the network topology during task assignment. Similarly, Georgiou et al. [108] developed a method that accounts for both the machine's topology and the application's communication characteristics to determine optimal node allocation, integrating this approach into the Slurm resource and job management system. Amaral et al. [109] extended topology-aware scheduling to GPU workloads in cloud environments, optimizing GPU placement based on interconnect bandwidth, NVLink topology, and PCIe locality. Their proposed approach significantly enhances deep-learning training performance by reducing inter-GPU communication delays. Mishra et al. [110] proposed communication-aware job scheduling algorithms within SLURM, aiming to allocate nodes in a manner that reduces network contention, thereby improving the performance of communication-intensive applications. Lan et al. [111] introduced Neural Simulated Annealing (NSA) for topology-aware job allocation. Their

method optimizes job placement dynamically, learning network congestion patterns to minimize inter-node communication overhead in HPC clusters.

Beyond network topology considerations, system performance optimization in HPC scheduling focuses on efficient task placement and resource utilization. Several studies have explored heuristic and metaheuristic approaches to improve scheduling efficiency. Augonnet et al. [112] introduced **StarPU**, a unified task scheduling framework for heterogeneous architectures. StarPU dynamically assigns workloads across CPUs and GPUs using heuristic and performance-aware strategies, optimizing resource usage in multicore systems. Li and Peng [113] introduced an improved genetic algorithm for cloud-based task scheduling, enhancing adaptability to dynamic workload changes and optimizing scheduling decisions for better efficiency. Gupta et al. [114] proposed an HPC-aware virtual machine (VM) placement strategy that accounts for inter-VM communication and workload co-location. Their heuristic-based approach minimizes network bottlenecks and improves execution efficiency for parallel HPC workloads in cloud environments. Grandl et al. [115] proposed **Tetris**, a multi-resource packing scheduler that optimizes job allocation by balancing CPU, memory, and I/O demands. Their approach addresses inefficiencies in traditional cluster schedulers, reducing fragmentation and improving resource utilization. Rekha and Dakshayini [13] explored a genetic algorithm-based task allocation strategy that improves load balancing in cloud environments. Their method optimizes resource utilization while reducing job execution time.

Researchers also focus on improving HPC scheduling by leveraging runtime estimation techniques to enhance resource allocation efficiency. Kumar et al. [116] propose a runtime-elastic scheduling approach that dynamically adjusts resource allocation based on job execution progress and system load. By enabling jobs to scale resource usage adaptively, their method optimizes system throughput and reduces job waiting times compared to static allocation policies. Similarly, Villapando and Rubio [117] address the issue of runtime estimation inaccuracies in traditional HPC schedulers by introducing a data-driven correction mechanism that refines job runtime predictions using historical execution data. This improves scheduling accuracy and resource allocation efficiency, minimizing job delays and system underutilization. Both studies highlight the significance of adaptive scheduling strategies that incorporate real-time job execution characteristics to enhance overall system performance.

Energy efficiency is an increasing concern in HPC scheduling, as power consumption significantly impacts operational costs. Recent studies have focused on workload-aware scheduling techniques that balance performance and energy efficiency. Viswanathan et al. [118] introduced an energy-aware VM allocation strategy that minimizes power consumption while maintaining workload performance. Their approach considers CPU utilization and energy constraints when scheduling jobs. Quang-Hung et al. [119] developed heuristic-based energy-aware VM allocation strategies tailored for HPC clouds. Their method dynamically adjusts VM placement to balance power efficiency and performance. Arabas and Niewiadomska-Szynkiewicz [120] proposed an energy-efficient workload allocation technique that optimizes power usage in distributed HPC environments. Their approach leverages heuristic scheduling to minimize energy consumption without degrading system throughput. Arabas [121] further extended energy-aware scheduling models by developing hierarchical task allocation strategies for HPC clouds, integrating energy constraints into workload distribution decisions.

Given the summary of resource allocation techniques, we identify a key challenge in resource allocation is that existing HPC scheduling approaches often treat job selection and resource allocation as independent processes, leading to inefficiencies in overall system performance optimization. This decoupled approach fails to account for the intricate dependencies between job selection and resource allocation, particularly in heterogeneous HPC environments where diverse resources such as CPUs, GPUs, and memory must be optimally matched to varying job requirements. As HPC systems continue to evolve with increasingly complex architectures and workload demands, an integrated scheduling approach that jointly optimizes both job selection and resource allocation is essential to improving resource utilization, minimizing fragmentation, and enhancing overall system efficiency.

Chapter 3

Advancements in RL-Based Job Selection in HPC

Job selection is a central challenge in HPC scheduling, particularly under dynamic workloads and complex resource requirements. Traditional heuristics such as First-Come-First-Served (FCFS) and Smallest-Job-First (SJF) often struggle to adapt to varying system situations and rely on static rules that fail to fully exploit scheduling opportunities. Recent advances in deep reinforcement learning (DRL) have shown promise in improving job selection strategies. However, existing RL-based schedulers face critical limitations, including reliance on fixed observation windows and separate backfilling processes. This chapter introduces Deep Backfilling (DBF), a unified DRL-based job selection framework that addresses these limitations through two key innovations. First, the Split Window Technique enables the agent to observe jobs from both the head and tail of the queue, enhancing visibility and allowing backfilling opportunities to be captured without external heuristics. Second, the Schedule Cycling mechanism redefines the scheduling trigger and reward assignment process, enabling the agent to learn effective placement decisions over multiple scheduling steps while maintaining training stability. DBF integrates job selection and backfilling into a single agent, eliminating the need for separate decision components and ensuring coherent scheduling policies. Experimental results demonstrate that DBF improves scheduling efficiency, reduces job waiting time and queue length compared to a range of HPC job selectors.

This chapter is derived from the following publication:

3.1 Introduction

In HPC scheduling, job selection is a critical step that determines which jobs from the queue will be dispatched for execution. Online job selection must operate in a dynamic environment where job arrivals are unpredictable, resource availability fluctuates, and scheduling decisions must be made in a timely manner. Traditional job selection policies rely on fixed heuristics to determine scheduling order. First-Come-First-Served (FCFS) enforces strict queue order, ensuring fairness but often leading to poor resource utilization where large jobs delay smaller ones that could otherwise run. Smallest-Job-First (SJF) prioritizes smaller jobs to improve throughput but risks job starvation, where large jobs experience prolonged delays. To mitigate inefficiencies, backfilling is commonly used, allowing smaller jobs to execute ahead of larger ones if they do not delay the first job in the queue. However, backfilling operates under predefined rules and does not dynamically adapt to system conditions.

Recently, HPC researchers have turned their attention to DRL for job selection, where a DRL agent learns a scheduling policy rather than relying on predefined heuristics. The advantage of RL-based job selection lies in its ability to adapt dynamically to workload variations, capturing complex patterns in job arrivals, resource contention, and execution times. Unlike traditional heuristics, which follow rigid rules, a well-trained RL agent can generalize across different scheduling scenarios and optimize decisions in ways that conventional methods cannot. Existing research demonstrates that RL-based job selection can outperform heuristic approaches [24, 26], and they share the following basic approach: an Agent observes a *fixed window of jobs at the head of the queue*, e.g., the first 128 jobs, and sequentially selects individual jobs from the window to be scheduled. When the queue length exceeds the observation window, the RL agent operates under partial observability, failing to consider jobs deeper in the queue. This limitation can lead to starvation of larger jobs, inefficient resource utilization, and scheduling bottlenecks.

- **Lingfei Wang**, Aaron Harwood, and Maria A. Rodriguez. A Deep Reinforcement Learning Scheduler with Back-filling for High Performance Computing. In *Proceedings of 2021 IEEE Asia-Pacific Conference on Computer Science and Data Engineering (CSDE)*, pp. 1-6. IEEE, 2021.
- **Lingfei Wang**, Aaron Harwood, and Maria A. Rodriguez. Deep Back-Filling: a Split Window Technique for Deep Online Cluster Job Scheduling. In *Proceedings of 2023 IEEE International Conference on High Performance Computing & Communications*, pp. 772-779. IEEE, 2023.

Moreover, traditional backfilling heuristics allow schedulers to scan the entire queue and opportunistically schedule smaller jobs that fit within available resource gaps. Some RL-based approaches incorporate backfilling by switching from the RL agent to a backfilling heuristic when the selected job cannot be scheduled [24]. While this method maintains high resource utilization, it breaks the end-to-end learning process, limiting the RL agent's ability to develop a fully optimized scheduling policy. More recent approaches replace heuristic-based backfilling with a separate RL-based backfilling agent, trained independently to identify jobs suitable for backfilling [26]. However, this multi-agent setup introduces coordination challenges, as each agent must implicitly account for the other's decision-making process to achieve an optimal policy. Since both agents operate on partially observed job queues, misalignment between job selection and backfilling decisions can lead to inefficient scheduling, suboptimal resource utilization, and increased waiting times. Addressing these limitations requires a unified RL-based approach that integrates job selection and backfilling within a single agent, ensuring seamless decision-making while maintaining scheduling efficiency.

To address these challenges, we propose a novel reinforcement learning-based job selection approach that integrates backfilling without relying on external heuristics or separate agents. Our method introduces a Split Window Technique, which enables the RL agent to observe jobs not only at the head of the queue but also at the tail, ensuring that backfilling opportunities are considered throughout the scheduling process. By maintaining a single-agent framework, this approach eliminates coordination issues inherent in multi-agent systems while retaining the benefits of reinforcement learning-based optimization. Additionally, we introduce Schedule Cycling, a mechanism designed to improve both scheduling decision-making and training stability. Instead of treating RL training as a step-by-step process, Schedule Cycling enables the agent to learn job selection patterns across multiple job selection steps, providing a more comprehensive understanding of system dynamics. The key contributions of this chapter are as follows:

- We introduce a novel Split Window Technique to address the challenge of unbounded state space in job selection. By allowing the RL agent to observe both the head and tail of the queue, this method ensures that backfilling opportunities are effectively utilized while maintaining a fully RL-driven approach.

- We develop a Schedule Cycling mechanism that improves RL model convergence and enhances long-term scheduling efficiency by enabling the agent to learn across multiple scheduling steps.
- Unlike existing approaches that rely on heuristics or separate agents for backfilling, our method trains a single RL agent to handle both job selection and backfilling. This unification removes coordination challenges between independent agents and enables a fully data-driven scheduling strategy that dynamically adapts to workload variations.

3.2 Related Work

Recent RL-based job selection approaches have demonstrated improvements over heuristics but face several key challenges. RLScheduler [24] applies PPO for batch job scheduling, but its improvements over heuristics remain limited due to reliance on a separate backfilling heuristic and a fixed single-objective optimization. SchedInspector [27] introduces an RL-based inspector that overrides heuristic decisions, yet its performance remains tied to the base heuristic and still requires heuristic-based backfilling. DRAS [26] improves flexibility by employing two separate RL agents for job selection and backfilling, but this introduces coordination challenges, as each agent must implicitly learn the other's decisions. Furthermore, its reliance on fixed workload patterns during training increases the risk of overfitting. RLSchert [25] incorporates runtime prediction to refine scheduling but is highly sensitive to misestimation, which can result in inefficient scheduling and resource wastage. Lastly, Orhean et al. [122] explored Q-Learning and SARSA for job scheduling in heterogeneous environments, but their approach failed to scale beyond small-scale systems.

Overall, while these works demonstrate that RL-based job selection can outperform traditional heuristics, they share common limitations, including reliance on separate heuristics for backfilling, lack of coordination between selection and allocation, and challenges in generalization due to static training methodologies. These gaps motivate the need for a more integrated, adaptable RL-based job selection approach capable of handling dynamic HPC workloads effectively.

3.3 Method

3.3.1 Split Window Technique

The agent's observation window defines the subset of jobs in the queue that the RL agent can consider for scheduling at any decision step. Given the unbounded nature of the HPC job queue, observing all jobs simultaneously in all conditions is infeasible. Existing approaches typically use a fixed-length window covering only the head of the queue, limiting visibility to the first M jobs. A major drawback of a head-only observation window is the potential accumulation of unschedulable jobs at the front of the queue. When this occurs, the agent may enter a "wait" state, where no jobs in its window can be scheduled, forcing it to passively wait for cluster resources to free up. This not only reduces scheduling efficiency but also prevents the agent from learning effective backfilling strategies. The backfilling opportunities are missed, as the agent lacks visibility into jobs that could efficiently fill resource gaps.

To mitigate these issues, we introduce a *split-window* strategy, allowing the agent to observe jobs from both the head and tail of the queue. Formally, the observation window consists of:

- M_h jobs from the head of the queue: $\{1, 2, \ldots, M_\mathrm{h}\}$.
- M_t jobs from the tail of the queue: $\{L(t), L(t) - 1, \ldots, L(t) - M_\mathrm{t} + 1\}$.

where $M_\mathrm{h} + M_\mathrm{t} = M$ and $L(t)$ is the total queue length at time t. This approach ensures that the agent maintains awareness of newly arrived jobs while also considering those that have been waiting the longest. By leveraging this expanded view, the agent can make more informed scheduling decisions without requiring a separate backfilling heuristic. By incorporating jobs from the tail of the queue, the split-window approach eliminates this issue. The agent maintains visibility into all newly arriving jobs, ensuring that it always has viable scheduling candidates. This dynamic perspective allows the agent to integrate backfilling naturally into its decision-making process, improving scheduling efficiency while avoiding reliance on external heuristics.

3.3.2 Deep Backfilling Design

To demonstrate the effectiveness of the split window technique, we developed a job scheduling simulation environment and framework for agent training, similar to existing approaches but with the following two significant differences:

- **Schedule Cycling** – at any given scheduling decision, the agent can choose not to schedule a job (even if a job could be scheduled) but rather just wait until either a running job has been completed or a new job has arrived on the queue.
- **Split Window** – the agent can observe M_h jobs at the head of the queue and M_t jobs at the tail of the queue at each scheduling decision.

For convenience, we refer to our overall approach as **Deep Backfilling** or DBF. In this section, DBF is explained in detail with the RL representations – state, reward, and action. Next, we present our Schedule Cycling, the training strategy, which defines when and how the scheduling decision is made, which makes it possible for our agent to learn higher-level scheduling strategies.

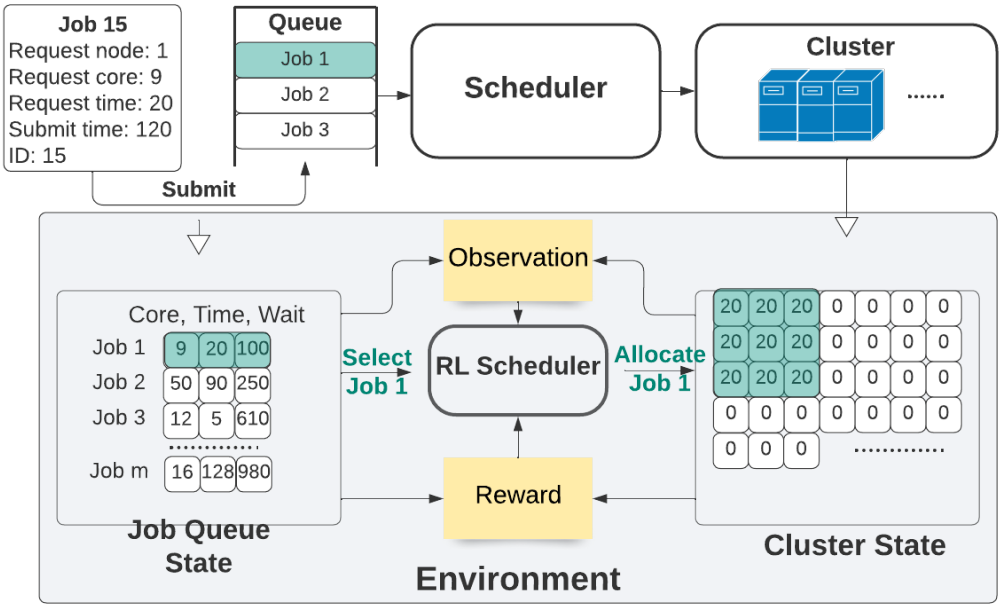

FIGURE 3.1: The DBF framework.

Fig. 3.1 shows an overview of the DBF framework. In the upper part of the figure, users submit their jobs to the job queue by specifying the resources needed. The scheduler makes scheduling decisions based on the cluster's status and the job queue. The bottom part shows the RL representations of the scheduling problem.

DBF receives **observations** from two primary sources: the cluster state and the job queue state. The cluster state, $\sigma(t) = \left[\sigma_1(t), \sigma_2(t), \cdots, \sigma_N(t)\right]$, describes the availability of compute resources, while the job queue state provides information about pending jobs in the queue. The cluster state at a given time consists of an array representing the status of each CPU core in the system. Each core is assigned a numerical value indicating its current availability. If a core is free, it is marked as 0, $\sigma_1(t) = 0$. If a core is occupied by a running job, the value represents the remaining time until that job completes and the core is released. This representation allows the scheduler to track resource availability dynamically. An example of this state encoding is illustrated in Fig. 3.1.

The job queue state is captured through a window-based observation strategy, where a subset of jobs from the queue is selected as input to the scheduler. Each job within this observation window is characterized by three terms which are the number of cores it requests, its requested runtime, and its waiting time in the queue. By combining these two states, the RL scheduler obtains a real-time snapshot of the system, enabling it to make informed job selection decisions based on available resources and job characteristics. We also explored a binary state design for a separate RL backfilling agent; this variant is documented in Appendix A for completeness but is not used in the results reported in this chapter.

The set of **actions**, $\{1, 2, \ldots, M\} \cup \{\text{Fwd}\}$, includes selecting one job from M jobs in the window, and *Fwd* which represents a "forward" action where the agent can wait for either a job on the cluster to complete running or another job to arrive before selection a job from the window, as described further in Section 3.3.3.

The **reward function** is designed to balance multiple scheduling objectives, including resource utilization, queue length, and total waiting time. Instead of optimizing a single metric, our approach simultaneously considers several objectives to maximize overall system performance. At time t, the reward function is defined as:

$$\Re(t) = -\alpha_1 \left(1 - \eta(t)\right) - \alpha_2 \frac{L(t)}{L_{\max}(t)} - \alpha_3 \frac{W(t)}{W_{\max}(t)} \tag{3.1}$$

where:

$$W(t) = \sum_{J_j \in \mathcal{Q}(t)} (t - w_j)$$

$$L_{\max}(t) = \max_{\tau \in [0,t]} \{L(\tau)\}$$

$$W_{\max}(t) = \max_{\tau \in [0,t]} \{W(\tau)\}$$

Here:

- $\eta(t)$ represents the instantaneous cluster utilization at time t.
- $L(t)$ is the number of jobs in the job queue at time t.
- $W(t)$ is the total waiting time of jobs in the queue, computed as the sum of the waiting times $(t - w_j)$ for all queued jobs J_j.
- $L_{\max}(t)$ and $W_{\max}(t)$ track the maximum queue length and waiting time observed up to time t, respectively.
- $\alpha_1, \alpha_2, \alpha_3$ are scaling factors that balance the contributions of resource utilization, queue length, and waiting time in the reward function.

This formulation ensures that the agent is incentivized to reduce unused resources, minimize queue length, and prevent excessive job waiting times, leading to improved scheduling efficiency. The terms $L(t)$ (jobs) and $W(t)$ (time) live on different scales and units; dividing by the running maxima $L_{\max}(t)$ and $W_{\max}(t)$ makes both signals dimensionless and bounded in $[0, 1]$ (since $L(t) \leq L_{\max}(t)$ and $W(t) \leq W_{\max}(t)$ by definition). This prevents either queue length or waiting time from numerically dominating the reward, stabilizes policy updates, and makes the reward comparable across traces and load levels. Using running maxima adapts the normalization to the observed workload intensity without introducing extra hyperparameters; early spikes are absorbed as the maxima update, while the coefficients $\alpha_1, \alpha_2, \alpha_3$ retain clear roles in balancing utilization versus queueing pressure.

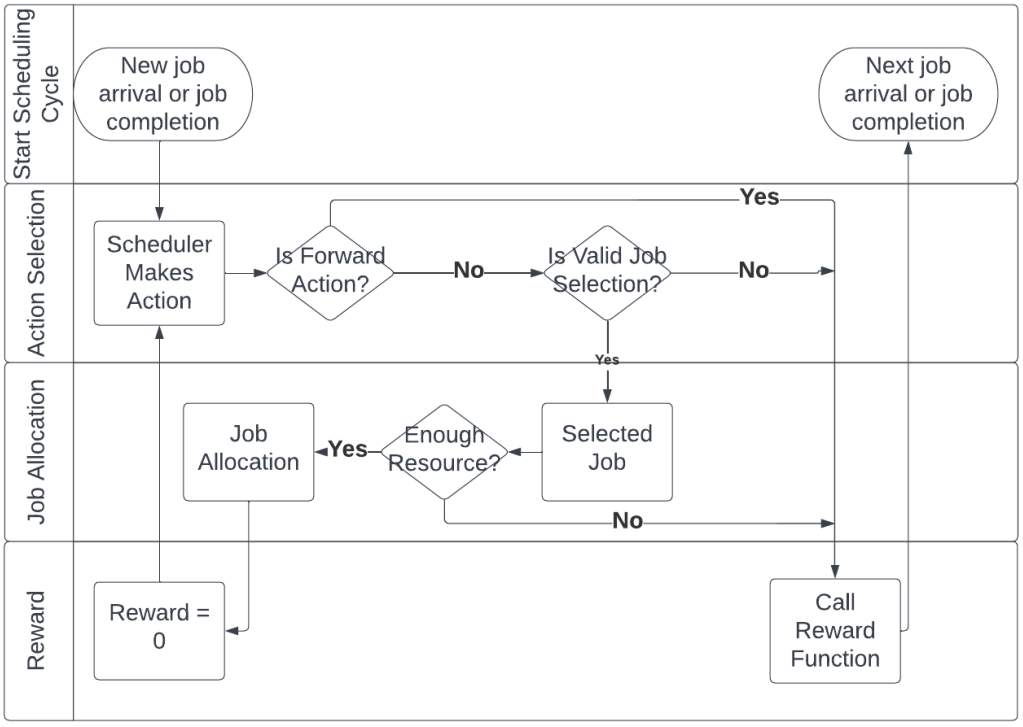

FIGURE 3.2: The flowchart of Schedule Cycling.

3.3.3 Schedule Cycling

In general, to train a reinforcement learning agent, the agent either receives the rewards immediately after the actions or receives a delayed reward when it reaches special states or rules. The agent can learn quickly for an immediate reward by evaluating its action step-by-step. However, the training may end up with a suboptimal policy when the agent tries to maximize the reward for each step and loses the "global vision" of the problem. For the delayed rewards, the agent can overview the problem and learn to achieve an overall better result. But it can fail to learn when the delay is too long.

For the existing works, such as RLScheduler [24], SchedInspector [27], and DRAS [26], if a job is selected by the agent when the cluster does not have sufficient resource to run the job, their schedulers will wait for other job completions and apply backfilling until the job can be run. Pathological cases where the selected job is large can significantly degrade the agent's performance since it may have to wait for some time before making any decisions.

To solve the problem, we designed a new scheduling mechanism – Schedule Cycling, shown in Fig 3.2. A scheduling cycle is triggered at time t when a new job j arrives on the queue at time t or a job j completes running on the cluster at the time t. The

agent receives the observation and selects an action from the action space. The Fwd action will terminate the current scheduling cycle, receive a negative reward $\mathfrak{R}(t)$, and move the system forward in time to the next scheduling cycle (next new job arrival or running job completion). If the selected action is in $\{1, 2, \ldots, M\}$, we check whether it is valid or not. An invalid selection arises when the scheduler chooses the i^{th} job from the window and either $L(t) < i$ or the chosen job cannot be run due to insufficient cores. For invalid selections, the system will be forwarded to the next scheduling cycle and receive a negative reward $\mathfrak{R}(t)$. If the selection is valid, then the job is removed from the queue and allocated to the cluster with a zero reward given to the agent, and the agent needs to continue making scheduling decisions until the system is forwarded. In this way, the Schedule Cycling mechanism allows the agent to learn to place jobs to maximize the reward, and the agent can also learn to achieve better job placement with the Fwd action when a more suitable job arrives in the near future.

Using Schedule Cycling, our scheduler not only leverages the delayed reward in a manageable number of transitions so as to take an overview of the problem but also has a chance to learn whether to wait or not, as only valid selections will be placed. The overall strategy causes the scheduler to learn to make valid job allocations without an explicit penalty. Furthermore, including the waiting time in the observation and in the reward function, $\mathfrak{R}(t)$, prevents job starvation. Therefore, our DBF framework can learn a more advanced policy compared to without Schedule Cycling and does not require a separate backfilling heuristic nor a backfilling agent.

3.4 Experimental Results

3.4.1 Simulation and Agent Implementation

We simulate an HPC cluster that is widely adopted in the existing Deep RL approaches. An HPC cluster is modeled as a number of cores, N, and a job queue. Each job, $j \in \{1, 2, \ldots\}$, submitted to the cluster has a submission time, a requested number of cores, and a requested total run-time. The online job scheduling problem requires selecting jobs from the queue, as or after they arrive, that can be run on the cluster and usually assumes that the job queue is unbounded in length. Job j can be run at time t if the number of available cores on the cluster at time t is at least the requested number of cores. Jobs

and job inter-arrival times are modeled based on commonly used traces, specifically the Lublin dataset [123]. The Agent's PPO algorithm is implemented using PyTorch [124]. In DBF, training occurs in an episodic manner, with an episode ending when 1,000 jobs have been successfully placed, whether there are remaining jobs in the queue or not. Since the scheduler can make invalid job selections, the length of each episode is not fixed. After an episode terminates, an update to the actor and critic networks is performed by stochastic gradient descent with a mini-batch size of 128. The learning rate is 3×10^{-4}, clip ϵ is 0.2 and the discount factor γ is 0.99. The hyperparameters for the reward function are $\alpha_1 = \alpha_2 = \alpha_3 = \frac{1}{3}$. The hidden layer size for actor and critic networks is $[1024, 512, 256]$. The listed hyperparameters come from a hyperparameter search.

We normalize the Agent's observation, $\bar{\sigma}(t) = \frac{1}{e_{\max}\sigma(t)}$, where $e_{\max} = \max_{j \in \mathcal{J}}\{r_j\}$ (the maximum requested runtime) and $\mathcal{J}$ is the set of jobs arriving overall time, as training with normalized states has been shown to speed up learning in RL problems [125]. In particular, we use the maximum allowed runtime of the system to normalize the cluster's state vector and the maximum number of processors, maximum allowed runtime, and maximum waiting time of all jobs to normalize the state vectors of queued jobs.

3.4.2 Evaluation Setup

As previously mentioned, we use the Lublin workload model [123] to evaluate our approach. The reason is twofold: *i*) it is widely used in the related literature, and *ii*) it represents a realistic HPC workload as it is based on execution traces of thousands of jobs spanning several months. Specifically, the trace data contains all jobs that arrived in 89 days on a 256-core cluster. Each job requests 1 to 256 cores, and 1 second to 124707 seconds (34.6 hours) runtime. The average requested core-time per job (requested number of cores × requested runtime) is $209,278$ seconds, and the average amount of core-time submitted to the cluster is 271.5 per second. To avoid over-fitting – which may arise if the same sequence of jobs is used across episodes – we use a sequence of jobs starting at a random point in the workload for each episode.

We compare the performance of DBF to three heuristics, as they are widely used in real-world systems, and one RL-based scheduler, the newest work in the related literature: First-Come-First-Served (**FCFS**) schedules the jobs in the order of their arriving time. FCFS with backfilling is the default scheduler for many HPC clusters and works reasonably

well in practice. Smallest-Job-First (**SJF**) gives higher priority to smaller jobs and thereby always places the job with the smallest number of requested cores. Last-Come-First-Serve (**LCFS**) schedules jobs with the latest arrival first. LCFS can suffer of starvation since the probability of scheduling the job at the head of the queue is low.

SchedInspector is an RL-based HPC scheduler [27] proposed by Zhang, Dai, et al. The main idea behind SchedInspector is to train an RL agent to determine whether to accept or ignore the job placement choice made by a heuristic-based scheduler. The authors hypothesize that ignoring certain scheduling actions in some states can lead to better overall performance. We implemented their approach by using the code released in their *github* repository[1] with the default objective of minimizing the average bounded job slowdown (the average fraction of the job waiting time plus job execution time to the job waiting time) and backfilling enabled. We use both SJF and F1 [16] as the base heuristic schedulers, as also used by Zhang, Dai, et al.

SchedInspector and DBF use different training methods. SchedInspector trains on a job sequence of 128, and it stops receiving new jobs when 128 jobs have been submitted to the system. The episodes end after the 128 jobs have been scheduled, no matter how long it takes. We call it offline training. However, an online system can always receive new jobs during operation, and we use this way to train our DBF. To make a fair comparison, we will present the results of SchedInspector in both online and offline testing methods in section 3.4.3.

To evaluate job scheduling efficiency, we consider several key performance metrics. These metrics assess resource utilization, queue dynamics, and job waiting times, providing insights into scheduling effectiveness.

Utilization measures how effectively computational resources are used in the system. It represents the proportion of CPU cores actively running jobs at any given time compared to the total available cores in the cluster.

Average Queue length refers to the number of jobs waiting in the scheduling queue at a given time. An efficient scheduler aims to keep the queue length manageable by balancing job execution and system availability. By averaging this metric over time, we can assess whether the scheduling policy prevents excessive job accumulation and ensures a steady flow of executions.

[1] https://github.com/DIR-LAB/SchedInspector

Average Waiting time represents the duration a job spends in the queue before it starts execution. The average waiting time across multiple jobs provides a measure of overall system responsiveness. A well-optimized scheduler aims to minimize waiting times while ensuring high resource utilization.

Average Job load refers to the total computational demand imposed by jobs waiting in the queue. It considers both the number of requested CPU cores and the expected execution time of jobs. This metric helps understand whether the system is handling a mix of small and large jobs efficiently.

3.4.3 DBF Performance and Schedule Cycling Evaluation

We first evaluate the effectiveness and efficiency of DBF when learning to schedule jobs and optimizing the system's objectives (minimizing queue length and waiting time and maximizing utilization) in an online environment. We test a range of window size configurations. We trained the agent for 100,000 episodes and saved trained models after every 5,000 episodes. Then, we evaluated each of the saved models for 100 episodes without updating. Fig. 3.3 plots the learning curves of DBF with a window size of 20 without a split window, $M = 20$ and $M_t = 0$; similarly, the results for window sizes of 32, 64, and 128 share the same patterns. The points are the average performance for the objectives at each checkpoint, and the error bars are the standard deviations. Fig. 3.3 shows that DBF converges successfully by continuously reducing the average queue length and job waiting times while increasing the utilization of the cluster. After approximately 5,000 episodes, which we consider a reasonable amount of interactions with the environment, the average utilization, waiting time, and queue length improve dramatically. Afterward, the improvements become smaller because the agent cannot observe all jobs in the queue as the window size is 20, where the average queue length is 31.

Next, we evaluate whether DBF can learn backfilling by comparing its performance to specific heuristics and RL-based models with backfilling. From Fig. 3.4, we can see the DBF with window sizes of 32, 64, and 128 can reach better average waiting times and queue lengths with a relatively good utilization in all compared algorithms. The heuristics can achieve slightly better average utilization and job load in the queue. The SchedInspector-based approaches lead to the lowest cluster utilization for both online and offline tests. The main reason is that SchedInspector skipped most of the scheduling choices made by

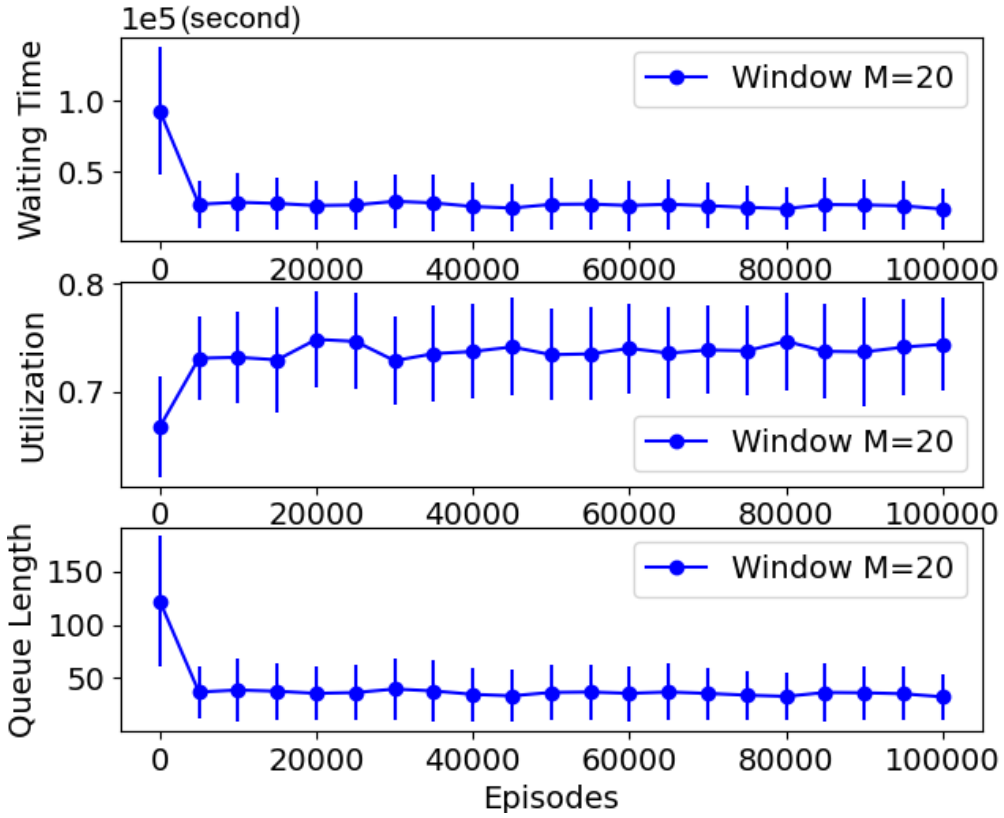

FIGURE 3.3: The learning curves of DBF with a window size of 20 for three different objectives.

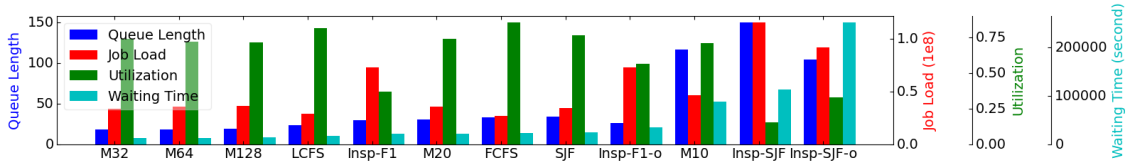

FIGURE 3.4: Performance comparison (sorted by average waiting time) on trained DBF of different window sizes with other schedulers. M10 means the DBF has a window size of 10. Insp-F1-o means the SchedInspector method has an F1 algorithm as the base scheduler and the test method is offline. Insp-SJF means testing SchedInspector in an online method with SJF as the base scheduler.

its base-heuristic algorithms, leaving the cores idle for long periods. Also, SchedInspector optimizes a single objective, average bounded job slowdown. In terms of average job slowdown, SchedInspector with SJF (average job slowdown 62.2) can outperform the SJF (average job slowdown 73.3). But SchedInspector is much worse than SJF when comparing the direct measurements, such as utilization and average waiting time. In our experiments, we observed such trade-offs between scheduling objectives, which means gains in one objective are achieved at the expense of other ones. For this reason, in DBF, the hyperparameters of each objective are adjustable in the reward function.

3.4.4 DBF Split Window Evaluation

The pre-defined size of the input limits the performance of the neural network-based scheduler. If the system received jobs more than the window size, the partially-observed state could cause the agent to learn a suboptimal policy. A natural method is to increase the window size. However, continuously increasing the window size can make the training more challenging and does not guarantee an increase in performance. According to the previous discussion, the split window technique can be a potential solution for the problem as it has a chance to observe all newly arrived jobs in the tail of the queue at least once. The agent gets at least one opportunity to schedule all jobs. In this section, we compare the performance of DBF with and without the split window technique to evaluate the method.

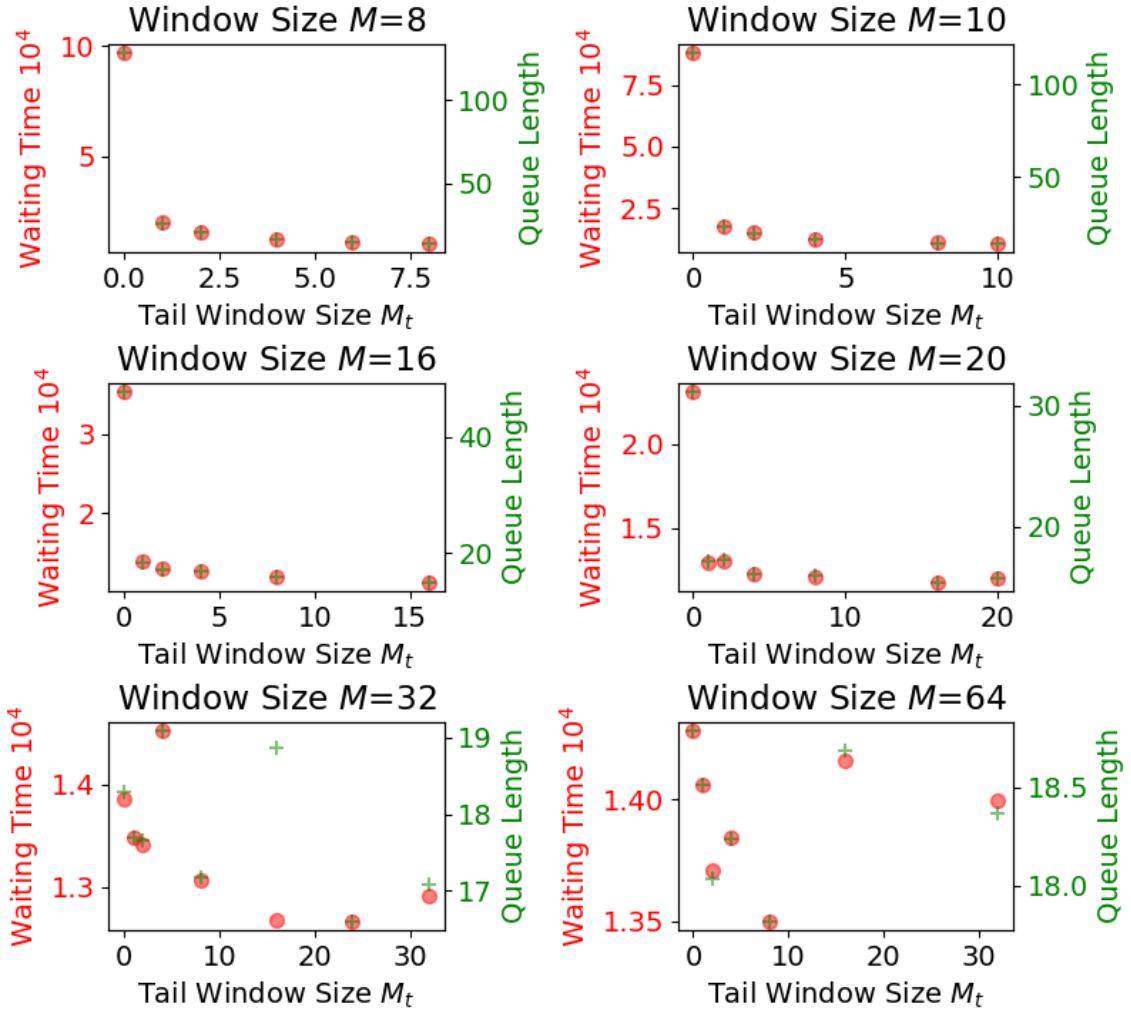

FIGURE 3.5: Performance comparison on different window sizes with various splitting configurations. The red points represent the average waiting time, and the green crosses represent the average queue length.

Fig. 3.5 plots the different splitting configurations with each window size. The titles of the subplots show the total window size, and the x-axis indicates the tail size. It shows the performance of each model after training for 100,000 episodes. The left y-axis is the average waiting time, and the right y-axis is the average queue length. The utilization is not plotted as all models share a similar utilization. With the window sizes of 8, 10, 16, and 20, we can clearly see that increasing the tail window size helps to improve the performance. The performance is significantly enhanced when it includes just one job from the tail with smaller total window sizes. It reveals that the split technique works extremely well, especially with smaller window sizes. It also suggests that we can achieve the same level of performance with less information by using the splitting. Moreover, agents with smaller window sizes are much easier to train, so it also improves sample efficiency. From the subplots of window sizes 32 and 64, the improvement of splitting is not as significant as in the smaller window sizes. Because 32 and 64 are larger than the average queue length. These agents can observe all jobs most of the time, but splitting windows can also improve performance. Unlike window sizes of 8, 10, and 16, continuing to increase cannot always have a positive impact on window sizes of 20, 32, and 64 as the performance gets worse when taking too many jobs from the tail.

Fig. 3.6 provides insights into how the splitting improves the average queue length in training. The plots on the left show the number of invisible jobs (how much the queue length exceeds the window size) on average. The plots on the right give information about the percentage of the partially observed state that the queue length exceeds the window size. For window sizes of 8 and 10, the splitting window can significantly decrease the average exceeding length and the ratio. The exceeding length can be largely decreased for window sizes 16 and 20 while the ratios remain at the same level using the splitting technique. When the window size is large compared to the average queue length, the improvement is limited in terms of average exceeding length for window size 32.

Moreover, Fig. 3.7 illustrates how the split technique improves the average waiting time for each type of job using the window size of 10 as an example. We use the requested cores and runtime to distinguish job types which are shown on the x-axis and y-axis. Every circle in the plot represents a type of job, while the average waiting time is shown in the colors, and the circle size is the number of times that type of job is placed. It is clearly demonstrated in Fig. 3.7 that even one split window at the tail of the queue can dramatically decrease the average waiting time for nearly every type of job. However, as

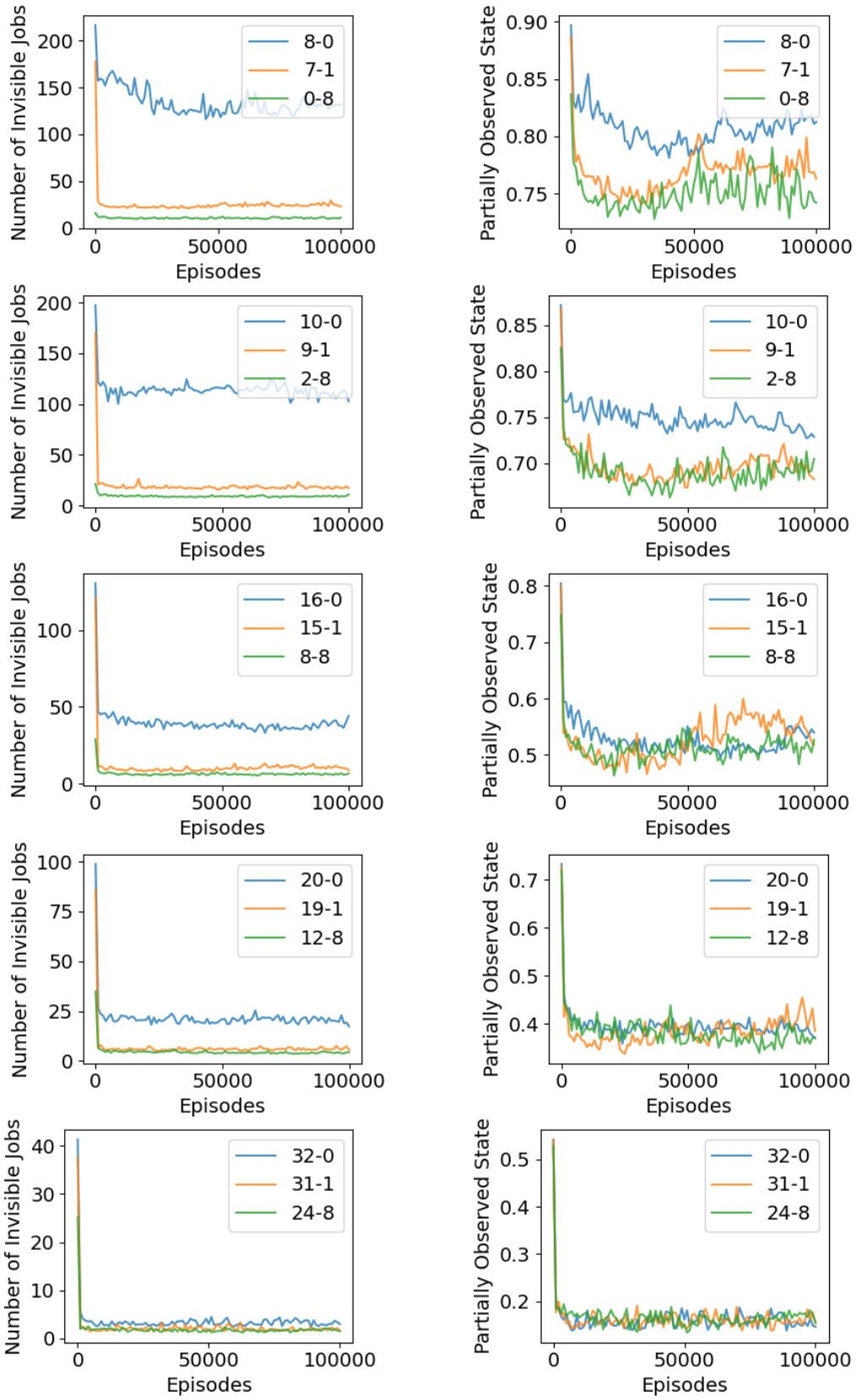

FIGURE 3.6: The average number of invisible jobs (cannot be observed by the agent) and the average ratio that the observation is partially observed in the training. The indicated window configuration M_{h}-M_{t} means the head window size is M_{h} while the tail window size is M_{t}.

discussed, if the tail window size is closer to the total window size, the risk of big jobs at the head of the queue being starved is higher. When comparing the sizes of the circles in the upper area in each subplot, the 0-10 configuration has smaller sizes for big jobs (the requested number of cores is greater than 128). It suggests that if the agent only takes information from the tail of the queue, the big jobs at the head of the queue are invisible to the agent so that the jobs cannot be starved. Although the configuration of 0-10 achieves the best average waiting time, such a window configuration should not be used in the real system to avoid job starvation.

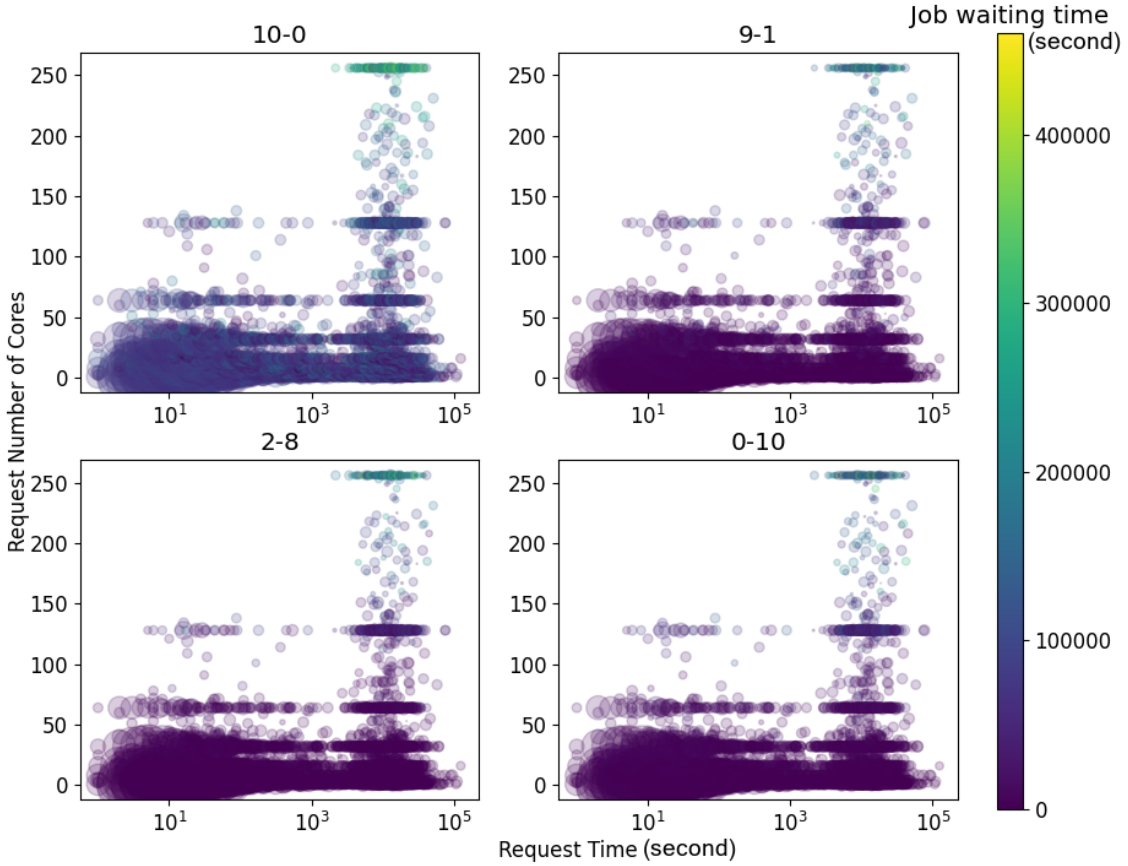

FIGURE 3.7: The average waiting time and the number of times being placed for each job type in 1,000-episode testing. The color indicates the average waiting time, and the circle size indicates the number of times the type of job is placed (a bigger circle means the job is placed more frequently). The titles indicate the window configuration M_{h}-M_{t} that means the head window size is M_{h} while the tail window size is M_{t}.

In conclusion, the splitting technique can increase performance, especially when the window size is smaller than the average queue length. Using the splitting window, the agent can achieve the same level of performance with less information (window size). However, increasing the splitting does not always have positive impacts. When the window

size and tail window size are big enough, continuously expanding the tail window size may cause a performance loss. Although the splitting seems to improve performance, it may lead to job starvation. Thus, the agent should not take all observations from the tail of the queue.

3.5 Summary

This chapter presented DBF, a DRL-based job selector designed to address key limitations in existing reinforcement learning selectors for HPC systems. DBF integrates job selection and backfilling into a single agent, eliminating the need for external heuristics or separate agents. To address the challenge of unbounded queue space, we proposed the Split Window Technique, which enables the agent to observe jobs from both the head and tail of the queue. This improves visibility and allows the agent to naturally learn backfilling behavior without relying on a separate process. Additionally, we presented Schedule Cycling, a training mechanism that restructures the timing of agent decisions and reward feedback, allowing the agent to learn more effective scheduling policies across multiple steps. Experimental results demonstrate that DBF improves scheduling objectives compared to both heuristic and RL-based baselines. The split window design shows significant benefit in scenarios with limited observation capacity, and Schedule Cycling enhances the agent's ability to learn long-term placement strategies.

While this chapter focuses on improving job selection through a single-agent reinforcement learning approach, it assumes a separate, predefined mechanism for resource allocation. However, job selection and resource allocation are inherently interdependent and must be coordinated to achieve globally efficient scheduling decisions. The next chapter addresses this challenge by introducing HeraSched, a hierarchical reinforcement learning-based scheduler that jointly optimizes job selection and resource allocation.

Chapter 4

HeraSched: Hierarchical Reinforcement Learning-Based Job Scheduler in HPC

Job allocation is a critical aspect in contemporary HPC systems, due to compute nodes possessing an increased capacity in terms of physical resources and having the capability to execute multiple jobs simultaneously. However, job allocation is often overlooked in existing RL-based schedulers that mainly focus on selecting suitable jobs from the job queue and leave allocation to overly simplistic policies, such as First-available allocation. The bin-packing nature at the node level of modern HPC necessitates more refined and intelligent allocation strategies. This chapter introduces HeraSched, a novel Hierarchical Reinforcement Learning (HRL)-based scheduler, adept at intelligent job selection with integrated backfilling and *heterogeneity-aware allocation, tailored for modern HPC environments. It efficiently manages diverse workloads across CPU and GPU cluster partitions. We evaluate HeraSched using real-world workloads, demonstrating significant improvements in reducing job waiting times and preventing job starvation compared to 27 scheduling combinations. In validation, the best maximum waiting time among compared methods is 78% higher than HeraSched's result in overloaded CPU partitions. This performance demonstrates HeraSched's ability to manage intensely stressed workloads and adapt to previously unseen, high-demand scenarios, thereby establishing a new standard in HPC job scheduling.*

This chapter is derived from the following publication:

4.1 Introduction

HPC scheduling involves two steps: *job selection* and *job allocation*. Job selection refers to choosing jobs from the waiting job queue, while job allocation involves assigning the necessary resources to the selected jobs for execution. A common solution in HPC scheduling involves using heuristic-based techniques for both job selection and job allocation, with each process generally handled independently [126]. For instance, an FCFS job selector might be paired with a first-available job allocator, assigning the longest-waiting job to the first available resource that can accommodate it. Recent advances in HPC systems have introduced new challenges in scheduling. The increasing complexity and diversity of workloads, which now include a wide range of applications with distinct computational demands and the integration of new resources like GPUs and accelerators [31], make heuristic-based scheduling methods increasingly inadequate, as they rely on a fixed set of predefined rules that fail to capture the complexity, diversity, and dynamic nature of the underlying environment. Consequently, there is a pressing need for *intelligent job selection and allocation strategies* that can effectively manage the workload complexity and resource diversity in contemporary HPC systems.

RL [38] is a promising method for addressing the challenges of HPC job selection and allocation. Its ability to learn and adapt in complex environments makes it suitable for optimizing these processes in dynamic HPC systems. While research on RL-based approaches for HPC scheduling has intensified in recent years, the majority of existing RL-based schedulers focus on job selection, often relegating job allocation to simple heuristics [24–28]. By combining RL-based job selectors and heuristic-based allocators, these schedulers prioritize enhancing job selection techniques to optimize system goals rather than addressing the complexities of job allocation. Such an approach is well-suited to HPC environments where the compute resources of nodes are limited and homogeneous (e.g., single CPU core per node), as variations in job-to-node allocations are likely to have minimal impact on overall system performance. In fact, the focus of existing RL-based schedulers on these types of HPC environments is evidenced by their use of workloads defined by the Standard Workload Format (SWF) [30]. However, the SWF is increasingly falling short of capturing all the necessary aspects of modern HPC structures and

- **Lingfei Wang**, Maria A. Rodriguez, and Nir Lipovetzky. Optimizing HPC Scheduling: A Hierarchical Reinforcement Learning Approach for Intelligent Job Selection and Allocation. *The Journal of Supercomputing* 81, no. 8 (2025): 918.

workloads. A significant limitation of SWF is its inability to account for resources beyond processors, such as GPUs and other accelerators. Additionally, it fails to capture node-level job requirements, such as the 'Requested Number of Nodes,' which are crucial for accurately selecting and allocating resources in contemporary HPC systems. Thus, to effectively manage modern HPC systems, we need not only a workload model that accurately reflects their complexities but also advanced job selection and resource allocation strategies, especially as these systems evolve to include complex node configurations and diverse resources such as GPUs and other accelerators.

Building on the discussion above, Hierarchical Reinforcement Learning (HRL) offers a powerful framework for addressing the complexities of HPC scheduling. HRL effectively separates scheduling tasks into distinct selection and allocation layers within a hierarchy, decomposing the problem into smaller subtasks, which reduces the size of the action space and simplifies decision-making. At the higher level, an HRL model can strategically prioritize job selection from the queue, aligning with global objectives like minimizing wait times or maximizing system throughput. Simultaneously, a lower-level HRL model can focus on the critical task of resource allocation, precisely assigning jobs to specific nodes while accounting for the various HPC hardware, including GPUs, CPUs, and memory. Also, HRL's capacity to coordinate multiple decision-making processes within a single framework is crucial for ensuring that job selection and allocation are not treated as isolated tasks but as components of a cohesive strategy. This approach allows HRL to optimize the overall performance of the HPC system.

In this chapter, we introduce **HeraSched**, a novel HRL-based approach tailored for intelligent job selection and node-level allocation in modern HPC systems. Building on an HPC simulation that leverages real trace data and system configurations from operating HPC systems based on Slurm [35], HeraSched is trained under conditions that closely mimic real-world scenarios, incorporating comprehensive information to reflect the complexities of modern HPC environments. Specifically designed to navigate the complexities of job scheduling in HPC environments, HeraSched addresses the challenges posed by heterogeneous CPU and GPU partitions. Through the application of two real-world HPC workloads, we show HeraSched's effectiveness in significantly optimizing average waiting times for jobs while preventing job starvation. Our approach integrates backfilling in job selection and exhibits an understanding of cluster heterogeneity for efficient job scheduling. The primary contributions of this chapter are:

- HeraSched represents an application of HRL in HPC job scheduling. To the best of our knowledge, it is the first system to employ HRL agents for intelligent job selection and node-level allocation in a heterogeneous HPC environment.
- We evaluate HeraSched on two distinct workloads from CPU and GPU partitions, confirming its scheduling ability in various HPC environments. This assessment highlights its success in integrating backfilling within job selection and its effectiveness in heterogeneous-aware allocation, demonstrating HeraSched's robust capabilities in practical HPC job scheduling scenarios.
- We design distinct reward mechanisms for the selector and allocator that are implemented, addressing their unique challenges and optimizing their performance in the HPC scheduling context.

4.2 Related Work

HPC job scheduling fundamentally comprises two critical components: job selection and job allocation. Job selection involves choosing appropriate jobs from the queue for execution. Job allocation deals with assigning the selected jobs to suitable computing resources. Job selection approaches initially centered around priority-based heuristics [5, 127]. As computational demands and scheduling complexity grew, meta-heuristic approaches emerged [14, 15], offering more sophisticated strategies for job selection. As the artificial intelligence developing, the focus has shifted towards scheduling approaches leveraging machine learning [16–18] and RL [20, 22–28, 104, 105]. However, most existing RL-based approaches primarily focus on solving the job selection process, overlooking the crucial aspect of job allocation. For example, DeepRM [20], Decima [23], RLScheduler [24], RLSchert [25], DRAS [26], and SchedInspector [27] all propose RL-based selectors that are coupled with heuristic-based allocators. Unlike these approaches, CuSH [22] introduces an RL agent capable of selecting jobs from the queue while also choosing between two elementary heuristic-based allocation strategies. While this incorporates allocation into the RL process, the agent's control over allocation decisions remains limited, as it can only make high-level choices between pre-defined strategies. This lack of fine-grained control restricts the potential for optimizing job allocation, potentially leading to suboptimal resource utilization and system inefficiencies in dynamic HPC environments. Additionally,

as HPC systems evolve to include technologies like GPUs and specialized accelerators, RL-driven approaches face challenges in effectively integrating with these modern, heterogeneous architectures. A key limitation of existing related research is their reliance on outdated trace data from SWF [30], which may not accurately reflect the complexities and dynamic nature of current HPC systems.

Research focusing on job allocation has predominantly concentrated on building topology-aware allocation strategies [107, 108, 110]. These strategies aim to minimize communication costs between different parts of the HPC system, often employing heuristic methods to achieve this objective. Some developments in this field target minimizing system performance [13, 113] like job runtime and makespan, while others target minimizing energy consumption [118–121]. However, the majority of allocation methods employ static strategies that do not dynamically adapt to fluctuating workloads and changing resource availability in real-time HPC environments. This static nature can result in suboptimal performance under varying operational conditions. Moreover, a notable gap in these allocation approaches is their frequent discussion as independent methods, often paired with simplistic selectors, such as FCFS. This disconnect highlights the need for more integrated approaches that consider both dynamic allocation strategies and dynamic scheduling methodologies to fully harness the capabilities of modern HPC systems.

4.3 HeraSched Design

In this section, we outline the design of the proposed HRL Scheduler, named HeraSched. Firstly, how HRL applying to HPC scheduling is presented in Section 4.3.1. Then, Section 4.3.2 provides an overview of HeraSched, introducing its role and functionality within the HPC environment and describing the job scheduling process it orchestrates. Following this, the specifics of the HRL framework are presented, including HeraSched's states (Section 4.3.3), actions (Section 4.3.4), and the construction of its reward function (Section 4.3.5). Finally, in Section 4.3.6 we discuss the architecture of the HRL agents and the underlying algorithm that governs HeraSched's operation, thereby offering a comprehensive view of its design.

4.3.1 Hierarchical Reinforcement Learning in HPC Scheduling

HRL simplifies complex tasks by breaking them into smaller subtasks using a hierarchy of learning policies. The highest-level policy selects subtasks from the main task and learns to sequence them effectively, using the main task's rewards [128]. Each chosen subtask forms its reinforcement learning challenge at lower levels. These lower-level policies focus on accomplishing the subtasks, guided by specific internal rewards [129] and, optionally, by rewards from the main task. The lowest level in the hierarchy involves policies selecting basic, or primitive, actions to execute these subtasks.

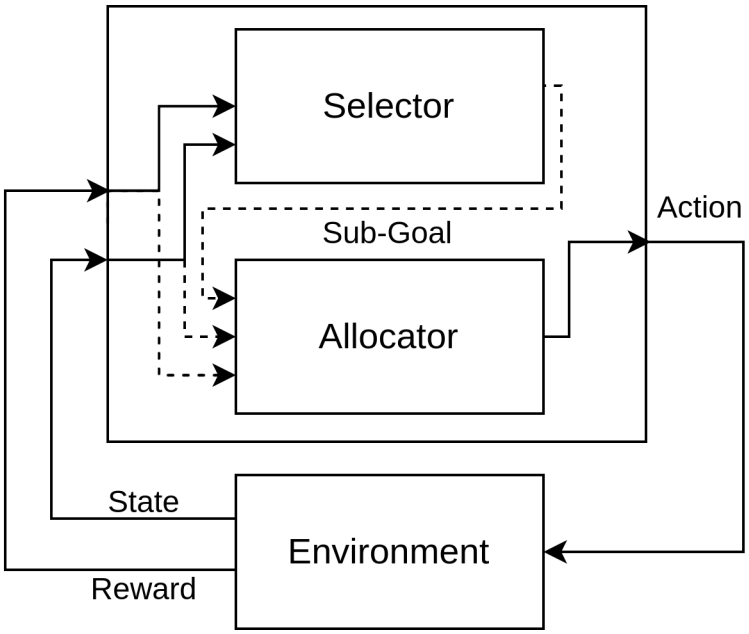

FIGURE 4.1: Framework of HRL with an Emphasis on the Integration of Selection and Allocation Processes

Fig. 4.1 presents a framework of HRL and its application in the HPC job scheduling process. The HPC job scheduling problem is decomposed into two tasks: job selection and job allocation. At the higher level, the job *selector* chooses an appropriate job from the job queue based on the current state of the HPC system (environment). The selected job becomes a sub-goal for the *allocator*, functioning as the low-level agent. The allocator's sub-goal is explicitly defined as 'identifying the most suitable node or nodes for executing the selected job'. To achieve this, the allocator evaluates the current state of the HPC system in conjunction with the defined sub-goal and determines the optimal allocation of the selected job, which constitutes its action. Once this action is implemented within the HPC system environment, the selector and the allocator, receive feedback (rewards).

These rewards are contingent on the efficacy of their actions. Following this process, the agents find themselves in a new state, ready to undertake the next iteration of decision-making. This cyclical process exemplifies the dynamic and interactive nature of HRL within the realm of HPC job scheduling.

4.3.2 Overview of HeraSched

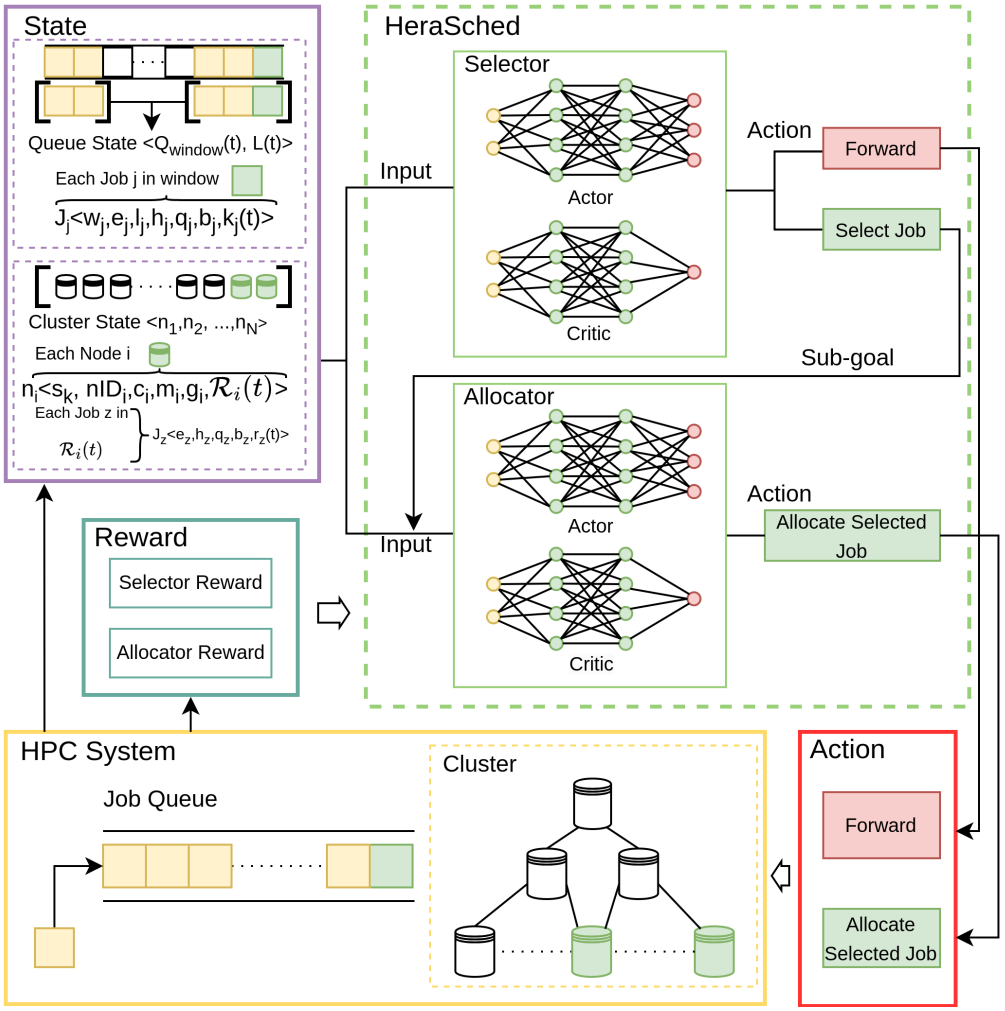

FIGURE 4.2: Overview of HeraSched

Fig. 4.2 illustrates an overview of HeraSched, showcasing its major components along with the scheduling process. The HPC system serves as the environment for HeraSched. The scheduling process is initiated either by the submission of a new job or the completion of a running job. As the process begins, HeraSched receives the current state of the job queue and the cluster from the environment as its input. The hierarchical structure of HeraSched dictates that the selector component acts first, making decisions based

on the state. The selector has two primary choices: it can select a specific job or opt for a 'Forward' action. The 'Forward' action is chosen when the agent decides not to schedule any job in the current scheduling process and moves to the next scheduling process triggered by the next job arrival or the completion of a running job. When the selector chooses a job, this selected job becomes a sub-goal for the allocator and is incorporated into the allocator's state as input. The allocator's responsibility is to sequentially identify appropriate nodes for the selected job, a process that may entail multiple steps to fully meet the job's node requirements. Once the allocation for the chosen job is determined, the job is placed in the cluster for execution. The rewards for the HeraSched agents, based on the reward functions presented in Section 4.3.5, are issued after the actions have been executed.

Importantly, the completion of a job allocation does not end the current scheduling cycle. Instead, after allocating the selected job, HeraSched receives an updated system state and can continue scheduling jobs within the same cycle if resources permit. This means that a single scheduling process can allocate multiple jobs, depending on the availability of nodes and the job queue. The process only concludes when HeraSched determines that no further jobs should be scheduled and selects the 'Forward' action, or when no further jobs can be scheduled in the current state. Noticeably, Backfilling in HPC job scheduling, developed as a strategy to optimize system performance, involves executing smaller or shorter jobs that fit into scheduling gaps created by larger jobs awaiting resources. Recent advancements in RL-based schedulers follow this idea to incorporate the concept of backfilling as a complementary strategy. Alongside the primary RL-based job selection mechanism, these scheduling systems typically have a separate backfilling scheduler. Their backfilling schedulers are either based on heuristics [24, 27, 28] or dedicated RL agents trained specifically for backfilling tasks [26, 29]. However, a separate backfilling mechanism is not needed for intelligent AI-based schedulers. When a scheduler, particularly an RL scheduler, comprehensively understands the cluster's state and the awaiting jobs, it can effectively manage job selection and allocation. This allows the traditional concept of backfilling as a separate process to become integrated into the selector's core functionality, where the selector's decision-making mechanism inherently incorporates identifying and exploiting opportunities for optimal scheduling goals. This methodology exemplifies the concept of Intelligent Integrated Backfilling. We design our HeraSched integrating the

principles of Intelligent Integrated Backfilling for enhanced system efficiency and effectiveness.

4.3.3 State Design

In HeraSched's design, the state space plays a key role as it defines the information accessible to the agents for decision-making. This includes details about the job queue status, resource availability, and the currently running jobs in the cluster. The state of the HPC system is divided into two main components: the job queue state and the cluster state.

4.3.3.1 Job Queue State

In HPC systems, the job queues are characteristically unbounded, allowing for an indefinite number of job submissions. This feature leads to a potentially arbitrary decision space for job selection, as the queue can continuously expand in both size and diversity of jobs. A common approach to managing unbounded queues in existing RL-based schedulers is to use a 'window' queue, where the agents only observe a fixed number of jobs at the head of the queue. However, this method often leads to 'large' jobs, which require more resources, accumulating at the queue's head. Consequently, newer jobs, especially those requiring fewer resources, may have to wait a considerable amount of time before being observed by the agents, decreasing system performance. To overcome this limitation, HeraSched incorporates a 'split window' technique, as previously proposed in 3.3.1. This approach allows agents to observe jobs from both the head and the tail of the queue, as shown in Fig. 4.2. By doing so, agents gain visibility of every job as soon as it arrives at the tail of the queue, ensuring a continuous influx of fresh candidates for selection and backfilling. This strategy prevents the agents from being constrained by a lack of new job options and enhances their ability to make more informed and diverse scheduling decisions.

In the job queue state, job J_j is represented as a vector consisting of the attributes, $\langle w_j, e_j, l_j, h_j, q_j, b_j, k_j(t) \rangle$ (notations in Table 2.1). A notable element within this vector is the parameter $k_j(t)$, which serves as a binary indicator of whether the job J_j can be executed given the current resource of the cluster. This parameter significantly reduces the learning time for HeraSched. The total number of jobs that HeraSched can observe is the

'Window Size'. The 'Tail Size' defines how many jobs in the window are gathering from the tail of the queue. 'Window Size' and 'Tail Size' are hyperparameters that can be fine-tuned to optimize HeraSched's performance in different HPC scheduling environments. If the job queue contains fewer jobs than the designated Window Size, it is padded with zero vectors. Then, a number $L(t)$, indicating the current queue length, is incorporated into the job queue state. This addition provides the agents with a sense of the actual workload in the queue when some jobs cannot be observed.

4.3.3.2 Cluster State

The cluster state is a comprehensive collection of all nodes' statuses, providing information about node identity, resource availability, and the status of running jobs. For each node, n_i, the state is defined as $\langle s_k, c_i, m_i, g_i, \mathcal{R}_i(t) \rangle$ (notations in Table 2.1) where s_k represents the switch ID connected to the node, and $\mathcal{R}_i(t)$ is a collection of running jobs at time t. The details of running job J_z within $\mathcal{R}_i(t)$ are encapsulated in a vector $\langle e_z, h_z, q_z, b_z, r_z(t) \rangle$, representing requested runtime, number of occupied cores, occupied memory, number of GPUs occupied, and the current runtime, respectively. One of the key insights gained from the running job information is the prediction of future resource availability. By analyzing the runtime of each job and its maximum requested time, the scheduler can anticipate when resources will be free for use by other jobs.

Designing the cluster state, however, is challenged when the number of running jobs on a node is dynamic, coupled with the need for a predefined cluster state structure. To address this, we introduce the concept of 'Job Placeholders' in the cluster state. These placeholders reserve space for potential jobs, ensuring a fixed size for $\mathcal{R}_i(t)$ despite the dynamic job activity. Each Job Placeholder is a zero vector. If a node is idle, the number of Job Placeholders equals the maximum number of jobs that can run in the node. The maximum number of running jobs for CPU partition nodes is the number of CPU cores c_i and the maximum number of running jobs for GPU partition nodes is the number of GPUs g_i, as the minimum occupation unit of critical resource is 1 per job. This allows HeraSched to maintain a consistent state representation while dynamically adapting to the fluctuating job activities within the cluster.

In HeraSched, the state input for the selector is a combination of the job queue state and the cluster state. As depicted in Fig. 4.2, when the selector chooses a job for allocation,

the index of that job within the window is incorporated into the allocator's state with the job queue state and the cluster state.

4.3.4 Action Design

The action space for the selector agent in HeraSched is determined by the Window Size, with an additional 'Forward' action. This configuration means the selector has a range of choices equal to the number of jobs it can observe based on the Window Size. With the flexibility provided by the Window Size and Tail Size settings, the selector can choose jobs from both the head and the tail of the job queue. This capability is crucial for optimizing system performance. The 'Forward' action plays a unique role in the scheduling process. When selected, it instructs HeraSched not to schedule any job and to allow the system to continue running as is until the next scheduling cycle. Through the 'Forward' action, the selector can bypass immediate scheduling in favor of waiting for potentially more beneficial conditions or jobs in the near future.

The allocator's action space in HeraSched consists of all the computing nodes in the cluster. When allocating resources for a specific job, the allocator selects one node at a time based on the job's required resources and the availability of resources within each node. If a job requires multiple nodes, the allocator will continue selecting one node per step until the job's requirements are fully satisfied. This step-by-step approach, rather than outputting all allocated nodes in a single action, is designed to reduce the dimensionality of the action space. By abstracting the allocation process to the node level, HeraSched simplifies the decision-making process, reduces the complexity of its action space, and remains effective in environments with various resources, all while ensuring that the jobs' resource requirements are accurately met.

4.3.5 Reward Mechanisms

In our design, we choose to focus on both **average waiting time** and **maximum waiting time**, valuing these two objectives as equally important. While these objectives can sometimes conflict, our aim is to reduce average waiting time while maintaining a low maximum waiting time to prevent job starvation. To achieve this balance, we have designed a reward function that effectively integrates both metrics. In traditional RL

systems, rewards are typically given on a step-by-step basis. However, such an immediate reward in HPC systems might inadvertently lead to suboptimal decisions focusing on reducing waiting times in the immediate term rather than optimizing them over a longer period. To mitigate this, we have adopted a delayed reward for HeraSched. Under this approach, rewards are accumulated and then distributed to the agents after a defined period, taking into account the cumulative impact of all actions made by the agents during that interval. This delayed reward mechanism aligns more closely with our long-term objective. By implementing a delayed reward system, HeraSched's agents are incentivized to make decisions that are beneficial not just in the immediate context but also contribute positively to the overall scheduling performance over time.

The reward for the selector is structured around the number of actions it takes, denoted as A_s. The selector is assigned a reward after it has executed a set of A_s actions. This reward is calculated as a negative value, reflecting the average waiting time of the jobs scheduled during these actions, combined with the waiting times of jobs currently in the queue. The reward function of the selector at system time t is:

$$\text{Reward}_s = \begin{cases} -\frac{\sum_j^{\mathcal{Q}_s}(d_j - w_j) + \sum_j^{\mathcal{Q}(t)}(t - w_j)}{\sum_j^{\mathcal{Q}_s} 1 + L(t)} & \text{After every } A_s \text{ selector's actions} \\ 0 & \text{Otherwise} \end{cases} \tag{4.1}$$

$\mathcal{Q}_s$ represents the collection of jobs that have been scheduled as a result of the selector's actions. When A_s actions have been made, the selector receives a negative reward, and $\mathcal{Q}_s$ is reset.

Unlike the selector, the allocator receives a negative reward after allocating A_a jobs rather than based on a fixed number of actions. The allocation process for a job in an HPC system can vary significantly, depending on the job's specific requirements. Some jobs may require multiple nodes, while others might need only one, leading to a variation in the number of steps required to allocate each job. As a result, a reward system based on the number of steps taken would not accurately reflect the efficiency or effectiveness of its decisions. Utilizing $\mathcal{Q}_a$ as the collection of jobs allocated by the allocator, the size of this collection is set to A_a jobs. After the allocator receives a negative reward, $\mathcal{Q}_a$ is

reset.

$$\text{Reward}_a = \begin{cases} -\frac{\sum_j^{\mathcal{Q}_a}(d_j - w_j) + \sum_j^{\mathcal{Q}(t)}(t - w_j)}{A_a + L(t)} & \text{After every } A_a \text{ jobs allocated} \\ 0 & \text{Otherwise} \end{cases} \tag{4.2}$$

The selector's rewards are based on the number of steps taken because the selector has a 'Forward' action. If the rewards only count the number of jobs selected, the impact of choosing 'Forward' would be neglected. The overarching aim of both the selector's and allocator's reward systems is to reduce the average waiting time. This common goal ensures that both components contribute to the overall scheduling efficiency, each from its unique operational perspective.

4.3.6 Algorithms

A common challenge in RL-based systems is the possibility of agents selecting invalid actions. In the specific context of HeraSched, these invalid actions might involve choosing a non-existent job or a job that cannot run due to insufficient resources for the selector. Similarly, for the allocator, an invalid action might be selecting a node that lacks the necessary resources for the chosen job. To eliminate the invalid actions, we implement a technique known as Invalid Action Masking [130]. This approach effectively filters out these invalid choices, preventing the agents from selecting them. This method enhances the efficiency of the learning process, as it guides the agents towards valid actions while reducing the time and computational resources spent on exploring unfeasible options.

HeraSched employs PPO [40] as its underlying Reinforcement Learning algorithm. Algorithm 2 presents the pseudocode of HeraSched. The update mechanism for the selector and allocator networks in HeraSched is structured around the selector action step count reaching a predetermined update threshold, denoted as x, rather than waiting for the end of a trajectory collection. This method allows HeraSched's agents to update their networks continuously as the system operates. This update mechanism is a significant step towards future implementations of HeraSched in real online scheduling environments.

Algorithm 2 HeraSched Scheduling Algorithm

Input: HPC job scheduling environment, policy networks for Selector and Allocator

Output: Scheduled jobs and resource allocations

Initialize policy and value networks for Selector and Allocator Set update threshold x

while *training not complete* **do**
    Reset buffers and selector step count
    foreach *Scheduling process* **do**
        Observe system state
        Apply action mask to Selector
        Selector chooses an action: select job or Forward
        if *a job is selected* **then**
            while *allocation requirements are not met* **do**
                Apply action mask to Allocator
                Allocator selects a node
            end
            Allocate job to selected node(s)
        end
        else
            Continue to the next scheduling process
        end
        Store states, actions, and rewards for Selector and Allocator
        Increment selector step count
        if *selector step count* $\geq x$ **then**
            Update Selector and Allocator networks
            Reset selector step count
        end
    end
    Reset environment if necessary
end

4.4 Evaluation and Results

4.4.1 Evaluation Workloads

We collected data from two partitions of Spartan[1]. These partitions, referred to as 'Physical' and 'Deeplearn', represent different computational environments within the Spartan HPC system. Table 4.1 presents the key characteristics of the Physical and Deeplearn partitions, highlighting their heterogeneity. The Physical partition comprises two types of nodes, each equipped with 72 cores but differing in memory capacity: one with 710 GB and the other with 1519 GB, the latter nodes being more suitable for memory-intensive jobs. The Deeplearn partition features five distinct node types, varying in both core count and memory capacity. A notable configuration within this partition is the 32-core node with 1000 GB of memory, specifically optimized for memory-intensive tasks. Additionally, each node in the Deeplearn partition is uniformly equipped with 4 GPUs. Users have the flexibility to submit jobs to the partitions, depending on their specific computational requirements. The maximum allowable running time for any submitted job is capped at 30 days. The full trace dataset used in this study can be found in the GitLab repository[2].

TABLE 4.1: Workloads: Physical and Deeplearn Characteristics

Name	Physical		Deeplearn				
Partition	CPU		GPU				
Cores/node	72	72	28		24	32	
Mem(GB)/node	710	1519	234	174	175	234	1000
Nodes	72	14	4	5	3	6	12
GPUs/node	0		4				
Period	2022-9-23 to 2022-9-30		2021-9-20 to 2022-9-30				
Max Runtime	30 days						
Jobs	84135		68720				
Low level Switch	4		4				
Total Cores	6192		900				
Total GPUs	0		120				

Fig. 4.3 offers insightful observations into the job arrival patterns and resource request characteristics. Subplots 4.3-(A) and 4.3-(B) show the distribution of job arrivals in the Physical and Deeplearn partitions. The job arrivals are subject to fluctuating peaks, with periods of high and low job submissions. Notably, the Physical partition consistently

[1] https://dashboard.hpc.unimelb.edu.au/.
[2] https://gitlab.unimelb.edu.au/lingfeiw/herasched.git.

experiences a higher volume of job submissions compared to the Deeplearn partition. Subplots 4.3-(C) and 4.3-(D) further demonstrate the resource requests in terms of nodes, CPUs, and GPUs for jobs in these partitions. It is observed that jobs in the Physical partition tend to request a greater number of nodes and cores than jobs in the Deeplearn partition. Some of these jobs even demand more than 15 nodes and hundreds of cores, indicative of their extensive computational requirements. Additionally, the accompanying CDF in subplot 4.3-(C) reveals a significant trend: 71.5% of jobs in the Physical partition request only 1 core. Tirmazi et al. [31] describe this phenomenon in CPU clusters as 'a very heavy-tailed distribution of job sizes' where a small portion of jobs consume most resources and a big portion of jobs request fewer resources. Conversely, GPU-oriented jobs in the Deeplearn partition, largely driven by machine learning applications, exhibit a different pattern. These jobs generally require substantial resources within a single node, often demanding multiple cores and GPUs. While they are less distributed across nodes, indicating a tendency to concentrate resources within fewer nodes, they exhibit a broader distribution across GPUs. This distinct resource allocation trend between CPU and GPU jobs in the Spartan HPC system presents unique challenges to HeraSched in effectively scheduling jobs across both CPU and GPU partitions.

The interconnect model used in our experiments. We use the real configuration of Spartan and emulate its two-tier (leaf–spine / fat-tree-style) switch hierarchy at the rack level. Nodes are grouped under top-of-rack (leaf) switches, and a topology file specifies the node–leaf mapping used by topology-aware placement. Spartan operates an Ethernet fabric with RoCE, with recent upgrades toward 400 Gb/s switching; our evaluation keeps runtimes fixed from the traces, so the interconnect influences only placement heuristics rather than per-job runtime.

4.4.2 HPC Simulation

To accurately replicate the dynamics of a real-world HPC system, we have developed a simulation framework named HPCsim. This simulation emulates the functionalities of modern workload managers, such as Slurm, by processing real trace and accounting data. HPCsim includes a Gymnasium [131] interface used in RL training and allows us to assess the performance of HeraSched under varied conditions and resource requirements. The simulation comprises six key components: Main Event-based Environment, Cluster

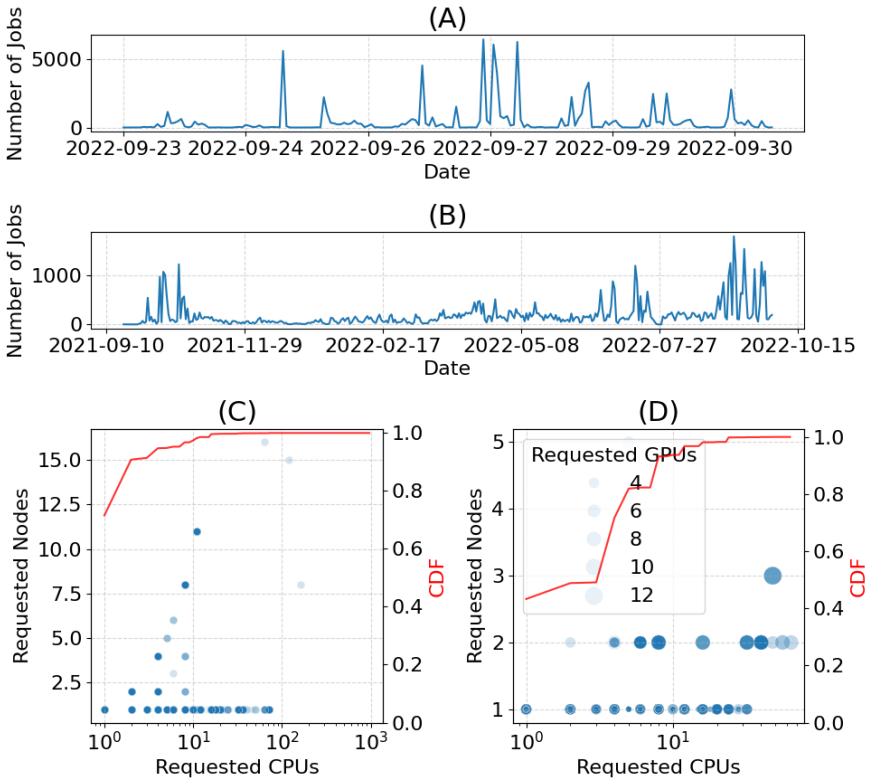

FIGURE 4.3: (A) Number of Jobs Received Over Time in the Physical Partition. (B) Number of Jobs Received Over Time in the Deeplearn Partition. (C) and (D) are Resource Request Patterns for jobs in the Physical and Deeplearn Partition respectively. The CDF line (right y-axis) indicates the cumulative distribution of CPU requests. The color reflects the number of jobs. The darker the color, the more the jobs. The sizes in (D) indicate the number of requested GPUs.

Simulator, Queue Simulator, Scheduler, Trace Reader, and Evaluator. The source code of HPCsim can be found in the GitLab repository[2].

The Cluster Simulator utilizes two primary sets of information: network topology, which defines the interconnects of nodes and switches, and node configuration, which details all resources such as cores, GPUs, and memory for each node. With this information, the Cluster Simulator can accurately structure a cluster mirroring a real-world cluster. It is responsible for allocating jobs to specific nodes and simulating the execution of these jobs. The Queue Simulator functions as the job queue in the system, where jobs are submitted and wait until they are scheduled. The Scheduler is divided into two parts: the selector and the allocator. The selector selects the most suitable jobs based on its rules, considering the current job queue and cluster state. The allocator, on the other hand, finds the appropriate nodes for the selected job, ensuring successful job execution.

[2] https://gitlab.unimelb.edu.au/lingfeiw/herasched.git.

The Trace Reader serves as the job pool, capable of reading jobs from real trace data or generating random jobs when random job generation is enabled. It simulates the job submission process to the job queue. The Evaluator collects system information, such as job waiting times and completion times, to compute system metrics.

The Main Event-based Environment, integrated with the Gymnasium interface, acts as the central controller of the entire simulation. It manages every process and interacts with all the aforementioned components to simulate real HPC operations. As an event-based environment, it recognizes two types of events that trigger processes: job arrivals and job completions. Both events signal the start of scheduling processes. Additionally, this environment supports HRL agents, including HeraSched's selector and allocator, by providing necessary observations for each and processing their actions accordingly.

4.4.3 Baseline Methods

In the HeraSched evaluation, we employ a set of established HPC job scheduling and allocation policies as baselines for benchmarking its performance. The compared job selectors include traditional algorithms, which are **FCFS** (First-Come First-Served), **LCFS** (Last-Come First-Served), **SJF** (Shortest Job First), **WFP3** [5], and **UNICEP** [5]. We incorporate two machine learning-based selectors, **F1** [16] and **F2** [16], to assess the impact of advanced, data-driven scheduling strategies. We include two RL schedulers as state-of-the-art, reproducible anchors: **RLScheduler** and **DRAS**. Both can be integrated into our simulator without changing their fundamental design. This makes them suitable RL-to-RL comparators alongside heuristic and ML baselines. **RLScheduler** [24] and **DRAS** [26]. However, as RLScheduler and DRAS are not GPU-aware scheduling, the two RL-based selectors are only trained and compared in CPU partition jobs. Specifically, RLScheduler was trained with our CPU jobs for ten million timesteps (stable-baselines3 version), and DRAS was trained for 100 episodes. Both of them were trained using their best-performing parameter values as presented by the authors in their corresponding papers. Furthermore, all selection methods are supplemented with a '**Backfilling**' [96] strategy, except DRAS which has its own backfiller.

We incorporate three distinct allocation methods with the above job selection strategies to assess the comprehensive performance of HeraSched. **First-Available** allocates jobs to the available nodes in the cluster. It is a straightforward strategy that prioritizes

immediate resource availability. **Best-Fit** focuses on allocating jobs to nodes that best meet their specific resource requirements. This method aims to find the nodes with the smallest available resources that can satisfactorily meet the job requirements to minimize the amount of unused resources. **Topology-Aware** [110] is designed to consider the physical and network layout of the system, focusing on optimizing job placement based on the connectivity and proximity of nodes. The method identifies the lowest-level network switch capable of handling the job's node requirements. Then, it applies a best-fit approach to allocate the available nodes under this switch, optimizing resource use.

4.4.4 HRL Scheduler Training

We developed two implementations of HeraSched for the Physical and Deeplearn partitions, taking into account the fundamental differences in their characteristics. To achieve HRL training, we designed an HRL-compatible PPO structure based on the stable-baselines3's PPO implementation [132]. This approach retains the core learning process of PPO from stable-baselines3, ensuring that our implementation is both comparable and reliable. The implementations are trained by the workloads described in 4.4.1 respectively. Given compute constraints, we adopted a targeted manual tuning protocol. We started from stable PPO defaults, ran brief pilot runs on a held-out validation slice to screen configurations, and then performed a small hand grid over the most sensitive knobs—learning rate, rollout length, batch size, policy clipping and entropy regularization, and network width and depth—while keeping other settings at conservative defaults. The selection criterion was the primary objective defined in this chapter. Promising settings were confirmed with longer runs; where feasible we verified with multiple seeds, otherwise a fixed seed was used for reproducibility. Where applicable, we also adjusted window and segment lengths to balance feedback timeliness and variance. The final configurations used in the evaluation are those reported in Table 4.2, with additional implementation details provided in the appendix/code repository. HeraSched is not supervised on historical allocations, and we do not attempt to infer counterfactual placements from logs collected under a single production policy. Instead, the agent learns by interacting with environment, which replays observed arrivals, resource requests, and durations while permitting the allocator to choose any valid node set at each decision point. The simulator deterministically advances the queue and cluster state under these counterfactual choices, revealing long-horizon consequences (e.g., future feasibility and backfilling opportunities) that are

TABLE 4.2: HeraSched Implementation

Partition	Physical		Deeplearn	
Window size	512		100	
Tail size	64		10	
A_s	512		512	
A_a	100		100	
Update per Selector Steps	2048		2048	
Hidden Layers	[4096, 2048, 1024]		[2048, 1024]	
Discount Factor	1		1	
Activation Function	Tanh		Tanh	
Batch Size	64		64	
	Selector	Allocator	Selector	Allocator
State Size (Input)	35340	35341	1451	1452
Action Size	513	86	101	30

unobservable in the logs. Following standard trace-driven evaluation, job runtimes are taken as fixed by the trace, so our objectives emphasize waiting-time metrics (average and maximum) that remain comparable across alternative allocation policies. The full implementation of HeraSched can be found in GitLab repository[2].

Fig. 4.4 shows the patterns in training HeraSched with full workloads presented in 4.4.1 respectively. In Fig. 4.4, 4.4-(A) and 4.4-(D) are the plots for selectors training indicating the timesteps the model trained versus the average return per episode in Physical and Deeplearn partitions respectively; 4.4-(B) and 4.4-(E) demonstrate the average episode length for selectors in the training. Since the selector has a 'Forward' action, the episode length changes with the agent's exploration when jobs per episode are fixed; 4.4-(C) and 4.4-(F) are training patterns for allocators with different workloads.

In the selector training for Physical partition jobs, the return experienced fluctuations in the early stage. The episode length can explain the fluctuations. Higher episode length means the selector chose more 'Forward' actions which can cause resource waste leading to a performance drop. After 10 million timesteps of training, the episode length became stable and the best performance of the selector was reached. The patterns for allocator training in 4.4-(C) shared a similar shape with 4.4-(A) as the rewards for the selector and allocator are the same system performance. Notably, the timesteps taken by the selector and allocator are different. Sometimes, the selector takes more steps to finish an episode when it chooses too many 'Forward' actions. Also, the allocator can take more actions when jobs request more nodes.

[2] https://gitlab.unimelb.edu.au/lingfeiw/herasched.git.

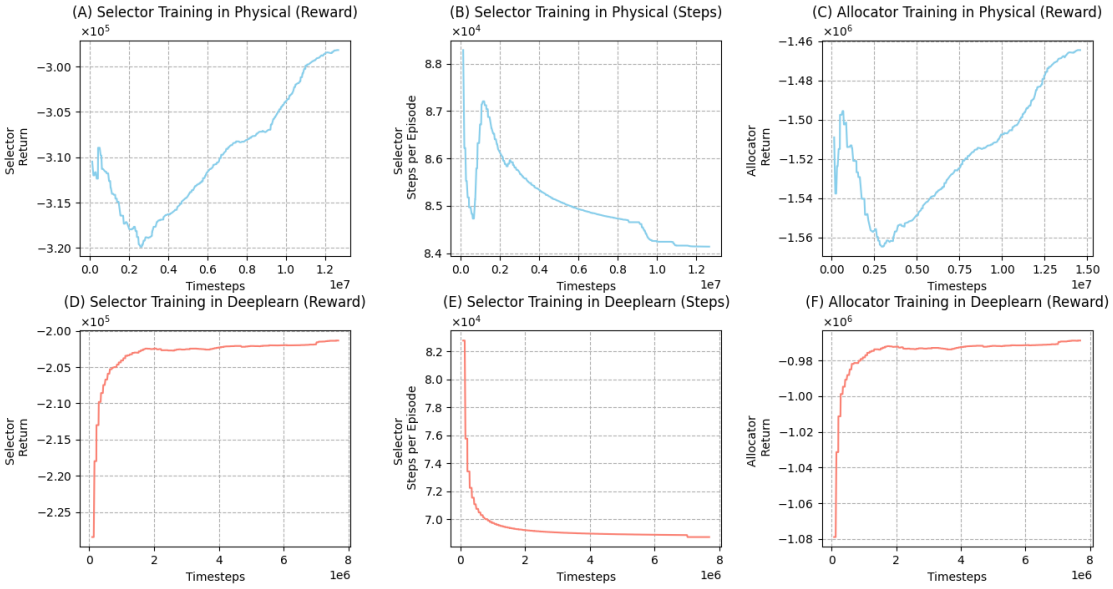

FIGURE 4.4: The training plots of HeraSched in Physical and Deeplearn partitions

For training in Deeplearn partition jobs, the system performance increased quickly in early training. Then, the performance gradually increased with the agents' exploration. The Physical partition job training for HeraSched took more timesteps since the Physical partition contains more nodes and is busier than the Deeplearn partition.

4.4.5 Performance Comparison

We thoroughly evaluated HeraSched's performance by comparing it with 9 different selectors, each coupled with a separate Backfilling, and combined with 3 allocators as outlined in 4.4.3. This resulted in a total of 27 unique scheduling methods for comparison (RLScheduler and DRAS only support CPU partition job scheduling). The objective of these comparisons is to assess the effectiveness of HeraSched in minimizing the average waiting time for jobs and maintaining a low maximum waiting time to prevent longer, resource-intensive jobs from experiencing undue delays or starvation.

Fig. 4.5 shows the comparative analysis of HeraSched against 27 other scheduling methods across two key metrics: average waiting time and maximum waiting time for jobs. The results are generated by using the corresponding scheduler to schedule entire workloads described in 4.4.1. The values are expressed as ratios compared to HeraSched. Specifically, for each selector, the average and maximum waiting times under three allocation methods are divided by the corresponding values for HeraSched. The ratios are

then adjusted by subtracting 1, so that a value of 0 indicates performance equivalent to HeraSched. Positive values represent worse performance (i.e., longer waiting times) relative to HeraSched, while negative values indicate better performance (i.e., shorter waiting times). The raw values can be found in Appendix B. The results indicate that HeraSched consistently outperforms all other methods in both metrics across the Physical and Deeplearn partitions, affirming its efficiency in job scheduling. The results also reveal variations in performance among the same scheduling methods when paired with different allocation policies. This variation underscores the significant impact allocation policies have on scheduling performance.

FIGURE 4.5: The heatmaps display the performance comparison of combinations of selectors and allocators relative to HeraSched in the Physical and Deeplearn Partitions. The values are ratios compared to HeraSched, with positive values indicating worse performance and negative values indicating better performance. ((value - HeraSched's value) / HeraSched's value)

In both the Physical and Deeplearn Partitions, the First-Available method consistently underperforms, showing higher average and maximum waiting times due to its simplistic, non-adaptive resource allocation. While Topology-Aware and Best-Fit methods offer some improvements by considering system layout and optimizing resource allocation, they still struggle with selectors like FCFS and WFP3. Best-Fit, in particular, improves resource distribution slightly but falls short in handling the complexity of Deeplearn workloads. HeraSched outperforms these methods by dynamically adjusting to job demands and system conditions in real-time. This adaptability allows HeraSched to efficiently handle diverse requirements, leading to significantly reduced waiting times.

The improvements observed in the Deeplearn partition are less pronounced compared to those in the Physical partition. This disparity can be attributed to the lighter workload

in the Deeplearn partition; the volume of jobs it handles in a year is less than what the Physical partition processes in a week. Most methods can maintain a minimal waiting time in low job submission periods (the flat area in Fig. 4.3-(B)) in the Deeplearn partition. The distinctions between these methods become more apparent during peak times. This indicates that while the performance differences are minimal during less busy periods, they are more significant in high job submission periods. Consequently, in less busy environments like the Deeplearn partition, the scope for selectors and allocators to enhance performance is relatively limited. Following this, we conduct a validation under a high load in 4.4.7.

4.4.6 HeraSched Feature Analysis

4.4.6.1 Integrated Backfilling Actions

Fig. 4.6 displays histograms detailing the selectors' actions in the Physical and Deeplearn partitions. The x-axis shows the actual actions taken by selectors according to their action spaces. For example, in Fig. 4.6-(A), the first bin indicating '0' demonstrates how often the selector chooses the first job in the queue for scheduling. The bin indicating '447 - 511' shows the frequency of selecting jobs from the tail of the queue (choosing newly arrived jobs). A notable observation is the selector's preference for scheduling jobs from the front of the queue, a strategy aimed at minimizing job waiting times. The histograms also reveal instances where the selector opts for jobs from the tail of the queue. This pattern of job selection is indicative of the concept of Integrated Backfilling, where HeraSched is capable of autonomously identifying and scheduling the most appropriate jobs, incorporating actions that are characteristic of traditional backfilling. The situations when the selector chooses jobs from the middle of the queue are more complex. The queue is dynamic, meaning its length may change dramatically over time. As a result, the concept of the 'middle' is fluid; what appears to be the middle at one point could shift to the tail or even the head of the queue as jobs enter and leave. The selector can adaptively choose jobs from this shifting 'middle' based on current queue conditions and system goals, balancing immediate scheduling efficiency with the anticipation of future resource availability.

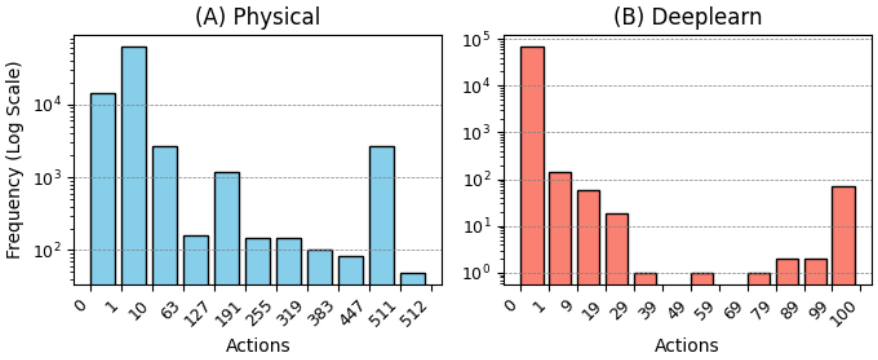

FIGURE 4.6: Histogram of Selector Actions

TABLE 4.3: Summary of Job Allocation in Different Nodes

	Physical		Deeplearn				
Cores/Node	72		28		24	32	
Mem/Node (GB)	710	1519	234	174	175	234	1000
Ave. Job Cores	2.6	2.8	4.4	3.9	3.6	3.9	4
Ave. Job Mem	25.8	35.1	44.2	35.4	37.6	36.8	42

4.4.6.2 Heterogeneity-Aware Allocation

There are two types of nodes in the Physical partition and five in the Deeplearn partition, which presents a complex landscape for job scheduling. Recognizing and leveraging this heterogeneity is critical to intelligent job allocation. Table 4.3 presents the average requested cores and memory for jobs allocated to different types of nodes. In the Physical partition, HeraSched successfully identifies and allocates jobs with higher memory demands to the nodes with larger memory capacities. Similarly, in the Deeplearn partition, HeraSched allocates jobs with lower core demands to nodes with fewer cores and directs more memory-intensive jobs to nodes with greater memory capacities. However, due to the constraint of running a maximum of four jobs per node in the Deeplearn partition, resource bottlenecks are more likely to occur with GPUs rather than with core and memory resources. These results underscore HeraSched's capability to understand and navigate the heterogeneity of the cluster environment, efficiently planning and directing jobs to the most suitable nodes.

4.4.7 Evaluation under High Load Situation

To test HeraSched under high-load conditions, we conducted a stress validation using two particularly busy workloads from the Physical and Deeplearn partitions which are

not included in 4.4.1. These validation sets represent a challenging environment for the scheduler. Specifically, the validation set for the Physical partition comprised an intense load of 52,985 jobs received in a single day, while the Deeplearn partition's validation set contained 1,137 jobs received in a single day. Such volumes provide a robust stress test for HeraSched.

FIGURE 4.7: The heatmaps display the performance comparison of combinations of selectors and allocators relative to HeraSched in high load Validation. The values are ratios compared to HeraSched, with positive values indicating worse performance and negative values indicating better performance. ((value - HeraSched's value) / HeraSched's value)

Fig. 4.7 provides a detailed performance comparison of selectors and allocators relative to HeraSched in Physical Partition and Deeplearn high-load validation sets. In the Physical Partition validation set, the performance of various selectors and allocation methods shows distinct patterns. HeraSched consistently performs better in minimizing maximum waiting times across all allocation methods. FCFS and LCFS perform poorly, particularly under the First-Available method. SJF, WFP3, and UNICEP perform well in average waiting times, with SJF achieving an average waiting time 12% lower than HeraSched. However, they struggle with high maximum waiting times, with the best among them still showing a 114% higher maximum waiting time than HeraSched. F1 and F2 show moderate performance across all methods, with higher waiting times compared to HeraSched, particularly under the First-Available method. Notably, F1 with Best-fit achieves the best maximum waiting time in compared methods but it is still 78% higher than HeraSched. In the Deeplearn Partition validation set, HeraSched consistently outperforms all other methods. LCFS, SJF, and WFP3 show high maximum waiting times under all methods, highlighting their inefficiency in managing deep learning workloads.

FCFS, UNICEP, F1, and F2 exhibit slightly worse performance compared to HeraSched but remain relatively consistent across different allocation methods. The actual waiting time for schedulers can be found in Appendix B.

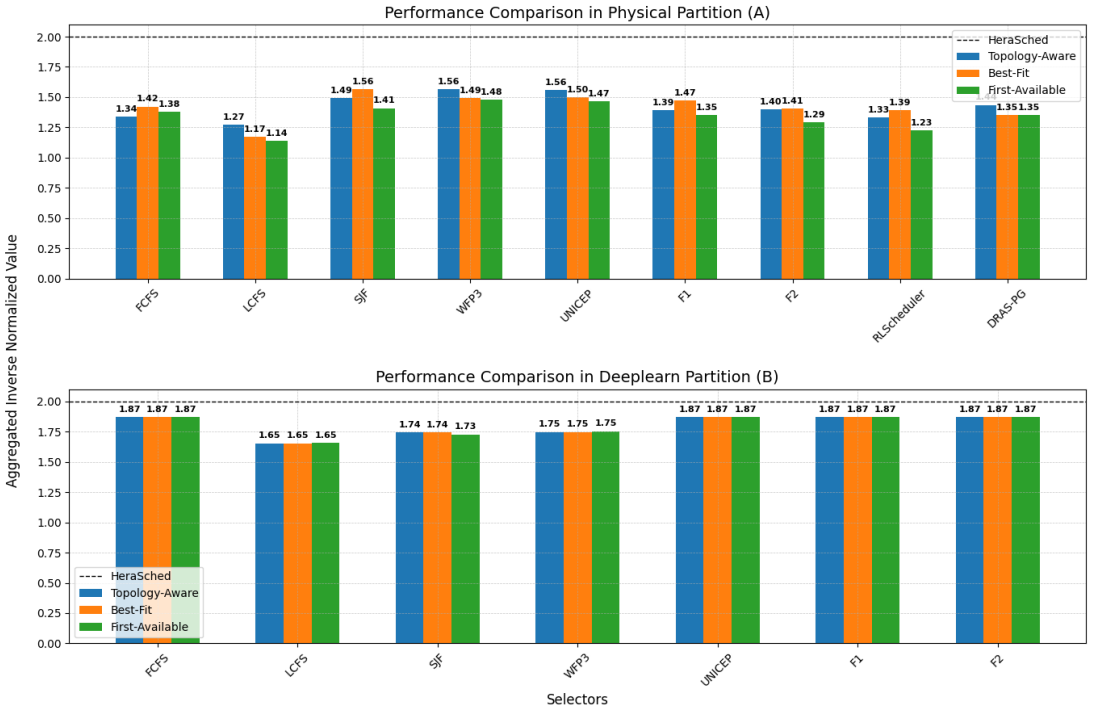

FIGURE 4.8: High Load Test Performance Comparison. Each bar represents the combined effect of both average and maximum waiting times, normalized against HeraSched's performance. The horizontal dashed line at 2 marks HeraSched's baseline performance. Bars below this line indicate worse performance, while bars at or above this line indicate equal or better performance. (HeraSched Ave. / Method Ave.) + (HeraSched Max / Method Max)

To further analyze the performance of HeraSched, we introduce a single metric that aggregates the key performance objectives: average and maximum waiting times. This metric treats both objectives as equally important and is calculated using an inverse normalization approach. Specifically, each scheduling algorithm's performance is normalized against HeraSched's performance for both average and maximum waiting times, and the results are then combined. A value of 2 represents the baseline performance of HeraSched. Values higher than 2 indicate that the compared method performs better than HeraSched, while values lower than 2 indicate worse performance. This method allows us to aggregate and compare the two metrics on a common scale, facilitating a fair and comprehensive evaluation of each scheduling algorithm. Fig. 4.8 shows the results. The results indicate that HeraSched consistently outperforms other algorithms. Other algorithms, such as SJF, Topology-Aware, and Best-Fit, show varied performance, but none match the

efficiency of HeraSched, highlighting its effectiveness in managing waiting times. These results also demonstrate HeraSched's advanced capabilities in dealing with high-stress situations, effectively optimizing scheduling jobs that were not part of its training set. Such resilience and efficiency in unanticipated, high-demand conditions highlight HeraSched's superiority in dynamic HPC environments.

4.4.8 HeraSched Computational Cost

HeraSched uses an HRL architecture consisting of two PPO agents: a *job selector* and a *resource allocator*. Each agent includes an actor and a critic network, totaling four neural networks in the system. The two-level structure increases model complexity compared to flat RL schedulers. The computational cost of using HeraSched can be divided into two parts: **the training time** involving updating 4 neural networks, and **the inference time**, which is the time to make a scheduling decision.

HeraSched's computational cost in two partitions: Physical and Deeplearn. Table 4.4 summarizes the wall-clock training and inference times, measured on a PC with a Ryzen 7700 CPU, Nvidia 3090 GPU, and 32 GB of memory. The detailed settings for each model are discussed in Table 4.2.

TABLE 4.4: HeraSched Complexity and Runtime

Metric	**Physical**	**Deeplearn**
Training Time (1 update)	23.225 s	1.288 s
Decision Time (1 decision)	6 ms	0.67 ms

Training Time (1 update) refers to the time required to complete a single training update in Algorithm 2, where the selector performs $x = 2048$ decision steps before triggering an update for both the selector and allocator networks. During this interval, the allocator may be called multiple times per job, depending on job requirements and node availability, resulting in a variable number of total environment interactions. The measured training time is the average wall-clock duration for one update under this setting, which includes all forward and backward passes required to train the four PPO networks (actor and critic for both selector and allocator). The total training time for HeraSched was approximately 37.78 hours for the Physical partition over 12 million selector steps, and 1.398 hours for the Deeplearn partition over 8 million steps, as shown in Fig. 4.4.

Decision Time (1 decision) refers to the average time it takes for either the selector or the allocator to make a single scheduling decision. In HeraSched, scheduling one job involves one decision by the selector and one or more decisions by the Allocator, depending on the job's required number of nodes. As shown in Table 4.4, the average decision time is 6 ms for the Physical partition and 0.67 ms for Deeplearn, primarily reflecting the difference in neural network size.

These computational costs are considered acceptable given the scale and complexity of HPC scheduling. In the Physical partition, HeraSched maintains a fast inference time of just 6 ms per decision for selection or allocation, enabling responsive scheduling. The training time per update is also reasonable, making HeraSched practical for both training and deployment in HPC scenarios.

These times are primarily influenced by a few key factors. Training time per update depends on the architecture of the neural networks and the number of optimizer steps. The total training duration is affected by the dimensionality of the state and action spaces, which grow with HPC system size, workload diversity, and frequency of job arrivals. Inference time is mainly determined by the size of the job selection choices, the number of available nodes for allocation, and the forward-pass cost of PPO networks. Notably, the Physical partition features significantly more nodes and frequent job submissions than Deeplearn, resulting in a larger observation space and deeper neural networks. Despite these increases, the additional computational cost remains well within practical bounds. These results suggest that HeraSched can potentially remain computationally feasible even as HPC systems grow in scale and architectural complexity.

4.5 Summary

This chapter introduced HeraSched, a hierarchical reinforcement learning scheduler that integrates job selection and resource allocation. HeraSched applies a two-level HRL framework where a high-level selector chooses jobs based on the status of the job queue and cluster, and a low-level allocator assigns those jobs to nodes while considering resource constraints and cluster heterogeneity. The scheduler is guided by a reward mechanism that promotes long-term performance, with specific emphasis on minimizing both average and maximum job waiting times. Through experiments in both CPU and GPU partitions

from real HPC traces, we demonstrated that HeraSched consistently outperforms a broad set of scheduling baselines across multiple workloads.

Building on HeraSched's integration of job selection and allocation, the next chapter addresses another critical challenge in HPC scheduling: managing conflicting scheduling goals and adapting to shifting objectives based on system conditions. It introduces MetaPilot, a deep reinforcement learning-based controller that enables dynamic prioritization between user-centric objectives — such as minimizing job waiting time — and system-centric goals like maximizing resource utilization.

Chapter 5

MetaPilot: A DRL-Based Controller for Balancing User-Centric and System-Centric Objectives in HPC Scheduling

HPC scheduling faces the challenge of balancing system-centric and user-centric objectives, as optimizing resource utilization often comes at the expense of job responsiveness and fairness. Traditional schedulers address this trade-off using fixed-weight heuristics or predefined scoring functions, which assign static importance to different objectives. Similarly, recent reinforcement learning-based schedulers use fixed reward functions that often combine multiple objectives into a predefined weighted sum. However, both fixed scoring functions and reward formulations lack adaptability, as they apply the same trade-off across all system states, failing to adjust dynamically to workload fluctuations and system demands. In this work, we propose MetaPilot, a deep reinforcement learning-based scheduling pilot that dynamically selects scheduling objectives in response to system states. Unlike conventional schedulers, MetaPilot does not directly schedule jobs but instead acts as an adaptive decision layer, guiding existing schedulers by determining whether to prioritize utilization-based or waiting-time-based objectives based on workload characteristics and resource availability. This separation of objective selection from scheduling execution allows MetaPilot to be easily integrated with well-established scheduling systems without

requiring modifications to their core algorithms. We evaluate MetaPilot using real-world HPC workload traces and demonstrate that it reduces maximum waiting time by 19% while maintaining competitive average waiting times and achieving higher resource utilization, particularly under high-load conditions. By enabling schedulers to dynamically adjust to workload changes, MetaPilot improves both system efficiency and user responsiveness, ensuring a more flexible and adaptive scheduling process.

5.1 Introduction

The choice of scheduling objectives directly impacts both system performance and user experience. Existing research has proposed various metrics to guide scheduling policies [33, 133–138]. However, there is no universally accepted metric that fully captures all aspects of HPC scheduling. Scheduling objectives in HPC can generally be categorized into **system-centric** and **user-centric** objectives. Some approaches prioritize system-centric objectives, such as maximizing resource utilization and minimizing job fragmentation, which improves overall computational throughput but focusing on overall metrics may lead to individual job starvation. Others emphasize user-centric objectives, such as reducing queueing delays and ensuring fairness, which enhances user experience but may result in suboptimal system efficiency.

These two sets of objectives are inherently conflicting, making it difficult to achieve an optimal balance. Traditional approaches tend to favor one-sided objectives — either maximizing system efficiency by prioritizing large, high-efficiency jobs or improving responsiveness by giving preference to short jobs. Attempts to balance these trade-offs often lead to multi-objective scheduling, where different metrics are combined through weighted scoring functions. Widely-used workload managers such as Slurm [35], PBS [101], and Borg [31, 32] rely on predefined scoring functions that assign different weights to system and user-centric objectives, such as different resource requirements of jobs and waiting time. Recent research has explored RL-based scheduling, where the scheduler learns a policy through mixed reward functions with different objectives, such as DRAS [26]. However,

This chapter is derived from a paper that is currently under review:

- **Lingfei Wang**, Maria A. Rodriguez, and Nir Lipovetzky. MetaPilot: A DRL-Based Controller for Balancing User-Centric and System-Centric Objectives in HPC Scheduling. Submitted to *Future Generation Computer Systems*.

the reward functions in RL-based scheduling also typically rely on fixed combinations of objectives, making them less adaptable to dynamic workload patterns where the importance of each objective may shift. Moreover, a fixed combination of utilization-based and waiting-time-based rewards leads to compromised scheduling rather than optimal decisions. As a result, these fixed-weight approaches inherently fail to achieve optimal performance in either system-centric or user-centric objectives.

In reality, workload characteristics and system conditions fluctuate over time, requiring dynamic prioritization mechanisms rather than a fixed combination of objectives. The optimal scheduling strategy depends on job arrival rates, resource contention, and cluster utilization. During periods of high system load, prioritizing resource utilization is essential to prevent underutilization as the resource becomes the system bottleneck. Conversely, under moderate or low load, prioritizing job waiting time enhances user satisfaction. The fixed-objective schedulers, whether rule-based or RL-based, fail to dynamically shift priorities based on these changing conditions.

In this chapter, we propose MetaPilot, an RL-based scheduling pilot that dynamically selects the most suitable scheduling objective based on real-time system states. At the core of MetaPilot is a DRL agent that learns a policy to adjust scheduling priorities dynamically based on system conditions. The agent continuously monitors system load, job queue characteristics, resource availability, and historical scheduling outcomes, selecting an objective function that aligns with current system needs. Unlike traditional schedulers, which rely on predefined trade-offs, MetaPilot serves as an adaptive decision layer, determining the optimal scheduling objective at any given moment. In this chapter, 'optimal' is defined with respect to an explicit preference between two system-level anchors—user-centric waiting time and system-centric utilization—selected based on system conditions. Additionally, MetaPilot does not replace existing schedulers; instead, it guides them by specifying the most appropriate objective. Once the objective is selected, job scheduling and resource allocation are handled by well-established schedulers, which then optimize the selected objective. This separation of objective selection from scheduling execution allows MetaPilot to be integrated with any existing scheduling system without requiring modifications to its core algorithms. The primary contributions of this chapter are:

- Develops MetaPilot, a DRL-based controller that dynamically balances user-centric and system-centric scheduling objectives in HPC environments. The proposed

framework adapts scheduling priorities between utilization-based and waiting-time-based objectives in response to real-time system conditions, offering greater flexibility compared to fixed-objective scheduling.

- Designs an efficient state representation and reward function that captures workload characteristics, system status, and resource availability to enhance decision-making. The reward function dynamically adjusts based on real-time workload fluctuations, enabling MetaPilot to align scheduling objectives with system demands and user expectations.

- Evaluates MetaPilot on real-world HPC workload traces, demonstrating its ability to intelligently switch scheduling strategies. Experimental results show that MetaPilot reduces maximum waiting time by 19% while maintaining competitive average waiting times and achieving higher resource utilization, particularly in high-load scenarios testing.

5.2 Related Work

The evaluation of HPC scheduling performance has been a long-standing research focus, with studies exploring different metrics and scheduling approaches. Frachtenberg and Feitelson [134] analyze utilization as a scheduling metric, noting that high utilization does not always translate to efficient job placement. Their study discusses how resource fragmentation can reduce overall system performance despite high utilization rates. Leung, Sabin, and Sadayappan [135] investigate fairness in parallel job scheduling, evaluating different job prioritization and resource-sharing strategies to prevent job starvation and improve equitable resource allocation. Verma, Korupolu, and Wilkes [136] examine job placement techniques in warehouse-scale computing environments, focusing on job-packing efficiency and its impact on system throughput and resource allocation. Goponenko et al. [138] propose alternative job-packing efficiency metrics to address resource fragmentation, suggesting that traditional utilization-based metrics may not fully reflect how effectively resources are allocated. Boëzennec et al. [33, 137] explore alternative optimization metrics in HPC batch scheduling, proposing utilization standard deviation as a measure of scheduling stability. Their findings indicate that standard deviation can help differentiate scheduling strategies, particularly in systems where average utilization fails to

capture workload variations. While some research emphasizes utilization and throughput as key performance indicators, others prioritize fairness and job waiting times to improve workload distribution. However, no single metric fully captures all aspects of scheduling quality, leading to a diverse range of evaluation methods in the literature.

Several studies have explored RL-based HPC scheduling with different optimization strategies. CuSH [22] introduced a dual-agent DRL approach, where one agent selects jobs and another chooses an allocation policy (depth-first or breadth-first) to optimize waiting and turnaround times. Decima [23] schedules interdependent jobs modeled as Directed Acyclic Graphs, selecting job stages and allocating executors to minimize completion time. RLScheduler [24] applies PPO for batch job scheduling, outperforming heuristics but lacking built-in back-filling and only optimizing a single objective. RLSchert [25] enhances PPO-based scheduling with an RNN-based runtime prediction model, improving scheduling efficiency but risking resource wastage due to inaccurate predictions. DRAS [26] employs two DRL agents for job selection and back-filling, achieving strong performance but facing overfitting risks due to training on fixed workloads. SchedInspector [27] builds an inspector based on reinforcement learning. The inspector takes into consideration the cluster's and queue's state to determine whether to execute or ignore the scheduling decision made by a heuristic algorithm. However, the performance of this method largely depends on the base heuristic. Further, it relies on a separate heuristic-based back-filling algorithm. Existing RL-based schedulers optimize fixed scheduling objectives but lack the ability to dynamically adjust priorities based on real-time system conditions.

5.3 Background and Discussion

5.3.1 HPC Scheduling Objectives

HPC job scheduling serves multiple stakeholders, from system administrators aiming to maximize resource utilization to end-users seeking reduced job waiting times. These competing priorities lead to two primary categories of scheduling objectives: **system-centric** and **user-centric** objectives. System-centric objectives focus on optimizing overall cluster performance, ensuring high throughput, efficient resource utilization, and long-term scheduling stability. These objectives are crucial for system administrators, as they directly impact the efficiency of an HPC system. User-centric objectives prioritize the

quality of service from the perspective of individual job owners. These objectives prevent starvation, reduce waiting times, and provide predictable scheduling behavior. One of the fundamental system-centric objectives is maximizing resource utilization, ensuring that CPU cores and memory are used effectively. CPU and memory utilization measures the fraction of CPU and memory resources actively used over a given time period.

User-centric objectives primarily focus on reducing job delays to improve user experience. Two widely used metrics are average waiting time and maximum waiting time. The average waiting time (AWT), Equation 2.5, measures the mean duration that jobs spend in the queue before execution, and the maximum waiting time (MWT), Equation 2.6, identifies the longest delay experienced by any job. A lower AWT indicates that jobs are scheduled promptly, reducing user frustration. MWT ensures that no job experiences excessive starvation, making it essential in scheduling.

We restrict the objective space to these two anchors because they capture the central trade-off in production HPC and can be measured consistently in our trace-driven setting. Administrators manage capacity through utilization, while users experience service quality through waiting time; these are the levers most often negotiated in practice. Many other goals move with these anchors on fixed workloads: higher utilization typically accompanies higher throughput and shorter makespan, and average slowdown is largely determined by waiting time when runtimes are fixed by the trace; tail guarantees can be expressed using maximum or high-percentile waiting time. Limiting the objective set also stabilizes learning and avoids mode-switching pathologies, while still spanning the practical system–user trade-off.

A scheduler that prioritizes maximizing utilization seeks to keep as many resources as busy as possible, reducing idle periods and improving overall throughput. However, this approach can inadvertently increase waiting times, as jobs with lower resource demands or shorter runtimes may be deprioritized in favor of larger jobs that maximize resource consumption. Conversely, a scheduler that minimizes average waiting time or maximum waiting time may prioritize short jobs or backfilling strategies, allowing small jobs to bypass larger ones to improve responsiveness. While this approach benefits users by reducing queue delays, it can reduce utilization efficiency by introducing fragmentation, where system resources remain idle because they cannot be perfectly packed with incoming job requests.

5.3.2 Insights on Scheduling Objectives

The importance of different scheduling objectives in HPC clusters is highly dynamic and depends on workload conditions and system status. In practice, system-centric and user-centric objectives often conflict, requiring careful trade-offs. To illustrate this, consider a simplified example of an HPC cluster with four compute nodes, each equipped with a single CPU core and sufficient memory. Five jobs arrive simultaneously, each with different resource requests and execution times. Job 1, 2, and 4 request one node each and require one unit of runtime. Job 3 requests two nodes and one unit of runtime. Job 5, the largest, requests four nodes and two units of runtime. Fig. 5.1 illustrates two different scheduling strategies: (A) minimizing average waiting time and (B) maximizing core utilization. In Strategy A, jobs are scheduled to minimize average waiting time, prioritizing smaller jobs to ensure quick turnaround. In contrast, Strategy B prioritizes maximizing resource utilization by allocating jobs in a way that keeps the most CPU cores occupied at all times. The average waiting time for Strategy A is 0.6 units of time, whereas for Strategy B, it is 0.8 units. Meanwhile, Strategy A leaves one or two nodes idle before Job 5 runs, resulting in wasted resources, whereas Strategy B achieves full utilization before Job 4 runs by ensuring all nodes remain occupied at all times.

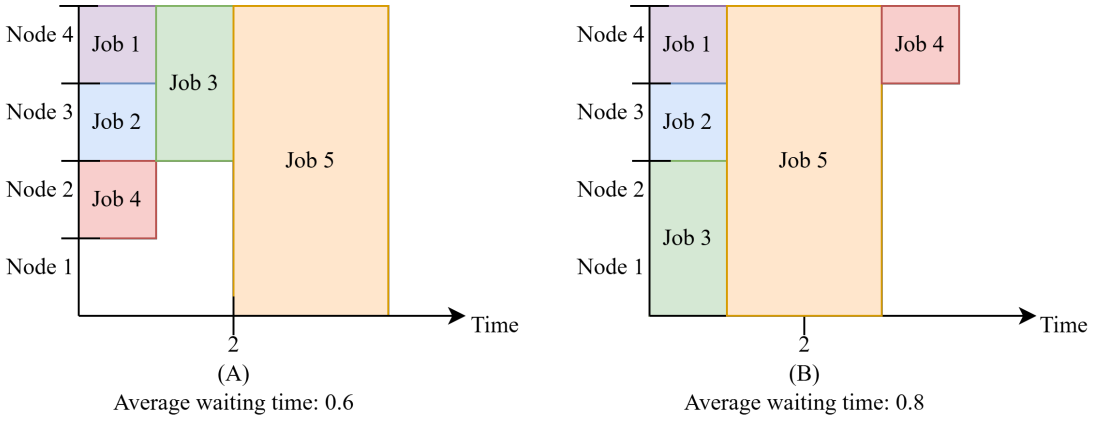

FIGURE 5.1: Example of two scheduling strategies: (A) minimizing average waiting time and (B) maximizing core utilization in a simplified four-node HPC cluster.

Intuitively, Strategy B may be considered better as it fully utilizes the available resources, ensuring that no compute node remains idle. However, this comes at the cost of delaying small jobs, which may negatively impact user experience. Thus, we cannot simply conclude which scheduling practice is superior without considering the broader context. A key factor influencing scheduling decisions is the *job arrival pattern*. System-centric

objectives, such as maximizing resource utilization, become more important in high-load scenarios where job arrivals are frequent, and resource contention is high. In such cases, prioritizing utilization prevents resource fragmentation and ensures that computational capacity is fully leveraged, ultimately improving overall system efficiency. On the other hand, user-centric objectives, such as minimizing average and maximum waiting times, are more critical in low-load scenarios, where job arrivals are infrequent and users expect quick turnaround times. In such situations, resource utilization is less of a concern because there is sufficient available capacity to execute jobs without causing contention. If the scheduler prioritizes utilization even when resources are abundant, it may unnecessarily delay short jobs by waiting for larger, high-efficiency jobs to arrive, leading to artificial queueing and degraded user experience. Under these conditions, the primary scheduling goal should shift toward reducing waiting times to ensure that jobs are executed as soon as resources are available, rather than deferring execution in an attempt to maximize efficiency. Given the dynamic nature of HPC workloads, an adaptive scheduling approach is necessary to balance these objectives based on real-time cluster conditions. Rather than adhering strictly to a single optimization criterion, schedulers should dynamically adjust their prioritization of system-centric and user-centric objectives in response to workload patterns.

5.4 MetaPilot Design

5.4.1 MetaPilot Framework

The MetaPilot framework is designed to dynamically balance system-centric and user-centric scheduling objectives in an HPC cluster. The workflow of the framework, as depicted in Fig. 5.2, consists of three primary components: Cluster and Queue Monitoring, System Dashboard, and MetaPilot RL agent. The system continuously monitors cluster resource availability, including CPU core utilization and memory usage. Simultaneously, the job queue maintains pending jobs. All monitoring information is summarized in the System Dashboard. The System Dashboard serves as an intermediary between the monitored HPC and the MetaPilot decision-making module. The dashboard consolidates system statistics into a structured representation, providing a real-time snapshot of system status, which serves as the state for the DRL-based MetaPilot agent. MetaPilot, a

DRL-based controller, processes the real-time system status from the System Dashboard to determine the most suitable scheduling objective at each decision point. The RL agent takes state information as input and, through a neural network-based policy, selects either Utilization or Waiting Times as the current scheduling objective. Once the scheduling objective is selected, the corresponding scheduler executes job selection and resource allocation decisions. This dynamic switching mechanism allows MetaPilot to adapt scheduling strategies to changing workload patterns and cluster conditions. This provides an adaptive, objective-aware scheduling framework, ensuring that HPC workloads are managed in a balanced and efficient manner.

A plausible alternative is to train a single scheduler under a continuously weighted objective. We adopt discrete meta-selection instead (choosing between specialized policies for waiting time or utilization) because the two anchors often induce conflicting short-horizon gradients; in pilot runs, mixing them in one policy compromised anchor-wise performance and complicated tuning. The modular design keeps the active preference explicit and allows reuse and transfer of specialized policies, yet remains compatible with a future soft-weight variant if desired.

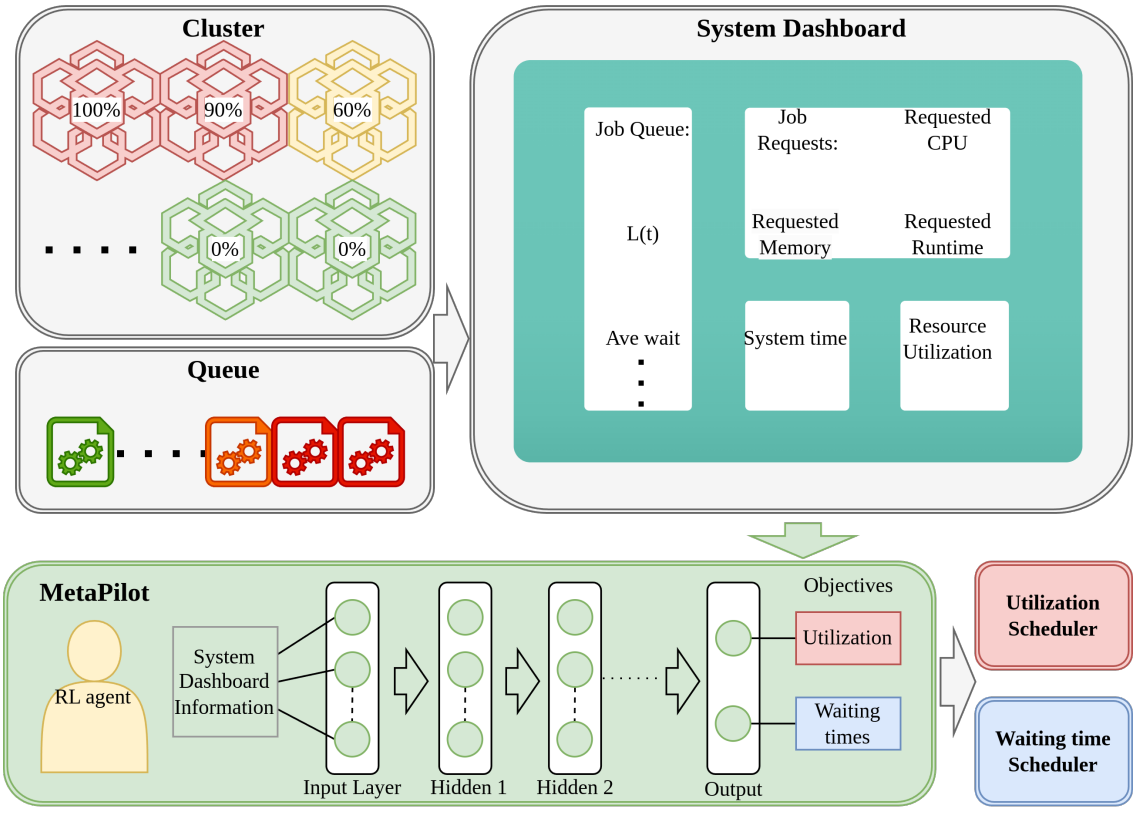

FIGURE 5.2: Overview of MetaPilot.

5.4.2 State Representation

As discussed earlier, system status and job arrival patterns are key factors influencing the importance of different scheduling objectives. The state representation should comprehensively reflect the scheduling environment while ensuring that it contains only the information necessary for MetaPilot to make high-level decisions. To maintain efficiency, MetaPilot relies on aggregated system statistics rather than fine-grained job details, keeping the framework lightweight while still capturing the key factors that influence scheduling objectives.

Job Queue Features capture essential information about the current job queue state, providing a snapshot of the pending workload. This includes the number of jobs in the queue $L(t)$. Additionally, the total requested CPU cores $\sum_{J_j \in \mathcal{Q}(t)} h_j \cdot l_j$, and the total requested memory $\sum_{J_j \in \mathcal{Q}(t)} q_j \cdot l_j$ are included. To give a clearer view of system load, these values are recorded not only as raw totals but also as fractions of the total available resources in the cluster. These features enable MetaPilot to assess how loaded the queue is and make scheduling decisions accordingly.

Job Requirements Statistics provide an aggregated view of job resource demands, summarizing the distribution of requested nodes, CPU cores, and memory for all jobs in the queue. Rather than considering each job individually, statistical measures such as maximum, minimum, quartiles, median, mode, and average values are used to describe job characteristics efficiently. This aggregation helps MetaPilot understand workload heterogeneity while maintaining a compact state representation.

System Time feature encodes temporal information relevant to scheduling. It includes two fields: the current hour and the day of the week. As shown in Fig. 5.3, job submissions exhibit strong dependencies on the time of day and the day of the week, reflecting user activity patterns and workload fluctuations. By incorporating system time into the state representation, MetaPilot can learn to anticipate peak and low-load periods, allowing it to adjust scheduling strategies accordingly. This temporal awareness helps the agent balance system-centric and user-centric objectives dynamically, ensuring that scheduling decisions align with workload characteristics.

Cluster Features provides a high-level summary of the system's resource availability. This includes the number of free nodes, available CPU cores, and memory, capturing

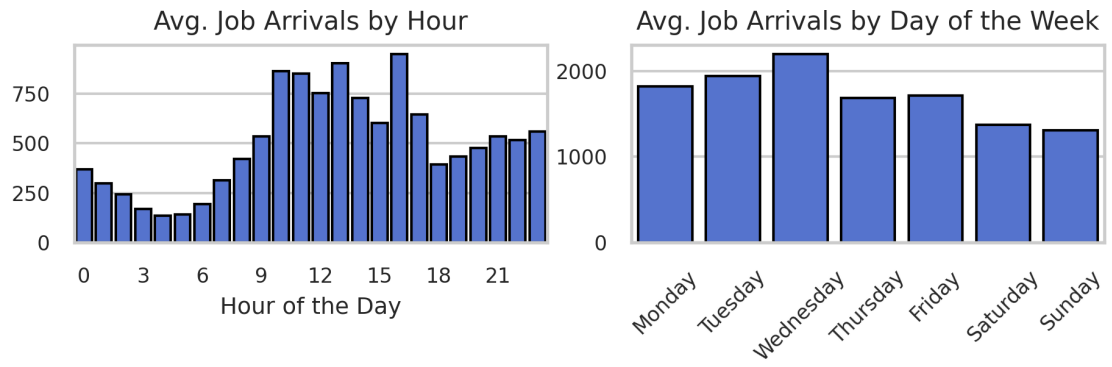

FIGURE 5.3: Average job arrival patterns for the Physical partition in Spartan [1] HPC system from August 20, 2021, to September 30, 2022.

the overall resource state at any given time. Additionally, the utilization ratios for each resource type are recorded, offering a relative measure of how much of the cluster's capacity is currently in use. These features help MetaPilot assess system load.

Each Node Features captures the state of individual compute nodes, detailing the available CPU cores, and memory for each of the N nodes in the cluster. Specifically, for each node, the number of free CPU cores, and memory is recorded along with their respective utilization ratios. Since there are N nodes, these features contribute a total of $2 \times N$ fields to the state representation. This per-node granularity allows MetaPilot to distinguish between uniformly loaded clusters and imbalanced ones, where some nodes remain underutilized while others are fully occupied.

5.4.3 Reward Function

The reward function is designed to guide MetaPilot in selecting the most appropriate scheduling objective based on real-time system conditions. The reward function must incorporate both system-centric and user-centric considerations. The implemented reward function is defined as follows:

$$reward = \begin{cases} 3, & \text{if } L(t) = 0 \\ \max\left(1 - \frac{\bar{W}}{\alpha}, 0\right) + \left(1 - \frac{C_{\text{free}}}{C_{\text{total}}}\right) + \beta\left(1 - \frac{M_{\text{free}}}{M_{\text{total}}}\right), & \text{otherwise} \end{cases} \tag{5.1}$$

where $L(t)$ is the number of jobs in the pending queue. $\bar{W}$ is the average waiting time of jobs in the queue. $C_{\text{free}}, M_{\text{free}}$ are the available CPU cores, and memory in the cluster, respectively. $C_{\text{total}}, M_{\text{total}}$ are the total CPU cores and memory in the cluster. α is a scaling factor for the waiting time penalty, controlling its impact on the reward.

β is a weighting parameter for memory utilization, allowing adjustments in its relative importance compared to CPU and GPU utilization.

If the job queue is empty, MetaPilot receives a fixed high reward. This condition reflects an optimal scheduling scenario where all submitted jobs have been dispatched and are currently running. By assigning a high reward, MetaPilot is encouraged to favor scheduling strategies that efficiently clear the queue, reducing job backlogs. When jobs are pending, the reward is penalized based on the average waiting time. A longer waiting time reduces the reward, prompting MetaPilot to prefer scheduling strategies that minimize job delays. Also, higher resource utilization leads to a higher reward, encouraging MetaPilot to select schedulers that maximize system efficiency.

MetaPilot dynamically adjusts its scheduling strategy based on real-time system conditions. When the job queue length is high but resource utilization remains low, the reward decreases due to prolonged waiting times and underutilization, prompting MetaPilot to prioritize rapid job execution. Conversely, when utilization is high but waiting times are growing, the system initially favors efficiency but eventually shifts toward a user-centric approach as delays exceed a threshold. In a balanced load scenario with moderate queue lengths, the reward remains stable, allowing MetaPilot to switch between objectives dynamically. In an underloaded system where resources remain idle despite pending jobs, utilization-based rewards decrease, encouraging more aggressive job dispatching. Unlike fixed-heuristic or mixed-reward approaches, MetaPilot continuously learns and adjusts to workload variations, ensuring an adaptive balance between responsiveness and efficiency.

5.4.4 DRL Agent

MetaPilot employs a DRL agent to adjust scheduling objectives based on system conditions dynamically. We implement PPO [40], a policy-gradient-based algorithm that ensures stable training through policy clipping. The DRL model is structured as a multilayer perceptron (MLP) with three hidden layers of sizes 1024, 512, and 64 neurons, shared between the policy network (actor) and the value network (critic). A Tanh activation function is applied throughout the architecture to regulate activation outputs and improve learning stability. Additionally, Training is performed with a discount factor ($\gamma = 1$), encouraging the agent to make long-term scheduling decisions rather than focusing solely on immediate gains.

5.5 Evaluation and Results

5.5.1 Evaluation Setup

To evaluate MetaPilot, we use real job traces and system configurations from Spartan [1], an operating HPC system. Specifically, we used a dedicated partition named Physical in Spartan, which consists of 86 compute nodes, each equipped with 72 CPU cores. There are in total 6,192 CPU cores in the partition These nodes have two different memory configurations: 72 nodes with 710 GB of memory and 14 nodes with 1510 GB of memory. The evaluation is based on real job traces collected over a 13-month period from August 20, 2021, to September 30, 2022. The dataset includes 4,608,924 recorded job submissions, capturing a diverse range of workloads with varying resource requests and execution times. The maximum recorded job runtime is 30 days.

For scheduling, we use **HeraSched** described in Chapter 4 as the base scheduling model. HeraSched is a hierarchical RL-based scheduler that hierarchically handles the two key subtasks of scheduling: job selection and resource allocation. This hierarchical structure enables the scheduler to coordinate job dispatching and resource allocation more effectively, providing a robust foundation for evaluating MetaPilot's ability to dynamically adjust scheduling strategies. Since running experiments directly on Spartan is impractical due to high resource costs and production constraints, we conduct our evaluation in HeraSched's simulated environment (described in 4.4.2). By leveraging real trace data and system configurations, the simulation closely reflects the operational characteristics of Spartan, ensuring a realistic and reliable assessment of MetaPilot's performance.

We use **HeraSched_W**, the original HeraSched model that prioritizes minimizing both average and maximum waiting times, and extend it to **HeraSched_U**, a variant designed to maximize CPU and memory utilization. Both models are trained following the methodology from HeraSched's setting in Table 4.2, using trace data from September 23, 2022, to September 30, 2022, ensuring consistency with prior work. This chapter focuses on MetaPilot's scheduling control rather than detailed scheduler performance comparisons.

[1] https://gitlab.unimelb.edu.au/lingfeiw/metapilot.git.

5.5.2 MetaPilot Implementation

MetaPilot is implemented using Stable-Baselines3 with its PPO algorithm. The key hyperparameters used in training include a discount factor (γ) of 1, and a learning rate of 3×10^{-4}. The agent is trained using mini-batches of size 64, with an update frequency of 2048 steps per optimization cycle. The training for MetaPilot is also based on Physical partition trace data from September 23, 2022, to September 30, 2022 (a week). Figure 5.4 presents the training progression of MetaPilot over 4 million steps. The episode return initially fluctuates as the agent explores different scheduling strategies but gradually increases, indicating the learning process stabilizing. Around the later stages, the curve plateaus, suggesting that the agent has converged.

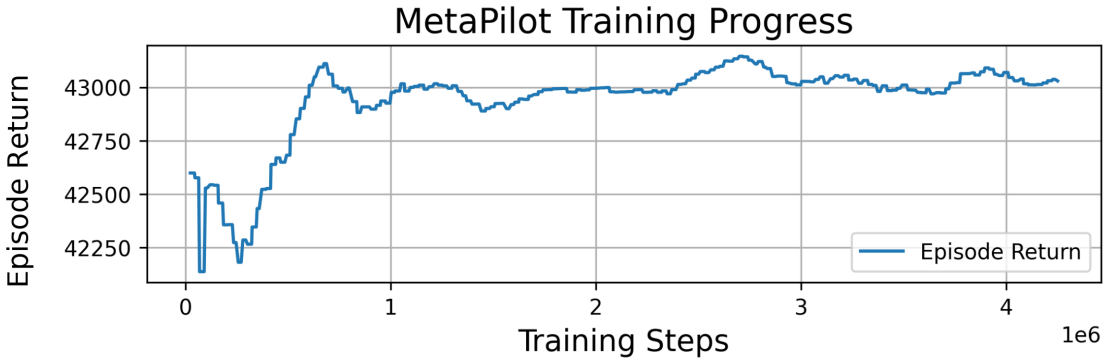

FIGURE 5.4: Training progress of MetaPilot.

5.5.3 Evaluation on MetaPilot

MetaPilot's effectiveness is evaluated by analyzing how it dynamically switches between system-centric (HeraSched-U) and user-centric (HeraSched-W) scheduling strategies based on real-time system conditions. MetaPilot should select the most appropriate scheduling objective depending on resource availability and queue characteristics. A key consideration in scheduling decisions is identifying when each strategy is most suitable. When the system has many idle resources and the queue remains small, prioritizing HeraSched_W reduces unnecessary delays, improving user experience. Conversely, if the queue grows significantly and jobs request large amounts of resources, maximizing utilization through HeraSched_U becomes essential for processing workloads efficiently.

HPC scheduling performance is significantly influenced by factors such as job arrival patterns, resource demands, and system load fluctuations [33]. We tested MetaPilot using the full 13-month dataset. However, presenting all results would be redundant and uninformative, as many periods exhibit similar scheduling behavior. Instead, we selected a continuous testing period that captures diverse workload conditions, including varying queue lengths, resource contention levels, and idle phases. The period from January 28, 2022, to February 25, 2022, was chosen for its representative nature, covering different scheduling challenges. Additional results from other testing periods are available in the source code repository[1].

5.5.3.1 Evaluation on MetaPilot's Actions.

Fig. 5.5 and Fig. 5.6 illustrate MetaPilot's decision-making process in dynamically selecting scheduling objectives based on system conditions in the training and testing sets, respectively. The CPU and memory load ratios, which represent the total requested resources in the queue normalized by the cluster's capacity, provide insights into workload variations over time. The shaded regions represent MetaPilot's chosen scheduling strategies—gray for waiting times optimization and green for utilization maximization. In the training set (Fig.5.5), the system experiences lower and more sporadic CPU and memory loads, with brief periods of high demand. The MetaPilot agent predominantly selects waiting-time optimization during these fluctuations. In contrast, the testing set (Fig.5.6) captures more diverse workload conditions, including sustained high memory loads, extreme CPU load spikes (e.g., 712.4% of total CPU cores), and periods of low activity. This mix of conditions allows for a more comprehensive evaluation of MetaPilot's adaptability, demonstrating its ability to adjust scheduling objectives based on varying system states.

During periods of low CPU and memory load, MetaPilot predominantly selects the waiting time minimization strategy (gray regions). This decision aligns with the objective of improving user experience by reducing queue congestion and preventing job starvation. As the system experiences an increase in queued job demands, indicated by spikes in CPU and memory load ratios, MetaPilot transitions to prioritizing utilization (green regions). In these instances, maximizing resource usage becomes critical to handling the incoming workload efficiently. By selecting utilization maximization, MetaPilot optimizes available CPU and memory resources, preventing resource wasting during high-demand periods.

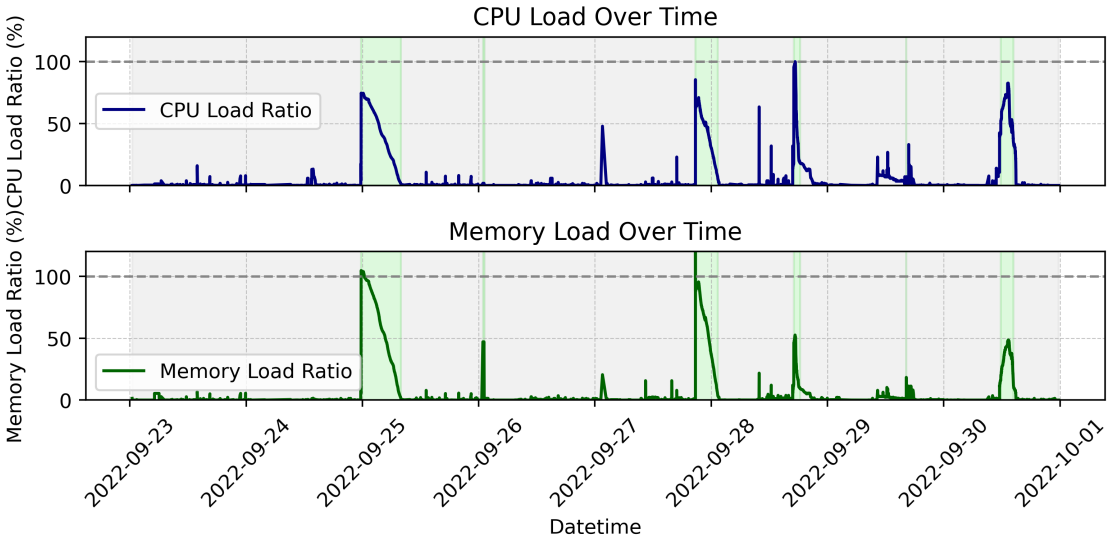

FIGURE 5.5: MetaPilot's action in **training** set.Requested CPU and Memory Load in the queue, normalized by the total available resources in the cluster. Shaded regions represent MetaPilot's scheduling decisions: gray indicates optimization for waiting time reduction, while green represents prioritization of resource utilization maximization.

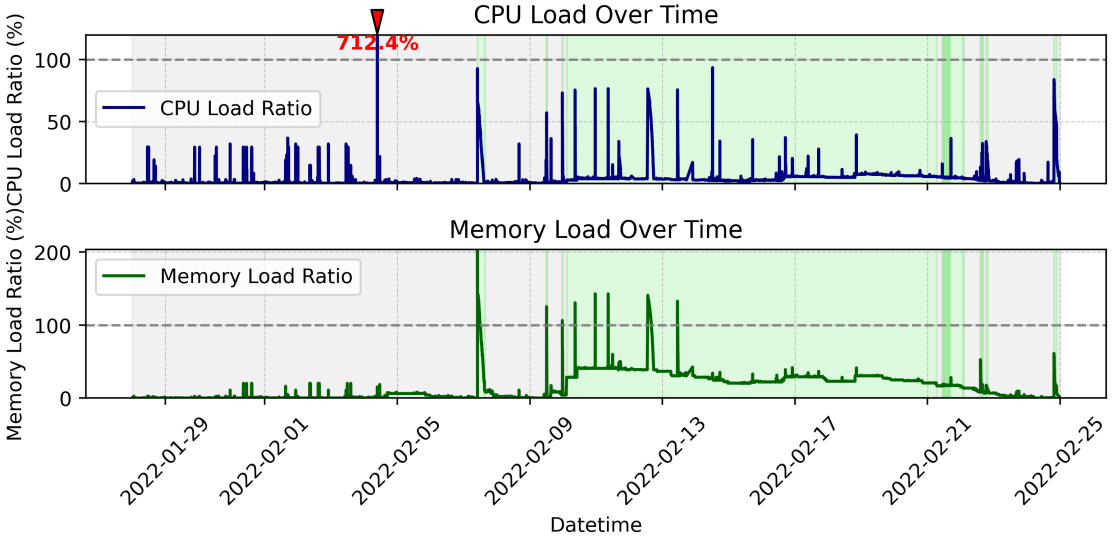

FIGURE 5.6: MetaPilot's action in **testing** set. Requested CPU and Memory Load in the queue, normalized by the total available resources in the cluster. Shaded regions represent MetaPilot's scheduling decisions: gray indicates optimization for waiting time reduction, while green represents prioritization of resource utilization maximization.

The alternation between these strategies suggests that MetaPilot effectively balances user-centric and system-centric objectives in response to workload variations.

Interestingly, MetaPilot is more sensitive to memory load than CPU load. Memory availability often better reflects the system's actual load since memory is typically more abundant per node compared to CPU cores. For instance, in the testing set, the CPU load peaks at 712.4% of total cluster cores, yet the memory load remains low. This indicates a burst of small, short-lived jobs that consume minimal memory. Since these jobs are completed quickly, MetaPilot chooses to optimize waiting times instead of prioritizing utilization, ensuring a better user experience.

5.5.3.2 Evaluation on MetaPilot's Performance

The bar plots in Fig. 5.7 compare the performance of MetaPilot, HeraSched_W, and HeraSched_U in both the training and testing phases, evaluating average waiting time (orange bars) and maximum waiting time (blue bars). In the training set, MetaPilot effectively adapts to varying system loads, demonstrating its ability to balance scheduling objectives. During high-load periods, it efficiently utilizes available resources by dynamically selecting the scheduling objective that optimizes system utilization. This strategy reduces maximum waiting time compared to HeraSched_W, which solely minimizes average waiting time without addressing extreme job delays, thereby lowering the risk of job starvation. Since most of the training period has low system demand, MetaPilot's average waiting time is slightly higher than that of HeraSched_W, which consistently prioritizes shorter job queues. However, this trade-off allows MetaPilot to respond more effectively when the system enters a high-load state, preventing excessive queuing and improving overall system fairness. Meanwhile, HeraSched_U strictly maximizes utilization, leading to significantly higher waiting times, further highlighting MetaPilot's ability to achieve a balanced scheduling policy.

In the testing set, the system experiences prolonged high loads, including extreme peaks that exceed cluster capacity. HeraSched_W still outperforms HeraSched_U in terms of waiting times, reducing maximum waiting time by 19% at the cost of an 8% increase in average waiting time. However, unlike in the training set, this advantage is less pronounced due to the prolonged high-load conditions. Under extreme congestion, minimizing waiting times alone is insufficient, as resource availability becomes the primary

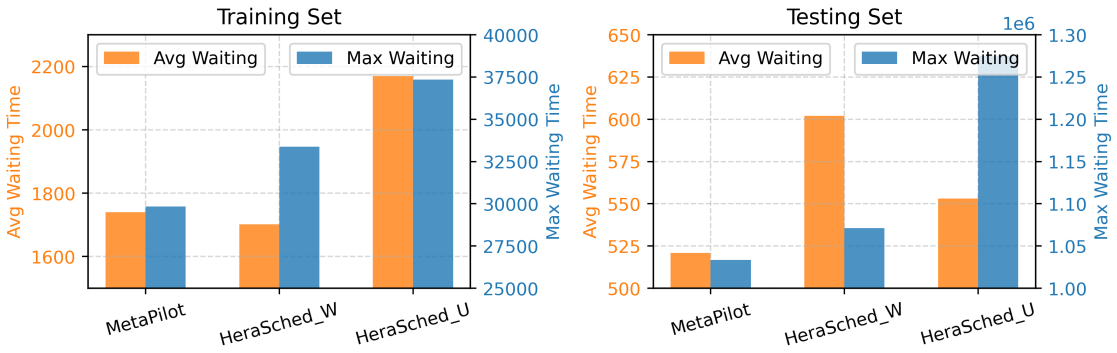

FIGURE 5.7: Comparison of maximum and average waiting times for MetaPilot-Control, HeraSched_W, and HeraSched_U during training and testing phases. Lower values indicate better scheduling performance.

constraint. In such cases, maximizing utilization becomes a more critical objective, as efficiently allocating resources helps prevent excessive job backlog and system stagnation. MetaPilot achieves the best performance in both maximum and average waiting times. A key reason for its superior performance is its ability to recognize when reducing maximum waiting time is more beneficial than strictly focusing on utilization. When demand is extreme, MetaPilot prioritizes efficient resource allocation to prevent resource wastage, which in turn reduces the likelihood of excessive delays. This is particularly important in long periods of high load because, under extreme demand, idle resources translate directly into longer waiting times. During more balanced periods, it shifts towards reducing waiting times to enhance user experience. This flexibility allows MetaPilot to maintain a significantly lower maximum waiting time while keeping the average waiting time lowest, demonstrating the advantage of an adaptive scheduling approach over rigid, single-objective strategies.

The utilization comparison in Fig. 5.8 demonstrates that MetaPilot achieves higher utilization, especially in memory, than both HeraSched_W and HeraSched_U during high-load periods. This improvement stems from its adaptive scheduling strategy, which dynamically shifts between objectives based on workload conditions. While the percentage improvement in utilization may appear modest, its impact is significant in large-scale HPC environments. Even a small increase in resource utilization translates to a substantial reduction in idle resources, leading to more jobs being processed within the same time frame. This effect is particularly critical in high-load scenarios, where MetaPilot's ability to maintain higher utilization ensures that available resources are not wasted. This adaptability allows the system to handle surges in workload more effectively while maintaining stable performance. MetaPilot is a meta-controller over objectives on top of a

fixed scheduler; to isolate the meta-control effect, we compare against the performance of HeraSched. Broader RL-to-RL comparisons are established in Chapter 4.

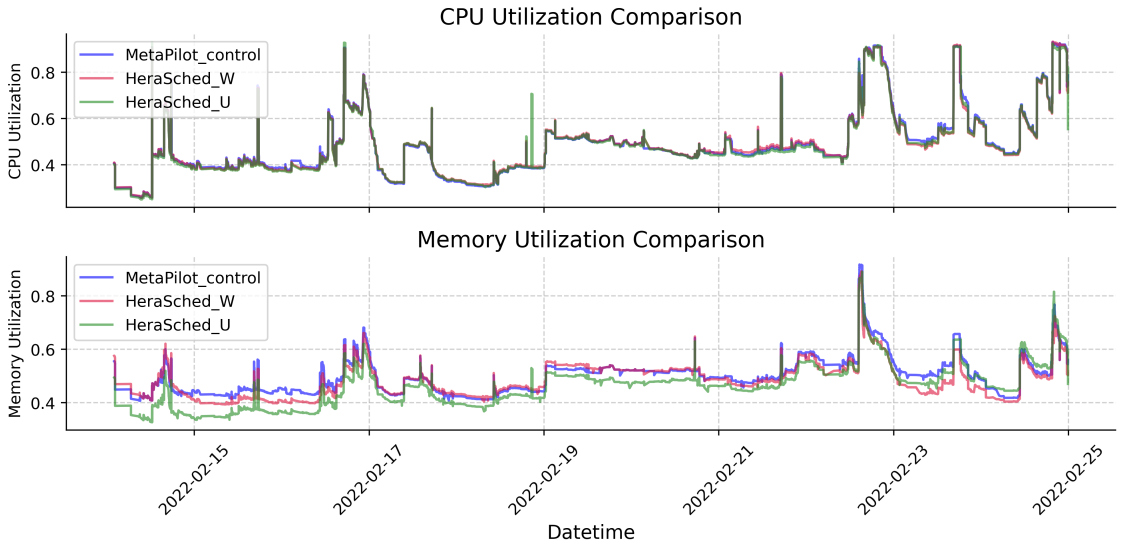

FIGURE 5.8: Comparison of CPU and Memory Utilization in high-load period for different scheduling approaches.

5.6 Summary

This chapter introduced MetaPilot, a deep reinforcement learning-based controller designed to dynamically balance user-centric and system-centric objectives in HPC job scheduling. By decoupling objective selection from scheduling execution, MetaPilot enables existing schedulers to adapt their behavior based on real-time system conditions without requiring changes to their core algorithms. Through a tailored state representation and adaptive reward function, MetaPilot learns when to prioritize job waiting times versus resource utilization. Evaluations on real-world HPC traces demonstrate that MetaPilot outperforms fixed-objective schedulers by reducing maximum waiting times and improving resource utilization, especially under high-load scenarios. These results highlight the value of adaptive objective selection in improving both user experience and system performance in complex, fluctuating HPC environments.

While MetaPilot focuses on dynamically balancing competing scheduling objectives based on real-time system states, it still relies on schedulers that must be trained from scratch for each new HPC environment. However, as HPC systems evolve — introducing

new hardware configurations, workload characteristics, and architectural changes — retraining reinforcement learning-based schedulers from the ground up becomes increasingly inefficient. The next chapter explores how transfer learning can accelerate the adaptation of RL-based schedulers in such evolving environments, enabling faster convergence and reducing the need for extensive retraining when migrating across HPC systems.

Chapter 6

Accelerating RL-Based Scheduler Adaptation with Transfer Learning in Evolving HPC

HPC systems frequently undergo architectural changes, such as hardware upgrades or the deployment of new clusters, to meet evolving computational demands. Traditional static schedulers and machine-learning-based approaches struggle to adapt efficiently to these changes, often requiring manual adjustments or extensive retraining. In this chapter, we propose a novel approach combining Separate Feature Extraction and Selective Transfer Learning to enable rapid adaptation of Reinforcement Learning-based HPC schedulers to new or modified cluster architectures. We evaluate our approach using three real-world HPC clusters, including both CPU and GPU architectures. Our experiments simulate scheduler transitions between these clusters, capturing a wide range of architectural changes and workload variations found in practice. Applied to a state-of-the-art hierarchical RL-based scheduler, our method demonstrates rapid adaptation across diverse system configurations and workloads. Across the six use cases we experimented with, the schedulers enhanced with our approach not only quickly outperformed all 21 baseline heuristic methods in terms of jobs' average and maximum waiting time but also achieved performance comparable to RL-based schedulers trained from scratch. Notably, even in the transition case that required the most retraining steps, integrating our approach reduced the required training timesteps to just 1.76% of the total timesteps needed for training a

scheduler from scratch—representing a 56.8× reduction in training effort. This demonstrates its ability to efficiently adapt existing schedulers to evolving HPC architectures with minimal cost, providing a practical solution for real-world HPC operations.

6.1 Introduction

HPC systems require regular upgrades and maintenance adjustments throughout their operational lifespan to sustain high performance. Advancements in hardware technology, including multi-core processors, high performance GPUs, and specialized accelerators, are rapidly reshaping the capabilities of computational systems. These innovations not only boost processing power but also enhance the efficiency of HPC infrastructures, enabling them to tackle increasingly complex and resource-intensive applications. To leverage these advancements, HPC systems often implement upgrades or develop entirely new clusters, ensuring they remain at the cutting edge of computational capacity. At the same time, maintenance and ongoing adjustments are essential to fine-tune resources in response to fluctuating workloads, ensuring that the system remains adaptable to user demands. This constant reconfiguration and new development present a significant challenge for maintaining efficient job scheduling. Therefore, there is an increasing need for flexible scheduling approaches that can quickly adapt to evolving system architectures while maintaining good system performance.

Traditional scheduling methods, such as heuristics, have been widely used for their simplicity and speed but are typically designed for fixed configurations. More recent Machine Learning-based schedulers [16], particularly those using RL [24, 26], have demonstrated significant improvements by dynamically optimizing scheduling decisions based on real-time feedback. Despite their success, these learning-based approaches, including RL-based schedulers, rely heavily on historical data and static cluster configurations for training. As a result, they often perform well only within the specific scenarios they were trained on. When HPC architectures undergo changes, such as hardware upgrades or adjustments

This chapter is derived from the following publication:

- **Lingfei Wang**, Maria A. Rodriguez, and Nir Lipovetzky. Accelerating RL-Based Scheduler Adaptation with Transfer Learning in Evolving HPC Architectures. In *Proceedings of 2025 IEEE 18th International Conference on Cloud Computing (CLOUD)*, pp. 1-11. IEEE, 2025.

to resource configurations, the environment and state space in which these schedulers operate are significantly altered, rendering previously trained models ineffective.

Transfer learning [78] offers a promising solution to address the limitations of RL-based schedulers in adapting to changing HPC environments. In contrast to training models from scratch for each new configuration, transfer learning allows a scheduler to leverage knowledge gained from previously trained models and apply it to new but related tasks [77]. This reduces the need for large amounts of new training data and minimizes the computational cost of retraining. By fine-tuning specific parts of the model, such as adapting the state and action spaces to reflect new hardware or workload characteristics, transfer learning enables schedulers to adjust more efficiently to changes in cluster configurations. This method supports a faster transition to new environments, such as hardware upgrades or entirely new clusters, while preserving scheduling performance.

In this study, we propose a solution that leverages Transfer Learning to enable RL-based schedulers to quickly adapt to evolving HPC architectures. Our approach incorporates *Separate Feature Extraction*, which isolates the distinct state changes in the cluster and job dynamics from the central RL-based scheduler, and *Selective Transfer Learning*, which retrains only the parts of the model most affected by the new environment. This approach minimizes the need for extensive retraining while ensuring that the scheduler can maintain high performance in newly developed or modified clusters. We validate our method with three distinct HPC partition cases drawn from real operational environments, encompassing both CPU-based and GPU-based clusters. We test scheduler transitions between each partition across six experiments, capturing a broad spectrum of HPC transition scenarios.

In these cases, the schedulers enhanced with our approach significantly reduce the retraining time and data required while achieving performance comparable to RL-based schedulers trained from scratch. At the same time, they consistently outperform traditional heuristic-based methods, demonstrating their efficiency and adaptability. We compare against heuristic-based methods because, in newly developed HPC architectures, there is often insufficient historical data to train machine learning models, making heuristics the default choice for initial deployment. The main contributions of this chapter are as follows:

- We propose a combination of Separate Feature Extraction and Selective Transfer Learning to efficiently adapt RL-based HPC schedulers to new or modified cluster architectures. To the best of our knowledge, this is the first study that addresses the adaptation of RL-based schedulers in response to changes in HPC architectures, providing practical guidance for the operation and maintenance of HPC systems in real-world environments.

- We conduct three case studies using real-world HPC configurations and workloads to demonstrate the flexibility of our approach. These cases cover a wide range of common challenges encountered in real-world HPC environments, showcasing the method's capability to handle diverse cluster transitions effectively.

- We use HeraSched described in Chapter 4, a Hierarchical RL (HRL)-based scheduler managing both job selection and resource allocation, as the base model for evaluation. The results show that Separate Feature Extraction and Selective Transfer Learning enable rapid adaptation with fewer updates and shorter training time compared to training from scratch, highlighting its effectiveness in dynamic HPC environments.

6.2 Related Work

RL has emerged as a promising approach for tackling the challenges of job scheduling in HPC systems. RL-based schedulers, such as DeepRM [20] and Decima [23], have demonstrated the ability to dynamically optimize scheduling decisions based on feedback from the environment. RLScheduler [24] and DRAS [26] proved that RL-based models can outperform traditional heuristics in various scheduling scenarios by learning from historical data and continuously updating their decision-making policies. Following this development, several approaches have proposed their own methods to enhance HPC scheduling. SchedInspector [27] trained an RL-based inspector to control the job selection choices of the based heuristics. However, the majority of these approaches primarily focus on job selection rather than resource allocation.

HRL has also been proposed as an approach to address the complexity of job scheduling by breaking down the problem into sub-tasks. HRL separates job selection and resource allocation into distinct layers, making it more suitable for environments where

heterogeneous resources need to be managed simultaneously. While models like CuSH[22] have made progress in incorporating job allocation into the RL process, they still rely on elementary strategies for resource allocation, limiting their effectiveness in complex, heterogeneous clusters. HeraSched designed an HRL-based scheduler to handle both job selection and resource allocation in heterogeneous HPC environments. By structuring these tasks in hierarchical layers, HeraSched optimizes global objectives like minimizing wait times while ensuring efficient resource allocation across CPUs and GPUs.

Despite these advances, most existing methods are designed to optimize performance for a specific cluster and workload, with limited ability to adapt to changes in cluster configurations or to new environments. Xie et al. [139] introduced an elite-led transfer learning approach for constrained workflow scheduling in dynamic cloud environments, demonstrating how transfer learning can accelerate adaptation to changing resources. However, applying transfer learning in HPC job scheduling, particularly in scenarios involving heterogeneous resources and large-scale workloads, remains relatively underexplored.

6.3 Transfer Learning in HPC Scheduling

Transfer learning is a machine learning technique in which a model developed for a specific task or environment is adapted to perform well in a new but related setting. Rather than training a model from scratch for each new environment, transfer learning leverages knowledge gained from prior tasks, reducing the need for large training datasets and extensive computational resources in the new domain. In RL, transfer learning is especially valuable when the system dynamics or the environment undergo significant changes. General RL transfer learning approaches often involve fine-tuning or modular adaptation of specific model components, such as feature extractors, policy networks, or value functions. These approaches allow the model to retain generalized knowledge while adapting only the parts most impacted by environmental changes.

In HPC scheduling, transfer learning can be particularly useful when clusters undergo hardware upgrades or when new clusters are introduced. However, its application to RL-based HPC schedulers comes with unique challenges, primarily in identifying which components of the model should be preserved from the previous environment and which need

retraining to adapt to the new environment. Retaining too much information from the old system could lead to suboptimal performance, as outdated assumptions may conflict with the new environment's requirements. Conversely, extensive retraining could negate the efficiency benefits of transfer learning. Achieving the right balance between knowledge retention and adaptation is, therefore, critical to maintaining effective scheduling in dynamic and evolving HPC settings.

It is essential to examine the changes and invariants that arise during the transition to a new HPC cluster in scheduling. By analyzing these changes and invariants, we can identify the key challenges that need to be addressed in the design and determine which aspects of the scheduler models should be preserved. This foundational understanding will guide the subsequent design process, ensuring that the scheduler adapts effectively to the new environment while maintaining its core functionality.

Changes: When transitioning to a new HPC cluster, several significant changes must be considered, each impacting the design of the scheduling system. First, the hardware configuration is likely to differ, with potential changes in CPU/GPU architecture, memory hierarchy, network topology, and storage systems. These hardware variations can introduce new constraints and opportunities, prompting adjustments in how resources are allocated and scheduled. Additionally, the workload characteristics in the new cluster may vary, with differences in job types, sizes, and arrival patterns. This shift in workload dynamics requires a corresponding adaptation in the state and action spaces used by the scheduler, ensuring that the scheduling decisions remain relevant and effective in the new environment. These changes collectively highlight the need for a flexible and adaptable design that can respond to the specific requirements of the new cluster.

Invariants: Despite the changes in hardware and workload, several core principles remain consistent during the transition to a new HPC cluster. The basic principles of scheduling, such as meeting the requirements of jobs in the cluster remain constant. These enduring principles provide a stable foundation for the scheduler, allowing it to adapt to new challenges while staying true to its core objectives. Although the specific representations of state and action spaces may need to be adjusted for the new environment, the fundamental knowledge of scheduling embedded in the models is largely preserved, enabling a quick adaptation to the new cluster.

6.4 Proposed Approach

6.4.1 Separate Feature Extraction

Given the substantial changes in hardware configurations and workload dynamics during the transition to a new HPC cluster, developing a scheduling system with enhanced flexibility and adaptability is essential. One of the primary challenges in this transition is adapting to the differences in cluster and queue states between environments. The cluster state includes node configurations, resource availability (e.g., CPU, memory, GPUs), and the running jobs in each node, which vary based on hardware setups and architecture. The queue state reflects waiting jobs and their characteristics, like resource requirements and arrival patterns, which can differ by workload type and user demand across clusters. The RL-based schedulers take the information they need from the states, called an observation. The observation is often well-structured and designed for the specific scheduler and environment. Then, the scheduler outputs the scheduling decisions. These variations in states resulting in observation changes pose a substantial challenge for RL-based schedulers designed and trained in one environment to make effective decisions in other environments. To address this, we present a separate feature extraction mechanism tailored specifically to interpret and adapt to these varying states.

Fig. 6.1 shows the overview of the separate feature extraction mechanism. The term *separate* has two key implications in this context. First, the feature extractors operate independently of the original RL-based scheduler. The feature extractors directly process the observation from the cluster and queue state and produce features that serve as inputs to the scheduler. This design ensures that the inputs to the RL-based scheduler match the original input size, regardless of changes in the cluster configuration. It avoids changes in the environment, such as increases in the number of nodes or CPUs in the cluster, which would typically require resizing the input layer of the RL-based scheduler, and does not affect the scheduler itself. Also, the separate feature extractors can process different forms of observations required by variant RL-based schedulers. This decoupling makes the approach versatile and compatible with a wide range of existing RL-based schedulers without requiring modifications to their architectures. Second, *separate* also refers to the architecture of the feature extractors, which consists of two distinct components: a cluster feature extractor and a queue feature extractor. These components independently

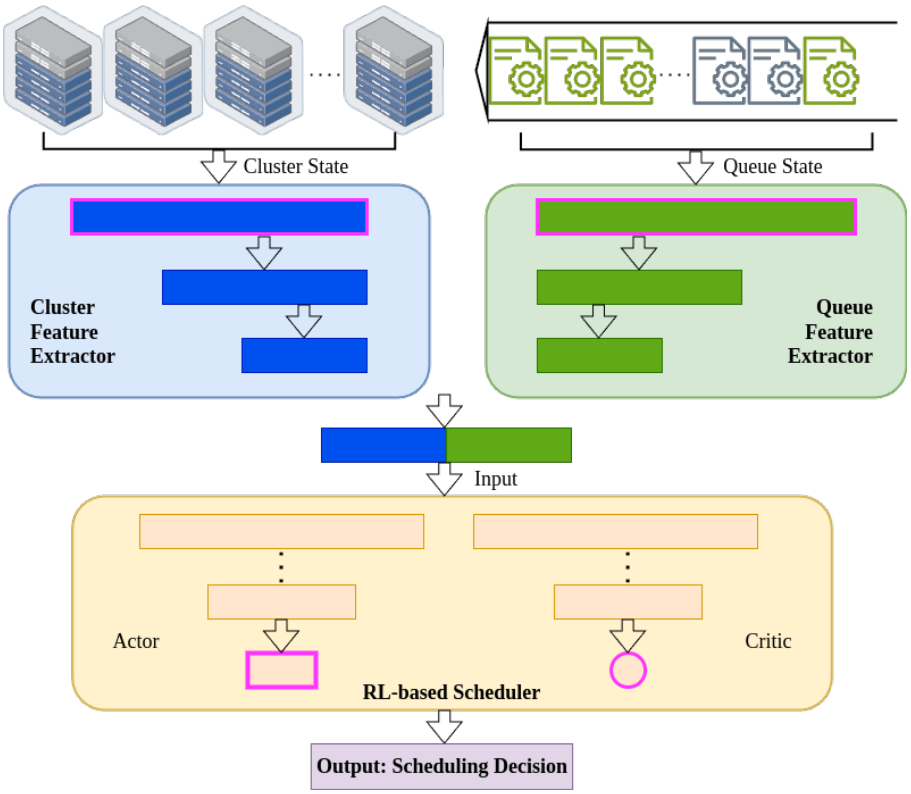

FIGURE 6.1: Separate Feature Extractor

process cluster state information and queue state information. The extractors ensure that the information from the two domains does not interfere with or distort each other before being taken by the RL-based scheduler. It follows the common practice in RL-based schedulers, where cluster and queue states are treated as distinct inputs, maintaining the scheduler's performance by producing an output format consistent with its original input design. The central RL-based scheduler and its original assumptions determine the exact details of these representations. Additionally, the separation of cluster and queue feature extractors also significantly reduces the number of trainable parameters compared to the combination of them. This reduction in complexity allows for more efficient training and faster convergence.

6.4.2 Selective Transfer Learning

Selective transfer learning is a method that enables an RL-based scheduler with separate feature extraction to adapt to a new HPC cluster by fine-tuning only the components of the model that are affected by changes in the environment. Components that are unaffected by the transition, and therefore still applicable, are left unaltered. In the

transition to a new HPC cluster, instead of retraining the entire model, we resize the first layer of the feature extractor and the output layer of the RL-based scheduler to accommodate changes in state and action spaces. Fig. 6.1 presents an example of an Actor-Critic RL-based scheduler coupled with separate feature extraction. When this scheduling system is transferred to a new cluster, our primary goal is to enable quick adaptation to the new environment. However, given that the state representation is likely to change, we reform the first layer of both feature extractors to accommodate the new input size corresponding to the new environment. Additionally, since the action space of the RL-based scheduler may also change, such as in the selection and allocation of jobs, the output layers of the scheduler's actor and critic must be adjusted accordingly. Specifically, only the layers highlighted in pink in Fig. 6.1, need to be retrained. By retraining just the essential components, we ensure that the core knowledge and learned behaviors of the scheduler are retained, reducing the risk of performance degradation and transfer learning costs during the transition. The advantage of this approach is that it allows for rapid adaptation to the new cluster while minimizing the need for extensive retraining, thereby preserving the efficiency and effectiveness of the original scheduling system.

6.5 Evaluation

We evaluate our approach through three case studies, based on three specific clusters and their corresponding workloads. In each case, we conduct two experiments where schedulers trained on the other two clusters transition to the target cluster. These case studies collectively cover wide real-world scenarios of cluster transitions, simulating new deployments, hardware upgrades, and system changes in HPC environments. Through the cases, we evaluate how quickly the scheduler adapts to the new environments and how much data is needed for successful transfer learning. Specifically, we investigate:

- Can the RL-based scheduler, with separate feature extraction and selective transfer learning, quickly adapt to new or modified clusters?
- Is it possible to achieve this adaptation without relying on large historical data from the new or modified cluster as they are likely unavailable?

To evaluate the effectiveness of our proposed approach, we selected HeraSched described in Chapter 4 as the base model. HeraSched is an HRL-based scheduler designed for heterogeneous HPC systems, managing both job selection and resource allocation in a hierarchical structure. This dual capability distinguishes HeraSched from other RL-based schedulers, which are typically limited to job selection. This comprehensive capability makes HeraSched an ideal candidate for assessing transfer learning across both scheduling tasks.

6.5.1 Baseline Schedulers

To assess the effectiveness of our proposed approach, we benchmark it against established heuristic-based methods in HPC environments. HPC scheduling involves two key processes: job selection and resource allocation. We selected seven heuristic-based job selectors for our baseline comparison, each paired with a backfilling mechanism and three heuristic-based resource allocators. The baseline job selectors include **FCFS** (First-Come First-Served), **LCFS** (Last-Come First-Served), **SJF** (Shortest Job First), **WFP3** [5], **UNICEP** [5], **F1** [16], and **F2** [16], all with **backfilling** [96] enabled. Among these, **WFP3** (Weighted Fair Priorities 3) adjusts job priority dynamically, favoring jobs with longer waiting times but normalizing based on resource requests. **UNICEP** refines WFP3 by incorporating additional system state factors to balance fairness and responsiveness. **F1** and **F2** are scheduling heuristics derived through simulation-based supervised learning, optimizing job prioritization based on historical job execution patterns. The resource allocators include **First-Available**, which assigns jobs to the first set of nodes that meet resource requirements, **Best-Fit**, which selects nodes to minimize resource fragmentation, and **Topology-Aware** [110], which places jobs on nodes that minimize the number of lowest-level network switches they traverse. Consequently, the baseline methods consist of 21 schedulers, representing all possible combinations of these job selectors and resource allocators.

These heuristic-based schedulers are selected as baselines because they represent practical and easily deployable scheduling strategies in newly developed or modified HPC clusters. Unlike learning-based approaches, heuristic methods do not require extensive training, making them immediately applicable in environments where historical job data may be limited or where rapid deployment is necessary. Additionally, our evaluation

aims to determine whether HeraSched, enhanced with our approach, can not only quickly outperform the baseline heuristic methods but also achieve performance comparable to HeraSched trained from scratch. This chapter studies transfer learning for a fixed scheduler (HeraSched) rather than proposing a new scheduler. To isolate adaptation effects, we compare the transferred model against the same HeraSched architecture trained from scratch on the target partition and a bunch of baseline methods. We do not repeat broader RL-to-RL scheduler comparisons here; those are established in Chapter 4.

6.5.2 Cluster Characterisation

The clusters used in our case studies are distinct partitions of a real HPC system called Spartan [1] and are referred to as *Physical*, *Deeplearn*, and *Sapphire*. Physical is a CPU-based partition optimized for distributed, CPU-intensive tasks. Deeplearn is GPU-centric and designed for machine learning workloads with powerful GPUs. Sapphire is a newly developed CPU partition with significantly more powerful cores than Physical. The transitions between these partitions cover a wide range of real-world scenarios, from adapting to CPU resource changes, shifting from CPU-based workloads to GPU-intensive tasks, and transiting from GPU partitions to CPU partitions.

Table 6.1: Cluster: Physical, Deeplearn, and Sapphire Characteristics

Name	Physical		Deeplearn					Sapphire
Partition	CPU		GPU					CPU
Cores/node	72	72	28		24	32		128
Mem(GB)/node	710	1519	234	174	175	234	1000	977
Total Nodes	72	14	4	5	3	6	12	52
GPUs/node	0		4					0
Period	2022-9-23 to 2022-9-30		2021-9-20 to 2022-9-30					2024-5-1 to 2024-5-7
Jobs	84135		68720					54263
Total Cores	6192		900					6656
Total GPUs	0		120					0
Max Runtime	30 days							

Table 6.1 highlights key differences among the three partitions. Physical is a CPU-based partition with 72 cores per node, while Sapphire, another CPU-based partition, features more powerful CPUs with 128 cores and 977 GB of memory. Transitioning between Physical and Sapphire involves adapting to Sapphire's higher core density, requiring the scheduler to efficiently handle this changing computational capacity to utilize the nodes. Deeplearn, a GPU-based partition, introduces another layer of complexity. Its

nodes feature fewer CPU cores (24 to 32 per node) but include 4 GPUs per node and between 174 and 1000 GB of memory. Transitioning to Deeplearn from either Physical or Sapphire involves a significant shift from CPU-bound workloads to GPU-accelerated tasks, requiring the scheduler to manage jobs that rely on both CPUs and GPUs. The challenge is maximizing GPU utilization while also accommodating Deeplearn's more limited CPU capacity. When transitioning from Deeplearn to either Physical or Sapphire, the scheduler faces the opposite challenge.

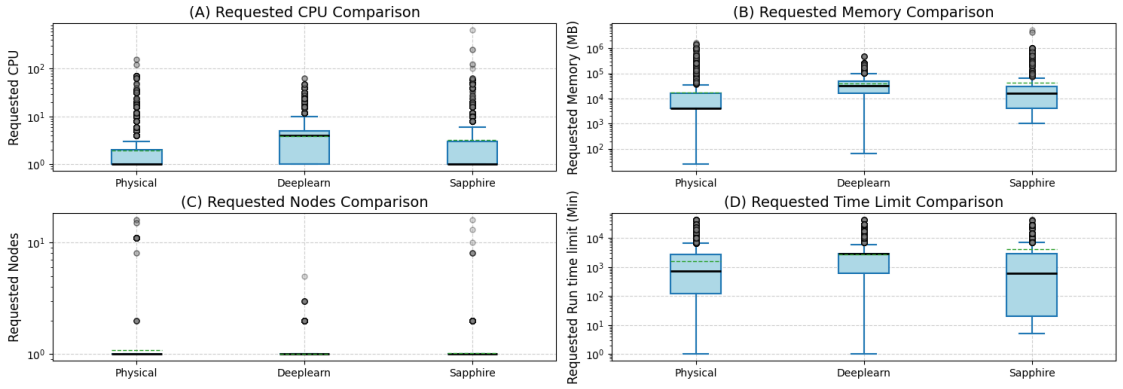

FIGURE 6.2: Comparison of requested resources across three job workloads: Physical, Deeplearn, and Sapphire. The subplots illustrate the distribution of requested CPU cores (A), memory (B), nodes (C), and time limit (D) for jobs in each workload. The boxplots show the spread and central tendency of resource requests, with the mean indicated by dashed lines and the median indicated by solid lines, highlighting variations in resource demands across the different workloads.

Fig. 6.2 illustrates the characteristics of job-requested resources across three workloads during the collected periods. The subplot for Requested CPU shows that the median for both Physical and Sapphire clusters is 1, while Deeplearn has a higher median, indicating that jobs in Deeplearn generally request more CPU cores per job. However, in the Requested Nodes comparison, while the median for all three workloads is 1, Physical and Sapphire exhibit slightly higher averages in the requested number of nodes compared to Deeplearn. Memory requests vary significantly. Deeplearn has a higher average and median requested memory than others. This can be explained by the nature of jobs in these clusters. In Deeplearn, which is a GPU partition, jobs tend to request more resources within a single node to fully leverage the computational power of GPUs. For example, machine learning tasks often require multiple CPU cores and more memory to support GPU usage, leading to fewer nodes being requested but more resources being concentrated in each node. On the other hand, Physical and Sapphire are CPU-based clusters, where jobs tend to be more distributed across multiple nodes. Some CPU tasks require extensive

communication between nodes, benefiting from a distributed setup. However, many CPU tasks fall under the category of embarrassingly parallel problems [140], where jobs run independently on individual nodes without requiring communication between them. In such cases, users often submit a large number of jobs that request only one node and a few cores per job. This behavior leads to a significant gap between the median and average number of requested CPU cores. While the median remains low, reflecting the fact that many jobs request minimal resources, the average is higher due to the presence of a smaller number of jobs that request significantly more cores for more complex, resource-intensive tasks. This pattern is especially evident in both Physical and Sapphire, where embarrassingly parallel jobs are common, contributing to the higher average despite the low median. In summary, the cluster configurations and job workloads across Physical, Deeplearn, and Sapphire highlight the distinct challenges faced by the RL-based scheduler in a transfer learning context.

6.5.3 Model Training and Transfer Learning

To experiment with cluster transitions, it is essential to first train the RL-based schedulers for each cluster individually. In our approach, we used HeraSched as the base RL-based scheduler by integrating the separate feature extraction mechanism and fully retraining the HRL models for each specific workload. The training is conducted using a simulation (described in 4.4.2) rather than directly on large-scale HPC clusters, as real-world training and testing would be prohibitively expensive. Instead, we leverage real trace data and actual HPC cluster configurations to build a high-fidelity simulated environment that closely mimics real scheduling scenarios. Table 6.2 provides an overview of the training settings, detailing the structure of the separate feature extractors for both cluster and queue states, as well as the hidden layers used in HeraSched's actors and critics. The window size defines the total number of jobs that can be observed by the scheduler. The tail size specifies how many jobs from the tail of the queue the scheduler can observe. This configuration allows the scheduler to consider both the current jobs at the front of the queue and a subset of jobs from the end of the queue, potentially improving scheduling decisions by incorporating a broader perspective 3.3.1. The sizes of the separate feature extractors are determined based on their proportional relationship to the original state size in HeraSched, ensuring alignment with its feature representation. Additionally, the hidden layer sizes and window settings are adopted from Table 4.2 to maintain consistency

with its training methodology. The trained models will serve as the base models, reflecting the realistic conditions encountered when adapting a scheduler to a newly established or modified cluster.

TABLE 6.2: Train from Scratch Settings (HeraSched)

Feature Extraction Layers	Cluster Extractor	Queue Extractor
	[4096, 3072]	[2048, 1024]
Extractor Activation Function	ReLU	
Output from Extractor	4096	
Hidden Layers	Actor	Critic
	[2048, 1024]	[2048, 1024]
HeraSched Activation Function	Tanh	

HeraSched Window Size	Window	Tail
Physical	600	60
Sapphire	600	60
Deeplearn	100	10

For transfer learning, we aim to adapt the base models with Separate Feature Extraction to a new or modified environment. Leveraging Selective Transfer Learning allows us to focus the adaptation process on specific parts of the model by updating the first layer of each feature extractor to account for changes in the cluster and queue states and adjusting the output layer of the RL agents to accommodate differences in the action space. The transfer learning is implemented based on the Maskable PPO from StableBaselines3 [132], and Table 6.3 details the hyperparameters used. These hyperparameters are derived from Table 4.2, ensuring consistency with its established training methodology. The number of steps per update determines how many steps (or experiences) are collected from the environment before the model updates its policy. The following experiments will highlight how fewer updates lead to quicker adaptation while still outperforming baseline methods, showcasing the method's efficiency in real-world HPC operations.

TABLE 6.3: Hyperparameters for Transfer Learning

Hyperparameter	**Value**
Learning Rate	3e-4
Number of Steps per Update	2048
Batch Size	64
Number of Epochs per Update	10
Discount Factor (Gamma)	0.99
GAE Lambda	0.95
Clipping Range	0.2

6.5.4 Case Study 1: Deeplearn Partition Adaptation

In this case study, we evaluate our approach by adapting schedulers to a significantly different environment. Specifically, we focus on transitioning schedulers trained and used in CPU-based partitions (Physical and Sapphire) to a GPU-based partition (Deeplearn). The Physical and Sapphire clusters consist solely of CPU nodes, supporting CPU-bound, distributed tasks, while the Deeplearn cluster, equipped with GPU-accelerated nodes, handles memory-intensive machine-learning jobs. This case study investigates whether our proposed approach can adapt schedulers trained on CPU-focused clusters to a GPU-intensive environment, enabling efficient GPU allocation and managing significantly different cluster configurations without extensive retraining or large amounts of new data.

Fig. 6.3 shows the heatmaps of average and maximum waiting times for different baseline heuristic scheduling methods across the three partitions. In subplots (A) and (B), the Deeplearn partition's performance is shown. In all baseline methods, **FCFS** with backfilling cooperating with **Best-Fit** yields the lowest maximum waiting time (55885 seconds) and ranks second in average waiting time (1361 seconds), making it the best-performing combination. In the following comparison, we use FCFS and Best-Fit as the best heuristics to compare the performance of the proposed methods.

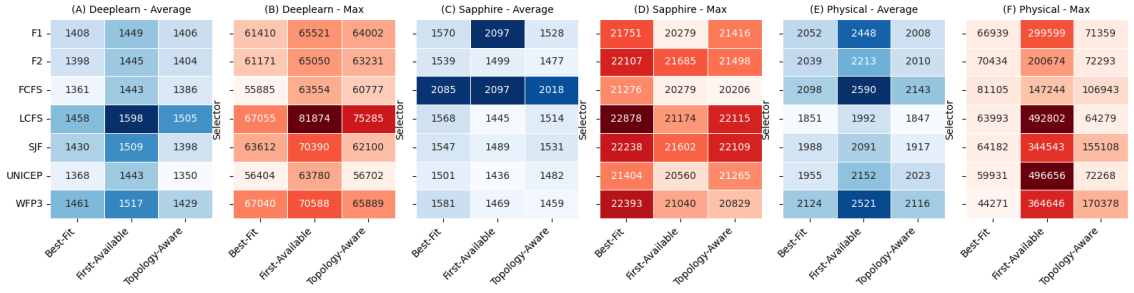

FIGURE 6.3: Heatmaps illustrating the average waiting times (seconds) and maximum waiting times (seconds) for the heuristic selectors combined with a backfilling mechanism across three allocation methods in the Deeplearn, Sapphire, and Physical workloads.

Fig. 6.4 compares the performance of the schedulers transferred from the Physical and Sapphire partition to Deeplearn partition, a scheduler trained in the Deeplearn partition from scratch, and the best-performing heuristics in the Deeplearn partition. The values are normalized by the best-performing heuristics, FCFS with Best-Fit (average waiting time: 1361 seconds, maximum waiting time: 55885 seconds), where the dashed line at 2 represents the performance of the best heuristics. The normalization combines average

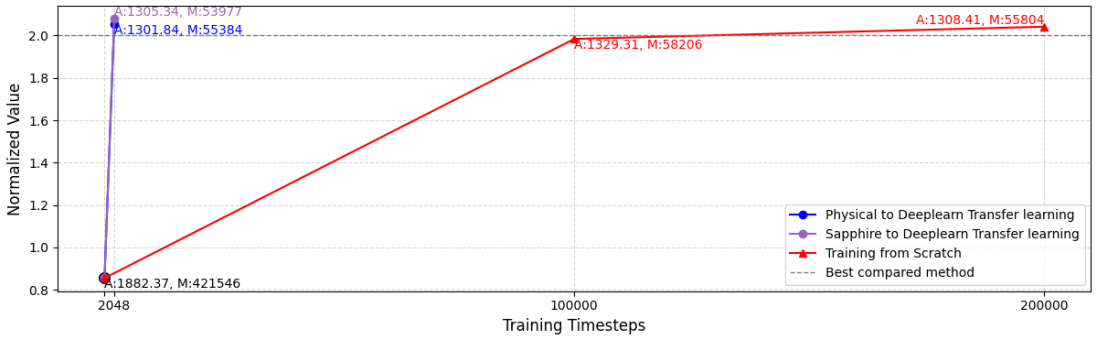

FIGURE 6.4: Comparison of transfer learning performance against training from scratch, showing normalized values of maximum and average waiting times by best performed Heuristics.Normalized Value $= \frac{\text{best\_heuristic}_{\text{avg}}}{\text{model\_waiting\_time}_{\text{avg}}} + \frac{\text{best\_heuristic}_{\text{max}}}{\text{model\_waiting\_time}_{\text{max}}}$.

and maximum waiting times into a single measurement, making it easier to compare performance across both metrics. This unified approach provides a clearer and more concise evaluation of the model's effectiveness relative to heuristic methods. A value above 2 indicates better performance than the best heuristics. Transfer learnings from both partitions demonstrate remarkable efficiency, achieving substantial adaptation with just **a single update** at 2048 timesteps, quickly outperforming the baseline of the best heuristics. This highlights how effectively the proposed methods leverage prior knowledge to adapt to new job workloads with minimal training effort. In contrast, training from scratch requires a significantly longer process to reach comparable performance. Even after 100,000 timesteps, the scratch-trained model only approaches the performance of the best heuristics, but it is still behind. After 200,000 timesteps, the model trained from scratch only begins to match the performance of transfer learning's single update.

For the transfer learning experiment, we utilized a subset of the total job data to perform one update. Specifically, out of 68,720 jobs in the workload, only about 2,100 jobs (3%) were used in 2048 timesteps to train the model. The remaining jobs (97%) were kept as a chronological hold-out test set with no gradient updates or tuning. This represents just 1% of the training timesteps required by the training-from-scratch approach, which involved training the model from the beginning using all available job data. Despite being trained on only 3% of the jobs, the transfer learning approach successfully performed well on the entire workload, as reflected in the results shown in the plot. Additionally, the training process was conducted on a PC with a Ryzen 7700 processor and an Nvidia 3090 GPU. The one update took approximately 7 seconds to complete, underscoring the fast adaptation capabilities of the approach. The combination of rapid training time and the small amount of data required for the update shows that the proposed method is highly

efficient, both in terms of computational resources and data utilization. This efficiency supports the argument that the approach can quickly adapt to new or modified clusters without needing large historical datasets, making it suitable for environments where data is scarce or unavailable.

6.5.5 Case Study 2: Sapphire Partition Adaptation

In this case study, we assess the effectiveness of the proposed transfer learning approach by adapting the models trained and used in Physical and Deeplearn partitions to the Sapphire HPC cluster. The transition from Physical to Sapphire represents adapting to an upgraded CPU cluster with higher core density and overall core count. On the other hand, the transition from Deeplearn to Sapphire involves adapting from a GPU-dominant environment to a CPU-centric one.

Fig. 6.3 shows the results of using combinations of baseline methods to schedule the entire Sapphire workload in two matrices: average job waiting time and maximum job waiting time. The performance of the allocation methods varies significantly across different heuristic selectors. Notably, the **First-Available** method shows consistent performance improvements for several job selection heuristics. Among these, **UNICEP** (backfilling included) achieves the best result in average waiting time (1436 seconds) and good performance in maximum waiting time (20560 seconds), making it the overall best performer in the baseline methods for the Sapphire partition.

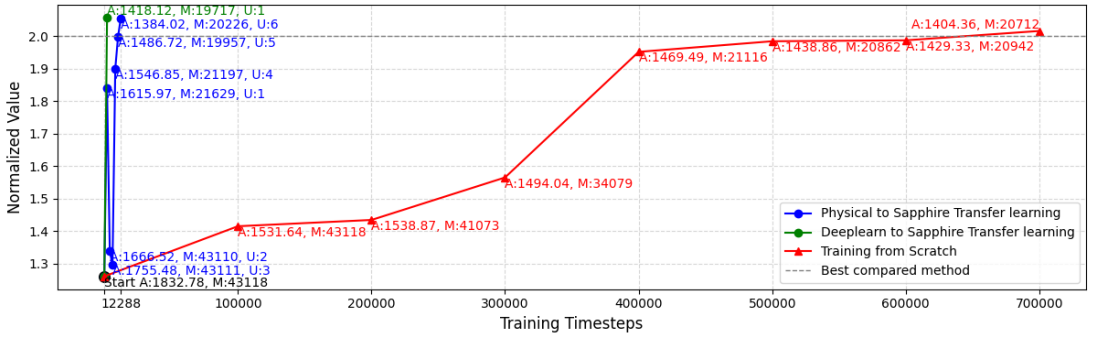

FIGURE 6.5: Comparison of transfer learning performance against training from scratch, showing normalized values of maximum and average waiting times by best performed Heuristics. Transfer learning points are plotted at various stages of the learning process, with detailed annotations for each step (A: average, M: maximum, U: update times).Normalized Value $= \frac{\text{best\_heuristic}_{\text{avg}}}{\text{model\_waiting\_time}_{\text{avg}}} + \frac{\text{best\_heuristic}_{\text{max}}}{\text{model\_waiting\_time}_{\text{max}}}$.

Fig. 6.5 compares the performance of transfer learning schedulers from the Physical and Deeplearn partitions and a scheduler trained from scratch in the Sapphire, showing normalized values of maximum and average waiting times, with the best-performing heuristics (UNICEP with First-Available) used as a baseline. The Deeplearn to Sapphire transfer learning (in green) achieves adaptation in only one update. This performance with 2048 timesteps training even outperforms the performance of training from scratch at 700,000 timesteps. The Physical to Sapphire transfer learning process (in blue) consists of multiple updates, with each update representing 2048 timesteps. As seen in the plot, the first update (U:1) already provides a significant performance improvement, showing that transfer learning quickly adjusts to the new cluster. However, the model's performance fluctuates slightly in the earlier updates, due to exploration, before stabilizing and improving again. By the fifth update, the model reaches the performance level of the best heuristics, and then at 12,288 steps, it outperforms the best heuristics. In contrast, the training from scratch process (in red) improves more slowly. Even after 100,000 timesteps, the scratch-trained model still struggles to yield relatively good performance, and it only begins to reach the performance level of the best heuristics after 500,000 timesteps. The first point where the scratch-trained model outperforms the best heuristics occurs much later, at around 700,000 timesteps.

Despite the increased complexity compared to both Deeplearn and Physical partitions, the adaptation remains remarkably efficient, as transfer learning significantly outperforms the best heuristics with only a small fraction of the training steps needed for training from scratch. Moreover, to outperform the best heuristics, the Deeplearn to Sapphire transition required just one update, using around 2,100 jobs out of the total 54,263 jobs, which represents about 3.8% of the full workload. The remaining jobs were kept as a chronological hold-out test set with no gradient updates or tuning. While the Physical-to-Sapphire transition required more steps (12,288 steps), this was still accomplished using only 1.76% of the training timesteps needed for training from scratch. The update was completed in 1 minute and 24 seconds using a PC with a Ryzen 7700 processor and an Nvidia 3090 GPU. In comparison, the Deeplearn case required just 7 seconds per update. This increased time is due to the significantly larger cluster state and queue state in Sapphire compared to Deeplearn. Although the models were larger and the adaptation process more computationally demanding, it's still notable that the adaptation was achieved in a quick and reasonable time frame using a regular PC setup. This highlights the scalability

and practicality of the transfer learning approach, even when dealing with high-density, core-intensive clusters like Sapphire.

6.5.6 Case Study 3: Physical Partition Adaptation

In this case study, we assess the ability of the RL-based schedulers to adapt to the Physical partition. This adaptation involves two transitions: from Sapphire to Physical and from Deeplearn to Physical. The transition from Sapphire to Physical reflects a shift to a less powerful CPU environment with fewer cores per node but with more total nodes. On the other hand, the transition from Deeplearn to Physical illustrates the scheduler's adaptation from a GPU-accelerated environment to a purely CPU-based setup, demanding a significant reconfiguration in job scheduling.

Fig. 6.3 shows the results of using baseline methods to schedule the entire Physical workload. In this setup, the **Best-Fit** allocation method yielded competitive performance across the compared allocation methods. When combined with **LCFS** and Backfilling, it achieves the second-best average waiting time (1851 seconds) and the third-best maximum waiting time (63993 seconds), making it the top-performing baseline heuristic for the Physical partition overall.

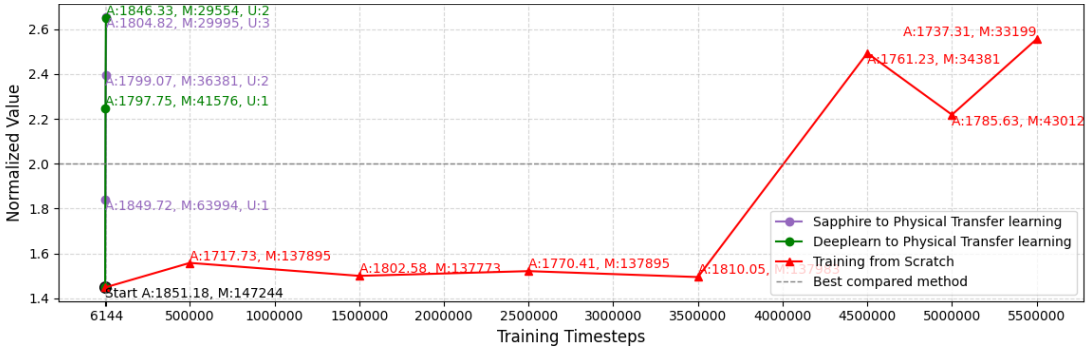

FIGURE 6.6: Comparison of transfer learning performance against training from scratch, showing normalized values of maximum and average waiting times by best performed Heuristics. Transfer learning points are plotted at various stages of the learning process, with detailed annotations for each step (A: average, M: maximum, U: update times).Normalized Value $= \frac{\text{best\_heuristic}_{\text{avg}}}{\text{model\_waiting\_time}_{\text{avg}}} + \frac{\text{best\_heuristic}_{\text{max}}}{\text{model\_waiting\_time}_{\text{max}}}$.

Fig. 6.6 compares the performance of transfer learning methods against a scheduler trained in Physical from scratch with the best-performing heuristics as the baseline. Both transitions, from Sapphire and Deeplearn to Physical, demonstrate that transfer learning achieves efficient adaptation. The Deeplearn to Physical transfer learning (in green)

adapts in just one update, achieving performance that surpasses the baseline heuristics with only 2048 timesteps. The Sapphire to Physical transfer learning (in purple) also shows significant improvement after two updates, already outperforming the baseline heuristics. By the second and third updates of the Deeplearn and Sapphire transitions respectively, the models achieve further performance gains, significantly surpassing the best baseline method. Compared to the best heuristics, the performance of the RL-based scheduler shows significant improvement in this case. One reason for this is that the Physical partition handles a higher volume of jobs compared to Sapphire and Deeplearn. In a busier environment, machine learning-based approaches, especially RL, can leverage the increased job traffic and have more opportunities to optimize job selection and resource allocation. This allows the scheduler to adapt more efficiently, making more informed decisions that result in better overall system performance compared to heuristic methods. In contrast, the training-from-scratch process (in red) struggles significantly compared to the transfer learning methods. Even after 3.5 million timesteps, the scratch-trained model still performs worse than the baseline heuristics.

The Deeplearn-to-Physical transition required just one update, using approximately 2,100 jobs out of the total 84,135 jobs (around 2.4% of the workload), and was completed in about 1 minute and 28 seconds on the same PC as previous experiments. The remaining jobs were kept as a chronological hold-out test set with no gradient updates or tuning. In comparison, the Sapphire-to-Physical transition required an additional update but still achieved remarkable efficiency. Specifically, while training from scratch required 5,500,000 timesteps, the transfer learning approach used only 4,096 timesteps and 6,144 timesteps across the two updates, representing just 0.2% of the training timesteps needed for training from scratch.

6.6 Summary

This chapter introduced a novel transfer learning framework to accelerate the adaptation of reinforcement learning-based schedulers in evolving HPC environments. By combining Separate Feature Extraction and Selective Transfer Learning, our approach enables schedulers to efficiently adjust to new or modified cluster architectures with minimal retraining. Through six transition experiments across three real-world HPC partitions — Physical, Sapphire, and Deeplearn — we demonstrated that the proposed method not

only rapidly outperforms 21 baseline heuristic schedulers but also matches the performance of RL models trained from scratch at a fraction of the training cost. Remarkably, this level of performance was attained while utilizing only 2–3% of the total workload jobs. Furthermore, across all experiments, this adaptation required just 0.2–1.76% of the training timesteps needed for training from scratch — representing up to a 500× reduction in training effort. These results highlight the strong potential of the proposed method to support adaptive scheduling in HPC environments with minimal overhead.

Chapter 7

Conclusions and Future Directions

7.1 Conclusions

This thesis investigated key problems in HPC scheduling with deep reinforcement learning, including designing job selection, integrating resource allocation with job selection, balancing system-centric and user-centric objectives in changing environments, and adapting RL-based schedulers to evolving HPC architectures. It identified research gaps within these areas, formulated targeted research questions, and developed novel RL-based frameworks and methodologies to address them. The key conclusions from each chapter are summarized below.

- **Chapter 1: Introduction**

  This chapter introduced the thesis, outlining the fundamental challenges in HPC scheduling with reinforcement learning. It identified four key challenges. The chapter also highlighted critical research gaps within these areas and formulated the research questions addressed in this thesis. Finally, it provided an overview of the thesis structure, linking each chapter to the research objectives and key findings.

- **Chapter 2: Background and Literature Review**

  This chapter provided the necessary background and literature review for this thesis. It introduced fundamental concepts of HPC job scheduling and reinforcement learning, establishing the theoretical foundation for the research. Additionally, it surveyed existing HPC scheduling techniques, including heuristic-based,

meta-heuristic-based, machine learning-based, and reinforcement learning-based approaches. The chapter highlighted key advancements and limitations in the literature, identifying gaps that motivate the proposed research.

- **Chapter 3: Advancements in RL-Based Job Selection in HPC**

  This chapter introduced an RL-based job selector, DBF, designed to address the challenge of unbounded state and action spaces in RL-based job selection. It presented the Split Window Technique, which enables the scheduler to observe both the head and tail of the job queue, extending the agent's observation when large jobs accumulate at the head of the queue. This ensures that newly arrived jobs are not overlooked while mitigating the limitations of fixed-window approaches. Additionally, by allowing job selection from the tail of the queue, the Split Window Technique provides more opportunities for backfilling, improving overall resource utilization. Unlike existing RL-based job selectors that rely on separate backfilling processes, DBF integrates a novel Schedule Cycling mechanism, allowing the RL agent to autonomously learn and apply backfilling strategies without extra processes. Experimental results demonstrated that DBF outperforms other job selectors by significantly reducing job waiting time and queue length, offering a more efficient and adaptive solution for HPC job selection.

- **Chapter 4: HeraSched: HRL-Based Scheduler**

  In this study, HeraSched, a novel HRL-based HPC job scheduling solution, was introduced to address the growing complexities of resource allocation and dynamic workload variations in modern HPC systems. Unlike existing approaches that primarily focus on job selection while relying on simplistic allocation policies, HeraSched integrates intelligent job selection with heterogeneity-aware node-level allocation, ensuring efficient job scheduling across CPU and GPU partitions. By incorporating backfilling directly into job selection and leveraging hierarchical decision-making, HeraSched enhances scheduling efficiency and minimizes job starvation. Evaluation results demonstrated significant reductions in job waiting times, particularly under high system loads, with HeraSched consistently outperforming compared methods, which are combinations of existing selectors and allocators. Additionally, its ability to adapt to varying workload demands underscores its robustness in real-world HPC environments.

- **Chapter 5: MetaPilot: Scheduling Controller for Balancing Objectives**
  This chapter introduced MetaPilot, a reinforcement learning-based scheduling controller that dynamically balances user-centric and system-centric objectives in HPC scheduling. Unlike traditional fixed-objective schedulers, MetaPilot adapts its decision-making to real-time workload characteristics and resource availability, effectively bridging the gap between maximizing resource utilization and minimizing job waiting times. By acting as an adaptive decision layer rather than replacing existing schedulers, MetaPilot significantly improves scheduling flexibility, demonstrated by a 19% reduction in maximum waiting time alongside enhanced resource utilization under varying workload conditions.

- **Chapter 6: Scheduler Adaptation in Evolving HPC**
  Chapter 6 introduced a novel approach to rapidly adapt RL-based HPC schedulers to evolving or newly configured HPC architectures. It proposed the Separate Feature Extraction and Selective Transfer Learning techniques, which isolate distinct changes in cluster configurations and workload dynamics, significantly reducing the retraining effort required during scheduler transitions to new environments. Evaluations conducted across three realistic HPC cluster scenarios demonstrated that schedulers enhanced with these techniques achieved performance comparable to schedulers trained from scratch, while requiring substantially fewer training steps. In particular, results indicated a reduction in retraining time and computational effort by up to 500 times compared to traditional training methods, highlighting the practical efficacy of this approach in managing dynamic HPC environments.

In conclusion, this thesis has made significant contributions to advancing reinforcement learning-based HPC scheduling by proposing innovative frameworks and methodologies that effectively address core challenges and enhance scheduler performance. Through novel solutions in job selection, hierarchical job scheduler, dynamic balancing of competing objectives, and rapid adaptation to evolving HPC architectures, the research outcomes provide robust foundations and open promising avenues for further exploration and real-world applications in intelligent HPC scheduling.

7.2 Future Directions

We present several promising research directions that have the potential to enhance scheduler adaptability, improve overall system efficiency, and support more practical deployment in real-world HPC environments.

Handling Resource Oversubscription

In production HPC environments, users frequently overestimate the resources their jobs require — particularly CPU cores, memory, and runtime. This behavior is often a precaution to avoid job failures due to underestimated needs, especially in systems with limited user feedback or resource monitoring. While understandable from a user standpoint, this practice, known as resource oversubscription, leads to substantial inefficiencies in cluster utilization. Resources that are reserved for a job but not actually used remain idle and unavailable to other pending jobs, increasing queue times and reducing system throughput. Effectively handling resource oversubscription requires intelligent, adaptive scheduling techniques that can dynamically adjust user-submitted resource requests to better reflect actual usage. This can be achieved through predictive models that learn from historical job data, user behavior patterns, and real-time monitoring. These models can estimate more accurate resource requirements at submission time or adjust them just before scheduling.

Integrating such predictive capabilities into RL-based scheduling frameworks presents a promising research direction. The RL agent can learn to interpret predictive estimates and make job placement decisions that increase cluster efficiency while maintaining high job success rates. However, the risk of under-provisioning must be carefully managed — allocating fewer resources than a job actually requires can lead to job failures, data loss, or performance degradation. Therefore, future solutions should include confidence estimation, safety margins, or dynamic runtime adjustments to ensure reliability while still improving efficiency. Another promising direction is online feedback loops. Rather than making static predictions at submission time, the scheduler could continually monitor resource consumption and adjust allocations at runtime, especially in systems that support elastic resource management or job migration. This can enable dynamic request scaling, where jobs are granted additional resources if needed or scaled back when over-provisioned.

Job Colocation Without Performance Interference

To further enhance resource utilization, colocating multiple jobs on partially used compute nodes — such as assigning a new job to idle CPU cores that are not fully utilized by an existing job — is a promising strategy. This fine-grained colocation approach aims to maximize per-node efficiency by filling in underused compute capacity. It is particularly relevant in modern multi-core and multi-threaded nodes, where a single job may not fully saturate all cores or memory bandwidth. By strategically placing complementary jobs on the same node, overall system throughput can be improved without needing additional hardware. However, naive colocation can lead to severe performance interference, especially when colocated jobs contend for shared resources such as memory bandwidth, I/O channels, cache hierarchies, or power budgets. These forms of contention are not always apparent at scheduling time and may result in significant slowdowns, quality-of-service violations, or unpredictable job performance. Moreover, interference can vary depending on workload types, resource usage patterns, and even hardware topology — making it difficult to generalize or hard-code rules for safe colocation.

Future research should focus on developing interference-aware job placement strategies. This involves learning interference models from prior job co-executions and incorporating them into the scheduling decision process. A well-designed scheduler must balance the benefits of improved utilization with the risk of contention, ideally learning to identify compatible job pairings and placement patterns that maximize throughput without sacrificing performance isolation.

Cold Starts in Real HPC Deployment

Most of the existing research on RL-based HPC scheduling is developed and evaluated within controlled simulators, where job characteristics, resource configurations, and scheduling events are simulated. While simulators are useful for prototyping and initial exploration, they often fail to capture the full complexity, and unpredictability of real HPC systems. This disconnect presents a significant challenge when transitioning from simulation to deployment — a problem often referred to as the cold start in real HPC environments. In real-world deployment, a new RL-based scheduler typically lacks access to rich historical data from the target cluster, making offline training less effective or even

infeasible. Moreover, naïvely applying a freshly trained model from a simulator can result in poor early performance due to mismatched assumptions or unseen behaviors, leading to job delays, resource underutilization, or user dissatisfaction.

Future research should investigate strategies for enabling safe and effective cold-start deployment of learning-based schedulers in real HPC systems. Key directions include designing bootstrapping techniques that allow the scheduler to operate with minimal historical data, and developing robust online learning mechanisms that continuously adapt the policy based on real-time observations. Techniques such as simulation-to-real transfer, warm-starting with domain-invariant features, or combining RL with conservative fallback heuristics during early deployment may help bridge the gap between simulated training and practical use. Ensuring that the scheduler can quickly converge to a performant policy in real clusters without significant degradation remains an essential step toward the broader adoption of RL-based scheduling in HPC.

Fairness-Aware Scheduling

Fairness is a longstanding but still unresolved challenge in HPC scheduling. Despite its importance, there is no universally accepted definition of fairness in this context, as fairness can depend on a wide range of factors. In shared HPC systems, fairness may involve distributing compute resources equitably across individual users, research groups, or organizational units. It can also consider historical usage patterns, such as how much computing time a user or group has consumed in the past, the types and amounts of resources requested (e.g., CPU cores, memory, GPUs), and even priority levels or funding allocations. A fundamental research need is to develop a principled and flexible definition of fairness that captures the multidimensional nature of HPC resource sharing. This includes not only who is receiving resources, but how much and what kind of resources they receive, and over what time horizon fairness should be evaluated (e.g., per job, per week, or per project).

Moreover, fairness in HPC scheduling is not static. As system usage evolves — with new users joining, project priorities shifting, or workload characteristics changing — the scheduler must be able to dynamically adjust fairness policies in real-time. A fairness-aware RL scheduler should therefore incorporate adaptive mechanisms that respond to

these changes, balancing short-term scheduling efficiency with long-term fairness objectives.

Future work should focus on designing fairness metrics that are both accurate and actionable, and integrating them into the RL framework in a way that allows the agent to learn fairness-aware policies. This may involve reward shaping, constrained RL, or multi-objective optimization. Importantly, fairness should not be treated in isolation — it must be balanced with competing goals such as resource utilization, job throughput, and user responsiveness. Developing RL-based schedulers that can make these trade-offs transparently and adaptively remains a critical direction for real-world deployment.

Energy-Aware Scheduling

Energy efficiency is an increasingly critical concern in modern HPC systems, driven by both economic and environmental factors. Large-scale clusters consume vast amounts of power, contributing to high operational costs and significant carbon footprints. As a result, minimizing energy consumption while maintaining system performance has become a key scheduling objective. Scheduling decisions affect energy usage through multiple pathways. At the node level, energy consumption depends not only on CPU and memory utilization but also on how jobs are placed across resources — idle but powered-on nodes still draw significant energy, and frequent power-state transitions (e.g., powering nodes up/down) introduce overheads. Job placement can also affect cooling efficiency: spreading high-power jobs across different racks may improve thermal balance, while clustering them in dense areas can create hotspots, increasing cooling demands. Furthermore, some modern systems include heterogeneous hardware (e.g., CPUs, GPUs, FPGAs), each with different power-performance, adding another layer of complexity to energy-aware scheduling.

Future work in this area should explore RL-based energy-aware scheduling policies that make use of fine-grained power monitoring data, thermal profiles, and hardware capabilities as part of the system state representation. These policies could learn to favor energy-efficient configurations such as job consolidation (packing jobs tightly to power down unused nodes), or intelligent load distribution that avoids thermal hotspots and reduces cooling energy costs. A key research direction is the design of reward functions that meaningfully incorporate energy-related objectives. These may include penalties

for excessive power draw, incentives for utilizing energy-efficient nodes, or cost-based formulations that reflect real-world energy pricing models. In addition, integration with hardware-level power management features (e.g., Dynamic Voltage and Frequency Scaling, power capping, or energy-aware BIOS settings) can further enhance the capabilities of an energy-aware scheduler. Coordination between software-level scheduling policies and hardware-level energy controls remains a largely unexplored but promising area.

Appendix A

Binary State Representation

Existing DRL-based scheduling methods in HPC systems often struggle to handle backfilling effectively. Many approaches either ignore backfilling actions entirely or fail to capture the idle time gaps between currently running jobs and those reserved for future execution. As a result, they miss critical opportunities to fill these system "holes" with short jobs, leading to inefficient resource usage. Explicitly encoding these idle gaps in the observation space is impractical due to the resulting unbounded and complex state space.

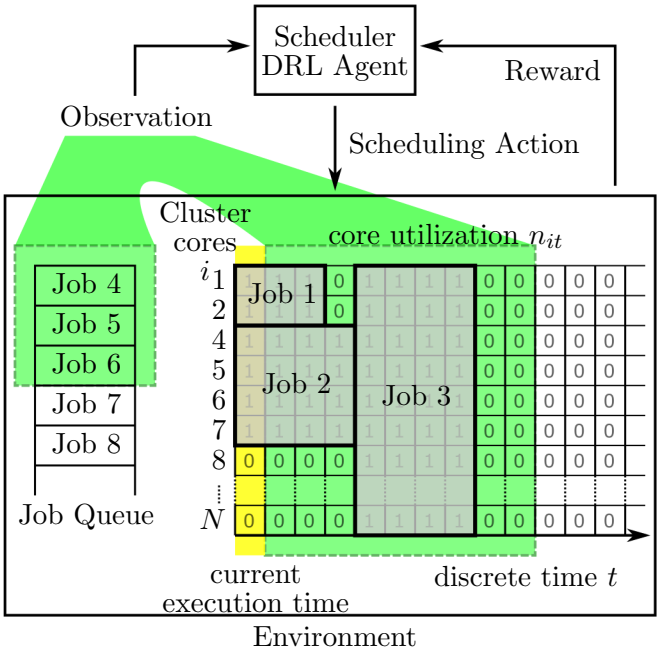

FIGURE A.1: Binary State Representation for DRL-based Scheduling. The agent observes a partial view of the binary core-time matrix and the job queue, allowing it to identify idle resources and learn effective backfilling strategies.

To address this, the method illustrated in Fig. A.1 introduces a partially observed cluster state space, allowing the agent to learn efficient scheduling—including backfilling—without requiring full visibility of the system. As shown in the figure, the environment includes two main components: a core-time matrix representing the cluster's resource allocation over discrete future time steps, and a job queue containing pending jobs. Each cell in the matrix indicates whether a specific core is busy or idle at a particular time using a binary value. The shaded green region highlights the subset of this matrix that is visible to the agent: a short time window across a limited number of compute cores. Likewise, only the earliest few jobs in the queue are exposed to the agent.

This design enables the agent to perceive upcoming scheduling opportunities — including fragmented idle regions that may be suitable for short jobs — without being overwhelmed by the full complexity of the system. Importantly, the method uses a binary representation of job-to-core reservations rather than summarizing availability through aggregate statistics such as total free cores. This fine-grained encoding makes inefficiencies in job placement, such as scattered idle cores between larger jobs, clearly visible to the agent. As demonstrated in our experiments, this representation significantly enhances the agent's ability to learn effective scheduling strategies, particularly in identifying and exploiting backfilling opportunities.

Appendix B

HeraSched Results

B.1 HeraSched Evaluation Results

Table B.1 presents a comparative analysis of job scheduling performance in the Physical Partition, focusing on the average and maximum waiting times for various schedulers, which are combinations of selectors (FCFS, LCFS, SJF, WFP3, UNICEP, F1, F2, RLScheduler, and DRAS-PG) and allocators (Topology-Aware, Best-Fit, and First-Available). The Topology-Aware allocator generally performs well, with the LCFS selector achieving the lowest average waiting time of 1846.93 seconds. The Best-Fit allocator shows exceptional performance for the WFP3 selector, recording the lowest maximum waiting time of 44271 seconds in compared methods. Conversely, the First-Available allocator exhibits higher variability in maximum waiting times, particularly for the LCFS and UNICEP selectors. Notably, our method, HeraSched, demonstrates superior efficiency with the lowest average and maximum waiting times (1702.41 and 33381 seconds, respectively), highlighting its effectiveness in reducing job waiting times and improving overall scheduling efficiency in the Physical Partition.

Table B.2 details the performance of job scheduling in the Deeplearn Partition. The Best-Fit allocator excels with the FCFS selector, attaining the lowest average (1361.02 seconds) and maximum (55885 seconds) waiting times in compared methods. While the First-Available allocator remains competitive, it shows higher maximum waiting times, especially for the LCFS and WFP3 selectors. HeraSched stands out with the lowest average and maximum waiting times (1343.30 and 54715 seconds, respectively), indicating

its proficiency in minimizing job waiting times and enhancing scheduling efficiency in the Deeplearn Partition, which is crucial for deep learning tasks.

Table B.1: Performance Comparison for Jobs in Physical Partition (seconds)

Workload	Topology-Aware		Best-Fit		First-Available	
Scheduler	Ave.	Max	Ave.	Max	Ave.	Max
FCFS	2143.12	106943	2098.10	81105	2590.34	147244
LCFS	**1846.93**	64279	1851.18	63993	1992.37	492802
SJF	1916.79	155108	1987.63	64182	2091.03	344543
WFP3	2115.51	170378	2123.69	**44271**	2521.39	364646
UNICEP	2023.23	72268	1955.33	59931	2152.24	496656
F1	2008.29	71359	2052.45	66939	2447.61	299599
F2	2009.63	72293	2038.79	70434	2212.52	200674
RLScheduler	2133.39	108000	2070.87	64279	2232.09	455932
DRAS-PG	1981.84	90337	2059.37	95557	2600.01	145490

	Ave. waiting	Max waiting
HeraSched	**1702.41***	**33381***

Table B.2: Performance Comparison for Jobs in Deeplearn Partition (seconds)

Workload	Topology-Aware		Best-Fit		First-Available	
Scheduler	Ave.	Max	Ave.	Max	Ave.	Max
FCFS	1386.44	60777	**1361.02**	**55885**	1442.71	63554
LCFS	1504.92	75285	1457.97	67055	1597.85	81874
SJF	1397.81	62100	1429.81	63612	1509.45	70390
WFP3	1429.3	65889	1461.31	67040	1516.97	70588
UNICEP	1349.89	56702	1367.84	56404	1443.13	63780
F1	1406.41	64002	1408.23	61410	1448.87	65521
F2	1403.69	63231	1398.25	61171	1444.63	65050

	Ave. waiting	Max waiting
HeraSched	**1343.30***	**54715***

B.2 HeraSched High Load Validation Results

Table B.3 compares the average and maximum waiting times for compared job schedulers in the Physical High Load Validation. The SJF selector with the Topology-Aware allocator achieves the lowest average waiting time of 1010.89 seconds, while the F1 selector with the Best-Fit allocator records the lowest maximum waiting time of 56814 seconds. The First-Available allocator shows higher variability in waiting times, particularly for LCFS and UNICEP. Although HeraSched's average waiting time of 1143.87 seconds is slightly higher than the best-performing SJF selector, it significantly improves the maximum waiting

time with 31863 seconds. This overall better performance highlights HeraSched's ability to prevent job starvation and enhance scheduling efficiency in the Physical Partition.

Table B.4 shows that Topology-Aware allocator combined with the UNICEP selector shows the best average waiting time of 15894.60 seconds, while the FCFS selector with the Topology-Aware allocator achieves the lowest maximum waiting time of 38269 seconds. Despite competitive performances, the First-Available allocator generally exhibits higher maximum waiting times, especially for LCFS and SJF selectors. HeraSched significantly outperforms other schedulers with the lowest average and maximum waiting times (14861.04 and 35865 seconds, respectively), indicating its effectiveness in minimizing job waiting times and improving scheduling efficiency in the Deeplearn Partition.

TABLE B.3: Performance Comparison for Validation Set in Physical Partition (seconds)

Workload	Topology-Aware		Best-Fit		First-Available	
Scheduler	Ave.	Max	Ave.	Max	Ave.	Max
FCFS	1203.06	81690	1237.28	64032	1236.82	70481
LCFS	1400.28	70304	1401.97	89490	1402.69	97756
SJF	**1010.89***	87682	1010.97	73824	1085.76	89622
WFP3	1036.23	69130	1049.51	78864	1029.65	86693
UNICEP	1048.75	68221	1047.59	78648	1047.04	84860
F1	1237.17	68221	1251.87	**56814**	1249.55	73374
F2	1243.69	66572	1233.36	66573	1290.90	78677
RLScheduler	1216.79	81722	1213.79	70683	1341.82	85165
DRAS-PG	1243.70	61807	1244.03	73931	1243.84	73692

	Ave. waiting	Max waiting
HeraSched	**1143.87**	**31863***

TABLE B.4: Performance Comparison for Validation Set in Deeplearn Partition (seconds)

Workload	Topology-Aware		Best-Fit		First-Available	
Scheduler	Ave.	Max	Ave.	Max	Ave.	Max
FCFS	15948.57	**38269**	15948.57	38269	15950.20	38269
LCFS	17821.91	43704	17821.91	43704	17820.66	43694
SJF	16715.91	42006	16715.91	42006	16935.81	42199
WFP3	16543.61	42243	16543.61	42243	16561.14	41869
UNICEP	**15894.60**	38318	15894.60	38318	15894.80	38318
F1	15951.14	38269	15951.14	38269	15950.16	38269
F2	15915.57	38272	15915.57	38272	15915.58	38272

	Ave. waiting	Max waiting
HeraSched	**14861.04***	**35865***

Bibliography

[1] Spartan. Spartan documentation. Available at: https://dashboard.hpc.unimelb.edu.au/ (Accessed: April 25, 2024).

[2] Daniel Reed, Dennis Gannon, and Jack Dongarra. Reinventing high performance computing: challenges and opportunities. *arXiv preprint arXiv:2203.02544*, 2022.

[3] V Rajaraman. Frontier—world's first exaflops supercomputer. *Resonance*, 28(4): 567–576, 2023.

[4] Dror G Feitelson, Larry Rudolph, Uwe Schwiegelshohn, Kenneth C Sevcik, and Parkson Wong. Theory and practice in parallel job scheduling. In *Job Scheduling Strategies for Parallel Processing: IPPS'97 Processing Workshop Geneva, Switzerland, April 5, 1997 Proceedings 3*, pages 1–34. Springer, 1997.

[5] Wei Tang, Zhiling Lan, Narayan Desai, and Daniel Buettner. Fault-aware, utility-based job scheduling on blue, gene/p systems. In *2009 IEEE International Conference on Cluster Computing and Workshops*, New Orleans, LA, USA, 2009. IEEE. doi: 10.1109/clustr.2009.5289206.

[6] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. Dominant resource fairness: Fair allocation of multiple resource types. In *Nsdi*, volume 11, pages 24–24, 2011.

[7] Thomas Bridi, Andrea Bartolini, Michele Lombardi, Michela Milano, and Luca Benini. A constraint programming scheduler for heterogeneous high-performance computing machines. *IEEE transactions on parallel and distributed systems*, 27 (10):2781–2794, 2016.

[8] Gonzalo P. Rodrigo, Erik Elmroth, Per-Olov Östberg, and Lavanya Ramakrishnan. Enabling workflow-aware scheduling on hpc systems. In *Proceedings of the 26th*

International Symposium on High-Performance Parallel and Distributed Computing, HPDC '17, page 3–14, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450346993. doi: 10.1145/3078597.3078604.

[9] Oliver Kramer and Oliver Kramer. *Genetic algorithms*. Springer, 2017.

[10] Dimitris Bertsimas and John Tsitsiklis. Simulated annealing. *Statistical science*, 8 (1):10–15, 1993.

[11] Marco Dorigo, Mauro Birattari, and Thomas Stutzle. Ant colony optimization. *IEEE computational intelligence magazine*, 1(4):28–39, 2006.

[12] Mihaela-Andreea Vasile, Florin Pop, Radu-Ioan Tutueanu, Valentin Cristea, and Joanna Kołodziej. Resource-aware hybrid scheduling algorithm in heterogeneous distributed computing. *Future Generation Computer Systems*, 51:61–71, 2015.

[13] P. M. Rekha and M. Dakshayini. Efficient task allocation approach using genetic algorithm for cloud environment. *Cluster Computing*, 22(4):1241–1251, January 2019. ISSN 1573-7543. doi: 10.1007/s10586-019-02909-1.

[14] Gang Zhao. Cost-aware scheduling algorithm based on pso in cloud computing environment. *International Journal of Grid and Distributed Computing*, 7(1):33–42, February 2014. ISSN 2005-4262. doi: 10.14257/ijgdc.2014.7.1.04.

[15] Gang Li and Zhijun Wu. Ant colony optimization task scheduling algorithm for swim based on load balancing. *Future Internet*, 11(4):90, 2019.

[16] Danilo Carastan-Santos and Raphael Y. de Camargo. Obtaining dynamic scheduling policies with simulation and machine learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '17, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450351140. doi: 10.1145/3126908.3126955.

[17] Mohammed Tanash, Brandon Dunn, Daniel Andresen, William Hsu, Huichen Yang, and Adedolapo Okanlawon. Improving hpc system performance by predicting job resources via supervised machine learning. In *Proceedings of the Practice and Experience in Advanced Research Computing on Rise of the Machines (learning)*, PEARC '19, New York, NY, USA, July 2019. ACM. doi: 10.1145/3332186.3333041.

[18] Daniel Nemirovsky, Tugberk Arkose, Nikola Markovic, Mario Nemirovsky, Osman Unsal, and Adrian Cristal. A machine learning approach for performance prediction and scheduling on heterogeneous cpus. In *2017 29th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, Campinas, Brazil, October 2017. IEEE. doi: 10.1109/sbac-pad.2017.23.

[19] Eric Gaussier, David Glesser, Valentin Reis, and Denis Trystram. Improving backfilling by using machine learning to predict running times. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–10, 2015.

[20] Hongzi Mao, Mohammad Alizadeh, Ishai Menache, and Srikanth Kandula. Resource management with deep reinforcement learning. In *Proceedings of the 15th ACM workshop on hot topics in networks*, pages 50–56, New York, NY, USA, 2016. Association for Computing Machinery. doi: 10.1145/3005745.3005750.

[21] Heyang Qin, Syed Zawad, Yanqi Zhou, Lei Yang, Dongfang Zhao, and Feng Yan. Swift machine learning model serving scheduling: a region based reinforcement learning approach. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–23, 2019.

[22] Giacomo Domeniconi, Eun Kyung Lee, Vanamala Venkataswamy, and Swaroopa Dola. Cush: Cognitive scheduler for heterogeneous high performance computing system. In *Workshop on Deep Reinforcement Learning for Knowledge Discover, DRL4KDD*, 2019.

[23] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. Learning scheduling algorithms for data processing clusters. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 270–288. Association for Computing Machinery, New York, NY, USA, 2019. ISBN 9781450359566. doi: 10.1145/3341302.3342080.

[24] Di Zhang, Dong Dai, Youbiao He, Forrest Sheng Bao, and Bing Xie. Rlscheduler: An automated hpc batch job scheduler using reinforcement learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '20, Atlanta, Georgia, 2020. IEEE Press. ISBN 9781728199986.

[25] Qiqi Wang, Hongjie Zhang, Cheng Qu, Yu Shen, Xiaohui Liu, and Jing Li. Rlschert: An hpc job scheduler using deep reinforcement learning and remaining time prediction. *Applied Sciences*, 11(20):9448, October 2021. ISSN 2076-3417. doi: 10.3390/app11209448.

[26] Yuping Fan, Zhiling Lan, Taylor Childers, Paul Rich, William Allcock, and Michael E. Papka. Deep reinforcement agent for scheduling in hpc. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Portland, OR, USA, May 2021. IEEE. doi: 10.1109/ipdps49936.2021.00090.

[27] Di Zhang, Dong Dai, and Bing Xie. Schedinspector: A batch job scheduling inspector using reinforcement learning. In *Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '22, page 97–109, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450391993. doi: 10.1145/3502181.3531470.

[28] Boyang Li, Yuping Fan, Matthew Dearing, Zhiling Lan, Paul Rich, William Allcock, and Michael Papka. Mrsch: Multi-resource scheduling for hpc. In *2022 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 47–57, Heidelberg, Germany, 2022. doi: 10.1109/CLUSTER51413.2022.00020.

[29] Elliot Kolker-Hicks, Di Zhang, and Dong Dai. A reinforcement learning based backfilling strategy for hpc batch jobs. In *Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*, SC-W '23, page 1316–1323, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400707858. doi: 10.1145/3624062.3624201.

[30] Dror G. Feitelson, Dan Tsafrir, and David Krakov. Experience with using the parallel workloads archive. *Journal of Parallel and Distributed Computing*, 74(10): 2967–2982, October 2014. ISSN 0743-7315. doi: 10.1016/j.jpdc.2014.06.013.

[31] Muhammad Tirmazi, Adam Barker, Nan Deng, Md E Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. Borg: the next generation. In *Proceedings of the fifteenth European conference on computer systems*, pages 1–14, 2020.

[32] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at google with borg. In *Proceedings of the tenth european conference on computer systems*, pages 1–17, 2015.

[33] Robin Boëzennec, Fanny Dufossé, and Guillaume Pallez. Optimization metrics for the evaluation of batch schedulers in hpc. In *Job Scheduling Strategies for Parallel Processing: 26th Workshop, JSSPP 2023, St. Petersburg, FL, USA, May 19, 2023, Revised Selected Papers*, page 97–115, Berlin, Heidelberg, 2023. Springer-Verlag. ISBN 978-3-031-43942-1. doi: 10.1007/978-3-031-43943-8_5.

[34] Rajendra K Jain, Dah-Ming W Chiu, William R Hawe, et al. A quantitative measure of fairness and discrimination. *Eastern Research Laboratory, Digital Equipment Corporation, Hudson, MA*, 21:1, 1984.

[35] Andy B. Yoo, Morris A. Jette, and Mark Grondona. *SLURM: Simple Linux Utility for Resource Management*, page 44–60. Springer Berlin Heidelberg, Berlin, Heidelberg, Germany, 2003. ISBN 9783540397274. doi: 10.1007/10968987_3.

[36] David Jackson, Quinn Snell, and Mark Clement. Core algorithms of the maui scheduler. In *Job Scheduling Strategies for Parallel Processing: 7th International Workshop, JSSPP 2001 Cambridge, MA, USA, June 16, 2001 Revised Papers 7*, pages 87–102. Springer, 2001.

[37] Christopher M Bishop and Nasser M Nasrabadi. *Pattern recognition and machine learning*, volume 4. Springer, 2006.

[38] Richard S. Sutton. *Introduction: The Challenge of Reinforcement Learning*, pages 1–3. Springer US, Boston, MA, 1992. ISBN 9781461536185. doi: 10.1007/978-1-4615-3618-5_1.

[39] TP Lillicrap. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

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

[41] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *International conference on machine learning*, pages 1861–1870. PMLR, 2018.

[42] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8:279–292, 1992.

[43] Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8:229–256, 1992.

[44] Richard S Sutton, David McAllester, Satinder Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. *Advances in neural information processing systems*, 12, 1999.

[45] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

[46] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *Proceedings of the AAAI conference on artificial intelligence*, volume 30, 2016.

[47] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado Hasselt, Marc Lanctot, and Nando Freitas. Dueling network architectures for deep reinforcement learning. In *International conference on machine learning*, pages 1995–2003. PMLR, 2016.

[48] Matteo Hessel, Joseph Modayil, Hado Van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Azar, and David Silver. Rainbow: Combining improvements in deep reinforcement learning. In *Proceedings of the AAAI conference on artificial intelligence*, volume 32, 2018.

[49] Oron Anschel, Nir Baram, and Nahum Shimkin. Averaged-dqn: Variance reduction and stabilization for deep reinforcement learning. In *International conference on machine learning*, pages 176–185. PMLR, 2017.

[50] Adrià Puigdomènech Badia, Pablo Sprechmann, Alex Vitvitskyi, Daniel Guo, Bilal Piot, Steven Kapturowski, Olivier Tieleman, Martín Arjovsky, Alexander Pritzel, Andew Bolt, et al. Never give up: Learning directed exploration strategies. *arXiv preprint arXiv:2002.06038*, 2020.

[51] Adrià Puigdomènech Badia, Bilal Piot, Steven Kapturowski, Pablo Sprechmann, Alex Vitvitskyi, Zhaohan Daniel Guo, and Charles Blundell. Agent57: Outperforming the atari human benchmark. In *International conference on machine learning*, pages 507–517. PMLR, 2020.

[52] Steven Kapturowski, Víctor Campos, Ray Jiang, Nemanja Rakićević, Hado van Hasselt, Charles Blundell, and Adrià Puigdomènech Badia. Human-level atari 200x faster. *arXiv preprint arXiv:2209.07550*, 2022.

[53] Volodymyr Mnih. Asynchronous methods for deep reinforcement learning. *arXiv preprint arXiv:1602.01783*, 2016.

[54] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation. *arXiv preprint arXiv:1506.02438*, 2015.

[55] John Schulman. Trust region policy optimization. *arXiv preprint arXiv:1502.05477*, 2015.

[56] Scott Fujimoto, Herke Hoof, and David Meger. Addressing function approximation error in actor-critic methods. In *International conference on machine learning*, pages 1587–1596. PMLR, 2018.

[57] Marcin Andrychowicz, Filip Wolski, Alex Ray, Jonas Schneider, Rachel Fong, Peter Welinder, Bob McGrew, Josh Tobin, OpenAI Pieter Abbeel, and Wojciech Zaremba. Hindsight experience replay. *Advances in neural information processing systems*, 30, 2017.

[58] Shixiang Gu, Timothy Lillicrap, Zoubin Ghahramani, Richard E Turner, and Sergey Levine. Q-prop: Sample-efficient policy gradient with an off-policy critic. *arXiv preprint arXiv:1611.02247*, 2016.

[59] Shixiang Gu, Timothy Lillicrap, Ilya Sutskever, and Sergey Levine. Continuous deep q-learning with model-based acceleration. In *International conference on machine learning*, pages 2829–2838. PMLR, 2016.

[60] Nicolas Heess, Gregory Wayne, David Silver, Timothy Lillicrap, Tom Erez, and Yuval Tassa. Learning continuous control policies by stochastic value gradients. *Advances in neural information processing systems*, 28, 2015.

[61] Carlos Diuk and Michael Littman. Hierarchical reinforcement learning. In *Encyclopedia of artificial intelligence*, pages 825–830. IGI Global, 2009.

[62] Peter Dayan and Geoffrey E Hinton. Feudal reinforcement learning. *Advances in neural information processing systems*, 5, 1992.

[63] Richard S Sutton, Doina Precup, and Satinder Singh. Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artificial intelligence*, 112(1-2):181–211, 1999.

[64] Ronald Parr and Stuart Russell. Reinforcement learning with hierarchies of machines. *Advances in neural information processing systems*, 10, 1997.

[65] Thomas G Dietterich. Hierarchical reinforcement learning with the maxq value function decomposition. *Journal of artificial intelligence research*, 13:227–303, 2000.

[66] Alexander Sasha Vezhnevets, Simon Osindero, Tom Schaul, Nicolas Heess, Max Jaderberg, David Silver, and Koray Kavukcuoglu. Feudal networks for hierarchical reinforcement learning. In *International conference on machine learning*, pages 3540–3549. PMLR, 2017.

[67] Ofir Nachum, Shixiang Shane Gu, Honglak Lee, and Sergey Levine. Data-efficient hierarchical reinforcement learning. *Advances in neural information processing systems*, 31, 2018.

[68] Andrew Levy, George Konidaris, Robert Platt, and Kate Saenko. Learning multi-level hierarchies with hindsight. *arXiv preprint arXiv:1712.00948*, 2017.

[69] Tejas D Kulkarni, Karthik Narasimhan, Ardavan Saeedi, and Josh Tenenbaum. Hierarchical deep reinforcement learning: Integrating temporal abstraction and intrinsic motivation. *Advances in neural information processing systems*, 29, 2016.

[70] Kevin Frans, Jonathan Ho, Xi Chen, Pieter Abbeel, and John Schulman. Meta learning shared hierarchies. *arXiv preprint arXiv:1710.09767*, 2017.

[71] Alexander Vezhnevets, Volodymyr Mnih, Simon Osindero, Alex Graves, Oriol Vinyals, John Agapiou, et al. Strategic attentive writer for learning macro-actions. *Advances in neural information processing systems*, 29, 2016.

[72] Chen Tessler, Shahar Givony, Tom Zahavy, Daniel Mankowitz, and Shie Mannor. A deep hierarchical approach to lifelong learning in minecraft. In *Proceedings of the AAAI conference on artificial intelligence*, volume 31, 2017.

[73] Nakul Gopalan, Michael Littman, James MacGlashan, Shawn Squire, Stefanie Tellex, John Winder, Lawson Wong, et al. Planning with abstract markov decision processes. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 27, pages 480–488, 2017.

[74] Daniel J Mankowitz, Timothy A Mann, and Shie Mannor. Iterative hierarchical optimization for misspecified problems (ihomp). *arXiv preprint arXiv:1602.03348*, 2016.

[75] Sainbayar Sukhbaatar, Emily Denton, Arthur Szlam, and Rob Fergus. Learning goal embeddings via self-play for hierarchical reinforcement learning. *arXiv preprint arXiv:1811.09083*, 2018.

[76] Jacob Rafati and David C Noelle. Learning representations in model-free hierarchical reinforcement learning. In *Proceedings of the AAAI conference on artificial intelligence*, volume 33, pages 10009–10010, 2019.

[77] Matthew E Taylor and Peter Stone. Transfer learning for reinforcement learning domains: A survey. *Journal of Machine Learning Research*, 10(7), 2009.

[78] Sinno Jialin Pan and Qiang Yang. A survey on transfer learning. *IEEE Transactions on knowledge and data engineering*, 22(10):1345–1359, 2009.

[79] Zhuangdi Zhu, Kaixiang Lin, Anil K Jain, and Jiayu Zhou. Transfer learning in deep reinforcement learning: A survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2023.

[80] Tom Schaul. Prioritized experience replay. *arXiv preprint arXiv:1511.05952*, 2015.

[81] Dan Horgan, John Quan, David Budden, Gabriel Barth-Maron, Matteo Hessel, Hado Van Hasselt, and David Silver. Distributed prioritized experience replay. *arXiv preprint arXiv:1803.00933*, 2018.

[82] André Barreto, Will Dabney, Rémi Munos, Jonathan J Hunt, Tom Schaul, Hado P van Hasselt, and David Silver. Successor features for transfer in reinforcement learning. *Advances in neural information processing systems*, 30, 2017.

[83] Emilio Parisotto, Jimmy Lei Ba, and Ruslan Salakhutdinov. Actor-mimic: Deep multitask and transfer reinforcement learning. *arXiv preprint arXiv:1511.06342*, 2015.

[84] Shikun Liu, Andrew Davison, and Edward Johns. Self-supervised generalisation with meta auxiliary learning. *Advances in Neural Information Processing Systems*, 32, 2019.

[85] Rich Caruana. Multitask learning. *Machine learning*, 28:41–75, 1997.

[86] Andrew Y Ng, Daishi Harada, and Stuart Russell. Policy invariance under reward transformations: Theory and application to reward shaping. In *Icml*, volume 99, pages 278–287, 1999.

[87] Deepak Pathak, Pulkit Agrawal, Alexei A Efros, and Trevor Darrell. Curiosity-driven exploration by self-supervised prediction. In *International conference on machine learning*, pages 2778–2787. PMLR, 2017.

[88] Yuri Burda, Harrison Edwards, Amos Storkey, and Oleg Klimov. Exploration by random network distillation. *arXiv preprint arXiv:1810.12894*, 2018.

[89] Haiyan Yin and Sinno Pan. Knowledge transfer for deep reinforcement learning with hierarchical experience replay. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 31, 2017.

[90] Andrei A Rusu, Sergio Gomez Colmenarejo, Caglar Gulcehre, Guillaume Desjardins, James Kirkpatrick, Razvan Pascanu, Volodymyr Mnih, Koray Kavukcuoglu, and Raia Hadsell. Policy distillation. *arXiv preprint arXiv:1511.06295*, 2015.

[91] Chrisantha Fernando, Dylan Banarse, Charles Blundell, Yori Zwols, David Ha, Andrei A Rusu, Alexander Pritzel, and Daan Wierstra. Pathnet: Evolution channels gradient descent in super neural networks. *arXiv preprint arXiv:1701.08734*, 2017.

[92] Ahmed Hussein, Mohamed Medhat Gaber, Eyad Elyan, and Chrisina Jayne. Imitation learning: A survey of learning methods. *ACM Computing Surveys (CSUR)*, 50(2):1–35, 2017.

[93] Eric Tzeng, Coline Devin, Judy Hoffman, Chelsea Finn, Xingchao Peng, Sergey Levine, Kate Saenko, and Trevor Darrell. Towards adapting deep visuomotor representations from simulated to real environments. *arXiv preprint arXiv:1511.07111*, 2(3), 2015.

[94] Josh Tobin, Rachel Fong, Alex Ray, Jonas Schneider, Wojciech Zaremba, and Pieter Abbeel. Domain randomization for transferring deep neural networks from simulation to the real world. In *2017 IEEE/RSJ international conference on intelligent robots and systems (IROS)*, pages 23–30. IEEE, 2017.

[95] Richard Bellman. Mathematical aspects of scheduling theory. *Journal of the Society for Industrial and Applied Mathematics*, 4(3):168–205, 1956.

[96] Edi Shmueli and Dror G. Feitelson. *Backfilling with Lookahead to Optimize the Performance of Parallel Job Scheduling*, page 228–251. Springer Berlin Heidelberg, Springer, Berlin, Heidelberg, 2003. ISBN 9783540397274. doi: 10.1007/10968987_12.

[97] Mohak Chadha, Jophin John, and Michael Gerndt. Extending slurm for dynamic resource-aware adaptive batch scheduling. In *2020 IEEE 27th International Conference on High Performance Computing, Data, and Analytics (HiPC)*, pages 223–232. IEEE, 2020.

[98] Jesus Carretero, Emmanuel Jeannot, Guillaume Pallez, David E Singh, and Nicolas Vidal. Mapping and scheduling hpc applications for optimizing i/o. In *Proceedings of the 34th ACM International Conference on Supercomputing*, pages 1–12, 2020.

[99] Chansup Byun, William Arcand, David Bestor, Bill Bergeron, Vijay Gadepally, Michael Houle, Matthew Hubbell, Michael Jones, Anna Klein, Peter Michaleas, et al. Node-based job scheduling for large scale simulations of short running jobs. In *2021 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7. IEEE, 2021.

[100] Daniel Nichols, Aniruddha Marathe, Kathleen Shoga, Todd Gamblin, and Abhinav Bhatele. Resource utilization aware job scheduling to mitigate performance variability. In *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 335–345. IEEE, 2022.

[101] Garrick Staples. Torque resource manager. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, pages 8–es, 2006.

[102] Inc. Adaptive Computing Enterprises. *Moab Workload Manager Documentation, Version 8.0*, 2024. URL https://docs.adaptivecomputing.com/suite/8-0/basic/help.htm#topics/moabWorkloadManager/topics/intro/productOverview.htm. Accessed: 2025-03-29.

[103] Mehrnoush Alemzadeh. *A toolset to aid MOAB scheduler configuration*. PhD thesis, UNIVERSITY OF CALGARY, 2009.

[104] Jargalsaikhan Narantuya, Jun-Sik Shin, Sun Park, and JongWon Kim. Multi-agent deep reinforcement learning-based resource allocation in hpc/ai converged cluster. *Computers, Materials & Continua*, 72(3), 2022. ISSN 1546-2226. doi: 10.32604/cmc.2022.023318.

[105] Jingbo Li, Xingjun Zhang, Jia Wei, Zeyu Ji, and Zheng Wei. Garlsched: Generative adversarial deep reinforcement learning task scheduling optimization for large-scale high performance computing systems. *Future Generation Computer Systems*, 135: 259–269, October 2022. ISSN 0167-739X. doi: 10.1016/j.future.2022.04.032.

[106] Renato Luiz de Freitas Cunha and Luiz Chaimowicz. An smdp approach for reinforcement learning in hpc cluster schedulers. *Future Generation Computer Systems*, 139:239–252, February 2023. ISSN 0167-739X. doi: 10.1016/j.future.2022.09.025.

[107] Tarun Agarwal, Amit Sharma, A Laxmikant, and Laxmikant V Kalé. Topology-aware task mapping for reducing communication contention on large parallel machines. In *Proceedings 20th IEEE International Parallel & Distributed Processing Symposium*, pages 10–pp, Rhodes, Greece, 2006. IEEE.

[108] Yiannis Georgiou, Emmanuel Jeannot, Guillaume Mercier, and Adèle Villiermet. Topology-aware job mapping. *The International Journal of High Performance Computing Applications*, 32(1):14–27, September 2017. ISSN 1741-2846. doi: 10.1177/1094342017727061.

[109] Marcelo Amaral, Jorda Polo, David Carrera, Seetharami Seelam, and Malgorzata Steinder. Topology-aware gpu scheduling for learning workloads in cloud environments. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2017.

[110] Priya Mishra, Tushar Agrawal, and Preeti Malakar. Communication-aware job scheduling using slurm. In *Workshop Proceedings of the 49th International Conference on Parallel Processing*, ICPP Workshops '20, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450388689. doi: 10.1145/3409390.3409410.

[111] Zekang Lan, Yan Xu, Yingkun Huang, Dian Huang, and Shengzhong Feng. Optimization of topology-aware job allocation on a high-performance computing cluster by neural simulated annealing. *arXiv preprint arXiv:2302.03517*, 2023.

[112] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. Starpu: a unified platform for task scheduling on heterogeneous multicore architectures. In *Euro-Par 2009 Parallel Processing: 15th International Euro-Par Conference, Delft, The Netherlands, August 25-28, 2009. Proceedings 15*, pages 863–874. Springer, 2009.

[113] Jian-feng LI and Jian PENG. Task scheduling algorithm based on improved genetic algorithm in cloud computing environment: Task scheduling algorithm based on improved genetic algorithm in cloud computing environment. *Journal of Computer Applications*, 31(1):184–186, March 2011. ISSN 1001-9081. doi: 10.3724/sp.j.1087.2011.00184.

[114] Abhishek Gupta, Laxmikant V Kale, Dejan Milojicic, Paolo Faraboschi, and Susanne M Balle. Hpc-aware vm placement in infrastructure clouds. In *2013 IEEE International Conference on Cloud Engineering (IC2E)*, pages 11–20. IEEE, 2013.

[115] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. Multi-resource packing for cluster schedulers. *ACM SIGCOMM Computer Communication Review*, 44(4):455–466, 2014.

[116] Dinesh Kumar, Zon-yin Shae, and Hani Jamjoom. Scheduling batch and heterogeneous jobs with runtime elasticity in a parallel processing environment. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, pages 65–78. IEEE, 2012.

[117] Aira Villapando and Jessi Christa Rubio. Simulation vs actual walltime correction in a real production resource-constrained hpc. In *Practice and Experience in Advanced Research Computing 2021: Evolution Across All Dimensions*, pages 1–7. 2021.

[118] H. Viswanathan, E.K. Lee, I. Rodero, D. Pompili, M. Parashar, and M. Gamell. Energy-aware application-centric vm allocation for hpc workloads. In *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, Anchorage, AK, USA, May 2011. IEEE. doi: 10.1109/ipdps.2011.234.

[119] Nguyen Quang-Hung, Duy-Khanh Le, Nam Thoai, and Nguyen Thanh Son. *Heuristics for Energy-Aware VM Allocation in HPC Clouds*, page 248–261. Springer International Publishing, Cham, 2014. ISBN 9783319127781. doi: 10.1007/978-3-319-12778-1_19.

[120] Piotr Arabas and Ewa Niewiadomska-Szynkiewicz. Energy-efficient workload allocation in distributed hpc system. In *2019 International Conference on High Performance Computing & Simulation (HPCS)*, pages 747–753, Dublin, Ireland, 2019. IEEE. doi: 10.1109/hpcs48598.2019.9188240.

[121] Piotr Arabas. Modeling and simulation of hierarchical task allocation system for energy-aware hpc clouds. *Simulation Modelling Practice and Theory*, 107:102221, February 2021. ISSN 1569-190X. doi: 10.1016/j.simpat.2020.102221.

[122] Alexandru Iulian Orhean, Florin Pop, and Ioan Raicu. New scheduling approach using reinforcement learning for heterogeneous distributed systems. *Journal of Parallel and Distributed Computing*, 117:292–302, 2018.

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

[124] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

[125] Marcin Andrychowicz, Anton Raichuk, Piotr Stańczyk, Manu Orsini, Sertan Girgin, Raphaël Marinier, Leonard Hussenot, Matthieu Geist, Olivier Pietquin, Marcin Michalski, et al. What matters for on-policy deep actor-critic methods? a large-scale study. In *International conference on learning representations*, 2020.

[126] Hameed Hussain, Saif Ur Rehman Malik, Abdul Hameed, Samee Ullah Khan, Gage Bickler, Nasro Min-Allah, Muhammad Bilal Qureshi, Limin Zhang, Wang Yongji,

Nasir Ghani, Joanna Kolodziej, Albert Y. Zomaya, Cheng-Zhong Xu, Pavan Balaji, Abhinav Vishnu, Fredric Pinel, Johnatan E. Pecero, Dzmitry Kliazovich, Pascal Bouvry, Hongxiang Li, Lizhe Wang, Dan Chen, and Ammar Rayes. A survey on resource allocation in high performance distributed computing systems. *Parallel Computing*, 39(11):709–736, 2013. ISSN 0167-8191. doi: https://doi.org/10.1016/j.parco.2013.09.009. URL https://www.sciencedirect.com/science/article/pii/S016781911300121X.

[127] Fatos Xhafa and Ajith Abraham. Computational models and heuristic methods for grid scheduling problems. *Future Generation Computer Systems*, 26(4):608–621, April 2010. ISSN 0167-739X. doi: 10.1016/j.future.2009.11.005.

[128] Shubham Pateria, Budhitama Subagdja, Ah-hwee Tan, and Chai Quek. Hierarchical reinforcement learning: A comprehensive survey. *ACM Comput. Surv.*, 54(5), jun 2021. ISSN 0360-0300. doi: 10.1145/3453160.

[129] Nat Dilokthanakul, Christos Kaplanis, Nick Pawlowski, and Murray Shanahan. Feature control as intrinsic motivation for hierarchical reinforcement learning. *IEEE Transactions on Neural Networks and Learning Systems*, 30(11):3409–3418, November 2019. ISSN 2162-2388. doi: 10.1109/tnnls.2019.2891792.

[130] Shengyi Huang and Santiago Ontañón. A closer look at invalid action masking in policy gradient algorithms. *The International FLAIRS Conference Proceedings*, 35, May 2022. ISSN 2334-0762. doi: 10.32473/flairs.v35i.130584.

[131] OpenAI. Gymnasium. https://gymnasium.farama.org/, 2024.

[132] Antonin Raffin, Ashley Hill, Adam Gleave, Anssi Kanervisto, Maximilian Ernestus, and Noah Dormann. Stable-baselines3: reliable reinforcement learning implementations. *J. Mach. Learn. Res.*, 22(1), jan 2021. ISSN 1532-4435. doi: 10.5555/3454287.3455016.

[133] Larry Rudolph and Paul H Smith. Valuation of ultra-scale computing systems. In *Workshop on Job Scheduling Strategies for Parallel Processing*, pages 39–55. Springer, 2000.

[134] Eitan Frachtenberg and Dror G Feitelson. Pitfalls in parallel job scheduling evaluation. In *Job Scheduling Strategies for Parallel Processing: 11th International*

Workshop, JSSPP 2005, Cambridge, MA, USA, June 19, 2005, Revised Selected Papers 11, pages 257–282. Springer, 2005.

[135] Vitus J Leung, Gerald Sabin, and Ponnuswamy Sadayappan. Parallel job scheduling policies to improve fairness: A case study. In *2010 39th International Conference on Parallel Processing Workshops*, pages 346–353. IEEE, 2010.

[136] Abhishek Verma, Madhukar Korupolu, and John Wilkes. Evaluating job packing in warehouse-scale computing. In *2014 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 48–56. IEEE, 2014.

[137] Robin Boëzennec, Fanny Dufossé, and Guillaume Pallez. Qualitatively analyzing optimization objectives in the design of hpc resource manager. *ACM Transactions on Modeling and Performance Evaluation of Computing Systems*, 9(4):1–28, 2024.

[138] Alexander V Goponenko, Kenneth Lamar, Christina Peterson, Benjamin A Allan, Jim M Brandt, and Damian Dechev. Metrics for packing efficiency and fairness of hpc cluster batch job scheduling. In *2022 IEEE 34th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 241–252. IEEE, 2022.

[139] Huamao Xie, Ding Ding, Lihong Zhao, and Kaixuan Kang. Transfer learning based multi-objective evolutionary algorithm for dynamic workflow scheduling in the cloud. *IEEE Transactions on Cloud Computing*, 2024.

[140] Ian Foster. *Designing and building parallel programs: concepts and tools for parallel software engineering*. Addison-Wesley Longman Publishing Co., Inc., 1995.