Automated Selection of Scheduling Algorithms using Reinforcement Learning in LB4MPI

Master Thesis

Faculty of Science of the University of Basel

Department of Mathematics and Computer Science

HPC Group
hpc.dmi.unibas.ch

Advisor: Prof. Dr. Florina M. Ciorba

Supervisor: Jonas H. M. Korndörfer

Andrei Bîrgovan

andrei.birgovan@stud.unibas.ch

21-063-425

12th of June 2023

Acknowledgments

I want to express my sincere gratitude to Prof. Dr. Florina M. Ciorba for allowing me to complete my Machine Intelligence Master's thesis in the HPC group. Her constructive feedback and valuable suggestions helped me stay motivated in chasing high-quality results. Also, many thanks towards Jonas H. M. Korndörfer for being a great advisor. He was *always* there to support me, and every interaction felt beyond beneficial. Special thanks go towards each member of the HPC group for making my Master's thesis an enjoyable experience. I am beyond grateful to my beloved family and friends for their continuous support and endless love.

Abstract

To meet the computational demands of scientific applications, high-performing computing systems chase the exascale by incorporating an ever-increasing number of computing nodes. One major source of overhead that results in sub-optimal usage of robust HPC systems is the load imbalance. To combat this phenomenon, numerous dynamic loop scheduling algorithms have been proposed. Deciding upon which loop scheduling technique to use is an NP-hard problem. The manual selection for a specific application, loop, and system context is error-prone and time-costly, due to the need for extensive experimentation. An automated selection is required, where the scheduling technique is selected dynamically.

In this thesis, we extend the `LB4MPI` library with reinforcement learning features for an automated selection of the most promising technique out of a portfolio of scheduling algorithms. The library features seven agent types (five main agents and two meta agents), four action selection policies (three main policies and a customizable one), and ten reward metrics. The two meta-agents have been developed to swerve away from the scheduling technique selection problem. The `ChunkParameterSelector` would select the most promising chunk parameter size for `SS`, while the newly-proposed `StealRatioSelector`, which is compatible with a distributed-data setup, would select the fraction of work to be stolen.

The implementation is validated and benchmarked using three scientific applications: PI-SOLVER (available in both replicate-data and distributed-data versions), Mandelbrot, and SPHYNX Evrard Collapse. The RL-based automated DLS algorithm or parameter selection results are compared against the theoretical highest-achieving selection, the Oracle. The agents using `looptime`-based reward metrics are shown to achieve higher performance than the `load-imbalance` rewarded ones. No long-term advantage is noticed regardless of using `QLearn` or `SARSALearn`. In some cases, the RL agent's selection achieves only a 0.69% performance loss over the Oracle. However, manually fixing the scheduling algorithm to a random value yields a performance decline of 9.43% to 120.27%. The overhead of using the RL features is between 0.001% and 0.022% of the application's execution time. An ANOVA examination is undergone to find the RL agent's configuration item that causes the most variance, and this is the *reward-type*.

Table of Contents

Acknowledgments ii

Abstract iii

List of Figures vi

List of Tables vii

List of Abbreviations viii

1 Introduction 1

2 Background 3

2.1 Workload Scheduling 3

2.1.1 The Message Passing Interface 3

2.1.2 The LB4MPI Library 4

2.1.3 Scheduling Algorithms 4

2.1.4 Work Stealing in Distributed Data Setups 6

2.1.5 Performance Metrics 6

2.2 Reinforcement Learning 8

2.2.1 Types of Policies 9

2.2.2 Types of Reinforcement Learning Agents 9

3 Related Work 13

4 Implementation 16

4.1 The LB4MPI Library 16

4.2 The Reinforcement Learning Extension 18

4.3 Environment Variables 23

4.4 List of Changes 25

4.5 Compiling the Library 26

4.6 Usage - Summary 27

5 Experimental Results **28**
5.1 The Computing System 28
5.2 Replicated-data Experiments 28
5.2.1 PISOLVER 30
5.2.2 Mandelbrot 34
5.2.3 SPHYNX Evrard Collapse 39
5.3 Distributed-data Experiments 43
5.3.1 PISOLVER (Distributed Version) 44
5.4 Replaying the DLS Selection 47
5.5 RL Component Overhead 48
5.6 Potential Performance Gained 49
5.7 Analysis of Variance 50
5.8 Discussion 52

6 Conclusions and Future Work **54**

Bibliography **56**

Appendix A Appendix **a**

List of Figures

1.1 Motivation 2

2.1 Components of an Agent-Environment RL system. 8

4.1 Communication between LB4MPI and the RL agent when automatically determining the most promising DLS technique 17
4.2 Components of the RL Agent 19
4.3 Application's workflow when using the C-based LB4MPI with RL features 27

5.1 Results summary for PISOLVER 31
5.2 Chunk parameter selection per timestep for PISOLVER (15 %) 32
5.3 DLS selection per timestep for PISOLVER, 15 % workload imbalance 32
5.4 RL DLS selector configuration components comparison for PISOLVER 33
5.5 Results summary for Mandelbrot 35
5.6 Chunk parameter selection per timestep for Mandelbrot 36
5.7 DLS selection per timestep for Mandelbrot 37
5.8 Similarity of DLS selection compared with Oracle's for Mandelbrot 38
5.9 RL DLS selector configuration components comparison for Mandelbrot 38
5.10 Results summary for SPHYNX Evrard Collapse 40
5.11 Chunk parameter selection per timestep for SPHYNX Evrard Collapse 41
5.12 DLS selection per timestep for SPHYNX Evrard Collapse 41
5.13 Similarity of RL DLS selection versus Oracle's for SPHYNX Evrard 42
5.14 RL DLS selector configuration components comparison for SPHYNX Evrard . . 43
5.15 Results summary for the distributed version of PISOLVER 45
5.16 StealRatio selection per timestep for distributed PISOLVER 46
5.17 Parallel execution times for all replicated-data experiments grouped by the RL configuration type 51
5.18 Results of benchmarking automated selection strategies using time-stepping applications. Formula to calculate the decline in performance compared with the Oracle: x% = (T_{par} - T_{par}^{Oracle}) / T_{par}^{Oracle} $\times$ 100, where T_{par} is the parallel execution time 53

A.1 Performance summary for PISOLVER 0% workload imbalance . . . . . b
A.2 DLS selection per timestep for PISOLVER with 0% workload imbalance . . . . . b
A.3 Performance summary for PISOLVER 5% workload imbalance c
A.4 DLS selection per timestep for PISOLVER with 5% workload imbalance c

A.5 Performance summary for PISOLVER 10% workload imbalance . . . . d
A.6 DLS selection per timestep for PISOLVER with 10% workload imbalance . . . . d
A.7 Performance summary for PISOLVER 15% workload imbalance . . . . e
A.8 DLS selection per timestep for PISOLVER with 15% workload imbalance . . . . e
A.9 Performance summary for PISOLVER 20% workload imbalance . . . . f
A.10 DLS selection per timestep for PISOLVER with 20% workload imbalance . . . . f
A.11 Performance summary for PISOLVER 25% workload imbalance . . . . g
A.12 DLS selection per timestep for PISOLVER with 25% workload imbalance . . . . g
A.13 Performance summary for PISOLVER 30% workload imbalance . . . . h
A.14 DLS selection per timestep for PISOLVER with 30% workload imbalance . . . . h
A.15 RL chunk size selection for PISOLVER i

List of Tables

2.1 Dynamic Load Scheduling techniques included in `LB4MPI` [1]. 5

3.1 Main characteristics of closely-related works. 15

4.1 Supported environment variables for `LB4MPI` / `LB4OMP` with RL features 24

5.1 Design of 1'835 factorial experiments for performance evaluation of LB4MPI with RL features in replicated-data applications 29
5.2 Design of 210 factorial experiments for evaluating the StealRatio selection 33 using RL features in a distributed-data setup 44
5.3 Differences in total application execution time when the sequence of DLS techniques achieved through automated or manual selection is replayed six times. For PISOLVER, the percentage in parenthesis refers to the workload imbalance. 48
5.4 Measuring the overhead introduced through using the RL component 49
5.5 Potential performance gained through using the RL-based selection feature. For PISOLVER, the percentage in parenthesis refers to the workload imbalance. . . . 50
5.6 One-factor ANOVA F-statistic 51
5.7 Tukey's Honestly Significantly Differenced analysis 52

List of Abbreviations

α	The learning rate
ϵ	Probability of exploiting the knowledge
γ	The discount rate
τ	Temperature parameter for `softmax`
L_0	The loop with ID 0
T_{par}^{loop}	Parallel loop execution time
T_{par}	Parallel execution time (per whole application execution)
ANOVA	ANalysis Of VAriance
c.o.v.	Coefficient of Variance
DLS	Dynamic Loop Scheduling
env var	Environment Variable
eps-greedy	The `epsilon-greedy` policy type
expl-1st	The `explore-first` policy type
HPC	High Performing Computing
LIB	Load Imbalance
LT	The `looptime` reward type
LT-avg	The `looptime-average` reward type
MPI	Message Passing Interface
PE	Processing Element
RL	Reinforcement Learning
RWS	Random Work Stealing
SRS	The `StealRatioSelector` meta agent

1
Introduction

The steady growth of computational demands in scientific applications indicates a need for more powerful high-performance computing (HPC) systems, chasing the exascale. The boost in performance is generally achieved through increasing the number of computing nodes. To exploit the full potential of a robust HPC system with multiple levels of hardware parallelism, parallelism in software needs to be exposed and expressed. Alongside the need for synchronisation, management of parallelism, and communication costs, *load imbalance* has been identified as a major overhead source leading to a decline in performance [2].

Load imbalance consists of idle processors, with work ready to be done that no processor has started yet. It leads to disparate advancement among the processing units. Load imbalance emerges from numerous application, algorithm, and system characteristics [3].

To mitigate the effects of load imbalance, *scheduling* aims to balance the workload in the processor's space and time. Dynamic loop scheduling (DLS) techniques efficiently improve *load balancing*. Software libraries such as `LB4OMP` [3], `LaPeSD libGOMP` [2], and `LB4MPI` [1] provide unified implementations of DLS algorithms, at thread or process levels. As there is no one-fits-all solution [2], the problem to be solved is an algorithm selection one. Manual selection of the DLS algorithm is proven to be inefficient, time-consuming, costly, and human error-prone; hence, an automated DLS selection is required.

Recent works by Dhandayuthapani [4], Banicescu et al. [5], Sukhija et al. [6], Korndörfer et al. [3], and Kury [7] use Machine Learning techniques for automated DLS selection during execution time. Automated selectors have been successfully developed for thread level, shared memory setups [7, 8]. Solving this problem for the more scalable distributed memory configuration at the process level is an understudied topic. Based on previous work, Expert Systems and Reinforcement Learning (RL) automated selection methods are worthwhile. However, opting for the RL solution is shown to be more flexible when the DLS portfolio needs to be changed. In addition to `QLearn` and `SARSA` RL agents being vastly encountered in literature, these are not outperformed by variations such as `DoubleQLearn`, `QVLearn`, or `ExpectedSARSA` [5, 7]. Rather, an agent's performance is shown to be influenced by other factors, such as the policy choice and reward metric used. Furthermore, Kury [7] demon-

strated the potential of an automated chunk parameter selection, the `ChunkLearner`, as an alternative to DLS selection. We will also explore this hypothesis by experimenting with this meta RL agent, renamed into the `ChunkParameterSelector`.

On another topic, when following a centralised-data approach and using workload scheduling, the scalability is compromised. To address these problems, the `LB4MPI` library also supports applications where the data is distributed instead of replicated. Within Wetten's Master's thesis [9], the `LB4MPI` library has been modified to support random work-stealing (RWS) for distributed-data contexts. We will explore an RL-based automated selection of the `StealRatio` through the `StealRatioSelector` meta RL agent.

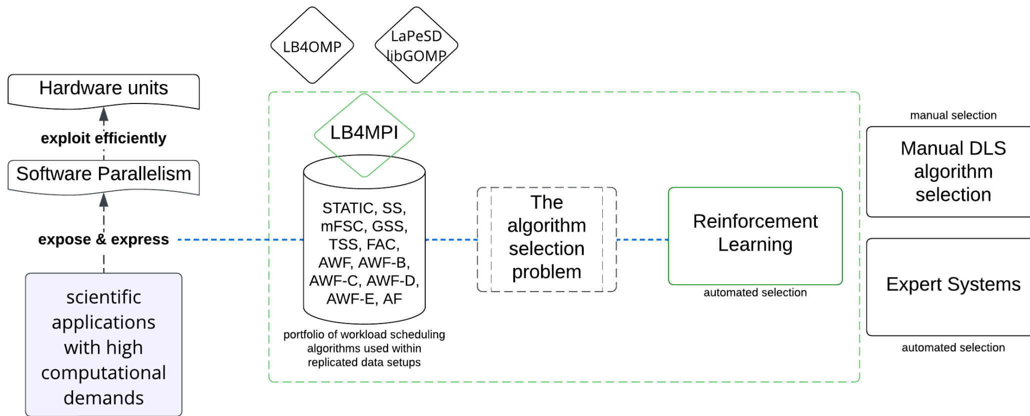


Figure 1.1: Motivation

Figure 1.1 illustrates the motivation behind this Master's thesis. The goal is to extend the `LB4MPI` library with RL capabilities to solve the automated DLS algorithm selection problem. Pursuing a similar methodology, we experiment with automatically selecting the chunk size parameter using a modified version of the Self Scheduling (`SS`) technique. Moreover, we analyse the performance of various RL agent configurations through benchmarking using three applications, PISOLVER [10], Mandelbrot [11], and SPHYNX Evrard Collapse [12], on the miniHPC cluster [13]. Followingly, the focus shifts from a replicated data setup to a distributed data one, and we use RL agents to select the most promising Steal Ratio dynamically. This selection is benchmarked using a distributed-data version of PISOLVER.

This Master's thesis is structured as follows. Chapter 2 offers the background information further needed. Chapter 3 summarises the literature findings that lead to this topic's existence. In chapter 4, the implementation choices are overviewed, while in chapter 5, a series of experiments are discussed. While chapter 6 offers the conclusions and future work prospects, the Appendix A brings completeness to the main body of the thesis. The Appendix mainly contains auxiliary figures and plots regarding some of the experiments.

2
Background

In this chapter, important notions such as workload scheduling algorithms and performance metrics are explained. Then, the focus shifts to Reinforcement Learning and the model-free learning components, such as `SARSA`, `QLearn`, and their variants.

2.1 Workload Scheduling

In the context of splitting the work among processes (as opposed to workload division at the thread level), the communication is achieved via the Message Passing Interface (MPI), as explained in subsection 2.1.1. To improve workload balancing, several approaches to dividing the workload among the Processing Elements (PEs) exist. The centralised data method implies a global centralised task queue and a master process handing new work to available PEs. Additionally, multiple algorithms able to perform this division exist, as explained in subsection 2.1.3. As the full data should be replicated to each process's internal memory, this approach is a bottleneck in memory-bounded contexts. The alternative is following a distributed data approach, as explained in subsection 2.1.4, where the work is equally divided among the PEs before the application starts. To dynamically balance the load, processes are now able to steal chunks of work from peers. The quality of the scheduling methods can be verified using multiple metrics, as explained in subsection 2.1.5.

2.1.1 The Message Passing Interface

Parallel programs make use of application programming interfaces to ensure communication among the processing units to achieve both data sharing and consistency in parallel regions. The Message Passing Interface (MPI) [14] is the most used standard in distributed memory contexts, where each parallel process only has access to its own local memory, and data-sharing is achieved through message passing. The messages can either be of the point-to-point type, such as MPI_Send or MPI_Recv, or collective communication, such as MPI_Bcast. By using MPI, scalability beyond the shared memory of geographically close components can be achieved. This standard is compatible with C, C++, and Fortran applications.

2.1.2 The LB4MPI Library

LB4MPI [1] is a library that contains a collection of Dynamic Loop Scheduling (DLS) algorithms. By allowing processes to communicate using MPI, this library supports the execution of scientific applications on High-Performance Computing (HPC) systems.

Available in both C and Fortran programming languages, this library is an extension of the DLB_tool, developed in 2007 by Cariño and Banicescu [15]. Mohammed et al. [1] built DLS4LB on top of DLB_tool, which in turn has been renamed as LB4MPI. The tool is used for parallelising and balancing the workload of scientific applications with simple parallel loops (1D loops) or nested parallel loops (2D loops) via efficiently distributing the workload among available resources. Based on a master-worker model, chunks of iterations (calculated based on the active DLS technique) are assigned by the master to workers whenever they become available. When not serving requests, the master also acts as a worker [1]. Alongside the centralised data approach, where each process replicates the whole work queue in its own memory space, the library also offers support for the distributed data method [9].

The LB4MPI library offers support for 14 different workload scheduling techniques: STATIC, SS, FSC, mFSC, GSS, TSS, FAC2, WF, AWF, AWF-B, AWF-C, AWF-D, AWF-E, and AF. These algorithms will be detailed in the following subsection.

2.1.3 Scheduling Algorithms

The techniques described in this section are suitable for a replicated data setup, with all processes able to access the full workload queue. In the context of HPC, scheduling is the process of organising the parallel computing elements and the corresponding data in the processor's space and time. Through scheduling, the system's performance boosting is obtained by optimising the resource allocation. This assignment of tasks is non-trivial, as the granularity of jobs differs, and the systems are often heterogeneous. Load balancing is achieved when the total workload is properly distributed among all the processing units, with the time corresponding to a process staying in an idle state being minimal.

Loops with little to no dependencies amongst iterations are major sources of parallelism [16]. Loop scheduling can either be *static*, where the tasks are divided and assigned to PEs before application execution, or *dynamic* (DLS), where the tasks are divided and *scheduled* during runtime [1]. Moreover, the DLS techniques can be further divided into *non-adaptive* and *adaptive*, based on the moment when information affecting the scheduling decision is gathered. In table 2.1, the two categories of DLS algorithms are expanded.

The **static** scheduling technique implies dividing the workload into a number of equally-sized chunks that correspond to the total number of PEs. As this division happens before executing the parallel code, the scheduling overhead is kept at a minimal value. Nonetheless, due to iterations requiring a non-equal amount of time to be completed (especially in irregular loops), workload imbalance often occurs.

Table 2.1: Dynamic Load Scheduling techniques included in `LB4MPI` [1].

Non-adaptive	**Adaptive**
Self Scheduling (**SS**) [17]	Adaptive Weighted Factoring (**AWF**) [18]
Fixed Size Chunking (**FSC**) [19]	AWF Batch (**AWF-B**) [20]
Modified Fixed Size Chunking (**mFSC**) [21]	AWF Chunk (**AWF-C**) [20]
Guided Self-Scheduling (**GSS**) [22]	AWF Batch + overhead (**AWF-D**) [20]
Trapezoid Self-Scheduling (**TSS**) [23]	AWF Chunk + overhead (**AWF-E**) [20]
Factoring2 (**FAC2**) [24]	Adaptive Factoring (**AF**) [25]
Weighted Factoring (**WF**) [26]	

Non-adaptive Scheduling Techniques

In this case, the scheduling decisions (e.g. determining the chunk size) depend on information gathered prior to the execution of the applications.

In Self Scheduling (**SS**) [17], when a PE becomes idle and requests work, it receives a chunk of size 1. Perfect load balancing is achieved, but the performance is degraded by the overhead produced by many individual requests, each with high communication costs.

In Fixed Size Chunking (**FSC**) [19], the optimal chunk size for *SS* is calculated based on the mean, standard deviation, and the overhead in the loop execution times, metrics received as input. The **mFSC** [21] relaxes the need to know these metrics a priori.

In Guided Self-Scheduling (**GSS**) [22], chunks of larger size are allocated at the beginning, with the chunk size decreasing based on the number of remaining iterations. Similarly, in Trapezoid Self-Scheduling (**TSS**) [23], the chunk sizes also decrease, but *linearly*.

In Factoring (FAC), the chunk size is calculated by using a probabilistic model based on the mean and standard deviation of execution times received as input. The modified version **FAC2** [24] is a more practical version. Half of the remaining iterations form a batch, and the chunk size is calculated by splitting the batch in a round-robin fashion.

In Weighted Factoring (**WF**) [26], speed-based weights are associated with each PE before the application's execution. The chunk sizes are calculated using FAC and scaled properly.

Adaptive Scheduling Techniques

The latest information regarding the system or the application is collected while the program is executed, and these are exploited for deciding the next chunk sizes.

Adaptive Weighted Factoring (**AWF**) [18] is based on WF, but the weight of each PE is flexible. No prior knowledge about the workloads is needed, as information regarding previous loops is considered. AWF Batch (**AWF-B**) [20] updates the weights not after each time step but after every batch of a scheduled loop. Similarly, in AWF Batch with scheduling overhead **AWF-D** [20], updates happen after each batch, and the scheduling overhead is considered. In AWF Chunk (**AWF-C**) [20], the weights are updated after each chunk, resulting in better load balancing, but increased overhead. In **AWF-E** [20], updates happen after the execution of every chunk, and the scheduling overhead is also considered.

Adaptive Factoring (**AF**) [25] behaves similarly to FAC, with the major difference that the mean and standard deviation of loop execution times are collected dynamically.

2.1.4 Work Stealing in Distributed Data Setups

In a distributed data setup, the total workload is divided among the PEs following a static division. In this way, each process would only hold in its memory the data it has to handle. This approach is particularly suitable when the application is memory-bounding or when the scalability of the system is desired. Regardless, the static division of work would likely introduce a level of load imbalance, as the iterations are not guaranteed to require the same level of effort. This effect can be reduced through work stealing.

In the work-shared policies exemplified in section 2.1.3, the idle PEs receive work from a master process, which might become a bottleneck. Nonetheless, in the work-stealing approach, idle PEs (the thieves) are responsible to find work themselves by querying busy PEs (the victims) and stealing their workload. In this way, the overhead of the victim is reduced. Yet, the cost of transferring the data among the processes is not negligible. There are multiple strategies for selecting the victim, such as probing potential victims, characterised by a tradeoff between larger searching overheads and having a lower number of tasks to move. Currently, the LB4MPI library only supports Random Work Stealing (RWS), where the victim is randomly selected from busy processes. In LB4MPI, the amount of work to be stolen is fixed from 0% to 100% through the StealRatio environment variable [9].

The efficiency of the scheduling strategies aforementioned in this section highly depends on the state of the environment, which is rather unpredictable and ever-changing. For example, by manually fixing the most promising scheduling algorithm for the whole application run, there is no guarantee this algorithm would remain the highest-performing choice during the whole execution of the application. In the following subsection, we introduce a series of metrics to be used in determining the performance of a DLS technique, which allow an objective comparison of different approaches.

2.1.5 Performance Metrics

Among the metrics characterising the timestep performance of a loop scheduling algorithm, *Time Measuring* and *Load Balancing* are widely encountered throughout the literature. Below are the definitions of such metrics that we will use further throughout this thesis.

Time Measuring Metrics

- $\mathcal{T}_{par}^{loop}$ (performance per loop) measures the execution time that the slowest process requires to finish its share of work during a single loop. Practically, the term *looptime* refers to the performance per loop for the slowest process. Likewise, the quantity $\mathcal{T}_{par}$ is the time needed to perform all the work during the whole application run and for all the loops, and it is approximated to the sum of all $\mathcal{T}_{par}^{loop_i}$.

Load Balancing Metrics

- The *percent load imbalance* [27] is calculated using Equation 2.1, where L_{max} is the maximum load of any process, and $\bar{L}$ is the mean load of all processes. Practically, the loop's parallel execution time can be used to quantify the load; hence, L can be viewed as $\mathcal{T}_{par}^{loop}$. This metric measures the performance that could be reclaimed through balancing the load. Any statistical property of the distribution is discarded.

$$\text{percent load imbalance} = (\frac{L_{max}}{\bar{L}} - 1) \times 100\% \tag{2.1}$$

- The *Coefficient of Variance* (c.o.v.) [24] from Equation 2.2 characterise the amount of variance in distributing the workload. The *Standard deviation* σ is needed to calculate the c.o.v. (in Equation 2.3, L_i is the load of the ith process out of n processes).

$$\text{c.o.v.} = \frac{\sigma}{\bar{L}} \tag{2.2}$$

$$\sigma = \sqrt{\frac{1}{n}\sum_{i=0}^{n}(L_i - \bar{L})^2} \tag{2.3}$$

- A positive *skewness* [27] (Equation 2.4) means that a small number of processes have a load higher than average, while a negative skewness implies having a lower workload on some processes. If the skewness is 0, the workload is normally distributed.

$$\text{skewness} = \frac{\frac{1}{n}\sum_{i=0}^{n}(L_i - \bar{L})^3}{\left(\frac{1}{n}\sum_{i=0}^{n}(L_i - \bar{L})^2\right)^{3/2}} \tag{2.4}$$

- The *kurtosis* [27] (Equation 2.5) characterises the size and frequency of deviations when distributing the workload. The workload is normally distributed if kurtosis is 0.

$$\text{kurtosis} = \frac{\frac{1}{n}\sum_{i=0}^{n}(L_i - \bar{L})^4}{\left(\frac{1}{n}\sum_{i=0}^{n}(L_i - \bar{L})^2\right)^{2}} - 3 \tag{2.5}$$

However, no performance metric can provide a complete picture by itself. Hence, a mixture of both time-measuring and load-balancing metrics will be used throughout this thesis.

Based on the assumption that the overall state of the application is not constant, a dynamic selection of the scheduling algorithm is required. This selection should take into consideration different performance metrics that are dynamically captured. An intelligent agent could be trained to learn the environment and make informed decisions in order to automatically select the most promising DLS. In the following section, several Reinforcement Learning methods and tools are introduced, which are later applied to resolve the automated scheduling algorithm selection problem.

2.2 Reinforcement Learning

Machine Learning (ML) is part of the wider field called Artificial Intelligence. The ML agents are trained in decision-making and aim to determine future outcomes based on input events. The vast domain of ML has three main branches, as follows [28]:

- **Supervised Learning** implies that the model learns from a labelled dataset under guidance. It helps to solve problems such as *Classification*, *Regression* or *Estimation*, by using *Neural Networks*, *Bayesian Networks* or *Support Vector Machines*.
- **Unsupervised Learning** hints that the model is trained on unlabeled data and features extracted. *Clustering* or *Prediction* problems help in e.g. *Big Data Visualisation* and are solved using algorithms such as *K-means* or *Gaussian Mixture Models*.
- **Reinforcement Learning** suggests that an agent learns the best actions in a trial-and-error fashion. The problem solved is of *Decision-making* type, and most representative algorithms are *Temporal-Difference Learning* (e.g. `QLearn` and `SARSALearn`) or *Markov decision process*, which in turn can be *model-based* or *model-free*.

In this Master's thesis, we focus on the RL part. This branch of ML focuses on a goal-oriented approach to learning problems that involve dealing with complex and unpredictable environments. An agent learns how an environment changes through trial and error - after performing an action, it is instantly presented with a *reward* and a new *state*, and it aims to receive higher rewards. The RL agent performs an action and receives feedback on it, the agent being guided in finding the correct solution, partially similar to what happens in unsupervised learning. However, the agent is not guided on what is the most promising action to perform next, behaviour specific to supervised learning.

Figure 2.1 illustrates the Agent-Environment interaction in an RL system at a random point in time. Given a state S, the agent performs action A. Based on the impact A has on the environment, the pair $\langle S, A \rangle$ is attributed a numerical value, the reward R.

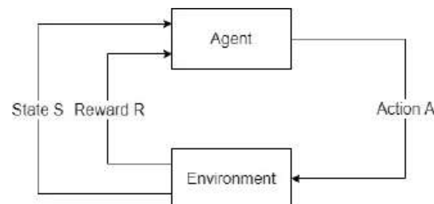


Figure 2.1: Components of an Agent-Environment RL system.

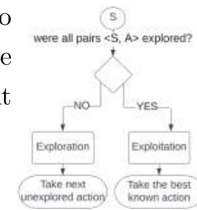
The reward R is a scalar value describing the extent to which the new state $S^{'}$ is desirable. Based on a *reward function*, the agent's utility is defined, and the agent must learn what action $A^{'}$ to perform next to maximize the expected rewards. Essentially, the current outcome is compared with all past actions. After measuring the performance of an action, the absolute value obtained is scaled into a relative one. For example, on the scale $[-1, 1]$, the action with the highest performance encountered so far is rewarded with 1, while a bad action is rewarded with -1. A performance situated between the two extremes would get a reward value closer to 0. When resolving the automated DLS algorithm selection problem, the agent can consider any performance metric from section 2.1.5 to be rewarded.

2.2.1 Types of Policies

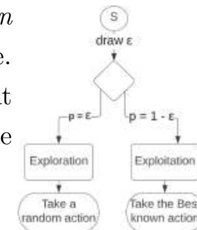
An action selection *policy* is a mapping between states and actions. Policies determine what action A should be performed in the current state S to get the agent to the next state $S^{'}$. The aim is to equip the RL agent with an action selection policy such that, in the long run, it converges to finding the optimal actions given a sequence of states.

One key characteristic of RL is that the system is assessed continuously with the learning process. There is a trade-off between *exploration*, where the agent performs new actions, and *exploitation*, through which the already experienced best action should be performed. Based on when the exploration phase happens, the following are possible policies.

- **Explore-First policy** implies that the first timesteps are used to *explore* all state-action pairs $\langle S, A \rangle$. Hence, the exploration phase would take no_states $\times$ no_actions steps. After exploring, the agent *exploits* the gained knowledge for all the remaining period.



- **Epsilon-Greedy policy** indicates the *exploration* and *exploitation* phases are not disjointed anymore, and each can happen anytime. Based on the value of the randomly drawn float $\epsilon \in [0, 1]$, the agent will explore a new action with a probability of $1 - \epsilon$ or exploit the already-gained knowledge with a probability of ϵ.



An *off-policy* learning strategy assumes that the policy is not relevant when deciding if a new action is good or bad. Oppositely, an *on-policy* assumes that experience comes from the agent, aiming to improve the agent's policy in real-time (an agent plays a game, and it improves based on the gained experience, and then plays the game again).

2.2.2 Types of Reinforcement Learning Agents

When learning, two main groups of strategies can be observed - *model free* and *model based.*

In a *model-based* learning (e.g. Dyna, Prioritized Sweeping, RTDP) approach, the model is used together with its corresponding utility function, which computes the expected reward when performing a certain action. Based on the utility function, a policy can be derived.

In a *model-free* learning (e.g. Monte Carlo Control, Temporal-Difference) approach, a model is not needed. Instead, an action-value function Q is used to derive a policy.

The model-based learning model is only appropriate when the environment allows finding a model. During DLS technique selection, the environment is heavily influenced by random factors, and determining a model is not feasible. Hence, a model-free approach is used.

Temporal-Difference (TD) is an implementation of model-free learning. Iteratively, the approximated value of the optimal action is updated after each cycle (one time-step) based on the new estimates. The next action to be performed is determined using a policy. Next, two important TD algorithms (`QLearn` and `SARSALearn`) are overviewed and compared.

QLearn

This model-free RL algorithm provides an off-policy TD control method. It is used to learn the action-value Q function, which provides the expected reward of performing an action in any given state. The Q function approximates the optimal value function Q^* independently of the policy being followed. The policy continues to have an impact on the sequence in which state-action pairs are visited and changed [28]. Correct convergence to the optimal solution occurs if all state-action pairs are continuously updated. An RL agent is tuned using the following two parameters that decide how new information is absorbed. These parameters are important when updating the Q-value in equation 2.6.

- The **discount rate** $\gamma \in [0, 1]$ balances the value of immediate and long-term rewards. Basically, if $\gamma = 1$ the whole execution, the agent would value the action far in the future just as much as the current rewarded action, and for $\gamma = 0$, only the current reward would matter. As the value of γ would impact the duration and quality of the learning process, its value should start at a high value and decrease over time, approaching a value closer to 0.
- $\alpha \in [0, 1]$ is the **learning rate**, and it dictates how much the agent values new observations over old knowledge. From a global perspective, α should start at a large value since, at first, all new information would bring the agent closer to discovering the optimal policy. It should decay over time, assuming every piece of new knowledge already gained places the agent in a state closer to the optimal one.

How the `QLearn` agent handles new information during any time step is defined by the Q-value update rule in equation 2.6 [28]. At this point, the agent is in state S and has performed action A. When updating the action-value function Q, it considers the old Q-value, $Q(S, A)$, the reward R gained for action A, and the potential Q-value of performing the next action with a maximal Q-value, a, while in state $S^{'}$ (the agent's new state).

$$Q(S, A) = Q(S, A) + \alpha[R + \gamma \max_{a} Q(S^{'}, a) - Q(S, A)]. \tag{2.6}$$

The `QLearn` pseudocode is displayed within Algorithm 1 [28].

Algorithm 1: Pseudocode for QLearn

Input: $\mathcal{S}$ - set of states; $\mathcal{A}$ - set of actions;
Output: Q - value function;

```
1  Initialise Q(S, A), ∀S ∈ 𝒮, A ∈ 𝒜(S) arbitrary and Q(terminal-state, -) = 0;
2  repeat
3  |   Initialise S;
4  |   repeat
5  |   |   Choose A from S according to the policy derived from Q;
6  |   |   Take action A and observe R (reward), S' (new state);
7  |   |   Q(S, A) ← Q(S, A) + α[R + γ max_a Q(S', a) − Q(S, A)];
8  |   |   S ← S';
9  |   until s is terminal;
10 until there are no time-steps left;
11 return Q;
```

SARSALearn

SARSA is a model-free RL algorithm providing an on-policy TD control method. In the SARSA context, the learning starts with being in the state S and performing an action A. Based on the reward R, the agent enters the state $S^{'}$ and picks a new action, $A^{'}$ - the name of the algorithm is based on this quintuple, $(S, A, R, S^{'}, A^{'})$. SARSA is used to learn the action-value function by estimating $Q(S, A)$. For a non-terminal state S, the update in equation 2.7 [28] takes place, while for a terminal state, $Q(S^{'}, A^{'})$ is defined as zero.

$$Q(S, A) = Q(S, A) + \alpha[R + \gamma Q(S^{'}, A^{'}) - Q(S, A)]. \tag{2.7}$$

The pseudocode for SARSA can be found in Algorithm 2 [28].

Algorithm 2: Pseudocode for SARSA Learning

Input: $\mathcal{S}$ - set of states; $\mathcal{A}$ - set of actions;
Output: Q - value function;

```
1  Initialise Q(S, A), ∀S ∈ 𝒮, A ∈ 𝒜(S) arbitrary and Q(terminal-state, -) = 0;
2  repeat
3  |  Initialise S;
4  |  Select A from S according to the policy derived from Q;
5  |  repeat
6  |  |  Take action A and observe R (reward) and S' (new state);
7  |  |  Choose action A' from S' according to the policy (e.g. explore-first/ϵ-greedy);
8  |  |  Q(S, A) ← Q(S, A) + α[R + γQ(S', A') − Q(S, A)];
9  |  |  S ← S';
10 |  |  A ← A';
11 |  until s is terminal;
12 until there are no time-steps left;
13 return Q;
```

QLearn versus SARSA

- These TD algorithms store the current state's reward and the most promising action for future use, and they converge provided the environment is finite.
- QLearn is an off-policy algorithm, while SARSA is on-policy. With a greedy policy, both algorithms would reach the same outcomes; using other policies yields differences.
- While QLearn should converge to an optimal policy Q^*, SARSA only provides an approximation of Q^*. QLearn is more *aggressive*, while SARSA is more *conservative*. Sutton and Barto [28] demonstrate such behaviour through the Cliff Walking experiment: given a task of moving from *Start* to *Goal* and avoiding falling off a cliff, the QLearn agent prefers taking the shortest path, ignoring the risk of falling, while the SARSA agent will take a longer but safer path, minimising the risks. Moreover, the QLearn agent receives a smaller mean reward when compared with the SARSA agent.
- The QLearn agent is more useful in fast-paced environments where mistakes are not costly. However, with expensive errors, SARSA is the preferable choice.

With larger state spaces, QLearn and SARSA do not scale well due to the memory-inefficient Q-tables. Variants of the two algorithms have been explored, aiming to resolve the scalability problem. Further, we look at the most promising alternative versions.

DoubleQLearn

It has been observed that the standard `QLearn` uses the same values twice, to both select and evaluate an action. This increases the likelihood of selecting exaggerated values, leading to overoptimistic value assessments. `DoubleQLearn` detaches the selection process from the evaluation. To do so, two value functions Q^A and Q^B are needed. For each update, one function is randomly picked to determine the policy, while the other function is used in determining its value. Update rules are found in Equations 2.8 and 2.9 [29].

$$Q^A(S,A) = Q^A(S,A) + \alpha(S,A)\Big(R + \gamma Q^B(S^{'}, \text{argmax}_a Q^A(S^{'}, A)) - Q^A(S,A)\Big) \tag{2.8}$$

$$Q^B(S,A) = Q^B(S,A) + \alpha(S,A)\Big(R + \gamma Q^A(S^{'}, \text{argmax}_a Q^B(S^{'}, A)) - Q^B(S,A)\Big) \tag{2.9}$$

QVLearn

In this variant of `QLearn`, both the Q and V functions are kept track of. While the V function is trained using normal TD methods, the Q-values are learnt from the V-values using the one-step `QLearn` algorithm (Algorithm 1). As the V-function does not consider actions and it is updated more often, it might converge faster than the Q function. Equation 2.10 is followed when updating the V-function, η_t is 1 if the state S occurred, and 0 otherwise, while Equation 2.11 explains how the Q-values are updated [30].

$$\begin{aligned} &V(S) = V(S) + \alpha\delta_t e_t(S) \text{ with eligibility trace} \\ &e_t(S) = \gamma\lambda e_{t-1}(S) + \eta_t(S) \text{ and } \delta_t = R + \gamma V(S^{'}) - V(S) \end{aligned} \tag{2.10}$$

$$Q(S,A) = Q(S,A) + \alpha(R + \gamma V(S^{'}) - Q(S,A)) \tag{2.11}$$

DeepQLearn

`DeepQLearn` improves `QLearn` by switching from a (not scalable) Q-table that expected reward values into using a Neural Network that approximates the Q-value for an action given a state. During the training process, the same iterative process is undergone, but instead of updating the values, the Neural Nets weights are tuned [31].

Expected-SARSA

Under the observation that the `SARSA`'s convergence requires every state to be visited infinitely often, sufficient exploration is provided. `ExpectedSARSA` is more flexible than the classic `SARSA`, as it can either be off-policy based (if a greedy expected return is employed, it gets transformed into `QLearn`) or on-policy (the expected return is computed for all actions). The action-value function differs from the classic `SARSA` algorithm, and it follows the rule in equation 2.12 [32].

$$Q(S,A) = Q(S,A) + \alpha\Big(R + \gamma\sum_a \pi(a|S^{'})Q(S^{'},a) - Q(S,A)\Big) \tag{2.12}$$

In a workload scheduling context, these techniques can be used for automatic algorithm selection. A selection of literature-reviewed texts, including the topic of how RL can improve automated DLS selection, is overviewed in the subsequent chapter.

3 Related Work

This chapter presents the literature context of this work. The related work is overviewed and critically analysed while underlining potential improvements.

Ciorba et al. [2] show that no scheduling technique is guaranteed to improve every parallel application's performance in all circumstances. This result has been achieved using the OpenMP-based library `LaPeSD-libGOMP` via benchmarking using different applications. Based on the observation that certain scheduling techniques outperform others in different scenarios, the automated scheduling algorithm selector we propose should be able to alternate between multiple scheduling techniques in the portfolio when the context changes.

The premise of Korndörfer et al. [3] is that the scheduling options in OpenMP are insufficient to address the load imbalance that arises during the execution of the multithreaded application, and `LB4OMP` is proposed. For numerous application-system pairs, the scheduling techniques in `LB4OMP` outperform the scheduling options in OpenMP. However, the manual algorithm selection requirement has proven to be time-consuming and demanding, and the selected algorithm is likely to underachieve in the future, demanding its change.

Mohammed et al. [8] proposed `Auto4OMP` as a novel automated load balancing tool for OpenMP applications. It aims to resolve the manual DLS selection problem of `LB4OMP` [3]. Expert systems are used to determine the *chunk* parameter and the most promising scheduling selection algorithm automatically during runtime. An 11% boost in performance, when compared to `LLVM`'s *schedule(auto)*, has been achieved by minimising the load imbalance at the thread level. The drawback of expert systems is that with the addition or deletion of any scheduling technique, the algorithm selection rules also need to change.

The work of Kury [7] is the closest related work to this thesis. RL techniques are used to select the optimal scheduling algorithm, aiming to reduce the effect of load imbalance on the performance of computationally-intensive applications. `LB4OMP` with RL can alternate multiple scheduling techniques during run-time. The software implements three action selection policies - Explore First, Epsilon Greedy, and Softmax, six reward methods based on the execution time of all threads and five learning agent types, among which `QLearn` and `SARSALearn`. Furthermore, a novel learning method is proposed, Chunk-Learning, which searches for the optimal chunk size to be used together with the *dynamic* schedule. The

evaluation methodology consists of benchmarking using Mandelbrot and SPHYNX Evrard Collapse. Results show that while `QLearn`, `SARSALearn`, or other variations achieve similar outcomes, the agent's nature is not as important as the policy used or the reward metric. Altogether, the results of using RL rarely outperform the expert systems proposed in `Auto4OMP`. We will extend the process-level `LB4MPI` load balancing library with Kury's software. As the amount of application load imbalance seems to influence the selection quality, we will study how different RL selection methods react to various levels of load imbalance. Furthermore, we plan to upscale the benchmarking setup.

Pearce et al. [27] shows the *percent imbalance* does not reveal how quickly a balancing algorithm can correct the imbalance. They proposed using statistical measures such as *standard-deviation* σ, *skewness*, and *kurtosis*. We will test if these measures can improve the RL agent's DLS selection quality by modifying its reward function. By combining information from Pearce et al. [27] and Hummel et al. [24], we conduct large-scale experiments with a reward function based on the processes' parallel execution time $c.o.v. = \sigma/\mu$.

Rashid et al. [16] use RL for automated DLS algorithm selection. The authors have considered the `QLearn` and `SARSA` methods and have investigated the optimal values for α/γ; just as Kury did, they concluded that `QLearn` and `SARSA` achieve similar results. A potential problem leading to this finding could be not allocating sufficient time for the exploration phase. To overcome this issue, we increase the number of time steps to 1'500 while spending around 10% of time exploring. As they only evaluated the performance using one application, QTM, one possibility is that the two RL agents are not the best fit for this particular setup. We plan to test our implementation on at least three scientific applications.

Banicescu et al. [5] studies how process-level DLS-with-RL online selection improves the load balancing in the time-stepping application QTM. For any number of processors, the automated DLS selection statistically outperforms the DLS-only approach. While `QLearn` and `SARSA` RL agents would differ from one another through their DLS selections, statistically, there is shown to be no advantage in picking one over another. It is also shown that for a fixed number of processes, the RL agent is insensitive to variations of α and γ.

Boulmier et al. [33] study RL-based automated DLS selection. They propose a new robustness metric as the reward function, the *flexibility*, that measures how DLS techniques resist facing execution time perturbations, but this is not always suitable. Experiments are run on the SimGrid framework, which is a *simulation* of a large-scale computing system. The `QVLearn` agent is shown to perform marginally better than other agents under extreme conditions, and the RL component accounts for less than 0.01% of the application execution.

Wetten [9] extended the `LB4MPI` library with distributed-data load balancing capabilities. Random Work Stealing is implemented as a way for processes that have finished their workload to help busy processes finish the work faster. The percentage of work to be stolen is fixed through the StealRatio, and their experiments show the value of StealRatio does not impact the application's performance. The authors only conducted experiments on a relatively small scale, with one application, Dist-D. We will verify this hypothesis by upscaling the experiments through benchmarking using a distributed version of PISOLVER and 200

processes. In our thesis, we conduct an RL-based selection of the StealRatio parameter through a novel approach, the StealRatioSelector, which behaves similarly to Kury [7]'s chunk parameter selector RL agent, but using a portfolio of StealRatios.

Table 3.1: Main characteristics of closely-related works.

	Banicescu, Ciorba, Srivastava, 2012 [5]	**Boulmier, Banicescu, Ciorba, Abdennadher, 2017** [33]	**Luc Kury, 2022** [7]	**This work**
Library	`MPI`-based	`MPI`-based	`OpenMP` based, `LB4OMP` [3]	`MPI`-based, `LB4MPI` [1]
Scheduling techniques	STATIC, AWF-B, AWF-C, AF, FAC2, GSS, mFSC	STATIC, FSC, GSS, FAC, AWF, AWF-B,-C,-D,-E, AF	STATIC, SS, GSS, GAC, TSS, STATIC STEAL, mFAC2, AWF-B,-C,-D,-E	STATIC, SS, FSC, mFSC, GSS, TSS, FAC2, WF, AWF, AWF-B,-C,-D,-E, AF
Automated selection methods	`QLearn`, `SARSALearn`	`QLearn`, `SARSALearn`, `E-SARSA`, `QVLearn`, `DoubleQLearn`	RandomSelection, ExhaustiveSelection, BinarySelection, ExpertSelection, `QLearn`, `SARSALearn`, `DoubleQLearn`, `E-SARSA`, `QVLearn`, `ChunkLearn`	`QLearn`, `SARSALearn`, `ChunkParameter-Selector`, StealRatioSelector
Benchmarking application	QTM	SimGrid-SMPI	Mandelbrot, SPHYNX Evrard Collapse	PISOLVER, Mandelbrot, SPHYNX Evrard Collapse
Number of workers P	$P \in \{2, 4, 8, 12, 16, 20, 24\}$	$P \in \{2^{10}, 2^{11}, 2^{12}\}$	$P = 20$	SPHYNX: $P = 40$ else $P = 200$
Observations, Limitations	`QLearn` and `SARSA` achieved similar results, independent of α and γ; One benchmarking application might be insufficient	Experimenting using one application - a simulation; Flexibility metric is mostly an unsuitable reward; `QVLearn` is marginally better than others; the overhead of using RL selection: <0.01%	RL-based selection outperforms manual selection, but not ExpertSelection; agent type is less important than other factors e.g. policy, reward	For a centralised-data context, analyse RL-based DLS selection and chunk parameter selection; compare results with RL-tuned RWS within distributed data

Among the reviewed literature, we have identified three that are closely related to our work. Table 3.1 puts into perspective the main features of each text. The red coloured algorithms are not compatible with the `LB4MPI` library, while the teal coloured ones have been implemented in `LB4MPI` library, but not in `LB4OMP`. In our work, when completing the DLS portfolio, we can only select the techniques compatible with a certain application and context (e.g. for RL-based selection, we would not consider WF, as its parameters are tuned over the whole run). Also, even if implementations of all RL algorithms have been successfully ported to `LB4MPI`, experiments will be conducted mainly with `QLearn` and `SARSALearn`. These two agent types are widely encountered in the literature, and studies show that variants exceedingly rarely outperform the originals. Rather, the variants were designed to boost the scalability and not improve the execution times through a more-qualitative action selection. We would rather focus our energy on finding the right combination ⟨`agent-type`, `policy-type`, `reward-type`⟩ that performs well for a particular application's context.

Notes on the implementation are provided in the next chapter.

4 Implementation

This section contains notes on implementing the Reinforced Learning tool as an extension to the LB4MPI library.

4.1 The LB4MPI Library

As described in Section 2.1.2, the LB4MPI [1] library is a Dynamic Loop Scheduling (DLS) tool that uses MPI to achieve workload balancing via process-level communication and synchronization. Due to the manual setup required by WF, FSC, and SimAS, these scheduling techniques have been removed from the DLS portfolio. We further consider STATIC, SS, mFSC, GSS, TSS, FAC2, AWF, AWF-B, AWF-C, AWF-D, AWF-E, and AF.

Algorithm 3: Calling the C version of the LB4MPI library (application perspective)

```
1  #include <mpi.h>
2  #include "LB4MPI.h"
3  #include "reinforcement-learning/c_connector.h"
4  MPI_Init(&argc, &argv);
5  MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
6  MPI_Comm_rank(MPI_COMM_WORLD, &myid);
7  foreach timestep do
8      distributedWorkLoad = ...// Distribute the data to all MPI ranks
9      infoDLS DLS_info;
10     ...// Setup parameters for DLS_Parameters_Setup and DLS_info
11     DLS_info.looptitle ← "loop_title";
12     DLS_info.workload ← work_load;
13     DLS_Parameters_Setup(MPI_COMM_WORLD, &DLS_info, nprocs, ...);
14     DLS_DataSetup(&DLS_info, distributedWorkLoad, ...);
15     DLS_StartLoop(&DLS_info, 0, work_load, DLS_method);
16     while !DLS_Terminated(&DLS_info) do
17         DLS_StartChunk(&DLS_info, &start, &chunk_size);
18         end = start + chunk_size;
19         if start < end then
20             Perform application-related computations
21         end
22         DLS_EndChunk(&DLS_info);
23     end
24 end
25 ...// Display results & report performance
26 MPI_Finalize();
```

Algorithm 3 contains information on how the C version of LB4MPI is accessed from an application perspective. The teal code highlights code to be added for the RL features in both the replicated and distributed application versions. The blue code indicates that data is distributed. Alongside the legacy parameters (e.g. h_overhead, sigma, Xeon_speed . . .), two extra parameters must be provided to use the RL features. As the new software is compatible with multiple synchronous loops, the *looptitle* has to be passed to the *DLS_info* structure. Multiple instances of *infoDLS* need to be created to access the multi-loop feature; the above code exemplifies a single-loop program. Furthermore, to calculate the size of the chunk parameter portfolio, *DLS_info.workload* needs to be initialised with the task's size prior to executing *DLS_Parameters_Setup*. As a prerequisite, the user should also export the environment variable for the desired RL agent type, e.g. `export MPI_RL_OPTION=QLearner`, as the RL component is ignored by default.

Figure 4.1 illustrate how the C (or Fortran) LB4MPI library can access the RL feature via accessing the C-to-C++ (or f90-to-C++) interface. Communication is facilitated through three functions: *DLS_Parameters_Setup*, *DLS_StartLoop*, and *DLS_EndLoop*. It can also be observed that the RL agent does not directly influence the application computations but rather helps pick the right DLS technique to balance the workload. The RL agent analyses the performance metrics for different DLS techniques and decides which DLS will most likely achieve higher performance during the next timestep.

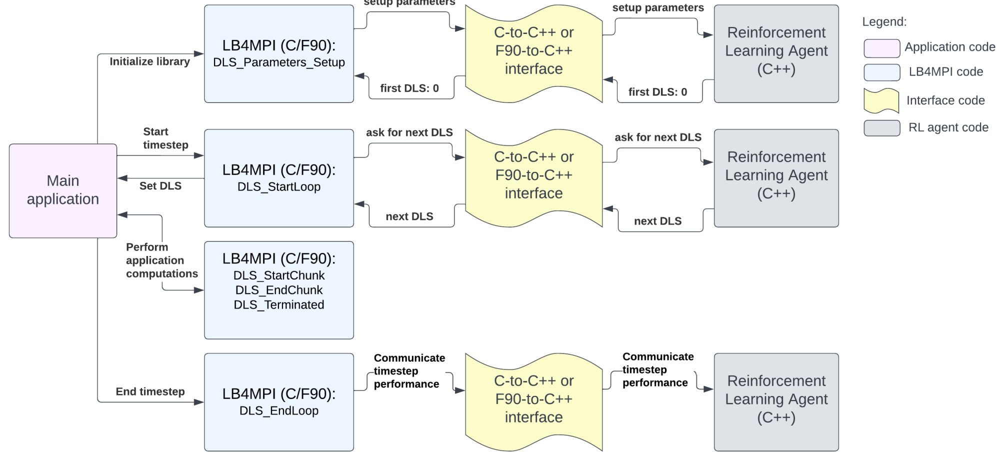


Figure 4.1: Communication between LB4MPI and the RL agent when automatically determining the most promising DLS technique

An initial version of the RL extension was developed in 2022 by Kury [7] during their Master's thesis. Initially, this tool is built as an extension to the LB4OMP/Auto4OMP library, which aims to accumulate a portfolio of thread-level scheduling algorithms. Kury claims the extension is *lightweight*, *encapsulated*, *extensible*, and *portable*, which allows easier integration with other tools, such as LB4MPI. Our Master's thesis is part of a strategy to achieve multi-level automated DLS algorithm selection at both process and thread levels. Naturally, our software solution updates Kury's implementation instead of building an entirely new program. Based on this decision, a future project unifying LB4OMP and LB4MPI with RL capabilities would be considerably easier to be developed.

As LB4OMP is entirely written in C++, Kury's extension can be straightforwardly integrated with C++ libraries. However, linking the C LB4MPI library and the C++ RL Agent extension has proven challenging since C is not directly compatible with C++ objects, e.g. AgentProvider. As an alternative to either rewriting the entire LB4MPI library in C++ or rewriting the RL component in C, a C-to-C++ interface has been developed by consulting Oracle resources publicly available online.

Such that a broader selection of scientific applications can use LB4MPI with RL features, integration between Fortran applications and the C++ RL agent software is desired. 16% of this Master's thesis has been invested into developing a solution. The first approach is to access the C LB4MPI library from the Fortran main application via an f90-to-C interface, which in turn has to access the RL agent via the C-to-C++ interface. However, due to considerable differences between Fortran and C (e.g. MPI_COMM_WORLD is represented as an integer in Fortran but struct-like in C), this task has proven not feasible. Hence, the Fortran version of LB4MPI has been updated to call the RL agent code via an f90-to-C++ interface. From a technical perspective, this approach is not as challenging from a technical perspective, as the C++ code only returns the subsequent DLS technique as an integer. The performance data is recorded by the library, and it is reported through LoopData.

The LoopData structure and other features of the RL agent extension are explained in the following section.

4.2 The Reinforcement Learning Extension

This section will discuss the components of the RL extension while highlighting the differences between the initial and the current software versions. For a better understanding, the reader is redirected to the *Implementation* notes of Kury's Master's thesis [7].

The main task of the RL agent is to analyse data regarding various dynamically gathered measurements for a portfolio of DLS techniques. Based on analysing multiple timesteps, the agent is able to make an informed decision about what DLS technique is most likely to achieve the highest performance during the next timestep. All the communication between the LB4MPI library and the RL component happens through a single MPI rank, the *foreman*. When reporting the performance of the latest timestep, the foreman consults all ranks and computes the reported quantity, e.g. load imbalance percent or looptime c.o.v.

The performance data is gathered by functions DLS_StartLoop() and DLS_EndLoop() of the LB4MPI library. Afterwards, it is reported to the RL agent via the LoopData structure. One instance of LoopData holds information about a single loop. Performance updating of the latest timestep can be done as follows: getLoopData(info->looptitle)-> cTime = time. Alongside the total timesteps, the number of MPI ranks, the execution time, and the load imbalance for the latest timestep, which can also be found in the previous version of the software, new statistical metrics are reported: cStdDev, cSkew, cKurt, and cCOV (detailed in Section 2.1.5). The LoopData structure also keeps evidence of the overall top metrics encountered during runtime through best- variables, e.g. bestTime.

As a feature, consulting the RL agent can cease after a certain number of timesteps. For example, instead of spending the full 1'500 timesteps searching for the DLS technique, synchronization and communication costs can be discarded by only searching during the first 500 timesteps: a user can manually set the `timeStepsLimit`, and the `autoSearch` flag is switched to false if this limit is crossed. This option is only recommended for environments where conditions remain constant throughout the whole execution.

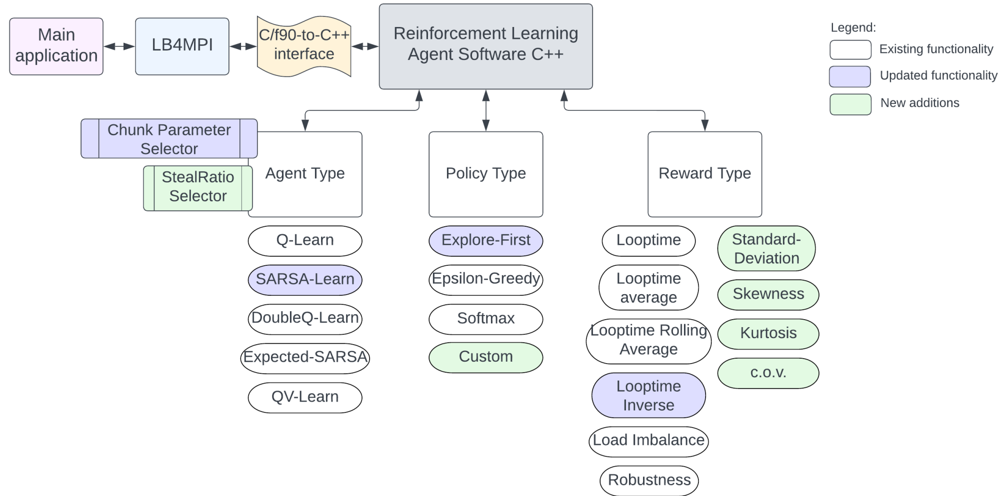


Figure 4.2: Components of the RL Agent

When the new agent is initialised, all the environment variables (env var) are read, and the agent's parameters are set accordingly, as opposed to using the default value when an env var is missing. Figure 4.2 illustrates the main components used to configure the RL agent.

Agent Types

The main agent types have been thoroughly described in section 2.2. The value can be user-provided through the `LB4MPI_RL_AGENT_STATS` environment variable via the leading integer code as it follows: 8 - `QLearner`, 9 - `DoubleQLearner`, 10 - `QVLearner`, 11 - `SARSALearner`, 12 - `ExpectedSARSALearner`. The software also supports two meta agents, as follows: 15 - `ChunkParameterSelector`, and 16 - `StealRatioSelector`. While the integer code for each agent can be misleading in the context of `LB4MPI`, this is a direct result of how `LB4OMP` represents different selection strategies (e.g. 1 - exhaustive search, 2 - binary search, 4 - expert systems, and values > 6 for RL Agents). The legacy identifiers are preserved so that a multi-level scheduling application using both `LB4OMP` and `LB4MPI` with RL would be straightforward to implement. To reduce the ambiguity, the user can now specify the more-intuitive full name of the RL Agent via a string.

It must be noted that a bug has been found and corrected in the implementation of the `SARSALearn` agent: while equation 2.7 states that the quantity $Q(S', A')$ should be calculated, the previous version of the software used the last performed action A instead of the next action A'. This way, the agent would be an *off-policy* one, hardly different from `QLearn`. The correct action is now retrieved through the in-use policy. When policies are based on randomness, A' is not guaranteed to be selected twice in a row.

Kury used a special kind of agent to select the optimal chunk parameter for the *dynamic* schedule. In short, this technique implies building an array of potential chunk values and, through monitoring the performance of each, finding the optimal chunk size. The number of chunk parameters in the array is calculated using equation 4.1, with N corresponding to the total workload size and P representing the number of workers.

$$no_chunks = \lfloor \log_2 \left(\frac{\mathrm{N}}{\mathrm{P}}\right) \rfloor - 1 \tag{4.1}$$

Equation 4.2 shows how each chunk size in the array is computed.

$$chunkArray[i-1] = \frac{N}{2^i * P}, i \in [1, \dots, no_chunks] \tag{4.2}$$

This meta agent requires creating a secondary agent, e.g. *QLearn*, whose type is specified through the env var `LB4MPI_RL_SELECTOR_TYPE`. Kury's name for this kind of agent is the `ChunkLearner`. While the problem solved is rather an *algorithm selection with configuration parameter selection*, a better fitting name is `ChunkParameterSelector`. While Kury's version would only work with the *dynamic* schedule (equivalent to `LB4MPI`'s *SS*), a modified version of this agent can be used to set the lower bound chunk for any DLS techniques (e.g. GSS) at run time instead of setting the chunk parameter directly.

Furthermore, we propose the `StealRatioSelector` as a novel strategy that uses RL to select the optimal StealRatio $\in [0, 100]$. This kind of agent is only compatible with a distributed data setup that allows RWS. Similarly to the ChunkParameterSelector, a secondary agent specified through the env var `LB4MPI_RL_SELECTOR_TYPE` needs to be instantiated. Dissimilarly, the values in this agent's portfolio are independent of the sizes of the problem N and P. Rather, the user can provide the size S of the portfolio through the env var `LB4MPI_RL_SR_PORTFOLIO_SIZE` (default: 10), and the space $[0, 100]$ will be divided into $S \times tileSize$ parts. Equation 4.3 displays how the StealRatio array is computed.

$$\begin{gathered} tileSize = \frac{100}{S} \\ stealRatioArray[i-1] = i * tileSize, i \in [1, S] \end{gathered} \tag{4.3}$$

Policy Types

To determine the next action, the RL agent can use three different strategies (provided via the `LB4MPI_RL_POLICY` env var) and a customizable one:

- `Explore First` (`explore-first`) - First timesteps are used to *explore* all possible combinations of states and actions exactly once. In this context, a state is the active DLS, while an action changes the current DLS with a new one. $DLS_MethodCount^2$ timesteps are needed for the exploration phase. One could argue that exploration should take $DLS_MethodCount$ timesteps, but the order in which the DLS techniques are tried is important (e.g. if *SS* has performed badly, thus setting the performance

bound to a bad value, it is likely that the DLS coming after *SS* would get rewarded better than in a hypothetical case where it was chosen prior to it, and this bias should be avoided). With this software version, it is guaranteed that all combinations are tried exactly once. After exploring, the agent *exploits* the knowledge to determine which DLS technique is likely to achieve the highest performance.

- `Epsilon Greedy` (`epsilon-greedy`) - *Exploration* and *exploitation* phases are not disjointed, and each can happen anytime. A float is randomly drawn, and the agent will explore a new state-action pair with $prob = 1 - \epsilon$ or exploit the knowledge with $prob = \epsilon$. The initial $\epsilon \in (0, 1)$ is user-specified through the environment variable `LB4MPI_RL_EPSILON`, and it decays over time based on `LB4MPI_RL_EPS_DECAY`.
- `Softmax` (`softmax`) - This is not based on either exploration or exploitation. Rather, given a state, each possible action is attributed a soft probability of it being selected, which is proportional to the action's Q-value. The temperature $\tau \in (0, +\infty)$ is user-specified through the environment variable `LB4MPI_RL_TAU`, and a higher value of τ implies a higher probability of selecting an action with a higher Q-value. The long-run performance increases by alternatively picking all the highly-rewarded actions.
- `Customizable` (`custom`) - The policy is useful when experienced users run custom experiments. For instance, if an ordered list of DLS techniques is provided, one can replay the selection to find the runtime standard deviation associated with the library (see Table 5.3). This policy works by hijacking other existing policies.

Reward Types

The RL agent can use reward functions to learn how a particular method has performed during the latest timestep. This incentive measure compares the current reward signal with other captured rewards; hence, this quantity needs to be translated from an absolute value to a relative scale. Aiming to gain high rewards, the agent is motivated to pick the most promising DLS techniques when not exploring.

Algorithm 4: Setting the reward based on input signal x

```
1  double x ← stats.reward_signal;
   // Read (r+, r0, r−) from env var LB4MPI_RL_REWARD_NUM
2  if x ≤ 1.05 × lower_bound then
   |  // Good case
3  |  if (x < lower_bound): set_lower_bound(x);
4  |  return r+;
5  end
6  else if x ≥ 0.95 × higher_bound then
   |  // Bad case
7  |  if (x > higher_bound): set_higher_bound(x);
8  |  return r−;
9  end
   // Neutral case 1.05 × lower_bound < x < 0.95 × higher_bound
10 return r0;
```

Algorithm 4 shows how an input reward signal x is transformed into the rewarded quantity. In our updated version of the software, the agent would be rewarded r_+/r_- if it is within

5% from the bound, as opposed to strict comparison with the limits. This strategy has proven to increase the quality of DLS selection by selecting the highest-performing DLS and increasing the reward of other highly-performing techniques.

The reward type can be user-specified through the `LB4MPI_RL_REWARD` environment variable. Two categories of rewards can be identified based on the nature of the reward signal. The following rewards are based on the loop execution time of the slowest process, $\mathcal{T}_{par}^{loop}$:

- `Looptime` (`looptime`) - Award with r_+, r_0, or r_- the looptime performance x.
- `Looptime Average` (`looptime-average`) - Keep track of the averaged looptime performance for **ALL** previous timesteps, and reward the looptime of the latest DLS with r_+ or r_-, based on its performance being situated below or above this quantity.
- `Looptime Rolling Average` (`looptime-rolling-average`) - Compare the input signal x with the averaged looptime of the last `ROLLING_AVG_WINDOW` (int, env var) timesteps, and reward a good performance with r_+, or a bad one with r_-.
- `Looptime Inverse` (`looptime-inverse`) - Reward the current DLS technique with $R(x) = \frac{1}{x} \times c$, where c is the `INVERSE_REWARD_MULT` (used to combat small looptimes). In the latest version of the software, the user can change the environment variable `LB4MPI_RL_INV_MULT` from its default value of 10.
- `Robustness` (`robustness`) - Quantify the ability of a DLS technique to resist variations in the loop iterations execution time [33]. The reward is calculated using the formula $\tau \times \mathcal{T}_{par}^{FASTEST} - x$, where τ is the tolerance factor (which can be user-provided via the env var `LB4MPI_RL_TAU`), and $\mathcal{T}_{par}^{FASTEST}$ is the fastest loop time recorded.

The following reward types use the imbalance of the parallel execution time as the reward signal, based on load balancing metrics introduced in Subsection 2.1.5.

- `Percent Loadimbalance` (`loadimbalance`) - Award with r_+, r_0, or r_- the load imbalance percent obtained by the current DLS. A load imbalance of close to 0 implies an even workload distribution among all processes and it is awarded r_+.

  The above reward does not provide insights on correcting any imbalance, as statistical data is ignored. The following reward types are based on statistical moments and were added with the newest version of the software. The load imbalance in the original definitions from subsection 2.1.5 has been replaced for practical reasons with the parallel execution time of each process.

- `Standard Deviation` (`stddev`) - metric detailed in Equation 2.3.
- `Coefficient of Varriance` (`cov`) - metric detailed in Equation 2.2.
- `Skewness` (`skewness`) - metric detailed in Equation 2.4.
- `Kurtosis` (`kurtosis`) - metric detailed in Equation 2.5.

Multiple strategies for setting reward values exist. The triple of doubles $\{r_+, r_0, r_-\}$ can be user provided via the environment variable LB4MPI_RL_REWARD. The default reward is the Negative reward [0.0, -2.0, -4.0]. The Neutral reward [+2.0, 0.0, -2.0] offsets the Negative reward to avoid the local extrema situation. A Positive reward can also be used, such as [+4.0, +2.0, 0.0]. Throughout this thesis, we will slightly modify the Negative reward to [0.01, -2.0, -4.0], so when a DLS achieves high performance, its reward would not be 0, to not be confused with the initial value in the agent's Q-table, zeroed.

The **Initialiser Type** can be user set through the LB4MPI_RL_INIT environment variable, and it specifies how to initialise the Q-table of a new agent. The Zero Initialiser (zero) fills the entire Q-table with 0s. The Random Initialiser (random) would fill the Q-table with random values bounded by the bad reward r_- and the good reward r_+. Lastly, the Optimistic Initialiser (optimistic) would fill the Q-table with values higher than realistically obtainable. Based on the experimental results derived by Kury [7], only the Zero Initialiser will be further considered.

The **Decay Type** is provided by the user through the LB4MPI_RL_DECAY env var, and it commands how the decay of the learning rate α and the exploration rate ϵ vary from timestep to timestep. The decay can be of type step, where after each timestep LB4MPI_RL_ALPHA is linearly reduced based on LB4MPI_RL_ALPHA_DECAY until the minimum is reached, ALPHA_MIN, and LB4MPI_RL_EPSILON is linearly reduced with LB4MPI_RL_EPS_DECAY down to EPSILON_MIN. Or it can be exponential, where the decay is also based on these parameters but follows an exponential behaviour. Throughout this thesis, we will use the exponential decay, which is more appropriate to our learning strategy.

The RL extension can enter a debugging mode by setting the _RL_DEBUG parameter from LB4MPI/reinforcement-learning/agents/kmp_agent.h to 1 or 2. Useful messages regarding the current state of the software would be printed to stdout.

Out of all the customizable components of an RL agent, the most impactful items are contained in the triple ⟨agent-type, policy-type, reward-type⟩. By altering these, the widest range of results can be achieved. We will experiment with changing these values and evaluate how component types affect each other. When not otherwise stated, all RL agent's parameters are unchanged from their default value. The default values are aggregated in the next section, in Table 4.1.

4.3 Environment Variables

All environment variables described throughout this chapter are centralised in Table 4.1. This table contains information on the environment variable's name in LB4MPI and, below it, the LB4OMP counterpart. The table also provides the variable's name internally used by the software, the default value, its type, description, and the range for the values.

Table 4.1: Supported environment variables for LB4MPI / LB4OMP with RL features

Environment Variable	Internally used Variable	Default value	Description and Possible Values
LB4MPI_RL_GAMMA KMP_RL_GAMMA	GAMMA	0.95	Agent's Discount factor; double $\gamma \in [0, 1]$
LB4MPI_RL_ALPHA KMP_RL_ALPHA	ALPHA	0.85	Agent's learning rate; double $\alpha \in [0, 1]$
LB4MPI_RL_ALPHA_DECAY KMP_RL_ALPHA_DECAY	ALPHA_DECAY_FACTOR	0.01	Learning rate decay; double $\in [0, 1]$
LB4MPI_RL_ALPHA_MIN KMP_RL_ALPHA_MIN	ALPHA_MIN	0.10	Minimum learning rate; double $\in [0, 1]$
LB4MPI_RL_EPSILON KMP_RL_EPSILON	EPSILON	0.90	Exploration rate for the policy `epsilon-greedy`; double $\epsilon \in [0, 1]$
LB4MPI_RL_EPS_DECAY KMP_RL_EPS_DECAY	EPSILON_DECAY_FACTOR	0.01	Exploration rate decay; double $\in [0, 1]$
LB4MPI_RL_EPS_MIN KMP_RL_EPS_MIN	EPSILON_MIN	0.10	Minimum exploration rate; double $\in [0, 1]$
LB4MPI_RL_TAU KMP_RL_TAU	TAU	1.50	Temperature of the `softmax` policy AND `robustness` reward; double $\tau \in [0, +\infty]$
LB4MPI_RL_INV_MULT KMP_RL_INV_MULT	INVERSE_REWARD_MULT	10	Constant multiplier for the reward `looptime-inverse`; int $\in [0, +\infty]$
LB4MPI_RL_AVG_WINDOW KMP_RL_AVG_WINDOW	ROLLING_AVG_WINDOW	10	Window size int $\in [0, +\infty]$ for the reward `looptime-rolling-average`
LB4MPI_RL_SR_ PORTFOLIO_SIZE	in LB4MPI: srArrayDim	10	Dimension of the StealRatio portfolio $\in [2, 100]$
MPI_RL_OPTION	in LB4MPI: mpi_rl_option	0	The type of the RL Agent String OR int values: `QLearner` (8), `DoubleQLearner` (9), `SARSALearner` (10), `ExpectedSARSALearner` (11), `QVLearner` (12) `ChunkParameterSelector` (15), `StealRatioSelector` (16)
LB4MPI_RL_SELECTOR_TYPE KMP_RL_SELECTOR_TYPE	SELECTOR_TYPE	8	Selector agent type for agents 15 and 16 Agent type as integer $\in [8, 12]$
LB4MPI_RL_POLICY KMP_RL_POLICY	POLICY_TYPE	explore-first	The type of Policy All accepted values: `explore-first`, `epsilon-greedy`, `softmax`
LB4MPI_RL_REWARD KMP_RL_REWARD	REWARD_TYPE	looptime	The type of Reward All accepted values: `looptime`, `looptime-average`, `looptime-rolling-average`, `looptime-inverse`, `loadimbalance`, `robustness`, `stddev`, `skewness`, `kurtosis`, `cov`
LB4MPI_RL_INIT KMP_RL_INIT	INIT_TYPE	zero	The type of Initialization All accepted values: `zero`, `random`, `optimistic`
LB4MPI_RL_DECAY KMP_RL_DECAY	DECAY_TYPE	exponential	The type of Decay All accepted values: `exponential`, `step`
LB4MPI_RL_AGENT_STATS KMP_RL_AGENT_STATS	AGENT_STATS	no output	Agent statistics output filename String values: the filename, e.g. `agent`
LB4MPI_RL_REWARD_NUM KMP_RL_REWARD	REWARD_STRING	0.0,-2.0,-4.0	Good/neutral/bad rewards Triple of comma-separated doubles
-	_RL_DEBUG	0	Inside the `kmp_agent.h` file, modify it to 1 or 2 to output useful debugging information

The following section offers a list of all the modifications suffered by the LB4MPI library or by the RL extension.

4.4 List of Changes

The LB4MPI library has been modified as follows:

- Both the C and f90 library versions can access the automated RL selection features. Our solution aims to be compatible with previous applications that use LB4MPI.
- The library supports multiple synchronous loops. Various performance metrics are dynamically recorded and stored such that in-depth analysis can be undergone.

The original version of the RL Agent software has been modified as follows:

- To facilitate compatibility with LB4MPI, the env var MPI_RL_OPTION provides the agent type, similarly to checking schedule(runtime) in an OpenMP environment.
- To allow compatibility with both the C and f90 versions of LB4MPI, the C++ RL extension should firstly be compiled together with the C++-to-C or the C++-to-f90 interface in .so objects, and just then linked with the main library. How the library is compiled differs from the initial approach (see section 4.5 for details).
- Moved the header file kmp_loopdata.h inside the reinforcement learning folder, from the library root folder.
- The error reporting system shows more explicit instructions when the user provides incorrect data through env vars. Moreover, if env var LB4MPI_RL_AGENT_STATS is not provided, crashing is now avoided by not outputting the statistical data.
- To use all available knowledge while exploiting, the argmax function now compares the averaged Q-table reward for all actions independently of the current state.
- We have updated the following features: the explore-first policy, the SARSA agent type, the Looptime-Inverse reward, and the step decay type.
- Added statistical reward types: stddev, c.o.v., skewness, and kurtosis.
- The functionality of the ChunkParameterLearner meta agent has been updated to set the minChunkSize for any DLS technique. Previously, it could only select the chunk size parameter for dynamic, which is the OpenMP counterpart of SS.
- The StealRatioSelector meta agent has been developed to perform an automated parameter selection to find the optimal StealRatio for a distributed-data setup where Random Work Stealing is allowed.

Information on how to compile the software is provided in the next section.

4.5 Compiling the Library

The instructions to link an application and the LB4MPI with RL library are explained in Algorithm 3. **Important note:** to maintain compatibility with legacy software, LB4MPI does not use the RL features by default, and MPI_RL_OPTION is the internal switch. Hence, the user must specify the agent type, e.g. export MPI_RL_OPTION=QLearner, before customising other optional RL parameters.

The script in Listing 4.1 shows how to compile the RL component and the C++-to-C connector in two .so objects, namely lib_rl.so and lib_connector.so. The Intel compiler version 2022a on the miniHPC-Broadwell cluster is used for compilation. The script is run from the LB4MPI/reinforcement-learning folder.

```
#!/bin/bash
module load intel/2022a

RL_files=$(find . -type f -name "*.cpp")
mpiicpc -lstdc++ -fpic -shared $RL_files -o lib_rl.so
mpiicpc -lstdc++ -fpic -shared c_connector.cpp -L. lib_rl.so
    -o lib_connector.so
```

Listing 4.1: Compiling the RL component and the C-to-C++ connector in .so objects

Paths to both .so objects should be provided when compiling the desired application alongside the LB4MPI.c library. Listing 4.2 displays how to compile a C-based application, Mandelbrot (note: in this example, Mandelbrot has the same parent folder as LB4MPI). Moreover, as the linker is required to load the compiled LB4MPI library, the library path should be added to the LD_LIBRARY_PATH environment variable. Below, the pwd indicates that the compiled library can be found in the current folder.

```
#!/bin/bash
module load intel/2022a
export LD_LIBRARY_PATH=$(pwd):$LD_LIBRARY_PATH

mpiicpc -O3 -lstdc++ -std=c++11 mandel_three_loops.c ../LB4MPI/LB4MPI.c
        -L. ../LB4MPI/reinforcement-learning/lib_rl.so
            ../LB4MPI/reinforcement-learning/lib_connector.so -o mandel
```

Listing 4.2: Compiling Mandelbrot

To compile the RL Agent's C++ code for a Fortran application (e.g. SPHYNX Evrard Collapse), the script in Listing 4.1 should be slightly modified to replace c_connector.cpp with fortran_connector.cpp, and lib_connector.so should be optionally renamed into lib_fortran_connector.so. Then, the DLS.f90 library has to be linked with the two .so objects, and form an object DLS.o, which is used by the main application.

The next section provides summarised information on how to use the library.

4.6 Usage - Summary

Below is a checklist to follow when using the `LB4MPI` with RL features.

- Copy to the source folder for `LB4MPI` the `reinforcement-learning` subfolder.
- On the application's side, modifications are required to transform the sequential program into a parallel one. The pseudocode in section 4.1 details the process.
- Compile the library by following the steps from section 4.5, preferably using an Intel compiler, through using `module load intel/2022a`. Then add the compiled library to the path using `export LD_LIBRARY_PATH=LB4MPI_path`.
- Export the desired environment variables by first consulting the table in section 4.1. Most importantly, set the RL type e.g. `export LB4MPI_RL_OPTION=QLearner`.
- Set the `slurm sbatch` configuration to use at least 2 ranks. Then use `srun` to run the application, eventually providing the `args` values.

Otherwise, the compilation is also possible through executing the `c_compile.sh` or the `fortran_compile.sh` scripts found in the RL folder.

In Figure 4.3, a typical application's workflow is shown. Generally, this workflow is also valid for `Fortran` applications.

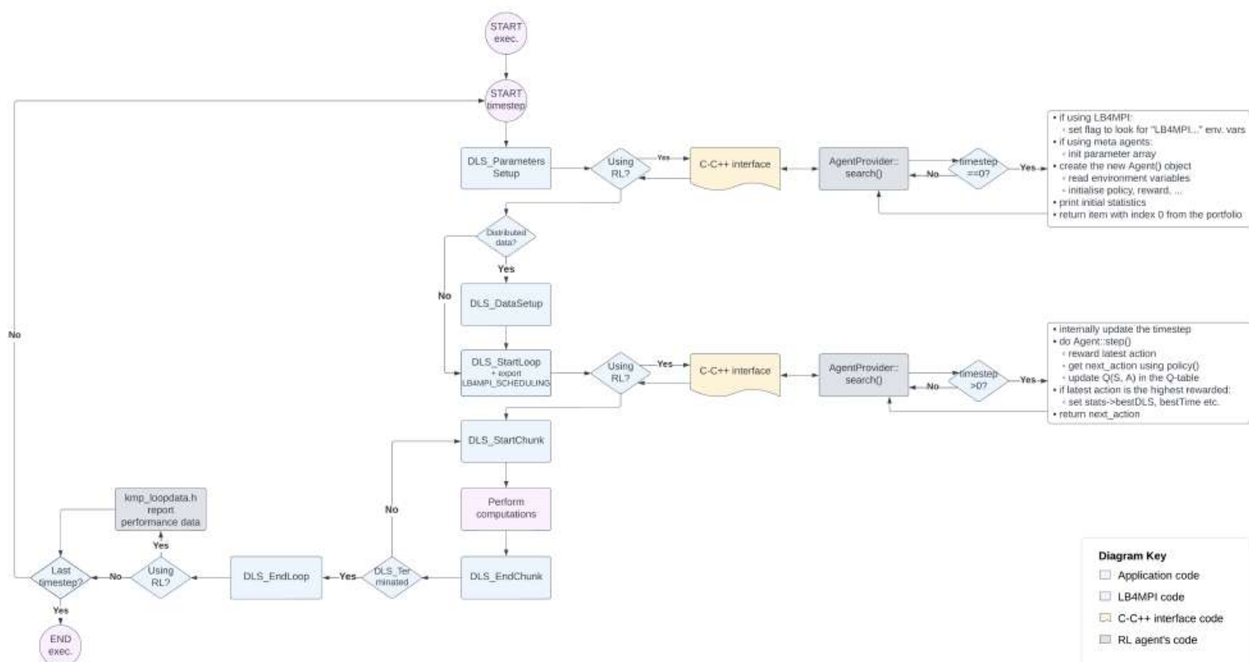


Figure 4.3: Application's workflow when using the C-based LB4MPI with RL features

The full potential of this software is explored through experiments in the following chapter.

5
Experimental Results

In this chapter, we evaluate the performance of the LB4MPI library with automated DLS selection using a Reinforcement Learning agent. A series of factorial experiments are run based on three different applications: PISOLVER, Mandelbrot and SPHYNX Evrard. A discussion based on aggregating all the experimental results is also given.

5.1 The Computing System

All the experiments were run on the high-performance computing cluster located at the University of Basel, miniHPC [13]. This computing system helps to create a controllable research environment, and it serves mainly educational purposes. MiniHPC contains four types of nodes: Intel Xeon E5-2640 v4 (Broadwell, 22 nodes), Intel Xeon Phi KNL 7210 (4 nodes), Intel Xeon Gold 6258 (1 node), and AMD EPYC 7742 (1 node), which are interconnected through an Intel Omni-Path 100 Gbit/s network. The experiments are performed on the miniHPC-Broadwell segment, using 10 nodes of type Intel Xeon E5-2640 v4.

All the applications' code and the LB4MPI library with RL features have been compiled on miniHPC-Broadwell, using the Intel Compiler version 2022a.

5.2 Replicated-data Experiments

Table 5.1 displays the design of experiments for the replicated data component. Each experiment consists in running a parallel application with diverse scheduling techniques active one at a time. The sum of performances for each timestep is used to determine the quality of the RL selector. These medium-to-large-scale experiments use 200 PEs for PISOLVER/Mandelbrot and 4 PEs with 10 threads each for SPHYNX Evrard Collapse. The 1'835 experiments took approximately 750 real hours to run.

The used applications span three different programming languages: PISOLVER is written in C, Mandelbrot in C++ and SPHYNX Evrard Collapse in Fortran90. Also, these parallel applications cover different scientific domains: PISOLVER [10] and Mandelbrot [11] cover mathematics, while SPHYNX Evrard Collapse [12] is an astrophysics simulation.

Table 5.1: Design of 1'835 factorial experiments for performance evaluation of LB4MPI with RL features in replicated-data applications

Factors		Values	Properties
Applications	Process-level parallelism	PISOLVER with 7 workload imbalance levels: + 0% | 5% | 10% | 15% | 20% | 25% | 30%	C++ | N = 500'000 | T = 1'500 | Total loops = 1
	Process-level parallelism	Mandelbrot	C | N = 1'048'576 | T = 1'500 | Total loops = 3 (L1, L2, L3) L1: constant, L2: increasing, L3: decreasing workload imbalance
	Process-level+ Thread-level parallelism	SPHYNX Evrard Collapse	f90 | N = 1'000'000 | T = 250 | Total loops = 37 Major workload imbalance source: the gravitation calculation loop
Scheduling Techniques	LB4MPI	`STATIC`	Straightforward parallelization
		`SS, mFSC, GSS, TSS, FAC2`	**Dynamic** and **non-adaptive** self-scheduling techniques
		`AWF, AWF-B, AWF-C, AWF-D, AWF-E, AF`	**Dynamic** and **adaptive** self-scheduling techniques
RL Agent Type	LB4MPI	QLearner (`QLearn`)	The type of the Reinforcement Learning Agent
		SARSALearner (`SARSA`)	
RL Selection Policies	LB4MPI	Explore-First (`expl-1st`)	Select the RL agent's next action
		Epsilon-Greedy (`eps-greedy`)	
		Softmax (`softmax`)	
RL Reward Metrics	LB4MPI	Loop execution time (`LT`)	Reward the RL agent's actions Reward values (r_+, r_0, r_-): (0.01, -2.0, -4.0)
		Loop execution time: average (`LT-avg`)	
		Percent imbalance of loop execution times (`LIB`)	
		c.o.v. of loop execution times (`cov`)	
RL Scheduling Technique Selector	LB4MPI	RL-based DLS selection	PISOLVER: {STATIC, SS, mFSC, GSS, TSS, FAC2, AWF, AWF-B, AWF-C, AWF-D, AWF-E, AF} Mandelbrot: {STATIC, SS, mFSC, GSS, TSS, FAC2, AF} SPHYNX: {STATIC, SS, mFSC, GSS, FAC2, AF}
RL Chunk Parameter Selector	LB4MPI	`SS` + `ChunkParameterSelection` secondary agent type: `QLearn`	PISOLVER: { 2, 4, 9, 19, 39, 78, 156, 312, 625, 1250} Mandelbrot: {2, 5, 10, 20, 40, 81, 163, 327, 655, 1310, 2621} SPHYNX: {3, 7, 15, 30, 61, 122, 244, 488, 976, 1953, 3906, 7812, 15625, 31250, 62500, 125000}
Size of the Experiments		PISOLVER ≈ 10 minutes per experiment run	Total: 1'470 experiments DLS: 12 (no. DLS) × 7 (levels imbalance) × 5 (rep.) = 420 runs RL DLS configurations: 2 (agents) × 3 (policies) × 4 (rewards) = 24 RL DLS selection: 24 (RL configurations) × 7 × 5) = 840 runs RL Chunk sel. configurations: 3 (policies) × 2 (rewards) = 6 RL Chunk selection: 6 (RL configurations) × 7 × 5 = 210 runs
		Mandelbrot ≈ 20 minutes per experiment run	Total: 185 experiments Plain DLS: 7 (no. DLS) × 5 (repetitions) = 35 runs RL DLS selection: 24 (RL configurations) × 5 (repetitions) = 120 runs RL Chunk selection: 6 (configurations) × 5 (rep.) = 30 runs
		SPHYNX Evrard Collapse ≈ 147 minutes per experiment run	Total: 180 runs 30 (DLS) + 120 (RL + DLS) + 30 (RL + chunk sel.) = 180 runs
Computing nodes		miniHPC-Broadwell	Intel Xeon E5-2640 v4 (Broadwell) nodes, no hyperthreading PISOLVER/Mandelbrot: P = 200; 10 nodes, 20 cores each; SPHYNX Evrard: P=40; 2 nodes each with 2 cores, 10 threads each
Metrics		T_{par}	Parallel execution time per application execution
		T_{par}^{loop}	Parallel loop execution time
		Percent imbalance [27] of loop execution times	Percent imbalance = $(T_{max}^{loop}/T_{mean}^{loop} - 1) \times 100\%$
		c.o.v. [34] of loop execution times	c.o.v. = σ/μ, with σ - standard deviation, and μ - mean loop execution time of all the workers

To ensure the correctness of results, each experiment has been run five times, with the execution achieving the median performance being representative of the configuration.

Kury [7] shows that out of the five different agent types tested (QLearner, DoubleQLearner, SARSALearner, ExpectedSARSALearner, QVLearner), no agent type would outperform the others. Since `QLearner` and `SARSALearner` are popular throughout the literature reviewed [5, 7, 16], we conduct our experiments using only these two agent types. Additionally, Kury shows that the policy type employed is one of the most important factors. Hence we will experiment with all the three policies available: `explore-first`, `epsilon-greedy`, `softmax`. Another impactful component is shown to be the reward type, for which we will experiment with four types: two performance per loop rewards, `looptime`, `looptime-average`, the `load-imbalance percent`, and the newly implemented statistical-based reward, the Sloop execution time `c.o.v`. Therefore, we conduct our DLS selection experiments using 24 agent configurations (2 agent types × 3 policy types × 4 reward types), where one configuration is of the form `agent-policy-reward`.

If it is not otherwise stated, all the experiments use the default parameters presented in Table 4.1. One difference is that the triple of rewards (r_+, r_0, r_-) is set to (0.01, -2.0, -4.0) instead of using the default negative reward. This change would help avoid a corner case where the agent would not distinguish between a high-performing action rewarded with 0 and the initial Q-table value of 0 set by the `zero` initialiser type.

The Oracle (also called *Ground-Truth*) measures a theoretical performance that is only achievable with a selection of DLS techniques. It can be used to facilitate the comparison between real DLS selections. To calculate the Oracle, the application runs five times for each scheduling technique in the portfolio. For each timestep, find the DLS technique that achieves the highest performance when comparing the median execution time of the five runs, and group these together by summing the looptimes.

In the following sections, we evaluate each RL agent's configuration based on the metrics presented in Table 5.1. The Oracle described above is used for an unbiased comparison.

5.2.1 PISOLVER

This C++ time-stepping application simulates a scalable mathematical problem. The initial codebase has been developed by Afzal et al. [10] and modified by HPC Group researchers at Basel Universität. As the name PISOLVER suggests, it calculates the value of π using the mid-point rule to approximate $\int_0^1 \frac{4}{1+x^2}\,dx$ repeatedly over T=1'500 timesteps. The size of π is computation-bounded by N=500'000 iterations. When allocating the workload among workers, a supernormal distribution with a user-provided mean=680'000 and programmable workload imbalance is generated. To calculate the workload imbalance, the application uses a user-provided percentage value and internally calculates the standard deviation. For example, for a mean of 680'000 and 10% imbalance, the standard deviation is 68'000; each worker has to resolve between 612'000 and 748'000 iterations. Therefore, our experiments compare the performance of several RL Agent configurations based on multiple levels of workload imbalance of $\{0\%, 5\%, 10\%, 15\%, 20\%, 25\%, 30\%\}$. The STREAM Triad kernel is not used for this set of experiments; hence the application is compute-bounded and not memory-bounded. A replicated-data approach is used, where the workload is split in a centralised manner, and each process has the same copy of the work array.

This set of experiments verifies how different RL agent configurations react to various levels of load imbalance and how the quality of automated DLS selection is affected. The full results for each level of workload imbalance can be found in the Appendix.

Figure 5.1 summarises the results for the replicated-data PISOLVER set of experiments. The percentage above the bars represents how much performance is lost through using a specific configuration versus Oracle's theoretical best performance. The colours in each bar represent the amount of time a DLS is active for a certain configuration; the detailed per-timestep DLS selection is found in Figure 5.3. Moreover, the last six orange bars in each plot represent the experiments where RL is used to determine the optimal chunk for `SS`; the timestep selection of this parameter is shown in Figure 5.2. The green dotted line represents the average performance for randomly selecting a DLS, the blue dotted line indicates the average performance achieved through automated RL DLS selection, and the

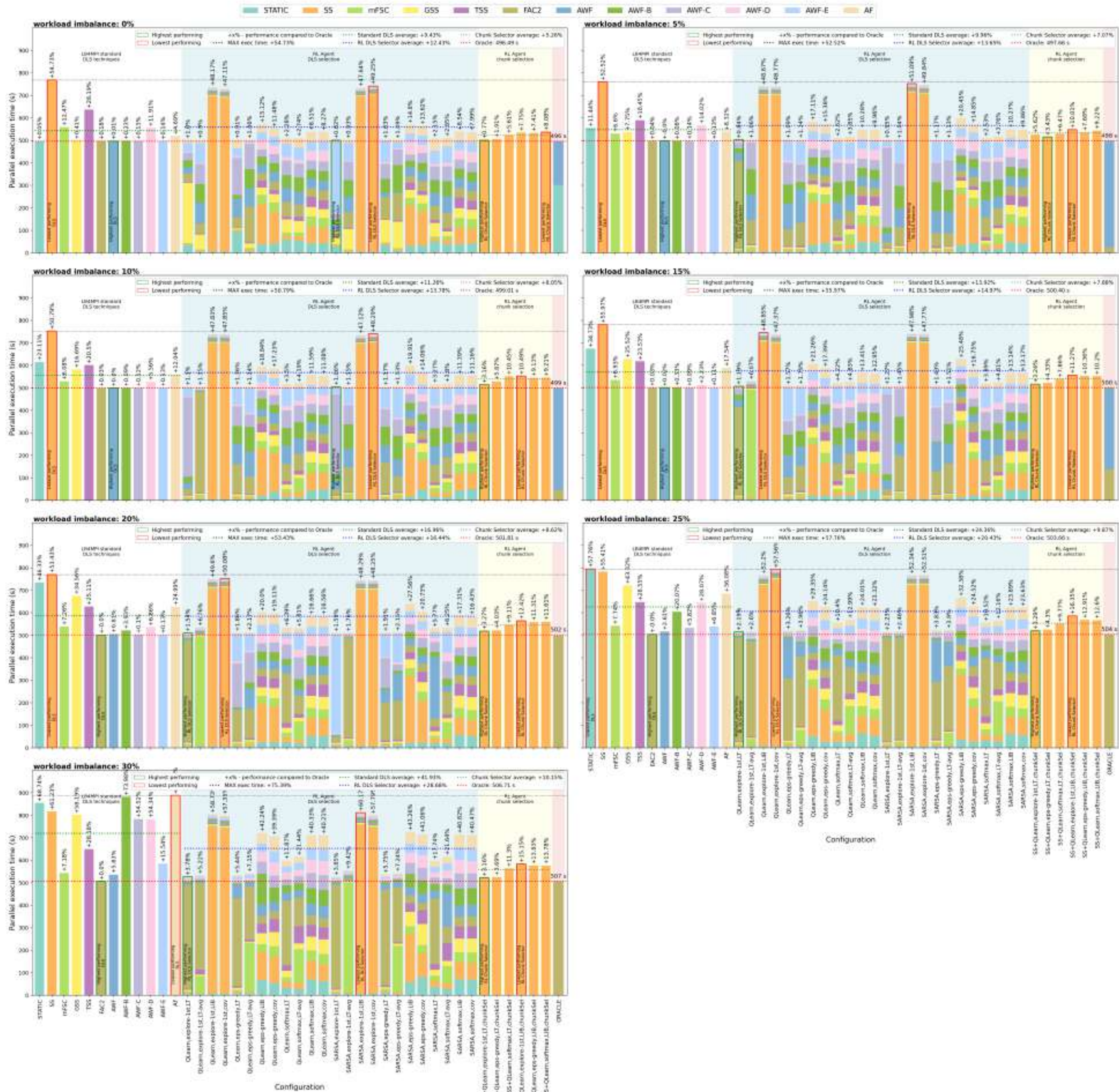


Figure 5.1: Results summary for PISOLVER

grey dotted line shows the average performance for the RL chunk parameter selector. For a level of workload imbalance of over 15%, the RL DLS selector with a random configuration outperforms randomly picking the DLS technique by up to 13%. Under this threshold, the RL DLS selector would be outperformed by around 3%. However, the RL chunk parameter selector would outperform both for all workload imbalance levels by a large margin. This agent achieves under 10% performance degradation versus Oracle in all cases. Generally, all agents that use the `looptime` or `looptime-average` rewards would achieve satisfactory results. Also, the `loadimbalance` and `cov` rewards would generally help achieve worse results than the agent's average execution time. No clear distinction between the `QLearn` and `SARSA` agent types is noticed, but we will further study it when discussing Figure 5.4.

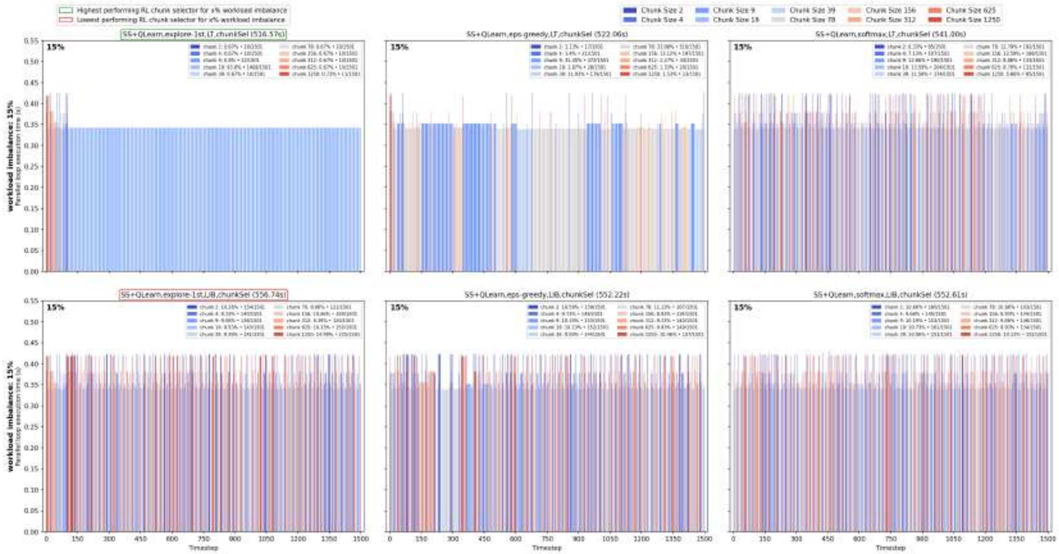


Figure 5.2: Chunk parameter selection per timestep for PISOLVER (15 %)

In Figure 5.2, the shade of each bar represents the value for the chunk parameter that has been selected for a certain timestep, while the height is its looptime, T_{par}^{loop}. While this figure refers to PISOLVER with a 15 % workload imbalance, plots for all setups are in the Appendix. A chunk size between 19 and 156 yields the lowest T_{par}^{loop}, and the first 2 agents achieved quality results, while the other agents seem rather undecided.

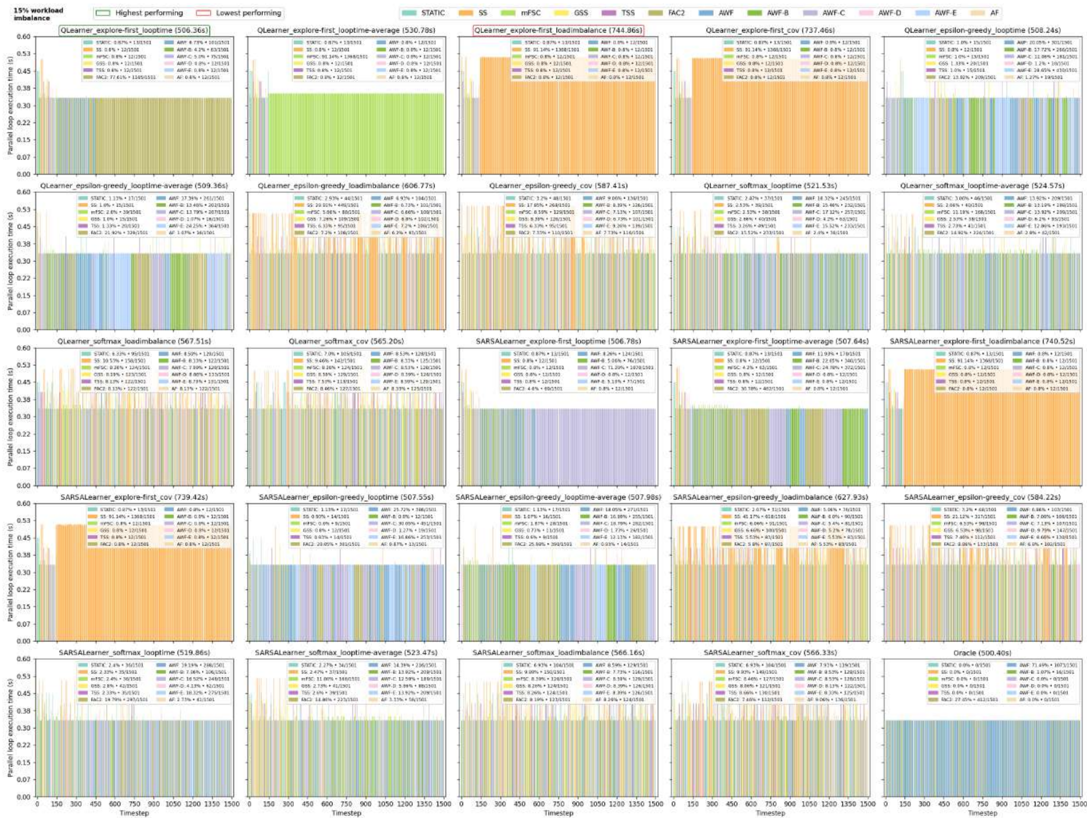


Figure 5.3: DLS selection per timestep for PISOLVER, 15 % workload imbalance

Similarly, in Figure 5.3, the bar's colour represents the DLS technique, while its height is the looptime, T_{par}^{loop}. This type of plot accompanies the previously seen plot 5.1, as each bar is expanded into a per-timestep selection description. Given that we only vary three configuration components, each agent seems to select the DLS in its own unique way. However, some patterns can be observed. An agent using the `explore-first` policy seems to change the DLS technique less often during the exploitation phase. Moreover, this technique excels in picking the DLS technique, which maximises his rewards - for the `loadimbalance` reward, the orange bar representing `SS` is selected in 90% of cases. `SS` with a chunk size of 1 is known always to achieve the highest possible load balancing, sacrificing the performance quality (which is rather obvious in the figure). Moreover, the highest-achieving configuration resembles Oracle's selection the closest; we will study if this is a coincidence or if the similarity level can quantify the selection's quality.

Figure 5.4: RL DLS selector configuration components comparison for PISOLVER

Figure 5.4 facilitates the in-depth analysis of each configuration component that an RL agent can equip: the agent type, the policy type, and the reward type. Each subplot displays data acquired for a level of application-induced workload imbalance. The green bar represents the performance of selecting a DLS technique randomly, with no prior knowledge. Also, the red

bar is Oracle, which represents the theoretical best performance achievable. The bottom-right plot is special since all data points in the previous subplots are aggregated in one figure. The olive-coloured lines above the bars show the standard deviation. When deciding upon which agent type to use, the `QLearn` and `SARSA` agents always have bars of the same height. Moreover, the performance data points in the aggregated plot are spread very similarly. Hence, no clear advantage is noticeable in picking one agent type over another. When analysing the results for the `explore-first` policy, such an agent is considered the lowest achiever in all workload imbalance cases. Nevertheless, the results aggregation subplot depicts a different story, as there are basically two groups of data points: the group situated at the bottom of the boxplot use an `looptime`-based reward, while the one above is based on an execution time imbalance metric. We will further study the very interesting `explore-first` policy type throughout the Mandelbrot and SPHYNX Evrard Collapse experiments. When analysing the `softmax` policy, a small level of standard deviation is characteristic; it can be noticed in the aggregated plot that the data points seem more condensed than for the `epsilon-greedy` policy. From this experiment set alone, we cannot clearly state which of the two policy types is more promising, and further analysis is required. By studying the yellow portion corresponding to the reward type comparison, two groups are obvious. The `looptime` and `looptime-average` reward-based agent is almost guaranteed to achieve better results than a `loadimbalance` or `cov` rewarded agent, purely because the performance degrading `SS` technique would be selected less often.

Generally speaking, the agent reacts to changes in the application's workload imbalance by selecting a more appropriate DLS technique. Not fixing the DLS technique for the whole application's execution is itself a performance-boosting strategy. However, for PISOLVER, the imbalance level is constant throughout the whole run; the Mandelbrot experiment set is designed to study the RL agent's reaction to unpredictable environmental changes.

5.2.2 Mandelbrot

This real-world, time-stepping, C-written application comes from the mathematical scientific domain, and it is highly computationally demanding, with a high amount of load imbalance being created. The Mandelbrot set [11] is computed repeatedly. The original Mandelbrot set contains the complex numbers z for which $f_c(z) = z^2 + c$, while the version in use modifies the condition to $f_c(z) = z^4 + c$, to increase the number of computations per task. The image has $1024 \times 1024 =$ 1'048'576 pixels that can be computed in parallel; hence 2^{20} parallel loop iterations are needed to build the Mandelbrot image. For each pixel, the maximum number of iterations is 10'000. This set is computed three times during each of the 1500 timesteps. The three loops running in synchronicity are loop L_1 with constant load imbalance over time, loop L_2 with increasing load imbalance, and loop L_3 with decreasing load imbalance. For this compute-bound experiment, the data is replicated to all the MPI ranks, and the workload is divided following a centralised-data assignment approach.

This set of experiments implies working with synchronous loops. The setup has required significant code modifications, such that each loop has its own RL agent and its own DLS technique active, with a unique set of performance data for each agent.

ATTENTION: The AWF and variants scheduling techniques exhibit unpredictable behaviour when scheduling the workload of Mandelbrot (random looptime spikes of up to 10× happen), leading to unforeseeable deadlocks. This problem is linked to how the AWF techniques work, as information about previous loops is used to determine the next chunk size. As no problem was encountered in PISOLVER, we can link this situation to running multiple loops synchronously and not isolating the internally-used variables correctly. We, therefore, remove these techniques for this set of experiments, downsizing the RL agent's portfolio to 7 scheduling techniques: STATIC, SS, mFSC, GSS, TSS, FAC2, and AF.

Figure 5.5: Results summary for Mandelbrot

The top subplot in Figure 5.5 depicts the results for Loop 0, which ensures a constant level of workload imbalance throughout the whole execution. The highest performance is achieved using `FAC2`, which entirely shapes the Oracle. The highest achieving RL DLS selector, `SARSA`, `explore-first`, `looptime`, preponderantly selects `FAC2`, reproducing the Oracle's selection in 96% of steps, according to Figure 5.8. Moreover, RL agents selecting mainly `SS` with a chunk size of 1 perform poorly. Just like the DLS selector, the highest achieving RL chunk selector is also based on the `explore-first` policy and the `looptime` reward. Likewise, poorly-performing agents use the `loadimbalance` as the reward.

The second subplot in Figure 5.5 shows the results for Loop 1, where the level of workload imbalance increases over time. The `SS,1` DLS technique degrades the Oracle 23 times. As a result, RL agents selecting this DLS often would also suffer from poor performance.

The results for Loop 2, with a decreasing level of workload imbalance, are shown in the third subplot in Figure 5.5. Overall, the results for each configuration for Loop 2 match the overall execution time results for Loop 1. Therefore, the order in which the load imbalance is inflicted is not as important as the fact that this level is not constant.

Figure 5.5 offers an ensemble look towards the three synchronous loops. It can be observed that the bar for `SS` is divided into three equal parts, meaning that it is guaranteed this technique would achieve identical performance results independently of the workload imbalance percentage. On average, picking a random RL DLS selector would yield a performance of 54% over the Oracle. Dissimilarly to PISOLVER's case, an RL DLS selector would perform 2.2% better than the RL chunk selector, which achieves 57% performance compared to the theoretical best. Due to `SS` and `AF` performing rather poorly, manually picking a DLS without expert knowledge is not indicated, as the average performance is 77% over Oracle. Once again, the RL agents achieving the highest performance use the `looptime` or `looptime-average` rewards. In contrast, the `explore-first` policy is equipped by agents achieving both top-quality and low-grade results.

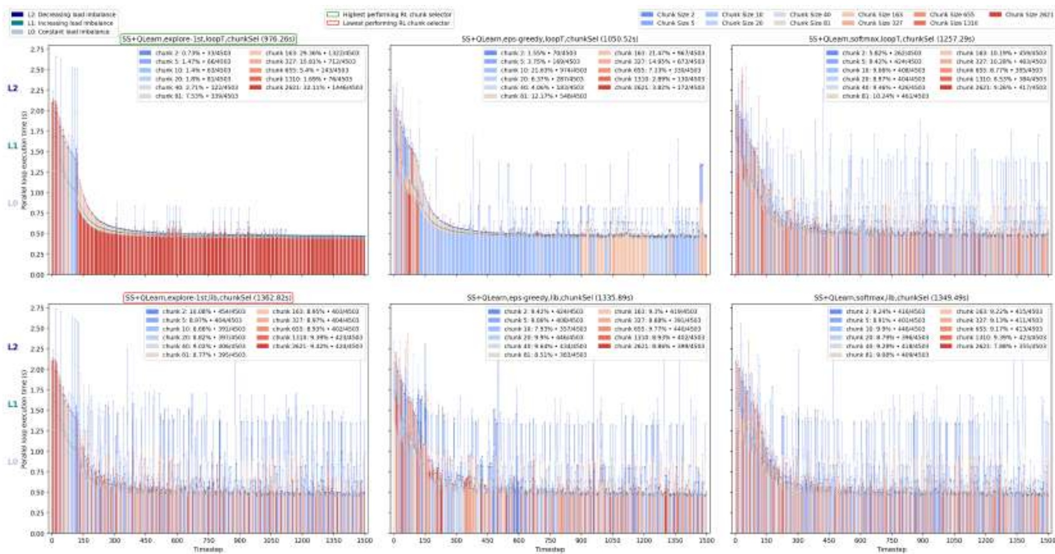


Figure 5.6: Chunk parameter selection per timestep for Mandelbrot

Figure 5.6 indicates the RL agent's `SS`' chunk selection during all three synchronous loops. The three loops are visually separated through coloured dashes on top of each bar segment. It is noticeable that high (red) values for the chunk size, e.g. 2621, would be the optimal choice for Loop 0, as the loop balancing is achieved regardless of dense or sparse communication. For loops 1 or 2, the optimal chunk size has a pink shade, as balancing the workload requires smaller chunk sizes and more process-wise synchronization. This value is not constant, as the imbalance level keeps adjusting. Only two RL agents achieved satisfactory results, the `explore-first` and `epsilon-greedy` agents based on the `looptime` reward. The other agents seem rather undecided, selecting each action around 9% of the time - this behaviour might be corrected by context-aware tuning the learning parameters, α and γ.

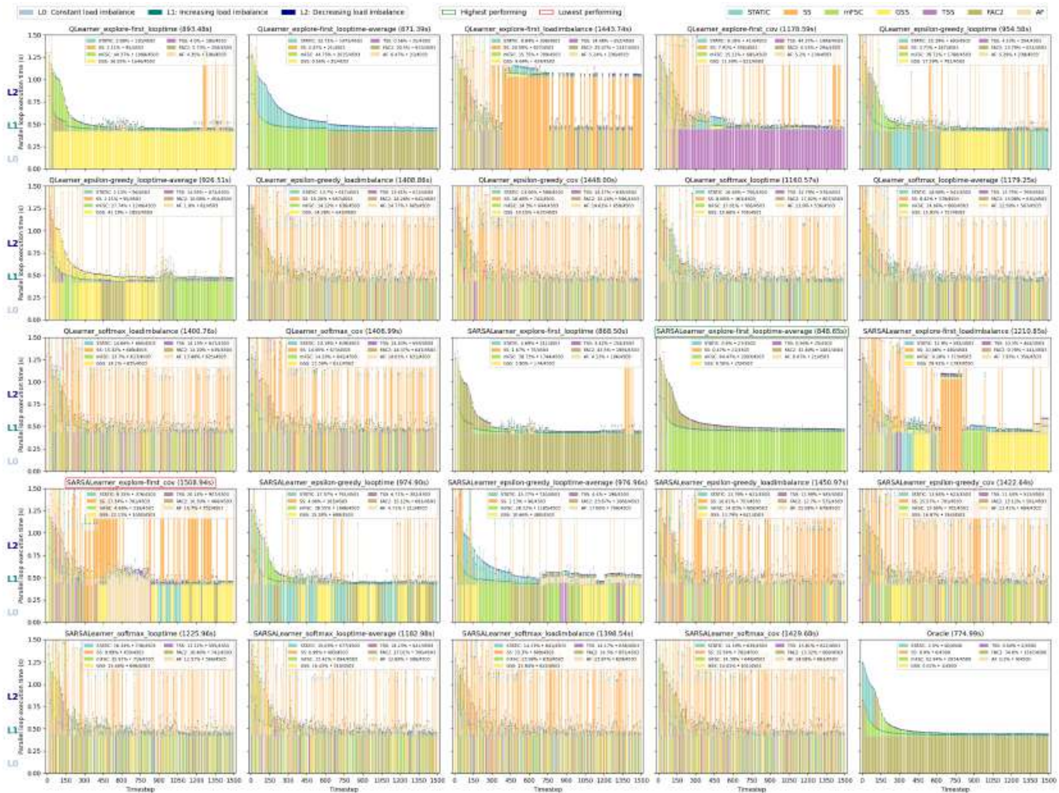


Figure 5.7: DLS selection per timestep for Mandelbrot

Figure 5.7 displays the DLS selection per timestep for the three synchronous loops of Mandelbrot, each loop being depicted one on top of the other. The total timestep performance is reported through the Y-axis, while the X-axis shows the timestep, and the colour of the bar hints at which DLS technique is selected. It should be mentioned that each loop is assigned a newly-instanced RL agent, each configured as in the subplot's title. The top four highest achieving agents are, as also seen in Figure 5.5, the `explore-first` agents using the `looptime` or `looptime-average` rewards. The lowest-achieving agents use, again, the `loadimbalance` or `cov` rewards. These agents seem rather undecided on what DLS to select, as the legend of each subplot shows similar selection rates of around 14%. This behaviour was also encountered during Kury [7]'s Master's thesis. A context-aware hyper-parameter tuning of RL variables τ, α, ϵ, and γ might benefit the future work.

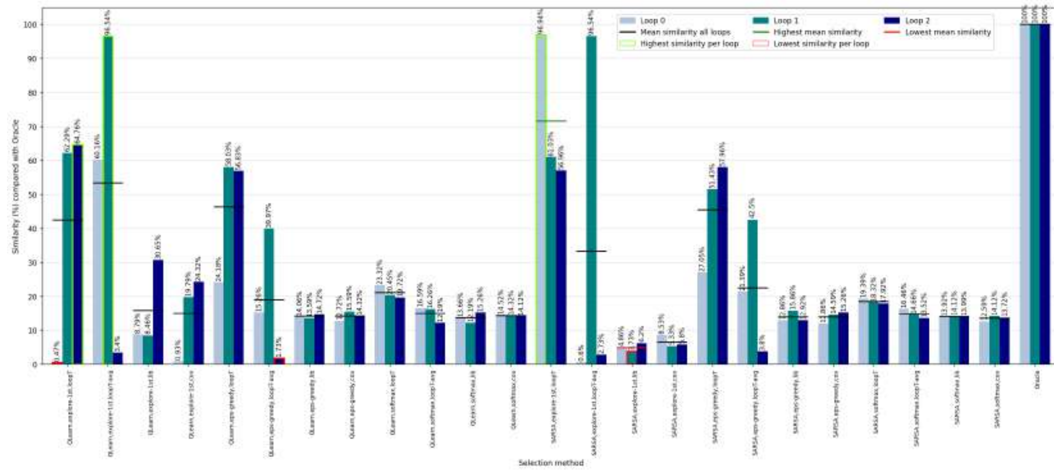


Figure 5.8: Similarity of DLS selection compared with Oracle's for Mandelbrot

Figure 5.8 displays the percentage of similarity when comparing the selection sequence of a specific RL DLS selector and the Oracle for each loop. Through analysing the highest performing configuration in 5.7, `SARSA,explore-first,looptime-average`'s selection diverges from the Oracle's selection in 2 of 3 loop cases, scoring 33% overall similarity. At the same time, the second highest achiever, `SARSA,explore-first,looptime`, is similar to the Oracle in 72% of timesteps. On the first hand, these cases underline the idea that the Oracle remains a theoretical selection, being rather unobtainable in reality without expert knowledge. On the second hand, it is not necessary that a selection replicates the Oracle's to be considered of quality - this is due to multiple DLS techniques achieving similar execution times. Still, the lowest achieving configuration, `SARSA,explore-first,cov`, is the most dissimilar to Oracle's selection, with only 5% similarity. Up to a point, a correlation between the selection similarity to the Oracle and its quality might exist.

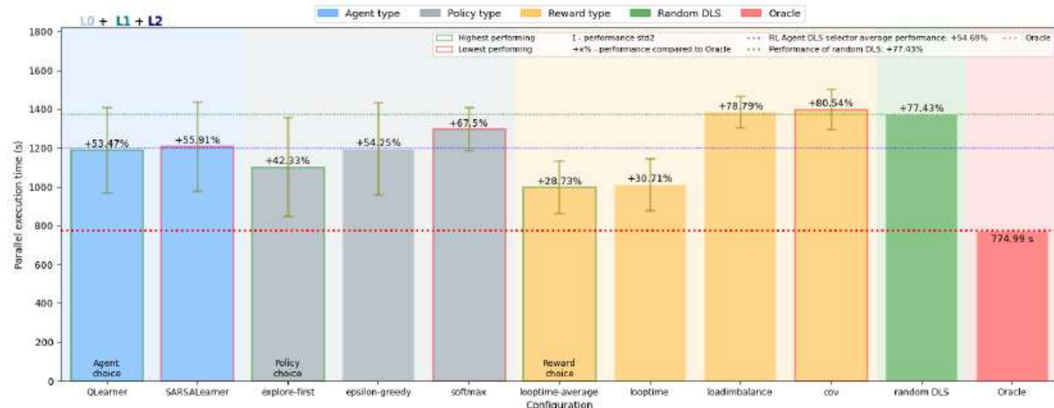


Figure 5.9: RL DLS selector configuration components comparison for Mandelbrot

Figure 5.9 aggregates data for the RL agent's main configuration components - the agent, policy, and reward types. Each bar represents the averaged performance per application run of all agents employing a specific component type. The olive-coloured bars show the performance's standard deviation. From the first two bars representing the agent type comparison, both `QLearn` and `SARSALearn` are shown to achieve very similar results,

approaching the blue dotted line signalling the RL agent's average performance. This result is also prevalent in the literature [5, 7, 16]. Moving to the policy type analysis, it is noticeable that the `epsilon-greedy` policy has achieved results close to the overall RL agent's average performance line. Furthermore, the standard deviation of `explore-first` is the highest among all the configuration types, which signals that its quality largely depends on the reward type employed. Two reward groups are formed: the looptime-based and the imbalance-based. An agent equipping any `looptime` reward is guaranteed to achieve a better performance since the std lines of the two groups do not overlap. Furthermore, it can be stated that a randomly configured DLS selector RL agent would be, on average, 200 seconds faster than when randomly picking and fixing the DLS.

Overall, the Mandelbrot set of experiments shows that the RL agent reacts correctly to workload imbalance changes by selecting a more appropriate technique. However, when this level fluctuates excessively, the agents would be destabilised and seemingly undecided, with most DLS techniques being chosen for an equal amount of timesteps. Through the next set of experiments, using the Fortran-based SPHYNX Evrard Collapse scientific application, we further test the agent's response to dynamic changes in the workload imbalance level. Furthermore, we facilitate multi-level scheduling at both process and thread levels.

5.2.3 SPHYNX Evrard Collapse

This Fortran F90 scientific application from the astrophysics domain simulates an Evrard collapse [12]. The version in use starts from timestep 1 and ends after T=250 timesteps, and the number of particles (thus iterations per timestep) is N=1'000'000. While 37 synchronous loops are running during each timestep, the gravity loop is observed to be the main source of load imbalance. Throughout this experiment set, we ignore the existence of synchronous loops and treat the application's execution as a whole. Initially, the level of workload imbalance is non-existent, as the interactions among the particles are rarer. Nevertheless, it raises gradually. This experiment set explores the two-level workload scheduling at both the process level using `LB4MPI` and at the thread level by using `OpenMP`.

ATTENTION: A bug has been detected where parameters used in the `update` subroutine are sometimes valued as NAN. This behaviour occurs for most hardware setups, e.g. for 10 nodes and 20 MPI ranks per node, the program would crash irreversibly at step 48. This behaviour is encountered independently of the `LB4MPI` library with RL being used or not. The bug persists when the code is compiled using either `intel` or `GNU`. When re-running a version of the software known to be working in 2019, the NAN error persists. A possible cause might be the incompatibilities between the application and the current configuration of the underlying HPC system. As an effect, we are only able to run the application up to 250 timesteps, by using the setup described below.

When setting the experimental setup, we follow the guidance of Mohammed et al. [35], such that each node executes two MPI ranks, one per socket, with 10 OpenMP threads within each MPI rank. In total, for each experiment, we use two nodes of the miniHPC. As our aim is to study the node-level scheduling quality, we set `OMP_SCHEDULE="static"`. Once again, in this compute-bound experiment, the data is replicated to each MPI rank.

Based on the literature reviewed, an RL agent would optimally explore the action space 10% to 20% of the time [28]. Due to running the application for only 250 steps, the portfolio of DLS techniques has to be downsized to the most representative six DLS techniques: STATIC, SS, mFSC, GSS, FAC2, and AF. The main implication is that the exploration phase of the RL DLS selector agent would take only 36 steps, as opposed to the 144 steps required when using the 12-DLS portfolio. Hence, this kind of agent would explore the action space for 15% of the time and not for the unreasonable 57% of the time.

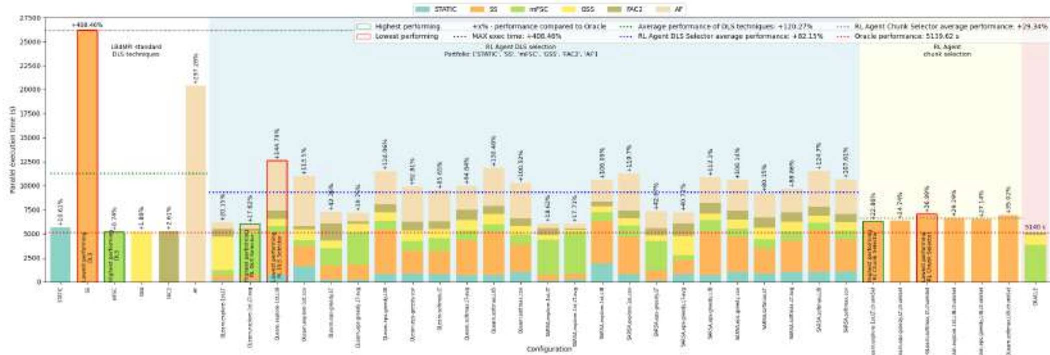


Figure 5.10: Results summary for SPHYNX Evrard Collapse

In Figure 5.10, it can be observed that the DLS techniques `mFSC`, `GSS` and `FAC2` would achieve the highest performance, within 2% of one another. These techniques are also selected throughout the Oracle. On the other hand, `SS` with a chunk size of 1 achieves the worst execution time, losing 400% performance compared to the Oracle, followed by `AF`, with 300% performance degradation. As for the RL agent's DLS selection, results similar to the previous two applications are observed. In all cases, a `looptime` (`-average`) reward achieved results of higher quality. The `explore-first` is equipped by both the highest-achievers and the lowest-achievers groups. Similarly, the agent being of type `QLearn` or `SARSA` does not seem to influence the outcome as much as the policy type and especially the reward type do. On average, an RL DLS selector would yield results 40% closer to the theoretical best than randomly selecting the DLS. For this application, the randomly-configured RL chunk parameter selector would outperform the RL DLS selector. In this case, the agent's configuration seems less important than in the DLS selection case, and good performance is achieved in all cases. Perhaps this effect is caused by having a smaller number of MPI ranks to synchronise, thus carrying smaller inter-process communication costs when compared to the overall execution times.

In Figure 5.11, the automated selection of the chunk parameters for the `SS` scheduling technique is illustrated. Due to the large number of iterations of 1 million and the small number of MPI ranks of 4, each process would have to handle 250'000 iterations per timestep. Based on the equation 4.1, the size of the chunks portfolio is 16, which implies an exploration phase for the `explore-first` agent of 256 timesteps, which exceeds the 250 steps of the application. Under the assumption that the value of the chunk parameter for one step does not influence the performance of a future step like DLS techniques do, we only explore each action once instead of 16 times, thus reducing the size of the exploration phase. It can be

observed that chunks with a size under 488 are associated with poorer performance. The sweet spot for the chunk size is between 976 and 7'812, which are depicted using a pink shade. Due to varying workload imbalances, the RL agents exhibit problems in maintaining the selected parameter for more than 66 timesteps. The performance is rather influenced by the number of times small chunk sizes are tried.

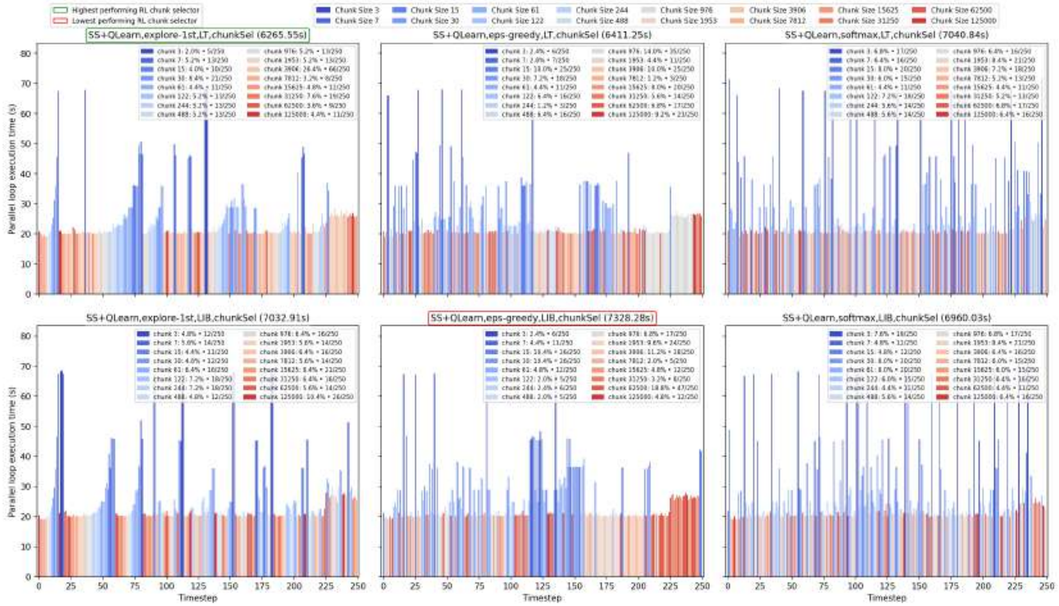


Figure 5.11: Chunk parameter selection per timestep for SPHYNX Evrard Collapse

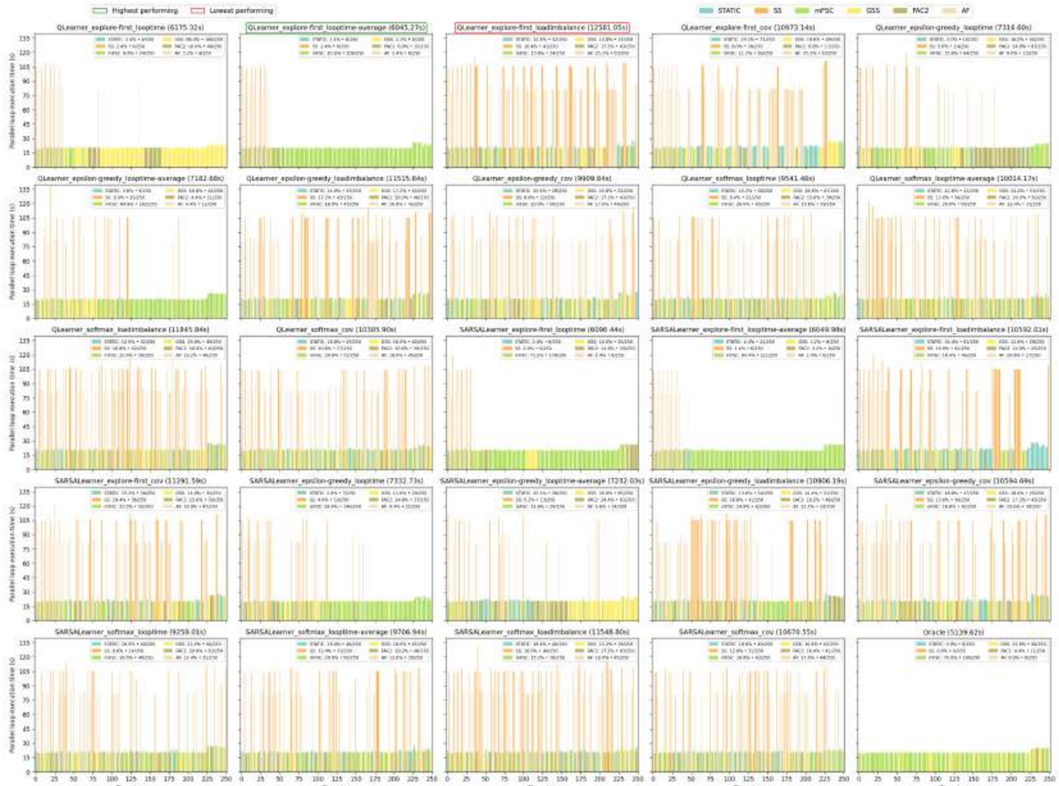


Figure 5.12: DLS selection per timestep for SPHYNX Evrard Collapse

As illustrated in Figure 5.12, a quality, top-performing selection is one where the SS, AF and even STATIC DLS techniques are selected for fewer timesteps. It can be noticed that a short exploration phase is beneficial for all applications employing the looptime reward together with the explore-first policy, as the low-performing techniques are now rarely encountered during the exploitation phase. This raises the question - Would exploring each action exactly once, such as in the above chunk parameter selection, result in a selection quality boost? Our belief is that the data gathered would not be sufficient to exclude lower-achiever techniques without also eliminating true negatives. The level of non-decidability would also be raised, just as observed in Figure 5.11. Instead, a quality improvement in the automated selection is observed when downsizing the DLS portfolio from 12 to 6 DLS techniques. It is also worth mentioning that the optimal DLS selection problem is harder for the SPHYNX Evrard Collapse setup, as the amount of load imbalance is not linear for the whole execution. Therefore, the performance of a DLS can improve or deteriorate over time. For example, FAC2 is chosen by the Oracle more often after timestep 225.

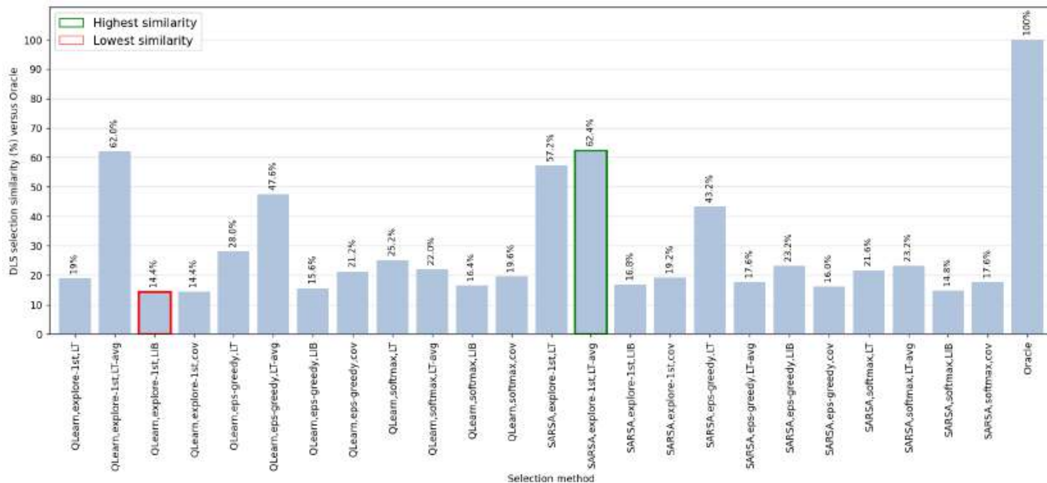


Figure 5.13: Similarity of RL DLS selection versus Oracle's for SPHYNX Evrard

Through analysing the results in figures 5.13 and 5.12, a correlation between the quality of the DLS selection and its similarity to Oracle's can be observed. However, there are exceptions. For example, the QLearn,explore-first,looptime configuration would only resemble the Oracle in 19% of cases, the 14th highest, would achieve the 4th highest performance. This observation situates comparing the selection similarity with Oracle's as insufficient in determining its quality by itself, while for most cases, it is a good indicator.

In Figure 5.14, performance results for different configuration components of the RL DLS selector are illustrated. Each bar represents the average performance of all agents employing this type. The olive-coloured bars show the performance's standard deviation. These measurements are in line with the results of PISOLVER and Mandelbrot. The agent type does not influence the selection quality in the long run. Agents operating in a epsilon-greedy manner achieve a performance level closer to the Oracle and under the all-average line, dissimilar to Mandelbrot's case. Judging by the standard deviation of explore-first, this policy helps achieve results closer to the two extreme points - the lowest and highest execu-

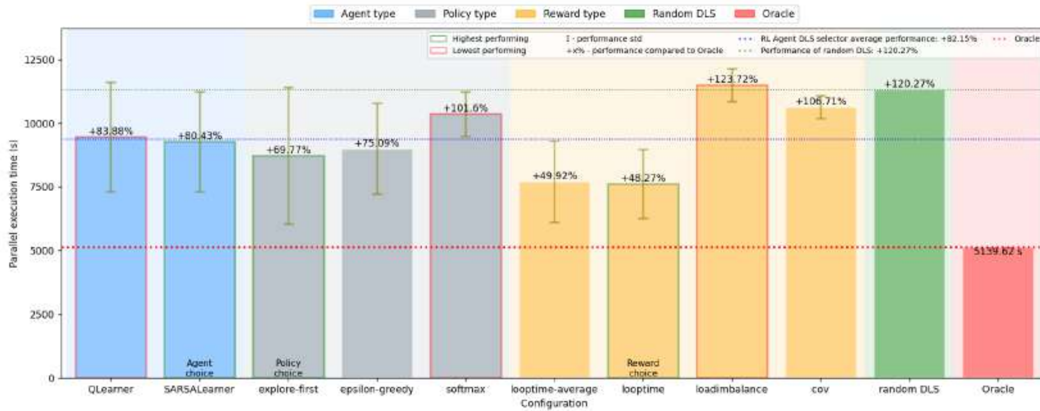


Figure 5.14: RL DLS selector configuration components comparison for SPHYNX Evrard

tion times. Meanwhile, the values for the `softmax` policy vary less, but an agent using this policy would yield results worse than the RL agent's mean performance. As for the reward types, selection based on `loadimbalance` or `cov` is considerably worse than the agent's mean performance. On average, choosing a randomly-configured RL-based automated DLS selector is better than choosing a random DLS manually.

Therefore, studying the SPHYNX Evrard Collapse application has revealed the potential benefits of shrinking the size of the DLS portfolio, as opposed to reducing the exploration phase to one timestep per action. Within this experiment set, we focused on the process-level scheduling technique selection by fixing the thread-level technique to `STATIC`. We believe that extending Mohammed et al. [35]'s study from a manual algorithm selection to an automated one would be beneficial for advancing state of the art.

The results achieved throughout studying the replicated data setup are consistent. We now shift the focus to distributed data and how RL can help improve the workload imbalance.

5.3 Distributed-data Experiments

For reasons that include the scalability of modern HPC systems, running an application using a distributed-data approach is viable. Among the benefits, the memory needed for each process is significantly lowered. Moreover, in a centralised data allocation approach, one rank responsible for dividing the workload among all workers could easily become overwhelmed with the task and become a performance-wise bottleneck.

At the application's start, in the distributed setup that we study, the work array is divided statically and workers would receive equal fragments of it. As we have previously discovered, a static workload distribution would introduce an unknown level of workload imbalance. This might be caused by some iterations being more demanding or the impact of uncontrollable external system factors on the workers' performance. One way to overcome this problem is through Random Work Stealing (RWS). This load balancing method has been implemented in `LB4MPI` by Wetten [9]'s work. RWS implies one rank acting as a coordinator, which is queried by workers with no iterations left to finish, called the *thief*. A potential *victim* is picked randomly such that it has not been a thief before, to avoid ping-pong style stealing,

and it is not the coordinator, to avoid potential deadlocks. The victim either sends some iterations based on the `StealRatio` parameter or a rejection message, if no iterations are left. The steal ratio $\in [1, 100]$ sets the percentage of remaining iterations to be stolen.

Table 5.2: Design of 210 factorial experiments for evaluating the StealRatio selection 33 using RL features in a distributed-data setup

Factors		**Values**	**Properties**
Applications	**Process-level parallelism**	PISOLVER in both replicated and distributed data versions with 7 workload imbalance levels: + 0% | 5% | 10% | 15% | 20% | 25% | 30%	C++ | N = 500'000 | T = 1'500 | Total loops = 1
Scheduling Techniques	**LB4MPI**	`STATIC` (for replicated data)	Straightforward parallelization
RL Agent Type	**LB4MPI**	`StealRatioSelector` (`SRS`), `QLearn`-based	The meta type of the Reinforcement Learning Agent StealRatioSelector internally uses a QLearn agent
RL Selection Policies	**LB4MPI**	Explore-First (`expl-1st`)	Select the RL agent's next action
		Epsilon-Greedy (`eps-greedy`)	
		Softmax (`softmax`)	
RL Reward Metrics	**LB4MPI**	Loop execution time (`LT`)	Reward the RL agent's actions
		Percent imbalance of loop execution times (`LIB`)	Reward values (r_+, r_0, r_-): (0.01, -2.0, -4.0)
RL-based StealRatio Parameter Selection	**LB4MPI**	`RWS` RL-based StealRatio selection (for statically distributed data)	Randomised work stealing with RL StealRatio selection for distributed data applications; StealRatio portfolio sized 10: PISOLVER: {10, 20, 30, 40, 50, 60, 70, 80, 90, 100}
Size of the Experiments		PISOLVER $\approx$ 10 minutes per experiment run	Total: 210 experiments Agent configurations: 3 (policy types) $\times$ 2 (reward types) = 6 RL StealRatio parameter selection: 6 (RL configurations) $\times$ 7 (levels workload imbalance)$\times$ 5 (repetitions) = 210 runs
Computing nodes		miniHPC-Broadwell	Intel Xeon E5-2640 v4 (Broadwell) nodes, no hyperthreading P = 200; 10 nodes, 20 cores each
Metrics		T_{par}	Parallel execution time per application execution
		T_{par}^{loop}	Parallel loop execution time
		c.o.v. [34] of loop execution times	c.o.v. $= \sigma/\mu$, with σ - standard deviation, and μ - mean loop execution time of all the workers
		N_{steals}	Number of successful steal attempts

For this set of experiments, we study the potential benefits of using the RL features to automatically select the StealRatio parameter. Results for the replicated-data experiments are reused from section 5.2. Again, we repeat each experiment 5 times and consider the execution achieving the median performance as representative.

By analysing the replicated-data experiments, the `QLearn` and `SARSA` RL agents achieve very similar results. Hence, we equip our `StealRatioSelector` (`SRS`) meta agent with only the `QLearn` secondary agent type. Moreover, when analysing the results for the reward types, 2 groups were noticed: the pure looptime-based (`LT` and `LT-avg`) and the workers' execution time imbalance-based (`LIB` and `cov`). We further use 2 reward types: `looptime LT` and `loadimbalance LIB`. We further consider all three policy types, as `explore-first`, `epsilon-greedy`, and `softmax` can outperform each other given the right context. Therefore, we study 6 RL configurations of type `policy, reward`.

5.3.1 PISOLVER (Distributed Version)

The PISOLVER application based on data replication has been modified into a distributed version. Following a static work distribution, only relevant data is copied to each processor's internal memory. The memory saved through not replicating unnecessary data is noteworthy. As the number of MPI ranks is 200, each rank's work array is downsized 200 times, from 500'000 `long` data-type entries to 2'500 and from 4 MB to 20 KB per worker. In total, for all MPI ranks, the initial memory allocated is reduced by 99.5%, from 800 MB to 4 MP. However, to facilitate RWS, additional memory is needed.

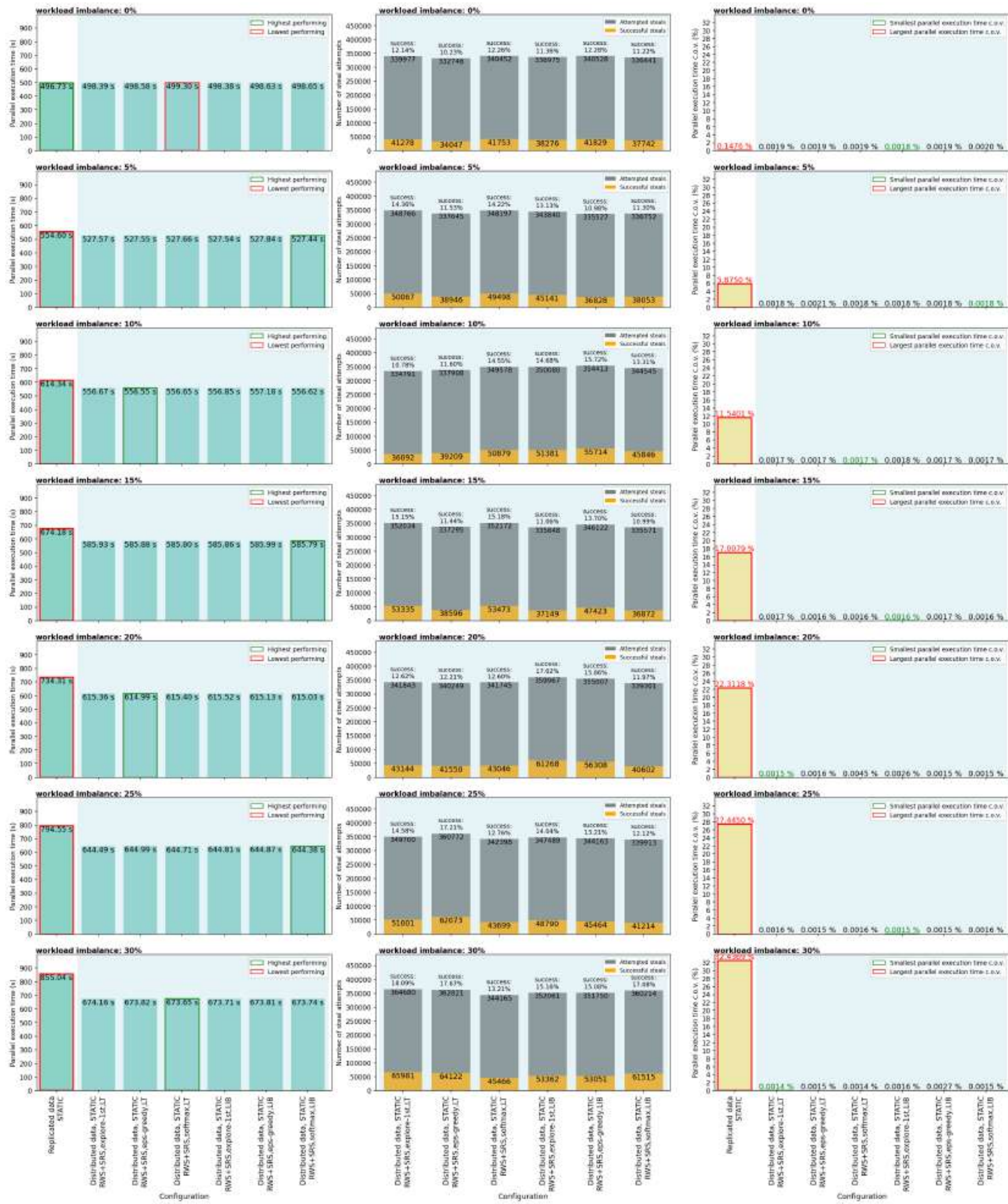


Figure 5.15: Results summary for the distributed version of PISOLVER

Each row in Figure 5.15 displays the results for a certain level of workload imbalance. Subplot column 1 depicts the parallel execution times for each configuration on the x-axis. In essence, we compare the results of the RL agent's StealRatio parameter selection in a distributed RWS context when compared to the static workload scheduling in the replicated version of PISOLVER. We explore the benefits of balancing the workload using random work stealing. In column 2, the steal attempts are shown, with the yellow portion representing the successful steals. Column 3 illustrates the c.o.v. of the processes' parallel execution times. For a workload imbalance of 0%, the replicated-static division achieves slightly better results. This roots in many time-consuming overhead-inducing work-stealing operations completed, all without gaining a performance improvement. When the imbal-

ance level becomes significant, the tradeoff between the stealing overhead and performance improvement becomes obviously beneficial. At the peak of 30% workload imbalance, the performance improvement is 21.2%. Also, it is interesting to note that in the 3rd column, the parallel execution time c.o.v. of the first bar raises proportionally to the workload imbalance, while it remains at a low level for the RWS case, which denotes that all RL agents selections achieved high-quality load balancing. Naturally, the number of steal attempts and the success rates are proportional to the level of imbalance. Overall, the RL agent's configuration seems irrelevant, as the execution times for all the configurations being verified remain relatively constant.

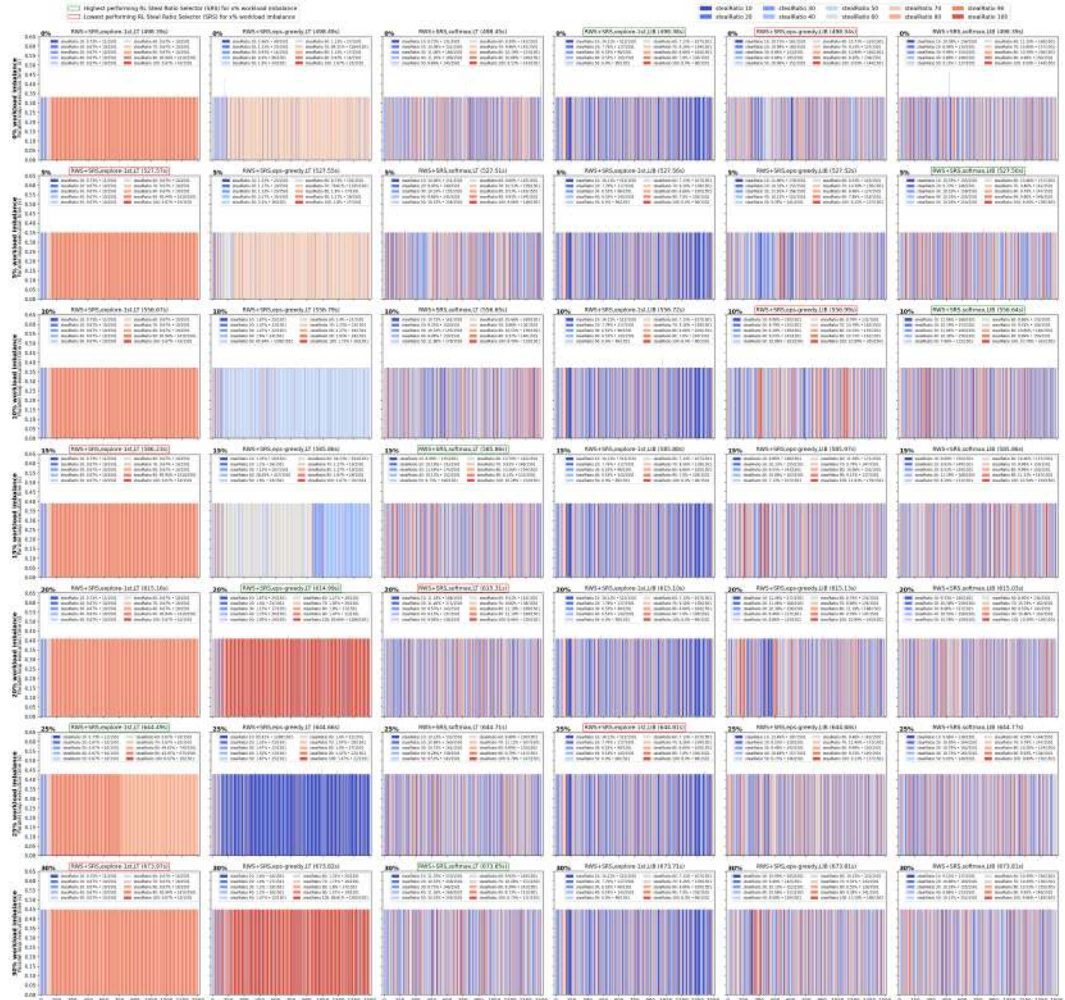


Figure 5.16: StealRatio selection per timestep for distributed PISOLVER

Figure 5.16 displays the selected StealRatio parameter per timestep for all the RL configurations. Each row contains information regarding a certain level of workload imbalance introduced at the application's level. Each column refers to an RL agent configured in a certain way. The bar shaded from blue to red depicts the parallel loop execution time for a certain StealRatio. The portfolio size is user provided as 10, and the colour-StealRatio value mapping is shown in the top-right corner. When compared to the DLS or chunk pa-

rameter selection per timestep from the previous experiments, one detail becomes obvious. The looptimes for all timesteps and for all StealRatio values achieve indistinguishable performance levels. This observation reveals some interesting facts about the selection process itself. During the exploration phase, the `explore-first,looptime` agent in column 1 tends to select one item from its portfolio and stick with it, as shifting happens infrequently. Meanwhile, the `epsilon-greedy,looptime`'s subplots from column 2 are shown more colourful, as shifting is often considered, but falling back to the highest-rewarded parameter prevails. However, other agents seem relatively undecidable, and the selection looks rather randomised, as each technique is picked in 10% of cases. Eventually, these agents might converge on a decision when running the application beyond the 1500 timesteps. Alternatively, the internally-used parameters α, γ, ϵ and τ need to be tuned in a context-aware manner.

Consequently, the experiments above show that the value of the StealRatio is rather irrelevant performance-wise, and simply using RWS provides a performance boost. Wetten [9] has reached a similar conclusion in his thesis by manually fixing the StealRatio's value to 1, 25, 35 or 45. Additionally, we show that an RL-based selection of this parameter holds no real value if no differentiation criteria can be derived. Rather, one can argue that using an RL-based selection in this context would negatively impact the performance since the overhead of using the RL agent is non-zero. We calculate this overhead in section 5.4; according to Boulmier et al. [33], it would account for 0.01% of the application's execution time.

Based on the observation that the software would achieve similar loop times independently of StealRatio's value, it is expected that the total execution times would remain constant within the same level of workload imbalance. However, it is not the case, and in Figure 5.15, the maximum ΔT_{par} in the first subplots column approaches 1 second. This might be caused by external system perturbations, which we analyse in the next section.

5.4 Replaying the DLS Selection

Through this experiment, we measure the impact that external factors have on the performance of an application. In order to replay an experiment, the selection of scheduling techniques is recorded and rewinded during a later execution. When replaying the RL agent's selection, through using the `custom` policy, the current outputted technique is hijacked, while the agent would follow the same states. This methodology introduces no overhead.

Based on the measurements available in Table 5.3, it can be observed that none of the configurations would produce the same results if the selection is replayed. There always exists an unpredictable component caused by uncontrollable system factors that ultimately affect the application's execution time. The Oracle can be seen as an unachievable performance due to the fact that this theoretical selection is based on measured performance. The variations in the table show that saving the learned path for later executions is not a viable approach, and there are no guarantees that the same performance would ever be reproducible.

Table 5.3: Differences in total application execution time when the sequence of DLS techniques achieved through automated or manual selection is replayed six times. For PISOLVER, the percentage in parenthesis refers to the workload imbalance.

Application	Configuration	Mean	Standard Deviation	Percentage of Variation	Max-Min Difference
PISOLVER (0%)	SARSALearner_explore-first_looptime-average	501.348 s	0.520 s	0.104 %	1.473 s
PISOLVER (5%)	QLearner_softmax_looptime	510.809 s	0.444 s	0.087 %	1.223 s
PISOLVER (10%)	QLearner_epsilon-greedy_cov	583.601 s	1.275 s	0.219 %	3.867 s
PISOLVER (15%)	SARSALearner_softmax_loadimbalance	567.019 s	0.951 s	0.168 %	2.776 s
PISOLVER (20%)	SARSALearner_epsilon-greedy_loadimbalance	639.639 s	1.887 s	0.295 %	4.865 s
PISOLVER (30%)	QLearner_explore-first_looptime	526.342 s	0.554 s	0.105 %	1.300 s
Mandelbrot	SARSALearner_explore-first_looptime-average	848.753 s	0.327 s	0.039 %	0.908 s
Mandelbrot	QLearner_softmax_cov	1402.620 s	4.107 s	0.293 %	11.500 s
Mandelbrot	SARSALearner_epsilon-greedy_loadimbalance	1448.702 s	3.100 s	0.214 %	8.466 s
PISOLVER (30%)	SS	818.071 s	11.993 s	1.466 %	36.534 s
PISOLVER (15%)	ORACLE	500.011 s	0.177 s	0.035 %	0.501 s
PISOLVER (30%)	ORACLE	505.693 s	0.454 s	0.090 %	1.255 s
Mandelbrot	ORACLE	774.900 s	0.150 s	0.019 %	0.358 s

Nevertheless, the RL agents based on the `looptime` and `looptime-average` reward achieve a promising standard deviation of under 0.56 seconds, similar in size to the Oracle. This might be seen as proof that the RL-based selection would not affect the stability of the system any more than the manual selection, and the impact of external factors is now evident and detrimental. On the other hand, larger differences are registered when the RL agent uses an execution time imbalance-based reward, `loadimbalance` or `cov`. The explanation is that the `SS` scheduling technique is picked more often. This technique implies large communication costs to achieve the highest possible workload balancing, and inter-process communication is an unmistakable root for execution time variance.

Therefore, uncontrollable system-level perturbations are demonstrated to exist. If replaying the selection sequence is shown to yield different results, repeating one experiment is even more unpredictable. A direct consequence is that the same RL agent is not guaranteed to have an identical DLS selection sequence in distinct runs, as there is no clear hierarchy in the performance times for the DLS techniques. To minimise the impact of external randomness, our experiments were repeated 5 times, with the median run being representative.

5.5 RL Component Overhead

In this section, we measure the impact of the RL component on the application's performance. The main factors that influence the RL component's overhead are the selection portfolio's dimensions, which are more important than its components (DLS, chunk or StealRatio parameters), and the agent's configuration (e.g. the active policy). Since MPI functions like MPI_Bcast and MPI_Reduce are needed to propagate the current state or to analyse workers' performance, they also impact the overhead of the RL component. Nonetheless, the total time needed for using the RL component is independent of the nature of the application, and the impact of the experiment size (e.g. number of MPI ranks) is minimal.

As shown in Figure 4.1 from a previous chapter, the `LB4MPI` library interacts with the RL component during parameters setup, at the start of the loop, when determining the active technique, and at the end of the loop, when analysing the workers' performance.

The results displayed in Table 5.4 are based on all experimental data gathered throughout this thesis. The mean parallel execution time $\overline{T}_{par}$ for the four applications aggregates data for the selection of the DLS algorithm, chunk parameter or StealRatio, where applicable. The times are as follows: PISOLVER distributed version - 586.03 s, PISOLVER replicated version - 577.82 s, Mandelbrot - 1203.48 s, SPHYNX Evrard Collapse - 8857.56 s.

Table 5.4: Measuring the overhead introduced through using the RL component

	$\overline{T}_{par}$ of the RL component	Std Dev T_{par} of the RL component	overhead % PISOLVER dist.	overhead % PISOLVER repl.	overhead % Mandelbrot repl.	overhead % SPHYNX repl.
Setup	0.00114 s	0.00014 s	0.00019 %	0.00020 %	0.00009 %	0.00001 %
StartLoop	0.06729 s	0.00778 s	0.01148 %	0.01165 %	0.00559 %	0.00076 %
EndLoop	0.06060 s	0.00260 s	0.01034 %	0.01049 %	0.00504 %	0.00068 %
Total	0.12903 s	0.00948 s	0.02202 %	0.02233 %	0.01072 %	0.00146 %

From Table 5.4, it can be noticed that the RL component's times fluctuate the most during the start of the loop. This is due to the fact that some RL configurations are more resource-demanding than others. For example, an agent using the explore-first policy would cost more performance-wise than one using epsilon-greedy due to how these configuration components work internally. The blocking MPI functions MPI_Bcast and MPI_Reduce are important sources of performance degradation, and asynchronous operations such as MPI_IBcast would likely improve the performance in a future software version. While the absolute cost of using the RL component remains relatively unchanging, the size of the experiment affects the overhead inflicted by using RL. As seen in the table, the RL component accounts for 0.001% to 0.022% of the total execution times ranging between 8857 seconds and 577 seconds. These results are comparable with the outcome of Boulmier et al. [33], where an overhead of under 0.01% is noticed for the application's execution time of 3'000'000 seconds (assuming the RL component's performance is calculated using data depicted in their figure 5).

Given the above measurements of the overhead, we will now study the potential gain in the performance of using the automated selection feature.

5.6 Potential Performance Gained

In this section, we measure the performance gained by using the automated RL DLS selection feature compared to randomly fixing the DLS technique with no prior knowledge.

In Table 5.5, the times in column 2 portray the mean parallel execution time of running an application where the DLS technique has been fixed manually to a random technique in the portfolio. Further, columns 3 and 4 display data based on a randomly-configured RL agent that automatically selects the DLS technique. It can be observed that the gain is negative for PISOLVER with under 15% workload imbalance. In the remaining cases, the performance is improved by up to 17.30%. Columns 5 and 6 show the mean execution time for an automated RL chunk parameter selector, and it is further compared with the baseline in column 1. The performance gained is between 2.63% and 41.28%. In most cases, the RL chunk parameter selection yields better results than the RL DLS selector, but it is not the case for Mandelbrot. Overall, the performance has been improved in 14 out of 18 cases.

Table 5.5: Potential performance gained through using the RL-based selection feature. For PISOLVER, the percentage in parenthesis refers to the workload imbalance.

	Manual Random DLS selection	Automated RL DLS selector		Automated RL SS chunk parameter selector	
	Baseline	$\overline{T}_{par}$	Gain (%)	$\overline{T}_{par}$	Gain (%)
PISOLVER (0%)	543.30 s	558.19 s	-2.74 %	522.58 s	3.81 %
PISOLVER (5%)	547.22 s	565.62 s	-3.36 %	532.86 s	2.63 %
PISOLVER (10%)	555.20 s	567.78 s	-2.27 %	539.17 s	2.89 %
PISOLVER (15%)	570.05 s	575.33 s	-0.93 %	539.82 s	5.30 %
PISOLVER (20%)	586.91 s	584.31 s	0.44 %	545.09 s	7.13 %
PISOLVER (25%)	626.33 s	606.56 s	3.16 %	553.36 s	11.65 %
PISOLVER (30%)	719.15 s	652.04 s	9.33 %	558.15 s	22.39 %
Mandelbrot	1375.08 s	1198.85 s	12.82 %	1216.71 s	11.52 %
SPHYNX Evrard	11320.81 s	9362.00 s	17.30 %	6647.60 s	41.28 %

Therefore, an informed decision of using or not the RL features must consider the following:

- The parallel execution time per loop should be large enough so that the overhead of the RL component becomes insignificant.
- The level of load imbalance at the application's level should be above a certain threshold; otherwise, the possible performance gains might be overcome by the RL agent's exploration costs. Nonetheless, most real-world scientific applications would introduce a significant level of load imbalance above zero.

Nevertheless, through this experiment, we assume a random configuration of the RL agent. We will perform an ANOVA analysis throughout the following section.

5.7 Analysis of Variance

In this section, we perform an Analysis of Variance (ANOVA) to find which RL agent's component category is more significant for the application's parallel execution time. This type of analysis was proposed in 1989 by Snedecor and William [36]. This tutorial available online has proven useful when learning about how to perform this kind of analysis. Furthermore, for the Python implementation, we consulted this online tutorial and used automated tools from the `bioinfokit`, `statsmodels` and `scipy` Python libraries for statistical measures.

In essence, the one-level ANOVA compares the means of multiple groups of data. In our case, a level is the RL component type (agent/policy/reward), while a group contains specific RL algorithms. The group mean is calculated using the variance-based F-test. The F-test quantifies how much items in a group vary through a comparison of the resulting F statistic and the significance threshold of 0.05. The F statistic takes into consideration both the between-group variance SSB and the within-group variance SSW. Large values for this metric imply a large variance, while a small value means that items inside the group are highly-similar.

We show in Figure 5.17 an aggregation of all the performance data recorded for the replicated data experiments set as a way to visualise the statements we soon make. By gathering all the items in a column, we define the one-level, while groups are represented by specific component types represented through boxplot objects. Each row displays information regarding one benchmarking application. The red dots inside each boxplot represent the performance recorded for a particular experiment where a certain configuration type is equipped.

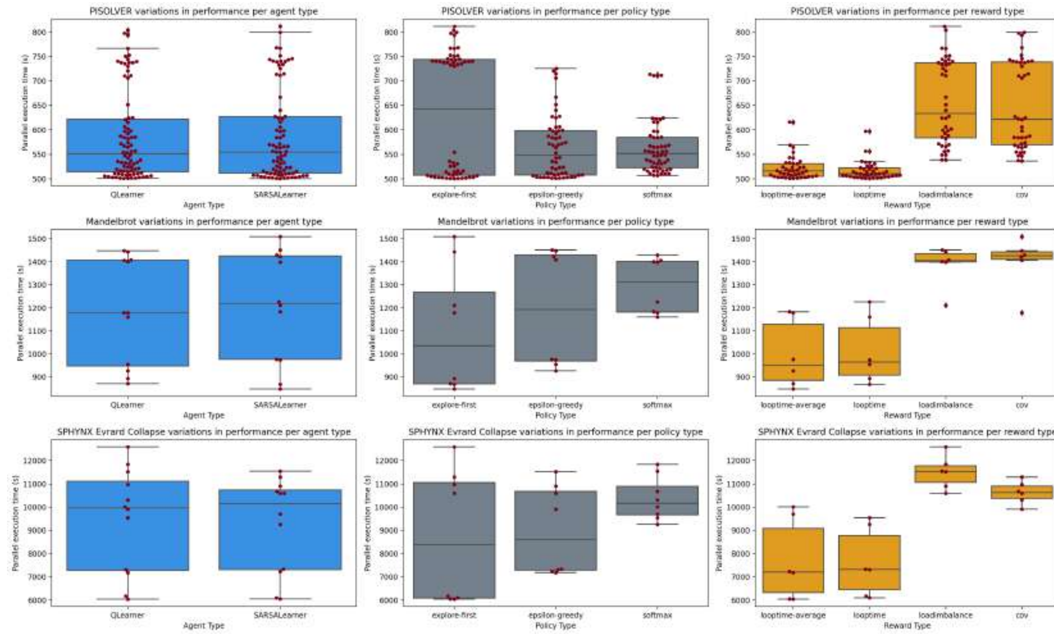


Figure 5.17: Parallel execution times for all replicated-data experiments grouped by the RL configuration type

In Table 5.6, we display the sum of squares, which measures the deviation from the mean, and the F statistic for an RL configuration type. For the agent type, we observe that the F statistic is under the threshold of 0.05, hence using any of `QLearn` or `SARSALearn` achieved no advantage in the 1'835 replicated-data factorial experiments we ran. As for the policy type, the F statistic is slightly above the significance threshold, and this component type is proven somewhat important. However, as graphically displayed in Figure 5.17, the configuration type that achieves performance results varying the most is the reward type.

Table 5.6: One-factor ANOVA F-statistic

Source	Sum of squares	F-statistic
Agent Types	15499	0.001921
Policy Types	1170257	0.072251
Reward Types	15071660	0.622446

ANOVA shows how significant the difference is in general among the groups but does not provide any data on particular components. With the results displayed in Table 5.6, we dive deeper into statistically measuring the impact of each item of a group. By using Tukey's Honestly Significantly Differenced (HSD) [37] test, we perform multiple pairwise comparisons

of the components and search for possible similarities. A lower `Diff` is associated with two components being similar. The *Lower* and *Upper* refer to the lowest and largest differences recorded between any two experimental results. The *q-value* measures the possibility that this discovery is a false-positive, and a lower value is considered better. Also, the *p-value* measures how likely this value of *Diff* would occur, starting from an initial hypothesis that the items in a pair items are not different. A *p-value* closer to 1 is better.

Table 5.7: Tukey's Honestly Significantly Differenced analysis

group1	group2	Diff (s)	Lower (s)	Upper (s)	q-value	p-value
QLearner	SARSALearner	16.94	-744.87	778.75	0.06	0.90
explore-first	epsilon-greedy	13.39	-1106.13	1132.92	0.04	0.90
explore-first	softmax	149.01	-970.51	1268.54	0.44	0.90
epsilon-greedy	softmax	162.41	-957.11	1281.93	0.48	0.90
looptime-average	looptime	11.60	-1404.29	1427.49	0.03	0.90
looptime-average	loadimbalance	568.24	-847.65	1984.13	1.47	0.70
looptime-average	cov	466.87	-949.02	1882.76	1.21	0.81
looptime	loadimbalance	579.84	-836.05	1995.73	1.5	0.69
looptime	cov	478.47	-937.42	1894.36	1.24	0.79
loadimbalance	cov	101.37	-1314.52	1517.26	0.26	0.90

In Table 5.7, we display a three-level comparison for the levels RL agent type, RL policy type and RL reward type. The most similar pair contains `looptime` and `looptime-average` rewards. Amongst the Agent type level, `QLearn` and `SARSA` are somewhat similar. The most unexpected result shows that the `explore-first` and `epsilon-greedy` policies are highly similar, hinting that *when* the `exploration` happens is less important than *if* it happens. As expected, the highest differences are among the execution time-based reward metrics and the imbalance-based metrics.

The true potential of a carefully customised RL agent is explored in the next section. The *discussion* will touch on all topics analysed in this chapter.

5.8 Discussion

In this section, we summarise the findings of all the above-described experiments.

We focus first on the replicated-data experimental results. We tested our software using 3 time-stepping applications. PISOLVER maintains a constant level of workload imbalance throughout the entire application's run, and this level can be user-specified. In Mandelbrot, the selection has been tested within three loops, with constant/increasing/decreasing load imbalance. Finally, SPHYNX Evrard Collapse is closest to how a real-life application would behave, and the timestep-to-timestep level of load imbalance is hard to predict.

All benchmarking results are condensed in Figure 5.18. The columns represent selection strategies, while the rows represent different application setups. Hence, their intersection represents the parallel execution time for one selection strategy, and the percentage above is the performance lost from the theoretical best, Oracle. A smaller performance loss percentage is, of course, better. The performance lost over Oracle by the RL DLS selector agent is between 0.82% for the `SARSALearner`,`explore-first`,`looptime` agent,

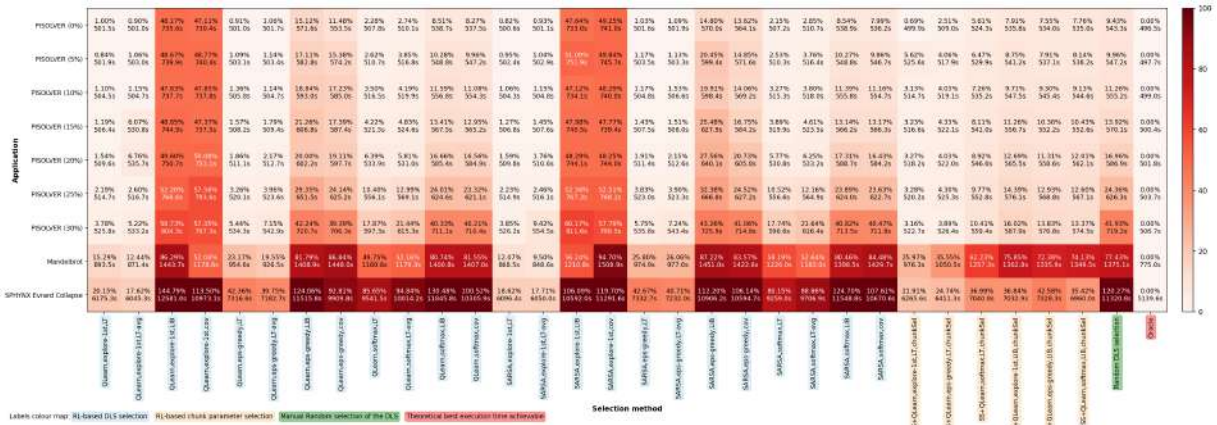


Figure 5.18: Results of benchmarking automated selection strategies using time-stepping applications. Formula to calculate the decline in performance compared with the Oracle: x% = (T_{par} - T_{par}^{Oracle}) / T_{par}^{Oracle} × 100, where T_{par} is the parallel execution time

and 144.79%, for `QLearner,explore-first,loadimbalance`. Similarly, for the RL chunk selector, the top performance is within 0.69% from Oracle, and it is achieved by `ChunkParameterSelector+QLearn,explore-first,looptime`. The poorest result is 75.85% performance degradation, and it is achieved by `ChunkParameterSelector +QLearn,explore-first,loadimbalance`. Nevertheless, for each application, at least an automated selector exists such that it would outperform the randomised manual DLS selection. Generally, the RL agent's configuration component that affects performance most is the reward type. Agents using the `looptime` or `looptime-average` reward achieve lower execution times than those equipped with `loadimbalance` or `cov`. The second most important configuration component is the policy type. Agents employing an `explore-first` policy would often achieve the highest rewards - having a reward function focusing on achieving high performance is significant. As for the agent's type, as most literature studies mention [5, 7, 16], using `QLearn` or `SARSALearn` would yield no long-term advantage. These intuition-based results are statistically proven through the ANOVA analysis we performed.

Moving from the replicated data to a distributed data setup would be beneficial in terms of scalability and memory consumption. However, for RWS, using the RL agent to set the StealRatio is excessive. The RL agent's overhead would deteriorate the performance; hence using any fixed value is preferred in our experimental setup.

Further, we demonstrated that using RL would not bring instability in the application's total execution time any more than external system perturbations would do. We measured the overhead of using the RL-based selection to be between 0.001% and 0.022% of the application's total parallel execution time. This overhead percentage is inversely proportional to the loop execution time, as using the RL feature requires a seemingly constant amount of time. Also, the performance gain through using a randomly configured RL agent for automated DLS selection is between -3.36% and 41.28%, when compared to manual random DLS selection. For applications with a notable level or workload imbalance, using the RL features to automatically select the optimal DLS or the chunk size parameter is advised.

The next chapter will conclude this thesis and will overview the potential future steps.

6 Conclusions and Future Work

This work extends the `LB4MPI` library with automated scheduling algorithm selection capabilities. Our software uses a C++-based Reinforcement Learning solution to build an autonomic computing instrument able to improve the performance of time-stepping applications. While the main functionality includes automatically selecting the most promising DLS, the software is also able to select various parameters for diverse load-balancing strategies. Through using the `ChunkParameterSelector` RL agent, the chunk size parameter for the `SS` technique can be automatically selected. The chunk parameter selection can also be coupled with other DLS techniques in order to set the minimum chunk size. As a novelty, we propose the `StealRatioSelector` RL agent, which would select the most promising value for the StealRatio in an RWS distributed-data context. The RL component is highly customizable, and the user can choose between 7 agent types, 3 action selection policies (or enter the expert-custom mode), 10 reward metrics, and can tune the hyper-parameters.

To establish the quality of the RL-based selection, we used 3 time-stepping scientific applications. For PISOLVER, we studied both the replicated-data and distributed-data versions. For Mandelbrot, three loops (constant, increasing, and decreasing workload imbalance) run synchronously. Through the third application, SPHYNX Evrard Collapse, we study the potential of multi-level scheduling at both process and thread levels. The performance deterioration from the theoretical best selection, the Oracle, is used as the main comparison metric. This quantity can vary from 0.69% for top-quality selections to the undesirable 144.79%. Meanwhile, manually fixing the DLS to a random value yields results in the range of 9.43% to 120.27% performance degradation when compared to the Oracle, depending on what benchmarking application is being used and the level of workload imbalance.

The reward's nature would affect the selection quality the most. An RL agent using `looptime` and `looptime-average` rewards outperform the `load imbalance` or `cov` rewarded selections in almost all cases. Our improved `explore-first` policy helps achieve the highest rewards, but having a performance-oriented reward type is very important. Regardless of using a `QLearn` or `SARSALearn` agent, no long-term advantage is created. Our findings are in line with the general consensus encountered through reviewing the literature. The intuition-based results have been statistically proven through an ANOVA examination.

As distributed-data solutions are considerably more scalable than replicated-data ones, as the memory usage at the node level would be drastically reduced, we apply RL to select the `StealRatio` in a distributed-data RWS load-balancing strategy. However, regardless of the amount of work that is set to be stolen, the looptimes are identical. Nevertheless, the overhead introduced by the RL component would negatively impact the overall performance.

Through replaying the selection process, we demonstrated that external system-level perturbations would affect the execution times for all DLS techniques. Moreover, we have shown that the RL component would not add additional variance to the application's performance. Also, the cost of using the RL-based selection is rather constant in the context of a loop. The overhead represents between 0.001% and 0.022% of the application's execution time. The performance gain of automating the selection process is between -3.36% and 41.28% when comparing a randomly-configured RL agent and the manual fixing of a random DLS.

Through this work, the all-automated selection of the cross-node load balancing algorithm is now possible. Since both the `LB4OMP` and `LB4MPI` libraries are now able to access the RL features, one logical future step is to study the multi-level automated workload scheduling technique selection for hybrid `MPI` + `OpenMP` applications. Furthermore, at the moment, the configuration and parameter tuning of the RL agent is manually done. Perhaps training a Machine Learning model specifically for hyper-parameter tuning of the RL agent can improve its performance. Additionally, building the DeepQ-Learning agent to learn the state-action space without using the memory-inefficient tabular structures would be desired. Also, variants of the `QLearn` and `SARSALearn` that promise to improve scalability (e.g. `DoubleQLearn`, `QVLearn`, `E-SARSA`) need to be experimented with using computation-intensive applications. Moreover, a neural network able to combine the signals of the 10 existing reward functions would most likely lead to a better-informed selection. Finally, a distributed data setup is shown to decrease the node-level memory needs, hence improving the scalability of HPC systems. Perhaps RL-based selection using meta agents can be applied in parameter selection for other scheduling strategies, such as locality-aware work-stealing.

Based on all of the above remarks, we achieved the Master's thesis goal of performing an Automated Selection of Scheduling Algorithms using Reinforcement Learning in LB4MPI.

Bibliography

[1] A. Mohammed, A. Eleliemy, F. M. Ciorba, F. Kasielke, and I. Banicescu. An approach for realistically simulating the performance of scientific applications on high performance computing systems. *Future Generation Computer Systems*, 111:617–633, 2019.

[2] F. M. Ciorba, C. Iwainsky, and P. Buder. OpenMP Loop Scheduling Revisited: Making a Case for more Schedules. In *International Workshop on OpenMP*, pages 21–36. Springer, 2018.

[3] J. H. Müller Korndörfer, A. Mohammed, A. Eleliemy, and F. M. Ciorba. LB4OMP: A Dynamic Load Balancing Library for Multithreaded Applications. *IEEE Transactions on Parallel and Distributed Systems*, 33(4):830–841, 2022.

[4] S. Dhandayuthapani. Automatic selection of dynamic loop scheduling algorithms for load balancing using reinforcement learning. Master's thesis, Mississippi State University, 2004.

[5] I. Banicescu, F. M. Ciorba, and S. Srivastava. Performance optimization of scientific applications using an autonomic computing approach. *Scalable Computing: Theory and Practice*, pages 437–466, 2012.

[6] N. Sukhija, B. Malone, S. Srivastava, I. Banicescu, and F. M. Ciorba. Portfolio-based Selection of Robust Dynamic Loop Scheduling Algorithms Using Machine Learning. In *2014 IEEE International Parallel & Distributed Processing Symposium Workshops*, pages 1638–1647. IEEE, 2014.

[7] L. Kury. Automated Selection of Scheduling Algorithms for Parallel Scientific Applications using Reinforcement Learning with OpenMP. Master's thesis, Universität Basel, 2022.

[8] A. Mohammed, J. H. Müller Korndörfer, A. Eleliemy, and F. M. Ciorba. Automated Scheduling Algorithm Selection and Chunk Parameter Calculation in OpenMP. *IEEE Transactions on Parallel and Distributed Systems*, 33(12):4383–4394, 2022.

[9] G. A. Wetten. Dynamic Scheduling in HPC using a Distributed Data Approach. Master's thesis, Universität Basel, 2022.

[10] A. Afzal, G. Hager, S. Markidis, and G. Wellein. Making Applications Faster by Asynchronous Execution: Slowing Down Processes or Relaxing MPI Collectives. *arXiv preprint arXiv:2302.12164*, 2023.

[11] B. B. Mandelbrot. Fractal aspects of the iteration of z$\rightarrow$ λz (1-z) for complex λ and z. *Annals of the New York Academy of Sciences*, 357(1):249–259, 1980.

[12] R. M Cabezón, D. Garcia-Senz, and J. Figueira. SPHYNX: an accurate density-based SPH method for astrophysical applications. *Astronomy & Astrophysics*, 606:A78, 2017.

[13] HPC Group at Basel Universität. miniHPC: SMALL BUT MODERN HPC. https://hpc.dmi.unibas.ch/en/research/minihpc/, 2016. Accessed: May 1, 2023.

[14] L. Clarke, I. Glendinning, and R. Hempel. The MPI message passing interface standard. In *Programming Environments for Massively Parallel Distributed Systems: Working Conference of the IFIP WG 10.3, April 25–29, 1994*, pages 213–218. Springer, 1994.

[15] R. Cariño and I. Banicescu. A tool for a two-level dynamic load balancing strategy in scientific applications. *Scalable Computing: Practice and Experience*, 8(3), 2007.

[16] M. Rashid, I. Banicescu, and R. Cariño. Investigating a dynamic loop scheduling with reinforcement learning approach to load balancing in scientific applications. In *2008 International Symposium on Parallel and Distributed Computing*, pages 123–130. IEEE, 2008.

[17] P. Tang and P. Yew. Processor self-scheduling for multiple-nested parallel loops. Technical report, Illinois Univ., Urbana (USA). Center for Supercomputing Research and Development, 1986.

[18] I. Banicescu, V. Velusamy, and J. Devaprasad. On the scalability of dynamic scheduling scientific applications with adaptive weighted factoring. *Cluster Computing*, 6(3):215–226, 2003.

[19] C. P. Kruskal and A. Weiss. Allocating independent subtasks on parallel processors. *IEEE Transactions on Software engineering*, 100:1001–1016, 1985.

[20] R. Cariño and I. Banicescu. Dynamic load balancing with adaptive factoring methods in scientific applications. *The Journal of Supercomputing*, 44(1):41–63, 2008.

[21] I. Banicescu, F. M. Ciorba, and S. Srivastava. Performance optimization of scientific applications using an autonomic computing approach. *Scalable Computing: Theory and Practice*, pages 437–466, 2012.

[22] C. D. Polychronopoulos and D. J. Kuck. Guided self-scheduling: A practical scheduling scheme for parallel supercomputers. *IEEE Transactions on Computers*, 100(12):1425–1439, 1987.

[23] T. H. Tzen and L. M. Ni. Trapezoid self-scheduling: A practical scheduling scheme for parallel compilers. *IEEE Transactions on parallel and distributed systems*, 4(1):87–98, 1993.

[24] S. Hummel, E. Schonberg, and L. Flynn. Factoring: A method for scheduling parallel loops. *Communications of the ACM*, 35(8):90–101, 1992.

[25] I. Banicescu and Z. Liu. A dynamic scheduling method tuned to the rate ofweight changes. In *High Performance Computing Symposium*, pages 122–129, 2000.

[26] S. Hummel, J. Schmidt, R. Uma, and J. Wein. Load-sharing in heterogeneous systems via weighted factoring. In *Proceedings of the eighth annual ACM symposium on Parallel algorithms and architectures*, pages 318–328, 1996.

[27] O. Pearce, T. Gamblin, B. R. De Supinski, M. Schulz, and N. M. Amato. Quantifying the effectiveness of load balance algorithms. In *Proceedings of the 26th ACM international conference on Supercomputing*, pages 185–194, 2012.

[28] R. Sutton and A. Barto. *Reinforcement learning: An introduction.* MIT press, 1992.

[29] H. Hasselt. Double q-learning. In J. Lafferty, C. Williams, J. Shawe-Taylor, R. Zemel, and A. Culotta, editors, *Advances in Neural Information Processing Systems*, volume 23. Curran Associates, Inc., 2010.

[30] M. A. Wiering and D. Leone. QV (lambda)-learning: A new on-policy reinforcement learning algorithm. In *Proceedings of the 7th European workshop on reinforcement learning*, pages 17–18, 2005.

[31] J. Fan, Z. Wang, Y. Xie, and Z. Yang. A Theoretical Analysis of Deep Q-Learning. In *Proceedings of the 2nd Conference on Learning for Dynamics and Control*, volume 120 of *Proceedings of Machine Learning Research*, pages 486–489. PMLR, 10–11 Jun 2020.

[32] H. Van Seijen, H. Van Hasselt, S. Whiteson, and M. Wiering. A theoretical and empirical analysis of expected sarsa. In *2009 IEEE symposium on adaptive dynamic programming and reinforcement learning*, pages 177–184. IEEE, 2009.

[33] A. Boulmier, I. Banicescu, F. M. Ciorba, and N. Abdennadher. An Autonomic Approach for the Selection of Robust Dynamic Loop Scheduling Techniques. In *2017 16th International Symposium on Parallel and Distributed Computing (ISPDC)*, pages 9–17. IEEE, 2017. doi: 10.1109/ISPDC.2017.9.

[34] S. F. Hummel, E. Schonberg, and L. E. Flynn. Factoring: A method for scheduling parallel loops. *Communications of the ACM*, 35(8):90–101, 1992.

[35] A. Mohammed, A. Cavelan, F. M. Ciorba, R. M. Cabezón, and I. Banicescu. Two-level Dynamic Load Balancing for High Performance Scientific Applications. In *Proceedings of the 2020 SIAM Conference on Parallel Processing for Scientific Computing*, pages 69–80. SIAM, 2020.

[36] G. W.C. Snedecor and G William. Statistical Methods. *Iowa State University Press, Eighth Edition*, pages 84–86, 1989.

[37] H. Abdi and L. J. Williams. Tukey's honestly significant difference (HSD) test. *Encyclopedia of research design*, 3(1):1–5, 2010.

A
Appendix

The appendix brings completeness to the topics discussed in the main body of the thesis. The following figures are grouped so that the reader can have an ensemble view of each replicated PISOLVER experiment, with a level of workload imbalance between 0% and 30%.

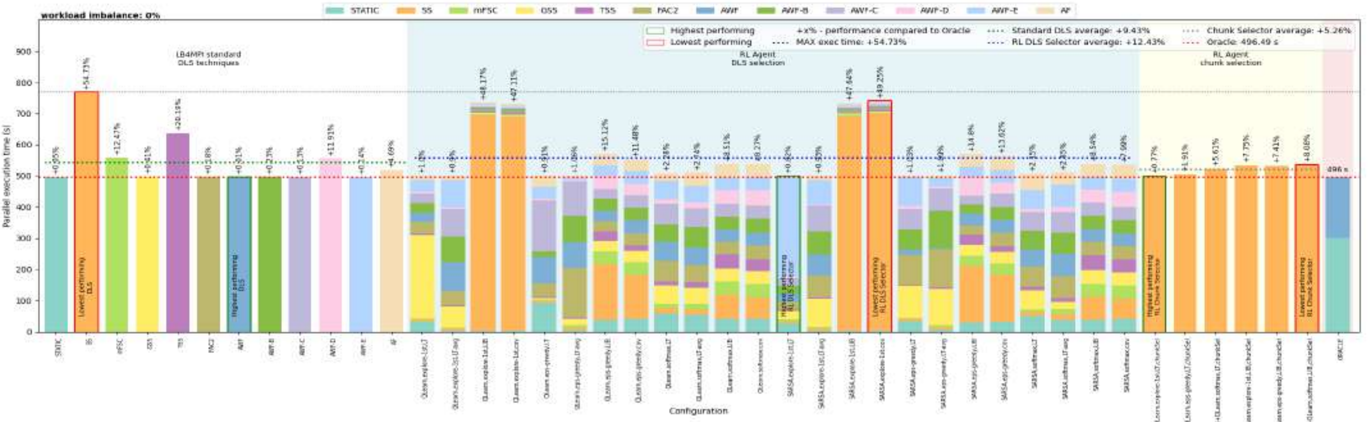


Figure A.1: Performance summary for PISOLVER 0% workload imbalance

Figure A.2: DLS selection per timestep for PISOLVER with 0% workload imbalance

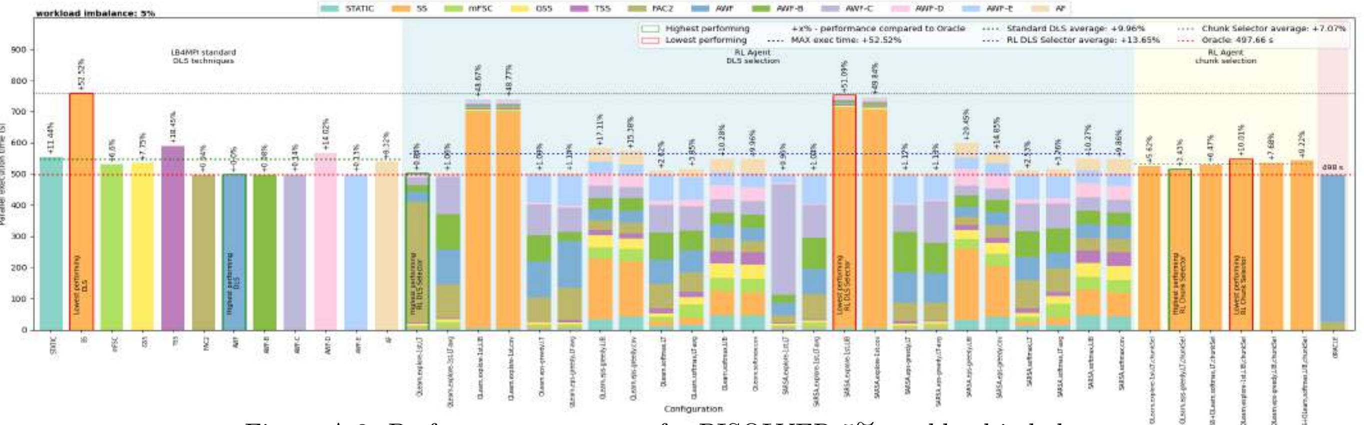


Figure A.3: Performance summary for PISOLVER 5% workload imbalance

Figure A.4: DLS selection per timestep for PISOLVER with 5% workload imbalance

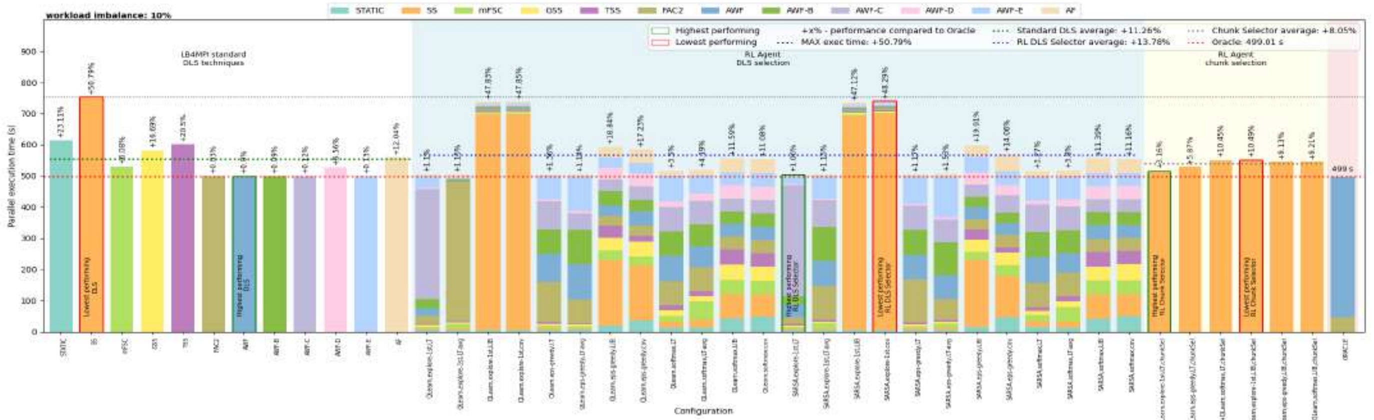


Figure A.5: Performance summary for PISOLVER 10% workload imbalance

Figure A.6: DLS selection per timestep for PISOLVER with 10% workload imbalance

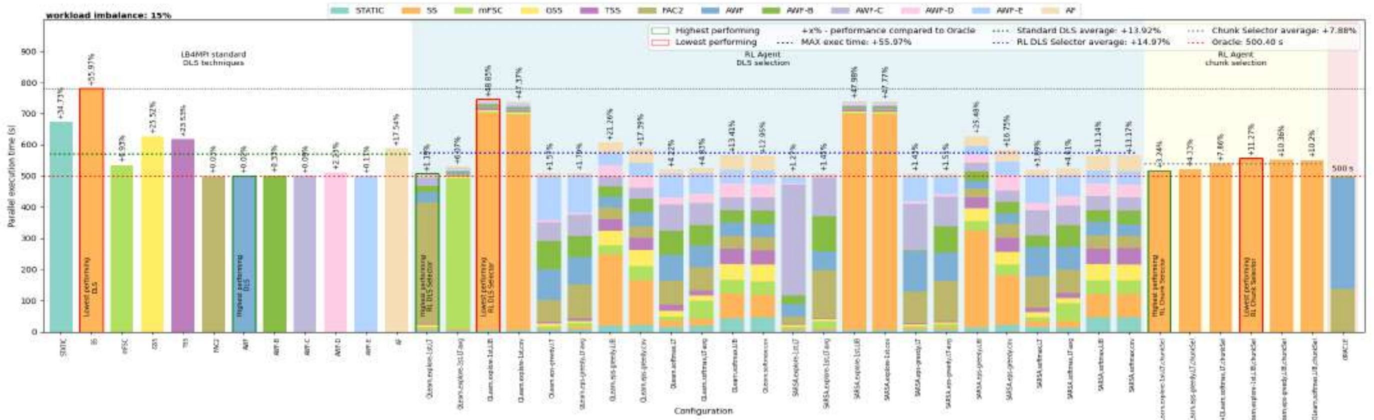


Figure A.7: Performance summary for PISOLVER 15% workload imbalance

Figure A.8: DLS selection per timestep for PISOLVER with 15% workload imbalance

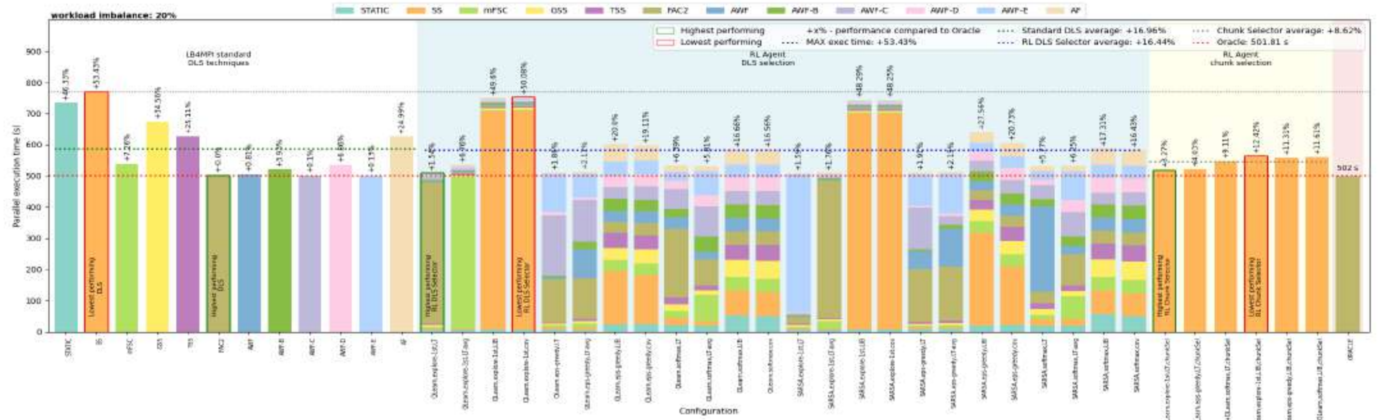


Figure A.9: Performance summary for PISOLVER 20% workload imbalance

Figure A.10: DLS selection per timestep for PISOLVER with 20% workload imbalance

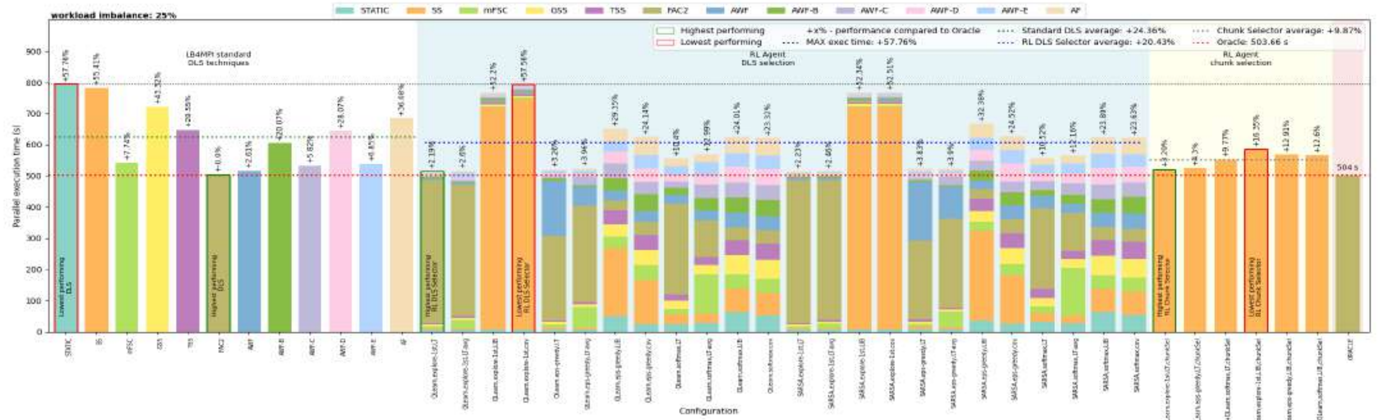


Figure A.11: Performance summary for PISOLVER 25% workload imbalance

Figure A.12: DLS selection per timestep for PISOLVER with 25% workload imbalance

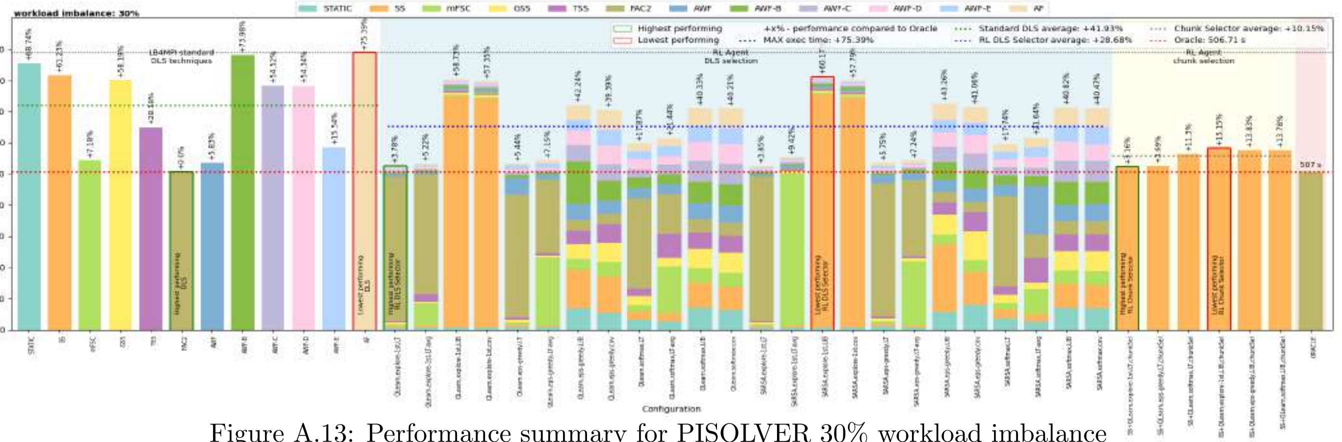


Figure A.13: Performance summary for PISOLVER 30% workload imbalance

Figure A.14: DLS selection per timestep for PISOLVER with 30% workload imbalance

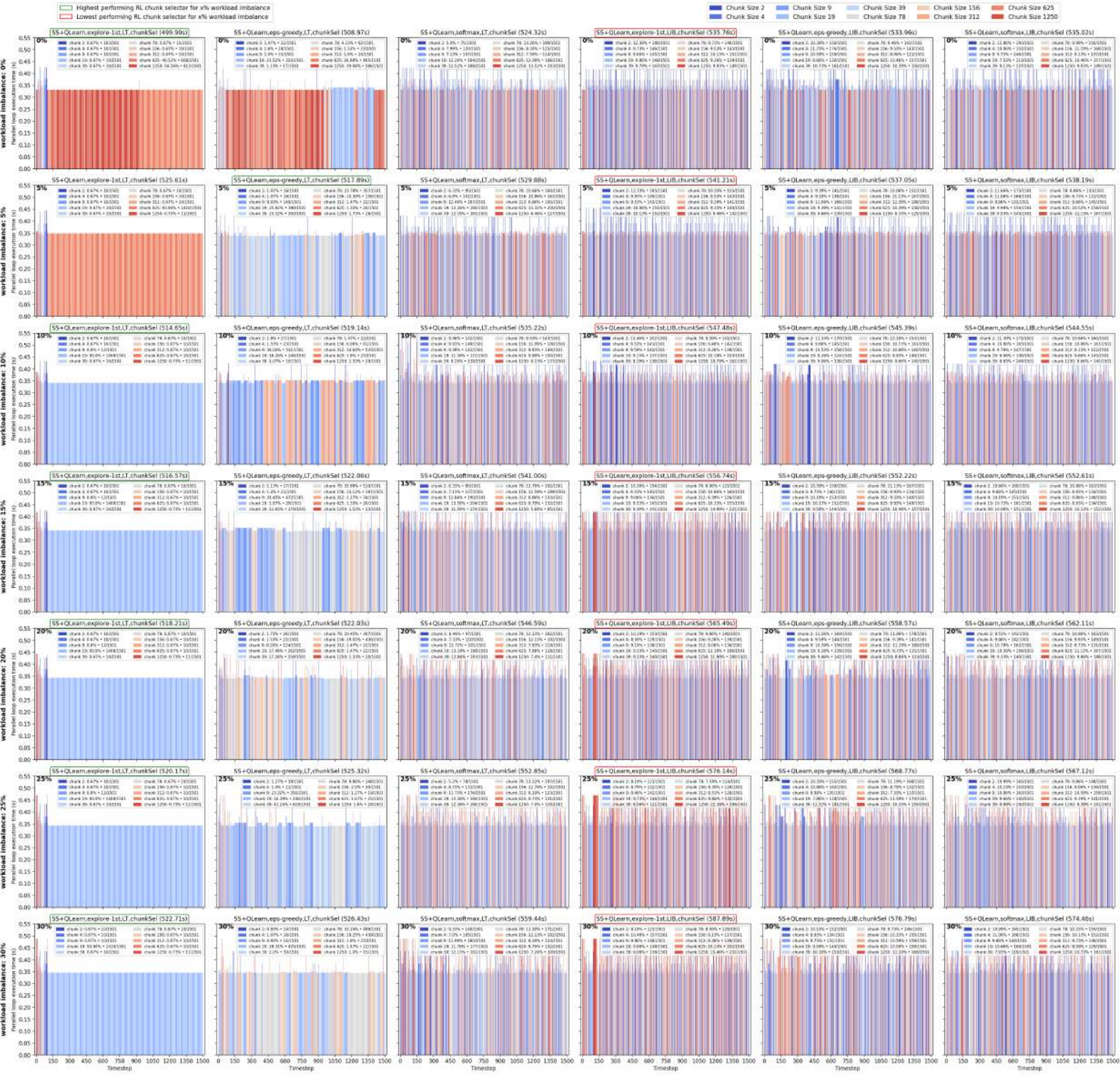


Figure A.15: RL chunk size selection for PISOLVER