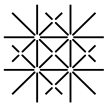

University of Basel

Automated Selection of Scheduling Algorithms for Parallel Scientific Applications using Reinforcement Learning with OpenMP

Master thesis

Natural Science Faculty of the University of Basel
Department of Mathematics and Computer Science
HPC Group
https://hpc.dmi.unibas.ch

Advisor: Prof. Florina M. Ciorba
Supervisor: Jonas H. M. Korndorfer

Luc Kury
luc.kury@unibas.ch
10-462-687

30^{th} of July, 2022

Abstract

Performance degradation due to load imbalance in computationally-intensive applications is a significant road block on the way of achieving higher parallel application performance. It is predominantly caused by idling processors, while there are other computation tasks ready to be executed. This results in uneven execution progress among the parallel processing units. Computationally-intensive applications often represent irregular workloads. The computing systems running such workloads consist of heterogeneous processors and may be affected by interference such as non-uniform memory access, operating system noise and contention. The resulting load-imbalance can effectively be combated by dynamic scheduling of computation units onto processing units. As a consequence many different scheduling heuristics have been devised over the past decades. Finding an optimal scheduling algorithm is a NP-hard problem. Through careful selection of a scheduling technique, the problem of imbalanced loads can be addressed effectively. However, a manual selection approach is time-consuming, tedious, and is fixed for the entire duration of the application's runtime. As a solution we propose RL4OMP, an extension to the LLVM OpenMP runtime, consisting of Reinforcement Learning agents, as a mean to achieve automated selection of DLS algorithms for OpenMP-loops. RL4OMP has 6 different agent types, 3 different action selection policies and 6 different reward functions at its disposal and is implemented in a extendable fashion through a class-based component system. Further we propose a new scheduling algorithm Chunk-Learn, which directly estimates the chunk-sizes for the scheduling rounds by the use of Reinforcement Learning.

The results of our performance analysis campaign show that our Reinforcement Learning extension RL4OMP in some cases can outperform state-of-practice loop scheduling algorithms and achieve the performance closest the ground-truth without any prior knowledge about the applications characteristics. However we also reveal that depending on the application, the configuration of the learning agent can have an non-negligible impact on its performance and therefore fall behind other expert knowledge based automated DLS algorithm selection methods.

This work reveals the feasibility of Machine Learning, Reinforcement Learning in particular, as a promising tool to increase parallel applications' performance in an unsupervised fashion. We also show the use of Reinforcement Learning as a scheduling algorithm can lead to better overall performance than trying to use it to solve the algorithm selection problem.

Table of Contents

1 Introduction

Performance degradation in parallel or distributed applications originating from load-imbalance is a significant barrier to achieve shorter parallel execution times. Load-imbalance is predominantly caused by idling processors, while there are other tasks ready to be executed but no processor has started doing so. This results in uneven execution progress among the parallel processing units. Computationally-intensive applications often represent irregular workloads and HPC clusters may be affected by non-uniform memory access, operating system noise, and contention due to the sharing of resources. These effects are also known as perturbations. Load-imbalance can effectively be reduced by dynamically and adaptively scheduling computation units. For this task, many different scheduling heuristics have been devised. However finding and selecting an optimal scheduling algorithm is non-trivial. [4]

Parallel applications, such as OpenMP programs, are especially susceptible to the effects of load-imbalance. Through careful selection of a scheduling technique for either every loop or the entire application, the problem of slow execution can be addressed effectively. However, a manual selection approach is time-consuming, can lead to decision paralysis, and is fixed for the entire duration of the application's execution. An automatic selection methods can address this problem and shortcomings in an effective manner. Previous work based on export knowledge and Machine Learning, shows that automatic selection methods perform as good or outperform a statically selected state-of-practice scheduling algorithm. [2, 13, 25]

For this thesis we intend to implement a Reinforcement Learning extension for the LLVM OpenMP runtime to augment its capabilities to automatically select between the available scheduling algorithms during runtime using Machine Learning. We will achieve this by applying an Object-Oriented approach using C++14, encapsulating every component in its own class. This allows for the extension to be lightweight, extensible and portable. Besides Q-Learning and SARSA, which have been explored in previous work, we plan to explore additional learning methods like Expected-SARSA, DoubleQ-Learning, QV-Learning to improve the selection process of DLS algorithms. Additionally the effects of three different actions selection policies and 6 different reward functions shall be investigated. For the performance evaluation we use two scientific benchmark applications on a real-world HPC system. [2, 20]

2

Background

Reinforcement Learning (RL) has gained popularity in the last decade with a series of successful real-world applications in robotics, games and many other fields.

In this chapter we will provide a high-level structural overview of classic Reinforcement Learning algorithms. The discussion will be based on their similarities and differences in the intricacies of algorithms.

2.1 Machine Learning

Reinforcement Learning is a part of the Machine Learning domain, along with supervised and unsupervised learning (Fig. 2.1). Supervised and Unsupervised learning are better suited for classification of clustering problems, while Reinforcement Learning enables an agent to learn how to make decisions under uncertainty for sequential decision problems.[26]

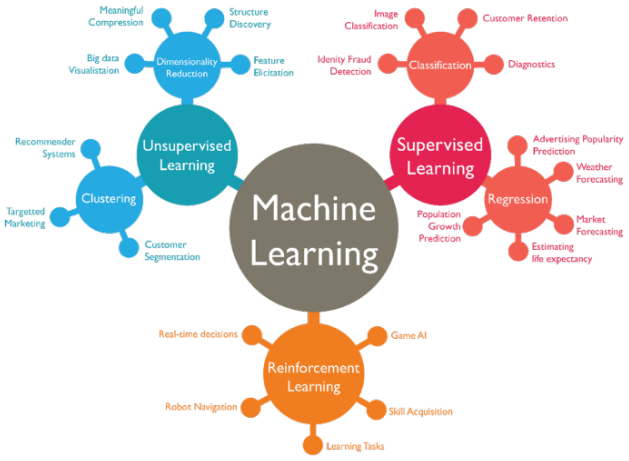

Figure 2.1: Overview of machine learning disciplines. The big three are: Unsupervised Learning, Supervised Learning and Reinforcement Learning.

2.2 Reinforcement Learning

There are two fundamental tasks in Reinforcement Learning: prediction and control. In prediction tasks, we are given a policy and our goal is to evaluate it by estimating the value $\mathcal{Q}$ of taking actions following this policy. In control tasks, we don't know the policy, and the goal is to find the optimal policy that allows us to collect the most rewards. As one might assume, the selection of scheduling algorithms belongs to the group of control tasks.

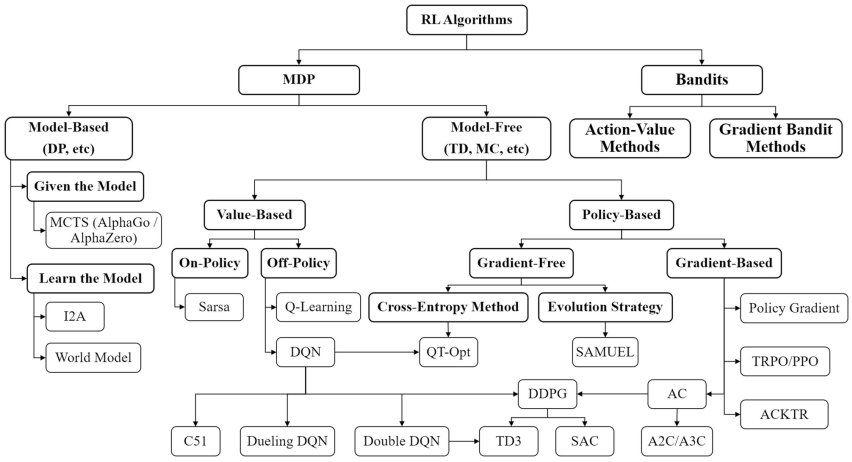

Figure 2.2: Overview of Reinforcement Learning method categories. The suitable learning methods for the control tasks presented in this thesis are the model-free (temporal-difference), values-based on-policy and off-policy methods.

2.2.1 Basic Principles

With Reinforcement Learning, the agent learns how to behave optimally in an unknown environment by taking actions and learning from the effects of that particular action. Each time the agent acts, the environment rewards the agent with a scalar value, and exposes the new state of the environment. The agent has to decide again which action to use, given the current state of the environment, to maximize its rewards. Figure 2.3 illustrates the main components and information flow in a RL system.

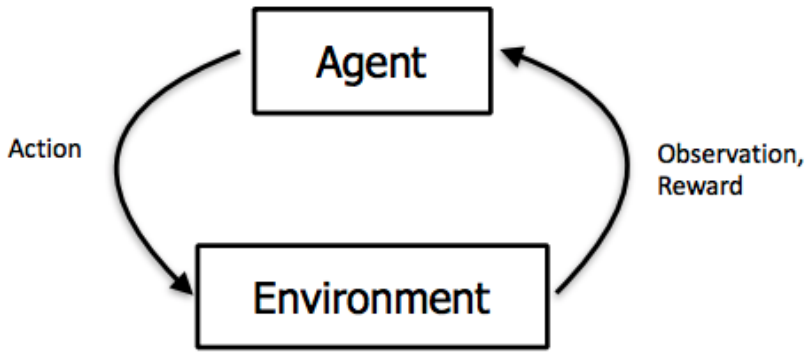

Figure 2.3: Basic working principle of a RL system. The figure shows the connection between the *agent* and *environment* entities via *actions*, *observations* and *rewards*.

In Reinforcement Learning there are different algorithms that govern the decision making of an agent and they can be divided into two groups: model-free and model-based. Model-free algorithms learn the optimal action-value function and use it to derive a control policy. This function represents the expected reward for taking a certain action. Model-based algorithms learn the model of the environment and use it to derive the control policy. In unpredictable environments (such as HPC systems that are influenced by random perturbations), the model-based algorithms are not suitable to derive any meaningful control policy.

To learn the problems optimal control policy, model-free agents iteratively approximate the optimal action-values via temporal difference (TD) learning. The agent updates the current approximation of the control policy after each action-reward cycle (i.e. a single time-step in a time-stepping application). Each model-free RL agent implements a behavioral policy (e.g. greedy policy, ϵ-greedy policy, etc.) that determines how to choose the next action [26]. This policy decides whether the agent should operate with the current best choice (exploitation) or test alternatives (exploration).

Learning methods like Q-Learning, SARSA and its derivatives are also know as tabular methods. They use a two-dimensional table like data structure to associate possible future rewards with a given state-action pair. This can become a problem with memory and lookup overhead when the environment has a large number of states and the agent can select amongst many actions. DeepQ-Learning differs from the tabular based approach and replaces it with a neural network to approximate the action-value function $\mathcal{Q}$. Instead of updating a single value in the lookup after a complete action-reward cycle, the weights of the network are updated via back-propagation. From this also follows that all the action-state values are output by the network at the same time and we can still employ any control policy that suits our problem the best. [21, 22] This design entails a few advantages and new possibilities:

- The network is able to generalize from states the agent has visited to states it has not visited $\rightarrow$ reduction in states that need to be visited to reach an approximate solution
- Allows action-value function approximation to be non-linear
- Can handle a large number of inputs
- Incremental training support

2.2.2 On-Policy v.s. Off-Policy Learning

Using either On-Policy or Off-Policy learning methods leads back to the exploration vs. exploitation dilemma. Essentially, an agent is forced to make a choice between making the best decision given the current information or start exploring and finding more information. If the algorithm for policy improvement always updates the policy greedily, meaning it takes only actions leading to immediate reward, actions and states not on the greedy path will not be sampled sufficiently, and potentially better rewards would stay hidden from the learning process.

On-policy methods solve the exploration vs. exploitation dilemma by either forcing exploration at the start or including randomness in the form of a policy that is soft, meaning

that non-greedy actions are selected with some probability. It will do so by evaluating and improving the same policy that the agent is already using for action selection. We say that the target policy is equal to the behavior policy (see Fig. 2.4). It is worth noting that because the optimal action will be sampled more often than the other actions, using on-policy algorithms the agent will generally converge faster but also have the risk of trapping the agent into a local optimum of the action-value function. An example of an On-Policy algorithm is SARSA.

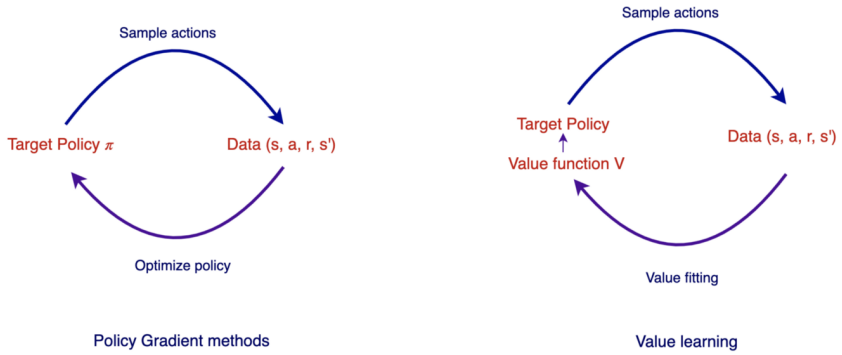

Figure 2.4: On-Policy learning with either policy gradients or value learning.

Off-Policy approaches have two different policies: a behavior policy and a target policy. The behavioral policy b is used for exploration, and the target policy π is used for function estimation and improvement. This works because the target policy π gets a "balanced" view of the environment and can learn from potential mistakes of b while still keeping track of the good actions and trying to find better ones. We say the target policy is **NOT** equal to the behavior policy (see Fig. 2.5). Some examples of Off-Policy learning algorithms are Q-Learning and Expected SARSA.

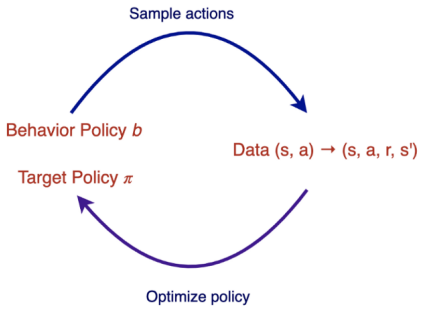

Figure 2.5: Off-policy learning with policy optimization.

2.2.3 Reward Function Design

The reward function is an incentive mechanism that tells the agent what is correct and what is wrong using reward and punishment. The goal of agents in Reinforcement Learning is to maximize the total rewards. In some cases designing a reward function is straightforward (i.e. if you have knowledge of the problem). For example, if we consider the game of chess.

There are three possible outcomes: win (good), loss (bad), or draw (neutral). Following that logic, we could reward the agent with $+1$ if it wins the game, -1 if it loses, and 0 if it draws.

However, in certain cases, the specification of the reward function can be a difficult task because there are many factors that could affect the performance of the RL agent. Considering the task of driving a car. In this scenario, there are many factors that affect the behavior of a driver. and it is difficult to incorporate these factors in a reward function.

Therefore designing a reward function is trial-and-error and an engineering process. Usually we define an initial reward function based on the knowledge of the problem, we then observe how the agent performs, then tweak the reward function to achieve better performance. It can be observed that the miss-specification of the reward function can have unintended consequences. To overcome this problem or improve the reward functions, there are methods such as:

- Learning from demonstrations (apprenticeship learning), i.e. do not specify the reward function directly, but let the RL agent imitate another agent's behavior, either to
  - learn the policy directly (known as imitation learning), or
  - learn a reward function first to later learn the policy (known as inverse Reinforcement Learning or sometimes known as reward learning)
- Incorporate human feedback in the RL algorithms
- Transfer the information in the policy, learned in another but similar environment, to your environment (i.e. use some kind of transfer learning for RL)

2.2.4 Learning Methods

This section will briefly explain the concepts of the Reinforcement Learning methods that will later be implemented for the automated DLS algorithm selection.

SARSA is an algorithm for learning a Markov decision process policy. The name simply reflects the fact that the main function for updating the Q-value depends on the current state of the agent S_1, the action the agent chooses A_1, the reward R the agent gets for choosing this action, the state S_2 that the agent enters after taking that action, and finally the next action A_2 the agent chooses in its new state. The pseude-code of the algorithm is listed in Alg. 1 and the action-value function in equation 2.1. [26]

$$Q(S_t, A_t) = Q(S_t, A_t) + \alpha(R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)) \tag{2.1}$$

Q-Learning is a model-free Reinforcement Learning algorithm to learn the value of an action in a particular state. It does not require a model of the environment, and it can handle problems with stochastic transitions and rewards without requiring adaptations. For any finite Markov decision process (FMDP), Q-learning finds an optimal policy in the sense of maximizing the expected value of the total reward over any and all successive steps, starting from the current state. Q-learning can identify an optimal action-selection policy

Algorithm 1: Pseudo-Code for SARSA Learning.

Input: policy π, positive integer *num_episodes*, small positive fraction α
Output: value function Q ($\approx q_\pi$ if *num_episodes* is large enough)
Initialize Q arbitrarily (e.g., $Q(s,a) = 0$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$)
for $i \leftarrow 1$ ***to*** *num_episodes* **do**
  $\epsilon \leftarrow \epsilon_i$
  Observe S_0
  Choose action A_0 using policy derived from Q (e.g., ϵ-greedy)
  $t \leftarrow 0$
  repeat
    Take action A_t and observe R_{t+1}, S_{t+1}
    Choose action A_{t+1} using policy derived from Q (e.g., ϵ-greedy)
    $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t))$
    $t \leftarrow t + 1$
  until S_t *is terminal*;
end
return Q

for any given FMDP, given infinite exploration time and a partly-random policy. "Q" refers to the function that the algorithm computes – the expected rewards for an action taken in a given state. The pseude-code of the algorithm is listed in Alg. 2 and the action value function in equation 2.2. [26]

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha(r_{t+1} + \gamma max_a Q(s_{t+1}, a) - Q(s_t, a_t)) \tag{2.2}$$

Algorithm 2: Pseudo-Code for Q-Learning.

Input: policy π, positive integer *num_episodes*, small positive fraction α
Output: value function Q ($\approx q_\pi$ if *num_episodes* is large enough)
Initialize Q arbitrarily (e.g., $Q(s,a) = 0$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$)
for $i \leftarrow 1$ ***to*** *num_episodes* **do**
  $\epsilon \leftarrow \epsilon_i$
  Observe S_0
  $t \leftarrow 0$
  repeat
    Choose action A_t using policy derived from Q (e.g., ϵ-greedy)
    Take action A_t and observe R_{t+1}, S_{t+1}
    $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t))$
    $t \leftarrow t + 1$
  until S_t *is terminal*;
end
return Q

Q-Learning vs. SARSA We will highlight the differences in learned agent behaviour by looking at an example environment called Cliffworld (see Fig. 2.6) - we can think of it as a playground for the agent. Cliffworld is an episodic task, with a start and goal state, and actions causing movement up, down, right, and left. The reward is given -1 on all transitions, except those into the region marked *Cliff*. Moving into this region awards the agent a score of -100 and resets the agent's position back to the start.

Q-Learning learns values for the optimal policy (no unnecessary steps taken), the path which travels right along the edge of the cliff. This leads to the agent sometimes stepping off the cliff because of the "epsilon-greedy" action selection. SARSA, on the other hand, takes the action selection into account and learns the longer but safer path through the upper part of the grid. Although Q-Learning actually learns the values of the optimal policy, its online performance is worse than that of SARSA, which learns the safer path. If ϵ were gradually reduced, then both methods would asymptotically converge to the optimal policy.

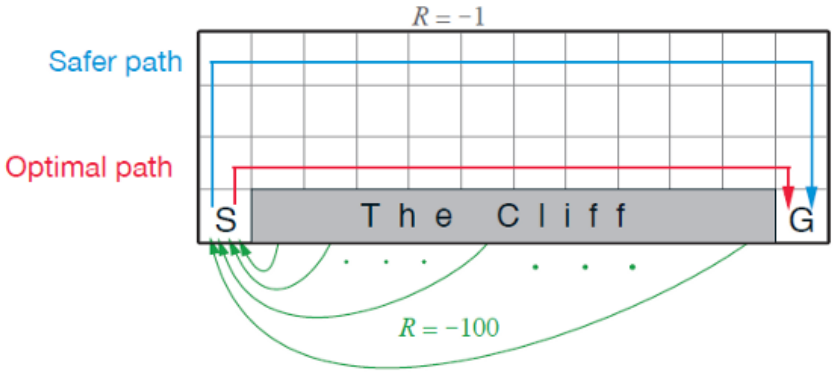

Figure 2.6: Comparison of paths taken by Q-Learning vs. SARSA in the Cliffworld example environment.

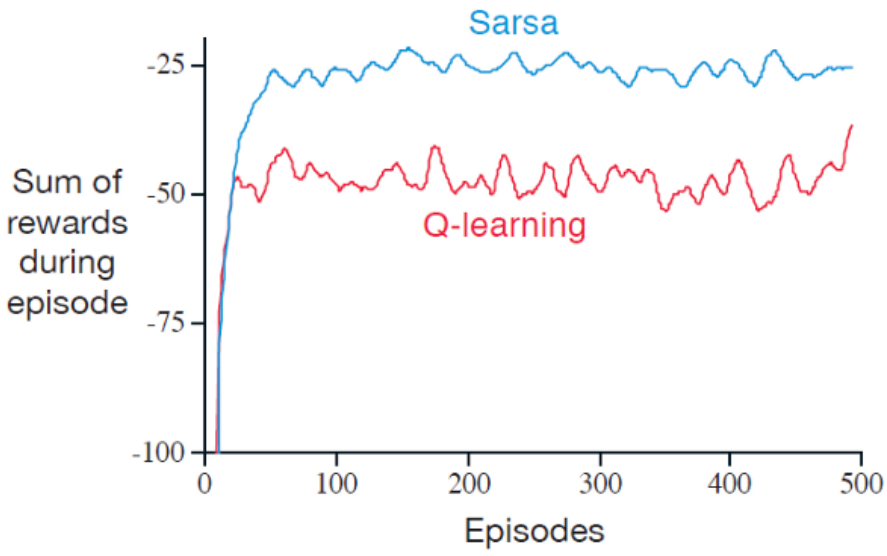

Figure 2.7: Comparison of rewards earned by Q-Learning vs. SARSA in the Cliffworld example environment.

Expected-SARSA is an alternative for improving the agent's policy. It is very similar to SARSA and Q-Learning, and differs in the action value function it follows (see equation

2.3). We have established that SARSA is an On-Policy and Q-Learning is an Off-policy technique. Expected-SARSA can be used either On-Policy or Off-Policy and is much more flexible. Expected SARSA takes the weighted sum of all possible next actions with respect to the probability of taking that action. If the Expected Return is greedy with respect to the expected return, then this equation gets transformed to Q-Learning. Otherwise Expected SARSA is On-Policy and computes the expected return for all actions. [26]

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha(r_{t+1} + \gamma \sum_a \pi(a|s_{t+1}) Q(s_{t+1}, a) - Q(s_t, a_t)) \tag{2.3}$$

DoubleQ-Learning is an Off-Policy algorithm that utilises double estimation to counteract overestimation problems with traditional Q-Learning. The max operator in standard Q-Learning uses the same values both to select and to evaluate an action. This makes it more likely to select overestimated values, resulting in overoptimistic value estimates. To prevent this, we can decouple the selection from the evaluation. For this purpose we maintain two Q-value functions Q_A and Q_B. In the update step we randomly select (with the same probability) either Q_A or Q_B. When updating Q_A we use the estimate from Q_B for the Q-value. When predicting the next action we consider both functions. [9]

$$Q\left(s_t, a_t\right) = Q\left(s_t, a_t\right) + \alpha_t\left(s_t, a_t\right)\left(r_t + \gamma \max_a Q\left(s_{t+1}, a\right) - Q\left(s_t, a_t\right)\right) \tag{2.4}$$

QV-Learning works by keeping track of both the Q-and V-functions. In QV-Learning, the state-value function V is trained with normal TD-methods. The adaptation from Q-Learning is that the Q-values are learned indirectly from the V-values using the one-step Q-Learning algorithm. The V-function converges faster to optimal values than the Q-function, since it does not consider the actions and is updated more often. Therefore, using QV-Learning, the Q-values can be learned and compared by the way an action in a state leads to different successor states. [29]

$$Q\left(s_t, a_t\right) = Q\left(s_t, a_t\right) + \alpha\left(r_t + \gamma V\left(s_{t+1}\right) - Q\left(s_t, a_t\right)\right) \tag{2.5}$$

3
Related Work

With the emergence of hyperthreaded and simultaneous multithreaded commodity hardware, the ability to parallelize applications grew at a fast pace. This lead to the realization that load imbalance is a major contributor to performance degradation and therefore needs to be addressed via (dynamic and adaptive) scheduling algorithms. It is agreed upon that no single statically selected algorithm yields the best overall performance ([2, 17–19]), instead there should be an automated system that selects the best scheduling strategy during execution according to some metric.

In [30], two scheduling techniques for applications executing on symmetric multiprocessors with simultaneous multithreaded processors are introduced: (1) a loop-based scheduler for individual loops (2) an off-line created hardware-counter directed scheduler using a decision tree. Similarly this work will also implement a selection process for every individual loop, but in contrast our decision logic is not created off-line to avoid the need for profiling.

Self-Scheduling is another important and commonly used principle when scheduling tasks on multiprocessors in multiprogrammed systems. Probabilistic Self-Scheduling is a self-scheduling algorithm proposed by *Girkar et al.* in [7]. Its goal is to minimize the run-time scheduling overhead by selecting an appropriate task size (chunks) based on the number of available (idle) processors and remaining iterations. This reduces the number of allocation points and in turn promises better performance. While not directly applicable to the present work, the observation of the system state (namely available processors) is an interesting proposition and can be adopted for the use with a reinforcement learning agent.

Using reinforcement learning to automatically select the most appropriate dynamic loop scheduling algorithm from a set of available algorithms is described in [20]. Our work will improve upon the findings in multiple ways. Instead of just the quantum trajectory method simulation for the performance evaluation we will use balanced, unbalanced and memory-bound kernels as well as two different scientific applications on native hardware. On top the results from [20] show that it did not matter weather the Q-Learning or SARSA algorithm was employed with the agent. This might stem from the fact that their time-stepping ap-

plications had to few time-steps or that the two learning algorithms are not the best fit for the problem. Instead in our work we focus on more recent learning algorithms.

Thoman et al. show in [28] that an automatic scheduling algorithm selection based on polyhedral compiler analysis can result in significant performance gains. They achieve this by generating an effort estimation function in combination with current runtime system behaviour monitoring. Opposing to [28] our work will not perform any sort of compiler analysis to improve the selection process - this has the advantage that the availability of the source-code is not a requirement. However as stated previously we will also employ the monitoring of the overall system state to estimate the fitness of a dynamic loop scheduling algorithm during runtime.

A similar approach to [20] is introduced by *Sukhija et al.* in [25]. The similarities lay in the portfolio-based selection of scheduling algorithms. The approach however differs in the machine learning technique that was applied. Supervised learning is used to build an empirical robustness prediction model. Performance evaluations were conducted with the SimGrid simulation framework. The usage of a robustness metric is an interesting proposition and is also applicable to our work - however we focus on performance evaluation on real HPC system instead of a simulated environment.

Adapting the insights from [25], *Boulmier et al.* devised a new robustness metric called "flexibility" to be used as reward input for a reinforcement learning agent to estimate the capability of a DLS technique to resist variations in the loop iteration's execution time. As in previous work, the performance was evaluated in a simulated environment using SimGrid.

Tying on to the research done in [2], a simulator assisted method called "Simulator in the Loop" (SiL) and also "Simulator-Assisted Scheduling Approach (SimAS) has been introduced in [17] and [18]. SiL is inspired by control theory, where a scheduler is used to achieve and maintain a desired load balance of the system by using the simulator to predict the performance of the system and then to dynamically select a DLS technique that maximizes application performance during execution.

Sreenivasan et al. proposes the concept of an autotuner in [24], which automatically determines the best combination of thread count, schedule type and chunk size associated with a loop based on a user defined search space. This is achieved by sampling every combination for a set period of time, then using the reported execution time for a loop to determine the best set of parameters. The downside of this method is the fast growth of the search space. In our work this is mitigated by assuming that the DLS method has the most impact on the applications performance.

A different approach to parallel performance optimization is taken in [11]. Instead of selecting a DLS method from a new set, the factoring self-scheduling (FSS) algorithm is extended to use Bayesian optimization. The new algorithm is appropriately called Bayesian

optimization augmented factoring self-scheduling (BO FSS) and achieves its performance improvements through solving an optimization problem. The tuning procedure only requires online execution time measurement of the target loop. BO FSS does not perform as well in some crucial scientific workloads that dynamically change during execution. We will investigate if reinforcement learning can overcome this weakness of BO FSS by selecting appropriate workloads.

Recently another supervised machine learning approach has emerged in [19]. The research explores a method called ADAPTIVELB which employs a K-Nearest Neighbors classifier to select the best suitable load balancing algorithm from the training set. It considers system workload information and communication among jobs during inference. The drawback of this method is the relatively high overhead during the inference step, which prevented the method from being the overall highest performing scheduling method during performance evaluation. In our evaluation we will record the execution time in and out of the loop to compare the overhead of the different automatic selection methods as well.

So far all of the above mentioned related work referred to speeding up parallel applications with dynamic loop scheduling. However similar algorithms exist for efficiently mapping processes with MPI in multi-/many core machines. This is predominantly interesting and challenging for cloud-based HPC systems where the architecture brings more diversity. [8] proposes two novel algorithms to construct efficient MPI mappings for any given architecture and application communication pattern. This is done by using low-level benchmarking utilities to extract machine information. Our goal is to use the same reinforcement learning agent as with the loop scheduling algorithm selection, but apply it to the process mapping problem to construct efficient communication graphs for MPI.

4

Implementation

In the following section we will discuss the architecture and the features of the Reinforcement Learning implementation (**RL4OMP**) for the scheduling algorithm selection problem as well as the libraries it builds on. The work presented in this thesis does not propose a new framework to improve thread-level scheduling with Reinforcement Learning. Rather it builds on, extends and improves an existing solution like the LB4OMP library - even though we took extra care to make sure our extension is lightweight, encapsulated, extensible and portable.

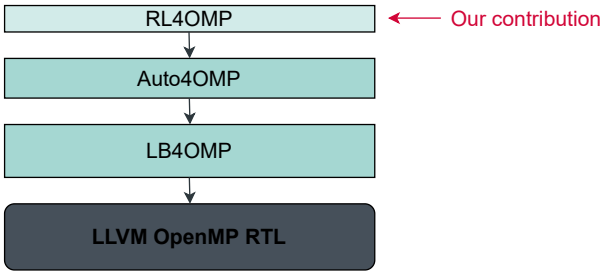

Figure 4.1: Showing the dependencies of RL4OMP in the "technology stack".

To understand how our Reinforcement Learning extension was implemented, we first need to understand the tools and libraries we employ to make the Reinforcement Learning work. The following sections are ordered according to their position in the "technology stack" (see Fig. 4.1).

4.1 The LB4OMP Library

LB4OMP extends the LLVM OpenMP RTL, which is widely used and compatible with various compilers (e.g. Intel, GNU, etc). Figure 4.2 shows the LB4OMP loop scheduling mechanism which extends the scheduling mechanism in the LLVM OpenMP RTL. The three main functions responsible for the chunk calculation are implemented in the file `kmp_dispatch.cpp`.

Upon initialization, each thread calls the `__kmp_dispatch_init_algorithm` function in-

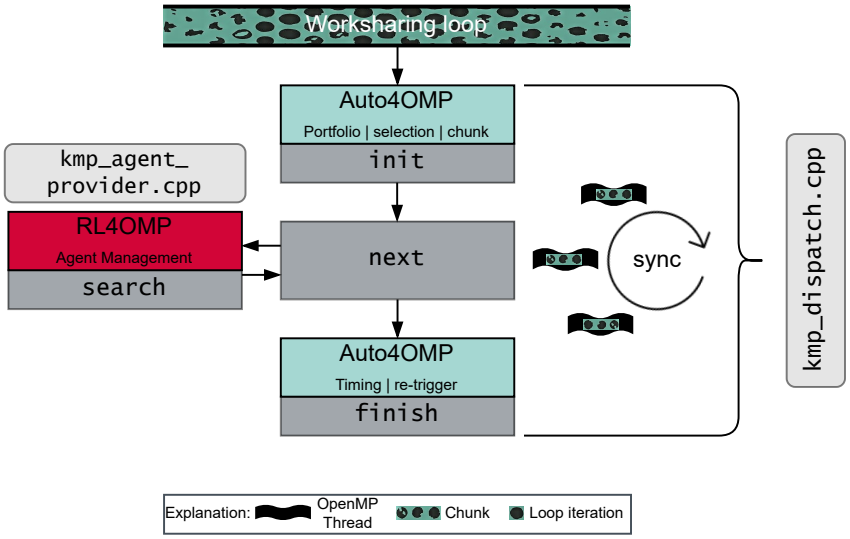

Figure 4.2: Extension of the OpenMP LLVM RTL scheduling process for work-sharing loops with LB4OMP and RL4OMP.

side the `kmp_dispatch.cpp` file (init in Fig. 4.2). This function then initializes the needed structures for the selected scheduling technique and calls `__kmp_dispatch_next_algorithm` (next in Fig. 4.2). The logic of the chunk calculation of all DLS techniques is implemented in the `__kmp_dispatch_next_algorithm` function. The `__kmp_dispatch_next_algorithm` is called each time a thread needs to obtain work. Since the threads obtain work from a shared queue, `__kmp_dispatch_next_algorithm` relies on different synchronization operations (sync in Fig. 4.2) depending on the scheduling technique in execution. Finally, the threads call `__kmp_dispatch_finish` (finish in Fig. 4.2) to reset variables or free allocated memory. The OpenMP standard scheduling techniques and the newly implemented scheduling techniques in LB4OMP support the declaration of a chunk parameter which bears different meanings among the scheduling techniques. For `schedule(static,chunk)` and `schedule(dynamic,chunk)`, the chunk parameter denotes the amount of iterations that the threads should receive for every work request. For the other techniques, the chunk parameter works as a threshold. When chunks sizes calculated by a scheduling technique fall below this threshold, they will be replaced by a chunk sizes equal to the size of the chunk parameter. The chunk parameter was introduced by the OpenMP standard to minimize the scheduling overhead and to improve data locality.

4.2 The Auto4OMP Extension

Auto4OMP is designed to address the scheduling algorithm selection problem in OpenMP. It leverages the existence of `auto` as a scheduling option in OpenMP and extends its implementation in the LLVM OpenMP runtime library with expert chunk selection, a portfolio of loop scheduling algorithms, and algorithm selection methods. The portfolio is an important concept which we will reuse in our Reinforcement Learning implementation. Scheduling algorithms are added to the portfolio if they posses certain characteristics in order to help reduce the search space and cost for automated selection methods. The Auto4OMP portfolio

includes the following 12 algorithms, sorted in ascending order of their scheduling overhead and load balancing capacity: `STATIC`, `SS`, `GSS`, `GAC`, `TSS`, `Static Steal`, `mFAC2`, `AWF-B`, `AWF-C`, `AWF-D`, `AWF-E`, and `mAF`. By default, Auto4OMP uses the expert chunk parameter. For our performance analysis campaign we will not use this feature and set the parameter `KMP_Golden_Chunksize=0`. Using the expert chunk feature has proven favourable for the performance of the combination of computing node and benchmark application chosen in our work. Leaving the option enabled leads to a similar parallel execution time for many scheduling algorithms which will not convey the true performance of the Reinforcement Learning agent.

4.3 The RL4OMP Extension

RL4OMP was created to further improve the ability of OpenMP to select the best available scheduling algorithm automatically, using Reinforcement Learning. This should in principle dismiss the need for expert knowledge to be transformed and hard-coded into the library's source code.

This extension is written in Object-Oriented C++14. For this thesis the source files were added to the `CMakeLists.txt` file of the LB4OMP extension for easier compilation. But the extension can be compiled stand-alone. Every component of the extension was designed to be extensible and as much logic as possible is contained within the abstract classes (see Fig. 4.3) in order for the implementation classes to be as small as possible. The **AgentProvider** class is used to pass information back and forth between the loop scheduling library and the RL4OMP extension. The class contains two static maps. The maps keep track of the elapsed time-steps and the agents for each loop. We use the loop name provided by the LLVM OpenMP runtime as the key. The static class method `AgentProvider::search` is called with every scheduling round (call to `__kmp_dispatch_next_algorithm`) and either creates a new agent for the corresponding loop name, or looks up the existing agent from the map and passes pointers to the data structures in order for the agent to learn and take an action. The selected action (next scheduling algorithm) is the only value returned to the LLVM OpenMP runtime. When a new agent gets created, the base class constructor gets called first which initialises the required fields and reads all the supported configurations from the environment. The specific implementation of an agent only implements the `Agent::update` function and any additional fields if needed. The base class **Agent** also takes care of instantiating the different sub-components like initialisers, policies or reward functions and using them during runtime.

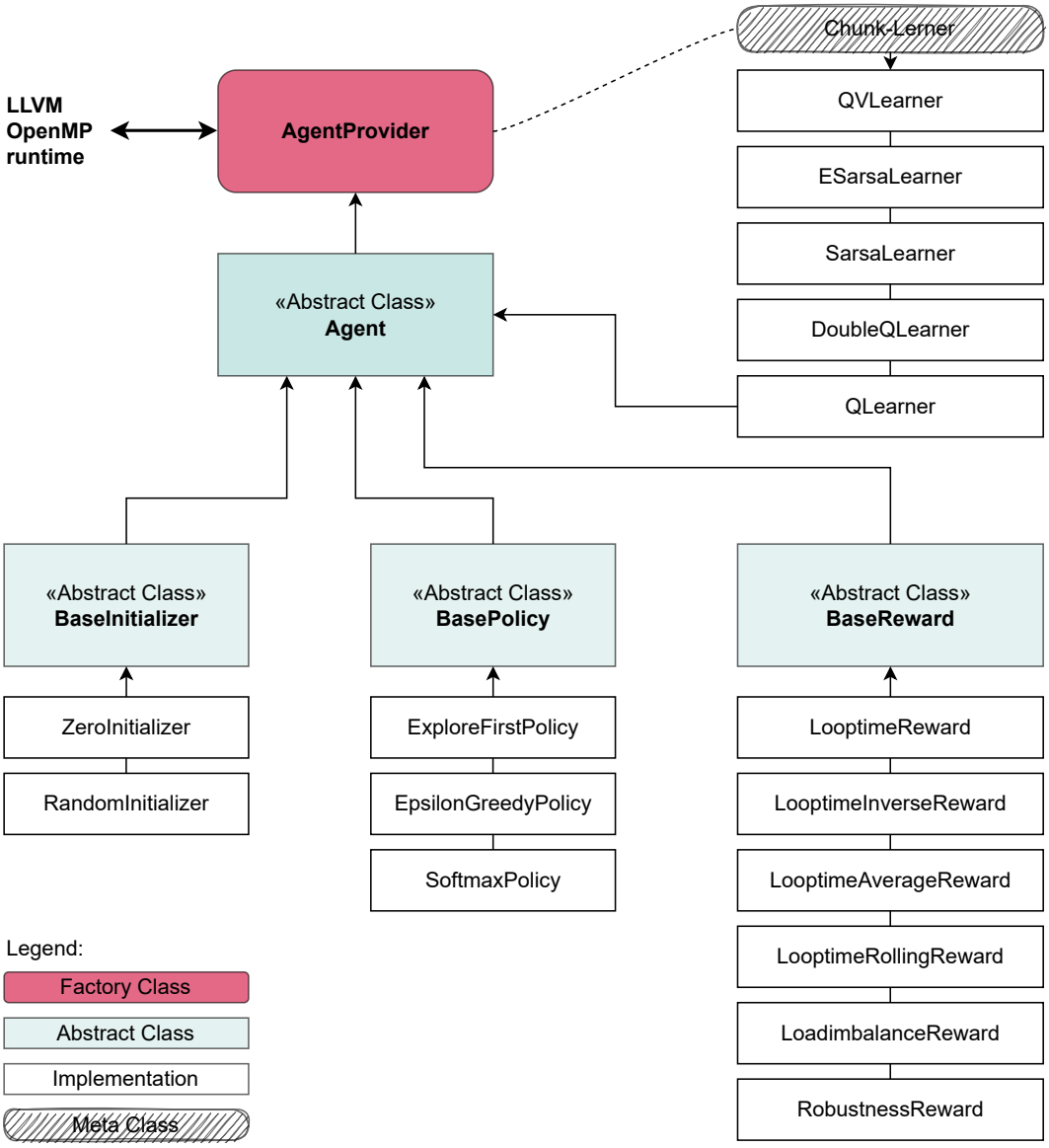

Figure 4.3: UML-Diagram of the RL4OMP extension's architecture and components.

The implementation for the Reinforcement Learning (RL) extensions hooks into the existing dispatch mechanism established by LB4OMP and Auto4OMP. It works by hijacking the chunk parameter for the `auto` schedule keyword (e.g. `auto,8`). While the Auto4OMP extensions reserves parameters 2 to 5 (`RandomSel`, `ExhaustiveSel`, `BinarySel` and `ExpertSel`) for itself, the RL extension uses parameters 8 to 15 (assignments listed below). When the LLVM's OpenMP runtime dispatcher encounters a chunk parameter belonging to the RL extension, it forwards the call to the newly created `AgentProvider` class[1]. The class-method `search` (see Fig. 4.2) serves as the only interface between the LLVM's OpenMP runtime and the Reinforcement Learning extension. It takes care of instantiating

[1] In this chapter when we talk about "classes", we mean it in the sense of Object Oriented Programming and not in the sense of a group of algorithms or things.

new agents and their sub-components, as well as keeping track of which agent belongs to which application loop. This minimizes the pollution of the existing codebase and facilitates extensibility and portability. When creating a new agent, a user is able to choose from the following learning methods (which also constitutes its own agent type):

- 8 → Q-Learning
- 9 → DoubleQ-Learning
- 10 → SARSA
- 11 → Expected-SARSA
- 12 → QV-Learning
- 15 → Chunk-Learning (special case)

How the agent types (8 - 12) are different in terms of their in learning behaviour, has been discussed in chapter 2. The Chunk-Learning agent (option 15) is a special case - we can think of it as a meta-agent. In itself the Chunk-Learning does not implement a classic learning algorithm from literature. Rather it wraps the other learning methods in such a way, that the output of its decision is not the next scheduling algorithm but the chunk-size directly. We use the chunk-size found by the agent in conjunction with the `dynamic` schedule to directly influence how many iterations we would like to get from the work-sharing loop. Theoretically we could let the agent try every chunk-size from $1..n/p$. This leads to a huge state-action space which performs poorly in time and space. Therefore we limit the available choices of chunk-sizes (for our implementation we choose to have the same number of choices as there are algorithms in the portfolio = 12). Equation 4.1 describes how the chunk-size C is derived. The total number of iterations in a loop is denoted by n, the numbers of available threads by p and x represents the desired size of the search space.

$$C_i = \frac{n}{2^i * p}, \quad i \in [1..x] \tag{4.1}$$

For all agents apllies, that besides the learning method for the agent (which governs the update process of new experiences to the agent's internal state), there are 3 sub-components to an agent which can be fine tuned by the user as well: **(1)** the initializer for the internal data structure, **(2)** the policy that selects the next action and **(3)** the function which calculates the agent's reward.

(1) Initializers For the initializer there are two choices - the `ZeroInitializer` class which sets all the values of the tabular data structure of the agent to 0 and the `RandomInitializer` class that sets the values to a random floating point number between the minimum and maximum reward value. [14]

(2) Policies The RL extension provides 3 options for the agent's policy. the `ExploreFirstPolicy` class selects every state-action pair at the beginning of the learning process exactly once in sequential order. After that, the `ExploreFirstPolicy` class acts greed-

ily. `EpsilonGreedyPolicy` class uses a random number generator to decide between exploitation (greedy behaviour) and exploration. Exploration is chosen with a probability of $1 - \epsilon$ and the greedy action is chosen with a probability of ϵ. This policy can be tuned with the two following environment variables: `KMP_RL_EPSILON` (initial value) and `KMP_RL_EPS_DECAY` (decay factor). The last available policy class is the `SoftmaxPolicy`. While the first two policies would look for the highest Q value (in the greedy case) when deciding the action, here the probability of selecting an action increases with a high Q value. This policy can be tuned with the `KMP_RL_TAU` environment variable. τ is also called the temperature. A high temperature increases the probability to select an action associated with a high Q value even further, while a low temperature arranges for the probabilities to be more uniformly distributed.

(3) Rewards Further we have several choices regarding the reward function. As discussed in chapter 2, the reward function is an important part when implementing a domain specific Reinforcement Learning problem. Since it is difficult to have an intuition for the behaviour of an agent to a specific reward signal, we implemented a wide range of different reward functions to test with a real world scientific applications. The implementations for the rewards generally follow the same principle across all classes (compare equation 4.2).

$$\mathcal{R}_t(x) = \begin{cases} r_+ & x \leq \min\limits_t(x) \\ r_0 & \min\limits_t(x) < x < \max\limits_t(x) \\ r_- & \max\limits_t(x) \leq x) \end{cases} \tag{4.2}$$

From the above equation we can gather that depending on the input x (which is usually the parallel execution time), the agent can be given 3 different values as the reward - we can think of them as good, neutral and bad values. To distinguish between these 3 cases we keep track of the min and max for the input x across all time-steps. The following reward function classes are available:

- `LooptimeReward`: Award is given according to the last thread finishing time for a time-step.
- `LooptimeInverseReward`: The inverse of the parallel execution time for a time-step is given as an award $\mathcal{R}_t(t_{PAR}) = 1/t_{PAR} * c$. This reward function is an exception! There are no distinct 3 cases for the reward value. This reward function can be customized with the `KMP_RL_INVERSE_REWARD_MULT` environment variable. It provides a value for the factor c and is used to combat small values.
- `LooptimeAverageReward`: An award is given based on the parallel execution time of the current time-step being above or below the running average for the execution time. This reward function is an exception! There are no distinct 3 cases for the reward value, only 2 ($\{r_+, r_-\}$).
- `LooptimeRollingAverageReward`: An award is given based on the parallel execution time of the current time-step being above or below the rolling average for the

execution time. This reward function can be customized with the KMP_RL_ROLLING_AVG_WINDOW environment variable. It determines the amount of previous time-steps are taken into account when calculating the average. This reward function is an exception! There are no distinct 3 cases for the reward value, only 2 ($\{r_+, r_-\}$).

- `LoadimbalanceReward`: Award is given based on the percent load imbalance (LIB) after the current time-step. $LIB = (1 - \frac{\text{mean of thread finishing times}}{\text{max of thread finishing times}}) * 100$
- `RobustnessReward`: Award is given based on the robustness metric after the current time-step. [2]

The class `AgentProvider` is designed to work as a factory that creates the right `Agent` class depending on the value of the `OMP_SCHEDULE` variable. The constructor of the `Agent` class reads all the environment variables (specified in listing 2 in section A.1) and initializes and configures all sub-components accordingly. During runtime, the learning rate α and the exploration rate ϵ are decayed exponentially according to their decay rate after each time-step. When the `ExploreFirstPolicy` is used, the decay process is delayed until the initial exploration phase is complete. Decaying these values ensures that the agent gradually uses more and more of its existing knowledge rather that learning for the entire duration of the application run. To prevent this behaviour, the user can set the decay factor to a value of 1. The initial values for all the agent's parameters are specified in listing 1 below. When not overwritten via the environment or a job-file, this way a scientific application can be run with minimal setup required.

```
1  namespace defaults {
2      const double SEED                 = 420.69f;//Random number generators
3
4      const double ALPHA                = 0.85f; // Initial learning rate
5      const double ALPHA_MIN            = 0.10f; // Stop learning rate decay
6      const double ALPHA_DECAY_FACTOR   = 0.01f; // Decay for learning rate
7      const double GAMMA                = 0.95f; // Initial discount factor
8      const double EPSILON              = 0.90f; // Initial exploration rate
9      const double EPSILON_MIN          = 0.10f; // Stop learning rate decay
10     const double EPSILON_DECAY_FACTOR = 0.01f; // Decay for exploration
11     const double TAU                  = 1.50f; // Temperature for softmax
12
13     const int CHUNK_TYPE              = 8;     // Type for chunk learner
14
15     const int ROLLING_AVG_SIZE        = 10;    // Sliding windows size
16     const int INVERSE_REWARD_MULT     = 10;    // Inverse reward multiplier
17
18     const RewardType REWARD_TYPE      = RewardType::LOOPTIME;
19     const InitType INIT_TYPE          = InitType::ZERO;
20     const PolicyType POLICY_TYPE      = PolicyType::EXPLORE_FIRST;
21     const DecayType DECAY_TYPE        = DecayType::EXPONENTIAL;
22
23     const std::string REWARD_STRING   = "0.0,-2.0,-4.0"; // r+, r0, r-
24 }
```

Listing 1: Default values for Reinforcement Learning agents (contained in its own namespace).

4.4 Usage

This section will provide a short overview on how to use the Reinforcement Learning selection method with a time-stepping application.

The usage of the Reinforcement Learning technique in any OpenMP application is designed to be as effortless as possible and explained in Fig. 4.4. As an initial step, the target OpenMP loops in the application must contain the `schedule(runtime)` clause. If this prerequisite is already satisfied, no further changes to the application's source code are required. Otherwise the existing scheduling clause needs to be altered to `runtime` in all target loops and the application must be recompiled. Further, the path to the compiled LB4OMP library has to be added to the environment variable that the linker uses to load dynamic and shared libraries from (e.g. `LD_LIBRARY_PATH` on UNIX/LINUX systems). Additionally the host CPU clock frequency as a system-related parameter is required. This is passed to LB4OMP via the environment variable `KMP_CPU_SPEED` as an integer in Megahertz.

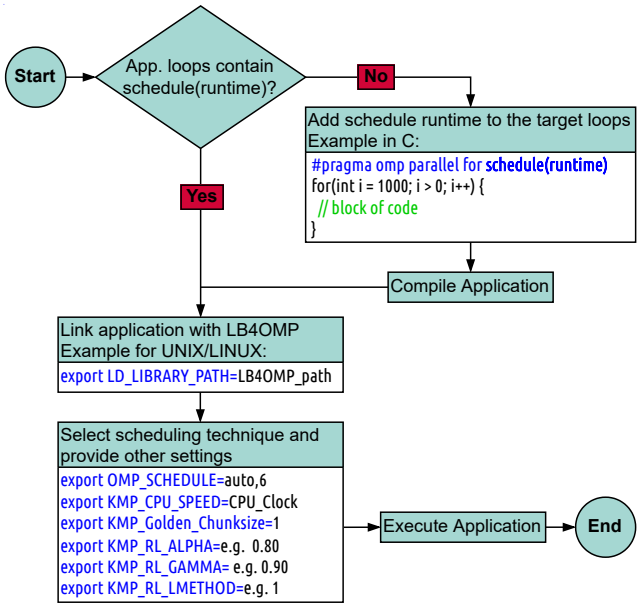

Figure 4.4: Showing the workflow for how to connect a time-stepping application to the Reinforcement Learning extension.

5
Benchmark & Results

In this chapter we discuss the experimental setup and results of the two benchmark applications that were used to measure the parallel performance of our proposed Reinforcement Learning extension.

As discussed in the previous chapter, the extension implements several different learning agents. Additionally the agent is composed of several sub-components which can be configured to the users needs via environment variables (see listing 2 in the appendix for all the available options). In table 5.1 we assembled the details of our factorial experiments for two different scientific time-stepping applications to asses the performance (parallel execution time) of different scheduling algorithms against the automated DLS algorithm selection methods - such as Auto4OMP or RL4OMP. From this table of experiments we derived 11 different configurations (specified in table 5.2) for the Reinforcement Learning agent, for which the performance has been evaluated. Running the 2 application with all the agents and reference methods for 11 different configurations, and repeating every run 5 times for consistency, resulted in over 1'440 experiments conducted in total, for which the log data have been collected and stored. On top of that we gathered the ground-truth for each application with respect to the portfolio of scheduling algorithm.

The factorial experiments add up to 70 individual runs per configuration. Each configuration only changes one aspect of the agent and its sub-components. We forewent the possibility to generate additional experiments by combining different settings for the sub-components of the agent. The evaluation process would become increasingly difficult and the source of the effect on the performance might be hard to trace. In table 5.2 we also describe what the anticipated effect of the different configurations and the difference to the default configuration is.

The experiments show that for a constant load-imbalance like the Mandelbrot application, the Q-Learning agent performs the best but cannot outperform some of the best automated DLS selection methods from previous work. The Chunk-Learn agent however achieves the performance closest to ground-truth for the Mandelbrot application. In the SPHYNX Evrard Collapse experiment the Chunk-Learn agent again outperform every other agent of the RL4OMP extension but falls behind automated DLS selection methods from Auto4OMP.

Table 5.1: Design of factorial experiments for the performance evaluation of the Reinforcement Learning extension to LB4OMP.

Factors		**Values**		**Properties**
Applications		Mandelbrot	OpenMP only	N = 262,144 T = 500 Total loops = 3, Modified loops L = 3
		SPHYNX Evrard Collapse	OpenMP only	N = 1,000,000 T = 400 Total loops = 37, Modified loops L = 2
Scheduling techniques		static		Straightforward parallelization
		ss		Dynamic and non-adaptive self-scheduling techniques
		gss		
		gac		
		tss		
		static steal		
		mfac2		
		awf-b		Dynamic and adaptive self-scheduling technique
		awf-c		
		awf-d		
		awf-e		
		maf		
Selection Methods	Expert	RandomSel	(auto,2)	Automated DLS algorithm selection
		ExhaustiveSel	(auto,3)	
		BinarySel	(auto,4)	
		ExpertSel	(auto,5)	
	Reinforcement Learning	Q-Learning	(auto,8)	Automated DLS algorithm selection
		DoubleQ-Learning	(auto,9)	
		SARSA	(auto,10)	
		Expected-SARSA	(auto,11)	
		QV-Learning	(auto,12)	
		ChunkLearning	(auto,15)	
Reinforcement Learning Initializer		Zero		-
		Random		-
Reinforcement Learning Reward Metrics		Loop Time		-
		Loop Time Inverse		-
		Loop Time Average		-
		Loop Time Rolling Average		-
		Percentage Loadimbalance		-
		Robustness		-
Reinforcement Learning Selection Policies		Explore First		-
		Epsilon Greedy		-
		Softmax		-
Chunk Parameter		Standard Chunk		-
Computing nodes		miniHPC-Broadwell		Intel Broadwell E5-2640 v4 (2 sockets, 10 cores each), P=20 cores w/o HT, Pinning: OMP_PLACES=cores, OMP_PROC_BIND=close

The LB4OMP library and the applications were compiled with the Intel compiler version 2019/a on the miniHPC-Broadwell cluster. We denote with the number of total iterations of the application with N, the amount of time-steps with T and he number of loops for which we modified the schedule clause with L.

Table 5.2: Explanation of the experiment configuration titles.

Configuraiton	**Description**
All Defaults	See listing 1 for default settings for agents.
Random	Uses the random intizializer for the agents internal data structure. The randomness could help the agent overcome its own learning bias and help select actions that have a higher reward.
Looptime Inverse	Instead of having distinct reward values, we return the inverse of the parallel execution time as a reward to the agent. The thought behind the principle is twofold: We use the dimension (parallel execution time) the agent should maximize as the actual reward. A small execution time results in a big reward and vise-versa.
Looptime Average	Instead of min and max, we track the average of the parallel execution time. This will help in a situation where early in the learning process the agent encountered a very extreme (fast or slow) parallel execution time and all subsequent rewards will mostly be neutral (not punishing or rewarding the agent enough for the chosen action).
Looptime Rolling	Instead of min and max, we track the average of the parallel execution time. But the average is only collected for the $x = 10$ last time-steps. The same principle as above applies, but for applications and system with strongly varying loadimbalance throughout the execution of the application this reward function might perform better.
Neutral Reward	Rather than setting the reward range to mostly negative values $[0, -2, -4]$, we set the reward range to $[+2, 0, -2]$. Offsetting the reward values might help the policy optimisation process and escape local extrema.
Positive Reward	Rather than setting the reward range to mostly negative values $[0, -2, -4]$, we set the reward range to $[+4, +2, 0]$. Same as above.
Loadimbalance	The standard reward of looptime gets replaced by the percentage loadimbalance (formula presented in section 2.2).
Robustness	The standard reward of looptime gets replaced by an adapted robustness metric (formula also presented in section 2.2).
Epsilon Greedy	Epsilon-Greedy is an alternate policy that relies on probability to explore new actions.
Softmax	The Softmax policy is an alternate policy that assigns a soft probability to every action instead of acting greedy. This increases the the chance for the second or third best action to be chosen as well, which might not yield the highest reward in the short term but increase the overall earned rewards.

5.1 Mandelbrot

This section describes the properties and results for the benchmarks of the modified Mandelbrot application. The application can be launched with a set of input parameters (`./mandelbrot.o maxiter [pixels x0 y0 size]`) which mainly increase the applications runtime, since Mandelbrot is able to run ad infinitum. The actual input parameters chosen for the test runs are: `./mandelbrot.o 1000 512 0 0 0.5`. The application also has been modified in such a way that it runs for 500 time-steps, which should give the agent enough time to learn and the exploit its knowledge in a meaningful way. All the experiments were run 5 times with the same configuration on the miniHPC-Broadwell of the university, to make sure the variance of the results is in an acceptable range.

5.1.1 Ground-truth

The ground-truth (GT) is the theoretical best scheduling method and serves as a base for comparison for the other scheduling algorithms. We construct the ground-truth by running the chosen scientific application 5-times with every scheduling algorithm from the portfolio defined in section 4.2. We then take the mean of the 5 runs and look for smallest parallel execution time amongst all the methods in the portfolio for every time-step. The sum of the smallest means over all the time-steps is the new theoretical best parallel execution time for any automated DLS algorithm method. The ground-truth should only be used to compare scheduling methods that use the same portfolio of algorithms.

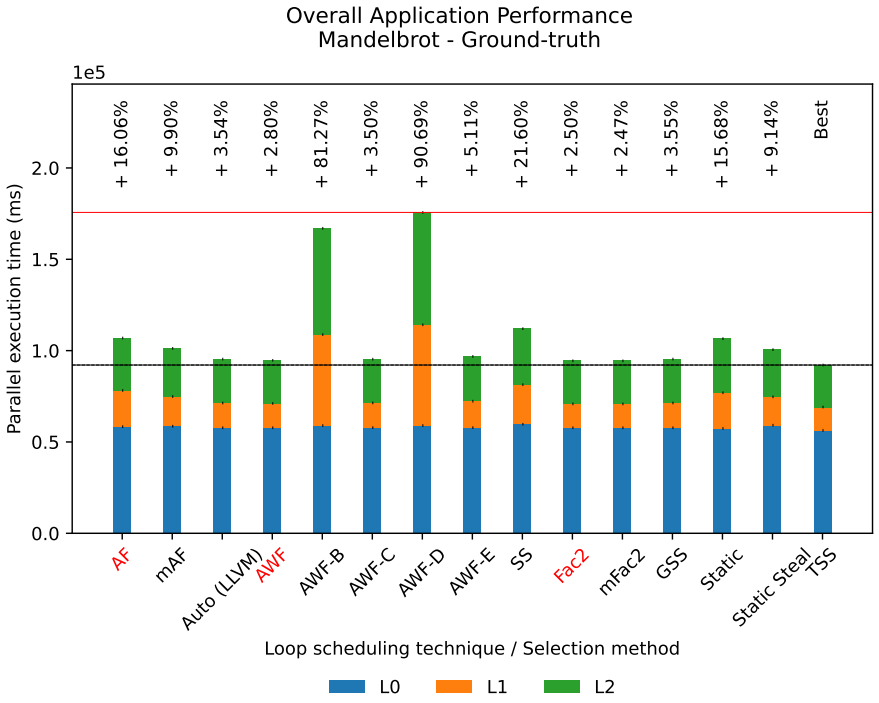

Figure 5.1: Results for the ground-truth for the Mandelbrot application. The numbers presented in the graph are the mean collected over 5 runs for every scheduling algorithm in the portfolio. Labels in red on the x axis represent scheduling algorithms not in the portfolio serving as comparison.

From Fig. 5.1 we observe that scheduling algorithm like *AWF, Fac2, mFac2* and *TSS* achieve a good performance when selected manually. We expect the automated selection methods to ideally choose the same algorithms as well.

5.1.2 Results

The following section presents the results for the Mandelbrot experiments. Fig. 5.2 and Tab. 5.3 show an overview of the all benchmarked methods and configurations. We want to show how the automated DLS selection methods from RL4OMP react to different configurations and also learn what the best configuration is. Additionally we compare them to other methods and algorithms. These results are presented in tabular form in Tab. 5.3 and as a line chart in Fig. 5.2. Bar plots with the results from a single configuration are available as well. The plot for the *All Defaults* configuration is included in this section in Fig. 5.3. The remaining plots are included in section A.2 in the appendix.

Table 5.3: Results for the benchmarks with the Mandelbrot application. The table shows the percentage difference in runtime against the ground-truth for different experiment configurations. Configurations are listed as rows and selected scheduling methods are shown as columns. A green cell-color indicates a performance closer to ground-truth, while a red cell-color indicates an overall bad application performance result. The graphs for all the results are located in the appendix in section A.2.

Selection Method / Configuration	RandomSel	ExhaustiveSel	BinarySel	ExpertSel	Q-Learn	DoubleQ	SARSA	ESARSA	QV-Learn	Chunk-Learn	Ground-truth
	Auto4OMP Reference				RL4OMP Extension						
All Defaults	23.30 %	16.11 %	52.61 %	13.72 %	20.40 %	22.05 %	22.75 %	21.63 %	21.23 %	2.21 %	0.00 %
Random	24.39 %	15.85 %	47.32 %	15.09 %	52.01 %	16.41 %	20.77 %	19.97 %	19.72 %	4.71 %	0.00 %
Looptime Inverse	22.14 %	15.34 %	48.37 %	14.44 %	13.06 %	21.37 %	20.12 %	20.67 %	20.73 %	8.67 %	0.00 %
Looptime Average	24.04 %	15.15 %	56.45 %	13.80 %	17.82 %	16.13 %	13.87 %	16.38 %	19.79 %	2.02 %	0.00 %
Looptime Rolling	24.20 %	16.03 %	49.33 %	15.44 %	20.30 %	21.62 %	10.04 %	16.06 %	19.87 %	2.03 %	0.00 %
Neutral Reward	22.26 %	16.14 %	50.91 %	12.69 %	12.50 %	19.64 %	20.05 %	19.58 %	14.38 %	8.33 %	0.00 %
Positive Reward	22.22 %	15.76 %	47.89 %	14.42 %	13.22 %	21.15 %	20.97 %	21.53 %	21.00 %	9.27 %	0.00 %
Loadimbalance	24.44 %	16.74 %	51.38 %	14.84 %	20.50 %	23.05 %	22.78 %	22.80 %	22.41 %	2.09 %	0.00 %
Robustness	26.50 %	14.98 %	55.85 %	13.77 %	12.97 %	20.55 %	21.51 %	20.85 %	20.15 %	8.95 %	0.00 %
Epsilon Greedy	24.04 %	16.74 %	54.92 %	13.12 %	20.55 %	22.22 %	22.07 %	21.79 %	21.27 %	1.76 %	0.00 %
Softmax	21.66 %	15.94 %	56.17 %	14.26 %	21.05 %	24.12 %	19.40 %	19.98 %	20.14 %	4.69 %	0.00 %
Average	23.56 %	15.89 %	51.93 %	14.14 %	20.40 %	20.76 %	19.48 %	20.11 %	20.06 %	4.98 %	0.00 %

When looking at the performance of the scheduling techniques in Tab. 5.3, we can see that on average the top three performing automated selection methods are Chunk-Learn, ExpertSel and ExhaustiveSel. Even though the Chunk-Learner's performance degrades under some configurations (low=1.76%, high=9.27% above GT), it can compete with or outperform state-of-practice scheduling algorithms like *Auto (LLVM)* or *AWF* for the time-stepping Mandelbrot application. This is certainly the case with the default configuration. The other learning agents from the RL4OMP extension cannot replicate the same performance and on average perform ∼20% above ground-truth. In comparison: Auto4OMP automated selection methods perform at best 14.14% above GT, while the worst case lies at 51.93%. When

analysing the performance of RL4OMP beyond the default configuration, it is apparent that most of the methods are able to slightly profit from the random initialization feature, but Q-Learn's performance takes a hit. This can be explained due to Q-Learning's tendency to find a less optimal policy faster and therefore not being able to counteract an unfavourable random initialisation. While all the methods do not really gain any significant performance with the *Looptime Inverse* reward configuration, the Q-Learn agent shows an above average improvement in this configuration. The performance could eventually be further improved by adjusting the scale factor $c = 10$. When looking at the two average reward configurations (*Looptime Average* and *Looptime Rolling Average*), the SARSA agent's performance could be improved significantly over its default configuration. An explanation for this behaviour is not easy to give - as SARSA is described to be more "conservative" than Q-Learning, the average reward might actually play well together with the learning behaviour of this agent. Offsetting the reward range (from mostly negative to mostly positive values) plays well together with the Q-Learn and QV-Learn agents, but the best performing agent (Chunk-Learn) so far does not benefit from this configuration. The *Loadimbalance* and *Robustness* reward configurations are a mixed bag. The two best performing methods from RL4OMP (Q-Learn and Chunk-Learn) show either no improvement over its base performance or contradictory reaction to the *Robustness* reward metric. Which is especially confusing when we consider that Chunk-Learn uses Q-Learning under the hood. The *Epsilon Greedy* and *Softmax* configuration do not have a huge positive or negative influence on the performance of the Mandelbrot application.

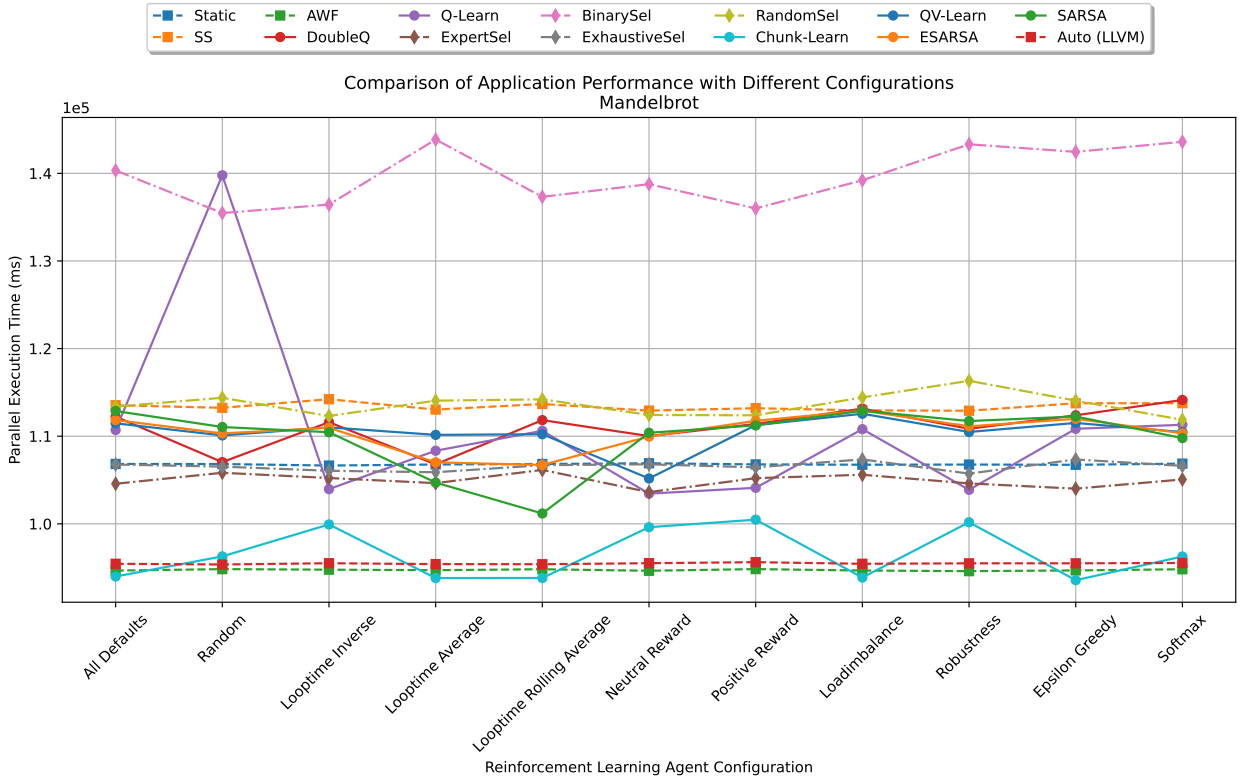

Figure 5.2: We show the results of the Mandelbrot application runs with $t = 500$ time-steps on the miniHPC-Broadwell cluster. The continuous lines with the ● markers represent results for the automated selection techniques implemented in RL4OMP. The dash-dotted lines with the ♦ markers depict the results for the automated selection methods from previous work (Auto4OMP), and the dashed lines with the ■ markers present the results for selected scheduling algorithms as a reference. The x axis lists the different configuration from table 5.2 while the y axis denotes the parallel execution time in milliseconds.

Fig. 5.3 lists the results from the *All Defaults* configuration in more detail. On the x axis the different scheduling methods are listed, on the y axis the parallel execution time in milliseconds is shown.

We observe that the four methods from Auto4OMP (*RandomSel, ExhaustiveSel, BinarySel* and *ExpertSel*) perform between 13.72% and 52.61% slower than ground-truth. Our automated DLS selection methods based on Reinforcement Learning perform between 2.21% and 22.75% slower than GT - with Chunk-Learn being the most effective method. As the performance of these 5 agents (Q-Learn, DoubleQ, SARSA, ESARSA and QV-Learn) lie close together, it confirms our initial assumption that the learning method is not the most important factor for optimizing the parallel execution time. Rather the configuration of the agent (i.e. initialization, reward function and policy) plays a much more significant role. The best result is achieved by the Chunk-Learn selection method. It makes intuitive sense that this method is able to perform better than the other automated selection methods. Instead of having the to interact with a black-box that selects the chunk-size, the agent can directly select and learn the best chunk-size on its own. A more detailed explanation is given in the next section.

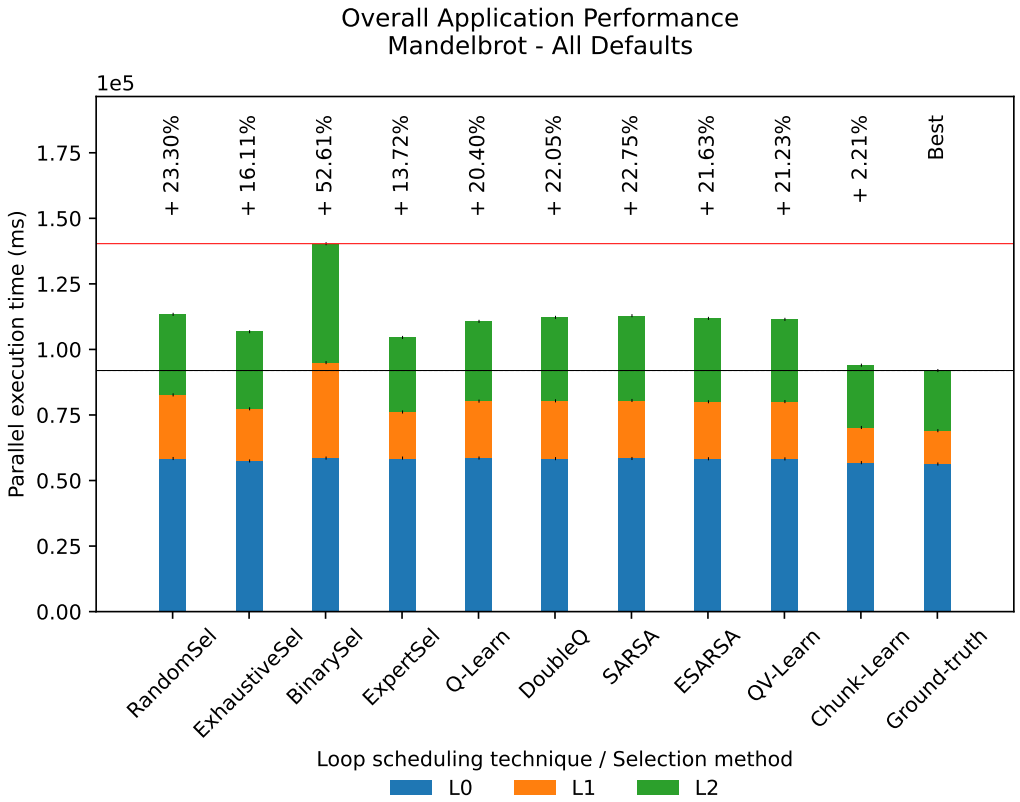

Figure 5.3: Shows the parallel execution time for each modified application loop on the y axis and the different scheduling methods on the x axis. Considering RL4OMP, *Chunk-Learn* is the only method that is close to in terms of performance (2.21%).

As a next step after looking at the performance results for the Reinforcement Learning

extension, it is interesting to inspect the agent's behaviour to be able to explain the results better and also understand the differences between the automated DLS selection methods. Figure 5.4 shows different aspects of the agent's behaviour.

The following figure (Fig. 5.4 on the next page) shows plots for every application loop ($L0, L1, ...$) and every automated DLS selection method benchmarked plus the ground-truth. A single sub-plot shows the time-steps on the x axis and the loop time in milliseconds on the y axis. The different colored bars represent the individual time-steps. The color of the bar indicates (in accordance with the legend on top of the plot) which scheduling algorithm from the portfolio the automated method chose for that particular time-step. Lastly in the top right of each sub-plot we listed the top 3 most selected scheduling algorithms for that particular selection method along with its percentage from the total time-steps.

Comparing the ground-truth for each loop for the Mandelbrot application to the automated selection methods, we see that only ExhaustiveSel from Auto4OMP has correctly identified the best performing scheduling algorithm `TSS` in their selection in two out of three loops. Even though ExhaustiveSel (15.89% above GT) took actions that are more in line with the ground-truth, ExpertSel is the automated selection from Aut4OMP with the best average performance of 14.14% above GT. Our Reinforcement Learning methods perform on average 19.48% to 20.76% above GT, with the low at 10.04% and the high at 52.01% (excluding Chunk-Learn). Considering ExpertSel for the three independent loops, we note that the most selected scheduling algorithm for $L0$ is `mAF` (78.2%), for $L1$ `STATIC` (24.2%) and for $L2$ `mAF` (31.6%) again. From the results we can gather that SARSA selected `mAF` in $L0$, Q-Learn `STATIC` in $L1$ and DoubleQ and SARSA `mAF` in $L2$ most of time as well. For the rest of the automated selection methods there is no similarity in the sequence of scheduling algorithms. The results for $L0$ need to be mentioned especially, since no matter what scheduling algorithm was applied, the loop time was only affected marginally (this is a property of the application and not the automated selection methods).

When looking at Q-Learn in more detail, we recognize that the agent learned to choose the same three scheduling algorithms (`STATIC, SS, TSS`) for the exact same amount of times for every loop. This is indeed not the expected behaviour, since we know there are better algorithms available (which have also been learned by the `ExploreFirstPolicy`). Notable as well is that the chosen algorithms by Q-Learn are the ones at the very start of the portfolio which leads to the conclusion that for Q-Learn, the first learning experience is the most defining one. The other Reinforcement Learning agents suffer from indecisiveness when selecting scheduling algorithms. Even late in the learning process, the agents keep selecting bad actions, which leads to an overall high parallel execution time. Looking at Figs. A.15 (SARSA), A.16 (SARSA) and A.17 (QV-Learn) we can see that the indecisiveness can be overcome with the right reward incentive.

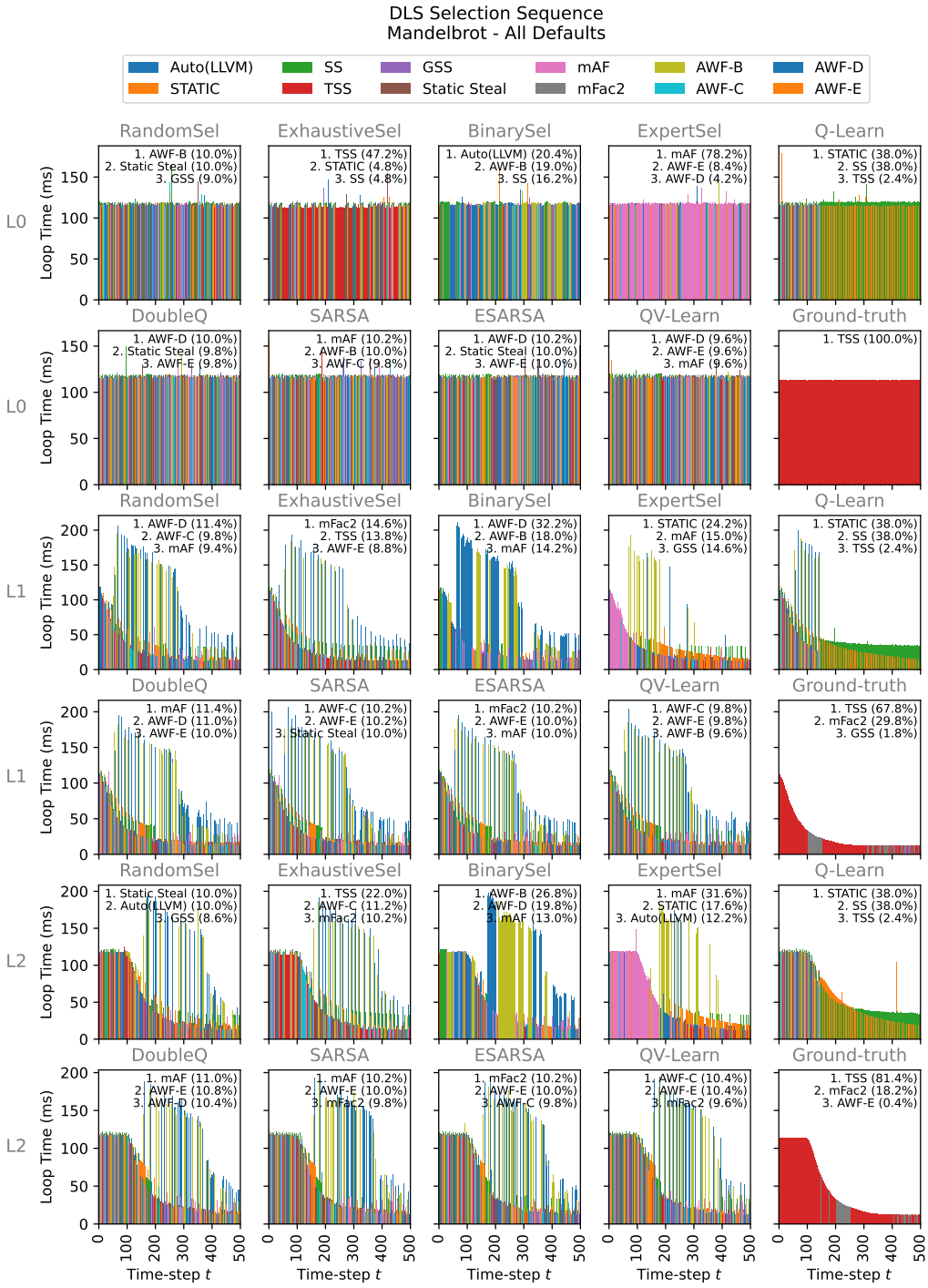

Figure 5.4: Sequence of selected scheduling algorithms for every time-step t during the execution of the application. The time-steps are shown on the x axis while the resulting loop time in milliseconds are shown on the y axis. The top 3 selected scheduling algorithms are listed in the top right of each plot together with its percentage.

5.2 SPHYNX Evrard Collapse

This section describes the properties and results for the benchmarks of the modified SPHYNX Evrard Collapse application. The application is launched without any input parameters. The version of SPHYNX used in this benchmark, starts the hydrodynamics simulation code at time-step $t = 2'000$ which changes the load-imbalance profile. We modified the `parameters.f90` file to end the application run at time-step $t = 2'400$ instead of $t = 2'200$ in order to increase the applications runtime and test automated method DLS algorithm selection capabilities over a longer duration. Every experiments was run 5 times with the same configuration on the miniHPC-Broadwell of the university, to make sure the variance of the results is in an acceptable range

5.2.1 Ground-truth

The ground-truth (GT) is the theoretical best scheduling method and serves as a base for comparison for the other scheduling algorithms. We construct the ground-truth by running the chosen scientific application 5-times with every scheduling algorithm from the portfolio defined in section 4.2. We then take the mean of the 5 runs and look for smallest parallel execution time amongst all the methods in the portfolio for every time-step. The sum of the smallest means over all the time-steps is the new theoretical best parallel execution time for any automated DLS algorithm method. The ground-truth should only be used to compare scheduling methods that use the same portfolio of algorithms.

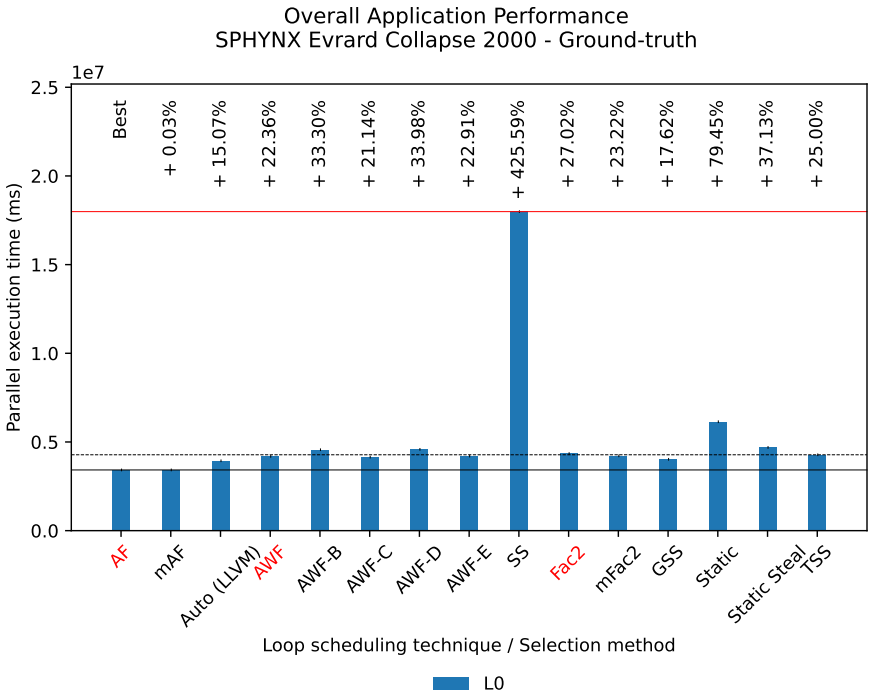

Figure 5.5: Results for the ground-truth for the SPHYNX Evrard Collapse application. The numbers presented in the graph are the mean collected over 5 runs for every scheduling algorithm in the portfolio. Labels in red on the x axis represent scheduling algorithms not in the portfolio serving as comparison.

From Fig. 5.5 we observe that the two scheduling algorithm *AF* and *mAF* achieve the best performance when selected manually. We expect the automated selection methods to ideally choose the same algorithms as well. The dynamic schedule (*SS*) was not able to provide adequate scheduling for the application to finish the 400 time-steps within the 6 hour time limit set for the individual jobs.

Possible bug: The SPHYNX Evrard Collapse particle simulation has two modified loops called `findneighbours` and `treewalk`. In a single time-step the `findneighbours` loop is called twice by the application. The two calls are executed with different arguments, which results in drastically different loop execution times. The problem we encountered stems from the underlying mechanism by the LLVM OpenMP runtime which derives the loop names. In this process the call location is not taken into account, only the definition of the called function. This might be fine for some logging purposes, but the implemented Reinforcement Learning agent has no other mechanism of distinguishing between functions calls with varying arguments. As a consequence the agent tracks twice as many time-steps for `findneighbours` as for the rest of the loops. On top, the agent's learning process is very sensitive to the reward input (in most cases the loop execution time). As a result the learning process is skewed by the big difference in execution time from two calls to `findneighbours`. This lead to the decision to ignore the `findneighbours` loop in the evaluation process.

5.2.2 Results

The following section explains the results for the SPHYNX Evrard Collapse experiments. Fig. 5.6 and Tab. 5.4 show an overview of the all benchmarked methods and configurations. As before, we want to show how the different configurations influence the automated DLS selection methods from RL4OMP and through that deduce what the best configuration is for the agent is. Additionally we compare them to other methods and algorithms. We expect the outcome to be different to the Mandelbrot experiment, since SPHYNX has a different load-imbalance profile. These results are presented in Tab. 5.4 and as a line chart in Fig. 5.6, as well as in bar plots for every single configuration are available. The graph for the *All Defaults* configuration is included in this section in Fig. 5.7 and the remaining plots are included in section A.3 in the appendix.

Table 5.4: Results for the benchmarks with the SPHYNX Evrard Collapse application. The table shows the percentage difference in runtime against the ground-truth for different experiment configurations. Configurations are listed as rows and selected scheduling methods are shown as columns. A green cell-color indicates a performance closer to ground-truth, while a red cell-color indicates an overall bad application performance result. The graphs for all the results are located in the appendix in section A.3.

Configuration \ Selection Method	RandomSel	ExhaustiveSel	BinarySel	ExpertSel	Q-Learn	DoubleQ	SARSA	ESARSA	QV-Learn	Chunk-Learn	Ground-truth
	Auto4OMP Reference				RL4OMP Extension						
All Defaults	397.53 %	5.04 %	14.05 %	80.76 %	278.62 %	65.80 %	63.80 %	66.35 %	67.93 %	25.75 %	0.00 %
Random	415.65 %	4.31 %	23.16 %	91.48 %	49.24 %	67.67 %	86.96 %	94.00 %	83.02 %	26.36 %	0.00 %
Looptime Inverse	420.96 %	4.38 %	13.56 %	72.21 %	30.42 %	82.92 %	82.47 %	83.27 %	88.40 %	79.45 %	0.00 %
Looptime Average	419.58 %	4.74 %	21.79 %	74.10 %	284.52 %	63.65 %	67.06 %	67.14 %	70.16 %	28.29 %	0.00 %
Looptime Rolling	261.00 %	10.42 %	32.45 %	55.14 %	277.83 %	214.49 %	80.80 %	88.83 %	65.81 %	27.18 %	0.00 %
Neutral Reward	427.83 %	2.29 %	22.42 %	78.98 %	278.01 %	77.33 %	79.24 %	63.73 %	76.67 %	27.78 %	0.00 %
Positive Reward	236.29 %	5.07 %	12.78 %	52.23 %	30.60 %	83.34 %	83.66 %	83.42 %	83.00 %	79.38 %	0.00 %
Loadimbalance	146.14 %	6.50 %	24.67 %	68.22 %	279.71 %	67.08 %	66.36 %	72.38 %	66.74 %	28.57 %	0.00 %
Robustness	419.47 %	2.39 %	15.49 %	73.20 %	278.17 %	63.95 %	63.07 %	63.83 %	63.71 %	27.28 %	0.00 %
Epsilon Greedy	135.59 %	2.99 %	22.45 %	75.27 %	304.28 %	78.51 %	74.80 %	81.93 %	69.84 %	44.65 %	0.00 %
Softmax	420.59 %	2.73 %	13.95 %	66.87 %	69.56 %	107.76 %	85.03 %	91.88 %	82.10 %	14.98 %	0.00 %
Average	336.42 %	4.62 %	19.71 %	71.68 %	196.45 %	88.41 %	75.75 %	77.89 %	74.31 %	37.24 %	0.00 %

The performance results in Tab. 5.4 show that the best performance for the SPHYNX Evrard Collapse application was achieved by ExhaustiveSel, BinarySel and Chunk-Learn. This stands in contrast with the results for the Mandelbrot application. Here Chunk-Learn is only the third best option in terms of automated DLS selection methods, performing 37.24% above ground-truth on average. ExhaustiveSel and BinarySel from Auto4OMP impress with a performance of 4.62% and 19.71% above GT respectively. While Chunk-Learn shows good performance across the board of all configurations, it performs best with the *Softmax* configuration (14.98% above GT) and has two bad performance results with the *Looptime Inverse* and *Positive Reward* configuration. The performance results for the SPHYNX Evrard Collapse application emphasize the importance and feasibility of automated scheduling algorithm selection, especially for longer running scientific applications. In Fig. 5.6 we see that the best performing method is still an automated selection method and not a fixed scheduling algorithm, even though *Auto (LLVM)* and *AWF* still perform well. The results for the Q-Learn agent really highlight the difference in performance when it comes to the the configurations for the sub-components. While on average the Q-Learner achieves a bad performance result, 196.45% above ground-truth, three configurations show a good overall performance compared to ground-truth: *Random* 49.24%, *Looptime Inverse* 30.42% and *Positive Reward* 30.60%. Interestingly although Chunk-Learn is configured to use Q-Learning as its agent, the configurations where Q-Learning has a performance advantage, do not overlap with those of Chunk-Learn's well performing configurations (expect for *Random*). Compared to Mandelbrot, the performance of the other agents (DoubleQ, SARSA, ESARSA and QV-Learn) during the execution of the SPHYNX Evrard Collapse application is more consistent - which means the configurations did not have a big impact on the individual performance results. A notable exception is the DoubleQ agent, which

performed comparatively poorly with the *Looptime Rolling* configuration (214.49%). On average these agents' performance is rated at 74.31% to 88.41% around GT (disregarding the results for Q-Learn). For comparison, the automated selection methods from Auto4OMP achieved results ranging from 4.62% up to 336.42% above ground-truth. It is worth mentioning that the best performing method from Auto4OMP for Mandelbrot was ExpertSel (with ExhaustiveSel performing only slightly worse), for SPHYNX the better performing method was ExhaustiveSel, with ExpertSel's performance not being comparable at all. In contrast, Chunk-Learn seems to offer a more consistent performance over the two selected applications.

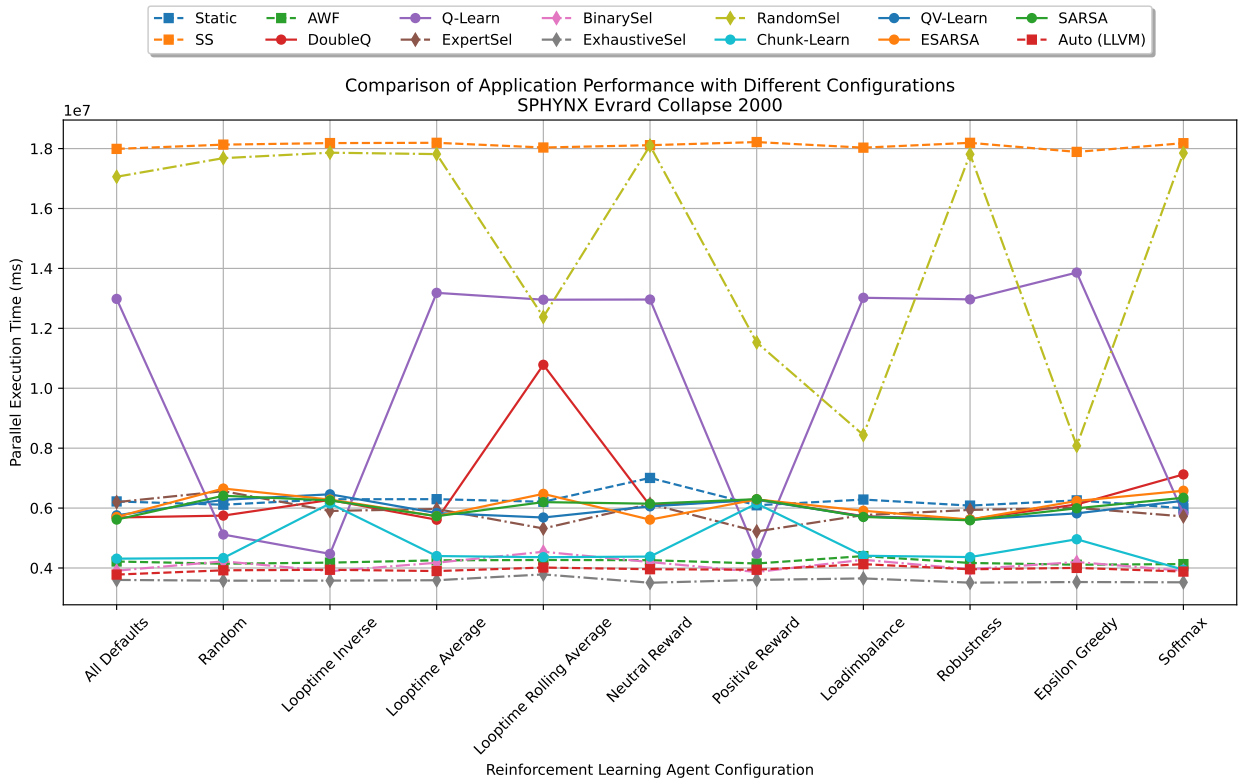

Figure 5.6: We show the results of the SPHYNX Evrard Collapse application runs with $t = 400$ time-steps on the miniHPC-Broadwell cluster. The continuous lines with the ● markers represent results for the automated selection techniques implemented in RL4OMP. The dash-dotted lines with the ♦ markers depict the results for the automated selection methods from previous work (Auto4OMP), and the dashed lines with the ■ markers present the results for selected scheduling algorithms as a reference. The x axis lists the different configuration from table 5.2 while the y axis denotes the parallel execution time in milliseconds.

Fig. 5.7 lists the results from the *All Defaults* configuration in more detail. On the x axis the different scheduling methods are listed, on the y axis the parallel execution time in milliseconds is shown. Contrary to the Mandelbrot application, from the two modified loops, only $L0$ is shown in the performance evaluation due to the aforementioned "bug" in the LLVM OpenMP runtime's assignment of loop identifiers.

While Auto4OMP presents the two best performing automated DLS selection methods, ExhaustiveSel with 5.04% and BinarySel with 14.05% around ground-truth, it also provides the method with the slowest performance (397.53% above GT), which did not let the application finish within the 6 hour time limit. Also the ExpertSel method which performed best on the Mandelbrot application, fell behind in the SPHYNX Evrard Collapse application. The default configuration derived from the Mandelbrot application experiments could not provide the same promising results here. This is especially apparent when looking at the Q-Learn agent (278.62% within GT) but also at the rest of the agents. Chunk-Learn delivers an acceptable performance result with 25.75% above ground-truth. The best performance for Q-Learn could be observed with the *Looptime Inverse* (see Fig A.25) and the *Positive Reward* (see Fig A.29) configuration. For Chunk-Learn the best configuration is by far *Softmax* (see Fig A.33).

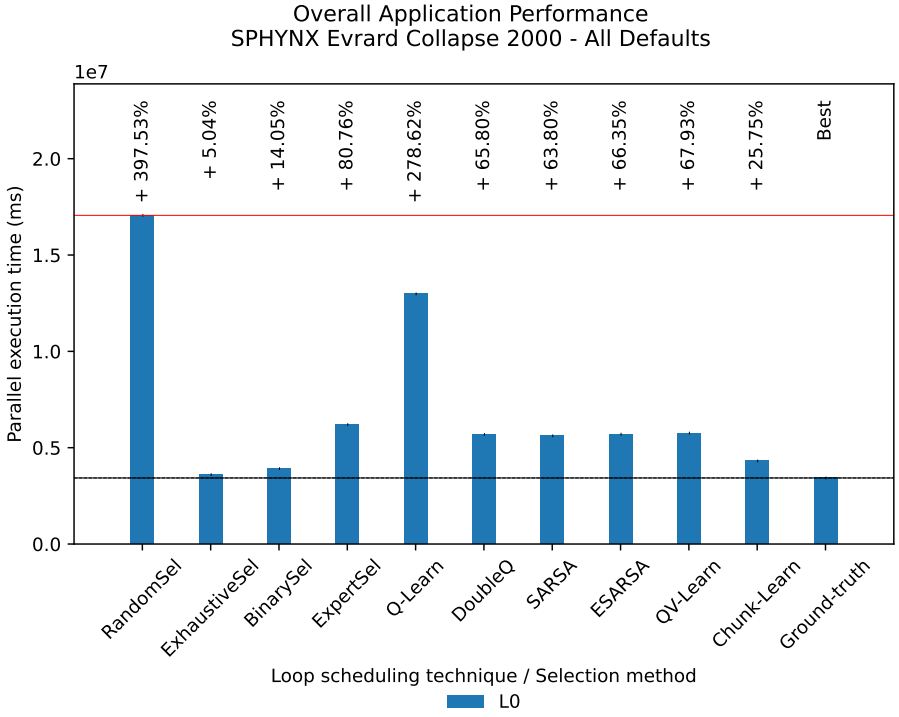

Figure 5.7: Shows the parallel execution time for each modified application loop on the y axis and the different scheduling methods on the x axis. The 4 automated selection methods from Auto4OMP show big differences in the performance achievable. While RL4OMP has a more consistent performance envelope, Q-Learn is not a valid contender for the best performance in the default configuration.

Fig 5.8 shows plots for application loop $L0$ of SPHYNX for every automated DLS selection method benchmarked in addition to the ground-truth. A single sub-plot shows the time-steps on the x axis and the loop time in milliseconds on the y axis. The different colored bars represent the individual time-steps. The color of the bar indicates (in accordance with the legend on top of the plot) which scheduling algorithm from the portfolio the auto-

mated method chose for that particular time-step. Lastly in the top right of each sub-plot we listed the top 3 most selected scheduling algorithms for that particular selection method along with its percentage from the total time-steps.

Comparing the ground-truth for loop *L*0 for the SPHYNX Evrard Collapse application to the automated selection methods, we see that only ExhaustiveSel from Auto4OMP has correctly identified the best performing scheduling algorithm *mAF* from almost the start of the application run. This is in line with the observations seen in the evaluation of the Mandelbrot experiments. This also explains the excellent performance of this method. Analysing the sequence of DLS algorithm selection for Q-Learn, we can see that the *ExploreFirstPolicy* forces the agent to select the same three scheduling algorithms (*STATIC, SS, TSS*) again. This confirms our suspicion that under the *All Defaults* configuration, the algorithms explored at the start under this policy will be seen favourably by the agent, even when the performance does not reflect this fact. Interestingly with the *Looptime inverse* (see Fig. A.36) and *Positive Reward* (see Fig. A.40) configurations, Q-Learn was able to identify *mAF* as the best scheduling algorithm after the initial exploration phase. As with Mandelbrot, the other Reinforcement Learning agents suffer from indecisiveness under every configuration when selecting scheduling algorithms. Even late in the learning process, the agents keep selecting bad actions, which leads to an overall high parallel execution time.

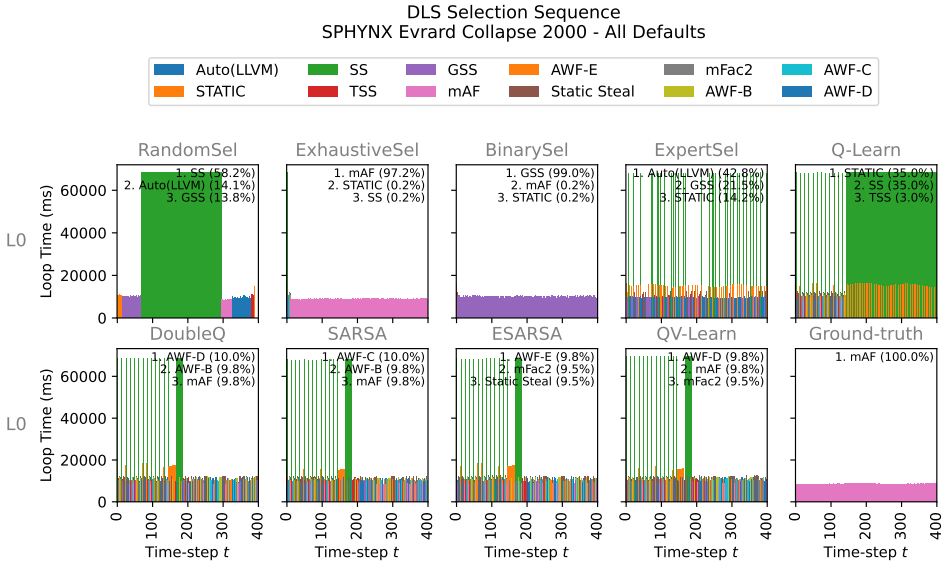

Figure 5.8: Sequence of selected scheduling algorithms for every time-step t during the execution of the application in the *All Defaults* configuration. The time-steps are shown on the x axis while the resulting loop time in milliseconds are shown on the y axis. The top 3 selected scheduling algorithms are listed in the top right of each plot together with its percentage.

6

Conclusion & Future Work

This work introduced RL4OMP, an automated approach for scheduling algorithm selection and load balancing based on Reinforcement Learning in OpenMP. RL4OMP provides six configurable agents, three action selection policies and six reward functions for an automatic selection of scheduling algorithms. We further propose a seventh meta-agent, Chunk-Learn, which can be viewed as a scheduling algorithm since it estimates the chunk-size for the next scheduling round directly. We evaluated the performance of RL4OMP for two applications, executing them on one multi-core system. We compared the performance achieved by RL4OMP with state-of-the-practice solutions and against ground-truth (the highest achievable performance which selects the highest performing scheduling algorithm for each loop, time-step and system). The proposed automated algorithm selection methods learn during the execution of the application, refine their selection policy over time, thereby minimizing load-imbalance and achieving performance that is closer to the ground-truth.

For the workload of the Mandelbrot application the SARSA agent with the `Looptime Rolling` configuration achieves the highest performance with **10.04%** above GT. The worst case scenario for the Reinforcement Learning extension is observed with the Q-Learn agent under the `Random` configuration, **50.01%** above GT. The special agent Chunk-Learn (using Q-Learn behind the scenes) achieved an even better performance result than SARSA. With the best performance stemming from the *Epsilon Greedy* configuration **1.76%** within GT. The results from the SPHYNX Evrard Collapse experiments paint a different picture about the performance of the Reinforcement Learning extension. Q-Learn can obtain a good performance only with two configurations (*Looptime Inverse* **30.42%** and *Positive Reward* **30.60%**), but has overall a bad impact on performance - worst case **304.28%**. But again Chunk-Learn achieved a solid performance with the *Softmax* configuration that lies within **14.98%** of GT.

On average, RL4OMP cannot outperform the best candidates of other automated DLS selection methods we compared against (e.g. Auto4OMP). Our performance however is still in line with other dynamic and adaptive scheduling algorithms. With the Chunk-Learn special agent, we presented a method that shows reliable performance with regard to the ground-truth and other automated selection methods. The direct estimation of the chunk-size is advantageous for the agent's learning process, since it can bypass any scheduling

algorithm which it interacts with as a black-box and might only distort the agent's perception of the environment. The downside to this approach is, that we cannot reason about the selection of the chunk-size anymore - since the learning agent has become a black-box itself. Additionally we also show that this method still benefits from hyper-parameter tuning and that to date there is no one-size-fits-all solution.

The benchmark results for the two chosen applications clearly show, that extensions like DoubleQ-Learning, Expected-SARSA or QV-Learning to the well known Reinforcement Learning methods Q-Learning and SARSA do not perform better in estimating the best action when it comes to scheduling algorithm selection. Rather they also suffer from decision paralysis like a human user selecting DLS algorithms manually. More impact on the performance has been observed through the design of the reward function or the action selection policy. DeepQ-Learning has not been implemented in RL4OMP, because the technical challenges (lightweight and standalone implementation of neural networks in C++) outweigh the theoretical benefits. DeepQ-Learning would lend itself if the state-action space that needed to be encoded as the agent's knowledge was massive and the action to estimate was more of continuous nature (e.g. how much to press a gas pedal) and not a discrete set of actions.

Seeing the promising results of the Chunk-Learn method, in future work this approach could be extended and improved upon. Here DeepQ-Learning could be applied by increasing the search space of chunk-sizes for more granular control over the load-imbalance (beyond the 12 chunk-sizes we calculated) without worrying about the increasing memory requirements as with tabular data-structures. Further, a neural network would simplify the use of more than one reward metric as the input for the learning process. Additionally it would be interesting to train a machine-learning model on one or multiple time-stepping applications and train it for many applications runs, then export the model and evaluate the performance on an unseen time-stepping application.

Lastly the Reinforcement Learning extension could be re-implemented in MPI (with the LB4MPI portfolio) for automated algorithm selection in order to achieve cross-node load balancing and multi-level scheduling for hybrid MPI+OpenMP applications.

Bibliography

[1] Bernd Bischl, Pascal Kerschke, Lars Kotthoff, Marius Lindauer, Yuri Malitsky, Alexandre Frechette, Holger Hoos, Frank Hutter, Kevin Leyton-Brown, Kevin Tierney, and Joaquin Vanschoren. ASlib: A benchmark library for algorithm selection. 237. doi: 10.1016/j.artint.2016.04.003.

[2] Anthony Boulmier, Ioana Banicescu, Florina M. Ciorba, and Nabil 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. doi: 10.1109/ISPDC.2017.9.

[3] Si-An Chen, Voot Tangkaratt, Hsuan-Tien Lin, and Masashi Sugiyama. Active deep q-learning with demonstration. 109. doi: 10.1007/s10994-019-05849-4.

[4] Florina M. Ciorba, Christian Iwainsky, and Patrick Buder. OpenMP loop scheduling revisited: Making a case for more schedules. URL http://arxiv.org/abs/1809.03188.

[5] Sumithra Dhandayuthapani. *Automatic Selection of Dynamic Loop Scheduling Algorithms for Load Balancing Using Reinforcement Learning*. Mississippi State University. Google-Books-ID: hrnVjwEACAAJ.

[6] Alla Evseenko and Dmitrii Romannikov. Application of deep q-learning and double deep q-learning algorithms to the task of control an inverted pendulum. pages 7–25. doi: 10.17212/2307-6879-2020-1-2-7-25.

[7] Milind Girkar, A. Kejariwal, Tian Xinmin, Hideki Saito, Alexandru Nicolau, Alexander Veidenbaum, and Constantine Polychronopoulos. Probablistic self-scheduling. pages 253–264. ISBN 978-3-540-37783-2. doi: 10.1007/11823285_26.

[8] Jahanzeb Hashmi, Shulei Xu, Bharath Ramesh, Mohammadreza Bayatpour, Hari Subramoni, and Dhabaleswar Panda. Machine-agnostic and communication-aware designs for MPI on emerging architectures. pages 32–41. doi: 10.1109/IPDPS47924.2020.00014.

[9] Hado V Hasselt. Double q-learning. page 9.

[10] A. Kejariwal and A. Nicolau. Reading list of self-scheduling of parallel loops.

[11] Kyurae Kim, Kim Youngjae, and Sungyong Park. A probabilistic machine learning approach to scheduling parallel loops with bayesian optimization. doi: 10.1109/TPDS.2020.3046461.

[12] Jonas H. Müller Korndörfer, Ahmed Eleliemy, Ali Mohammed, and Florina M. Ciorba. LB4omp: A dynamic load balancing library for multithreaded applications. 33(4):830–841. ISSN 1558-2183. doi: 10.1109/TPDS.2021.3107775. Conference Name: IEEE Transactions on Parallel and Distributed Systems.

[13] Michail Lagoudakis and Michael Littman. Algorithm selection using reinforcement learning.

[14] Marlos C Machado, Sriram Srinivasan, and Michael Bowling. Domain-independent optimistic initialization for reinforcement learning. page 2.

[15] Hongzi Mao, Malte Schwarzkopf, Shaileshh Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. Learning scheduling algorithms for data processing clusters. pages 270–288. ISBN 978-1-4503-5956-6. doi: 10.1145/3341302.3342080.

[16] Michael Melnik and Denis Nasonov. Workflow scheduling using neural networks and reinforcement learning. 156:29–36. doi: 10.1016/j.procs.2019.08.126.

[17] Ali Mohammed and Florina Ciorba. *SiL: An Approach for Adjusting Applications to Heterogeneous Systems Under Perturbations.* .

[18] Ali Mohammed and Florina Ciorba. SimAS: A simulation-assisted approach for the scheduling algorithm selection under perturbations. 32, . doi: 10.1002/cpe.5648.

[19] C. Oikawa, Vinicius Freitas, Marcio Castro, and Laércio Lima Pilla. Adaptive load balancing based on machine learning for iterative parallel applications. pages 94–101. doi: 10.1109/PDP50117.2020.00021.

[20] Mahbubur Rashid, Ioana Banicescu, and Ricolindo Carino. Investigating a dynamic loop scheduling with reinforcement learning approach to load balancing in scientific applications. pages 123–130. doi: 10.1109/ISPDC.2008.25.

[21] Km Vaishali Rastogi, Anand Prakash Shukla, Anubhav Patrick, and Navin Kumar Mittal. DEEP q LEARNING AND ITS VARIANTS: A CONCISE REVIEW. 7(18): 11.

[22] Nimish Sanghi. Deep q-learning. pages 155–206. ISBN 978-1-4842-6808-7. doi: 10.1007/978-1-4842-6809-4_6.

[23] Shahaf S. Shperberg, Solomon Eyal Shimony, and Avinoam Yehezkel. Algorithm selection in optimization and application to angry birds. 29:437–445. ISSN 2334-0843. URL https://ojs.aaai.org/index.php/ICAPS/article/view/3508.

[24] Vinu Sreenivasan, Rajath Javali, Mary Hall, Prasanna Balaprakash, Thomas Scogland, and Bronis Supinski. A framework for enabling OpenMP autotuning. pages 50–60. ISBN 978-3-030-28595-1. doi: 10.1007/978-3-030-28596-8_4.

[25] Nitin Sukhija. Portfolio-based selection of robust dynamic loop scheduling algorithms using machine learning. doi: 10.1109/IPDPSW.2014.183.

[26] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. Adaptive Computation and Machine Learning series. A Bradford Book, . ISBN 978-0-262-19398-6.

[27] Richard S. Sutton and Andrew G. Barto. *Reinforcement learning: an introduction*. Adaptive computation and machine learning series. The MIT Press, second edition edition, . ISBN 978-0-262-03924-6.

[28] Peter Thoman, Herbert Jordan, Simone Pellegrini, and Thomas Fahringer. Automatic OpenMP loop scheduling: A combined compiler and runtime approach. volume 7312, pages 88–101. ISBN 978-3-642-30960-1. doi: 10.1007/978-3-642-30961-8_7.

[29] Marco A Wiering. QV(λ)-learning: A new on-policy reinforcement learning algorithm. page 2.

[30] Yun Zhang and M. Voss. Runtime empirical selection of loop schedulers on hyper-threaded SMPs. pages 44b–44b. ISBN 978-0-7695-2312-5. doi: 10.1109/IPDPS.2005.386.

[31] S. Peer Mohamed Ziyath and Senthilkumar Subramaniyan. An improved q-learning-based scheduling strategy with load balancing for infrastructure-based cloud services. ISSN 2191-4281. doi: 10.1007/s13369-021-06279-y. URL https://doi.org/10.1007/s13369-021-06279-y.

A
Appendix

In the appendix we show all the additional plots for reference and completeness. These plots should aid in the understand of the effects the varying configuration have on the Reinforcement Learning agents and the application's performance.

A.1 Environment Variables

```
export KMP_RL_ALPHA=0.85
export KMP_RL_ALPHA_DECAY=0.01
export KMP_RL_GAMMA=0.95
export KMP_RL_EPSILON=0.9
export KMP_RL_EPS_DECAY=0.01
# Reward Options: looptime, looptime-average, looptime-rolling-average,
↪  looptime-inverse, loadimbalance, robustness
# Default: looptime
export KMP_RL_REWARD=looptime
# Initializer Options: zero, random, optimistic
# Default: zero
export KMP_RL_INIT=zero
# Policy Options: explore-first, epsilon-greedy, softmax
# Default: explore-first
export KMP_RL_POLICY=explore-first
# Chunk Learner Type Options: Range [8-14] inclusive
# Default: 8
export KMP_RL_CHUNK_TYPE=8
# Reward Number Options: Comma separated triple of doubles
#Default: 0.0,-2.0,-4.0
export KMP_RL_REWARD_NUM=0.0,-2.0,-4.0
```

Listing 2: Supported environment variables by the reinforcement learning extension.

A.2 Mandelbrot - Extended Results

A.2.1 Overall Application Performance

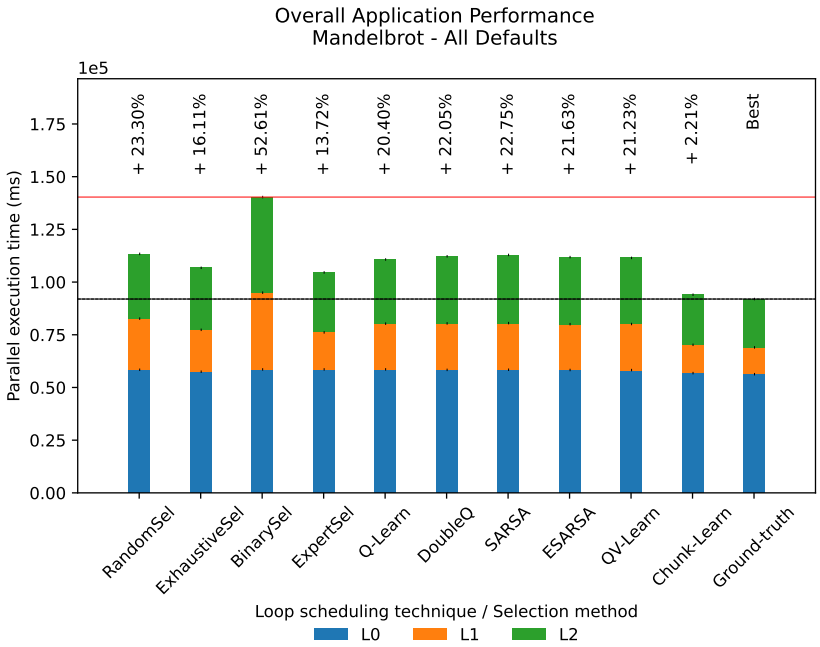

Figure A.1: Mandelbrot Overall Application Performance with *All Defaults* configuration.

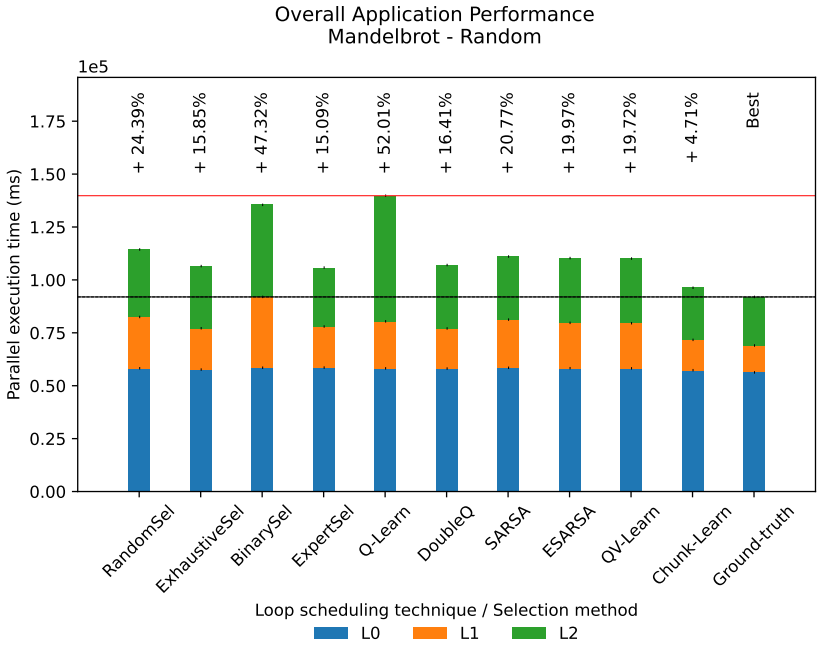

Figure A.2: Mandelbrot Overall Application Performance with *Random* configuration.

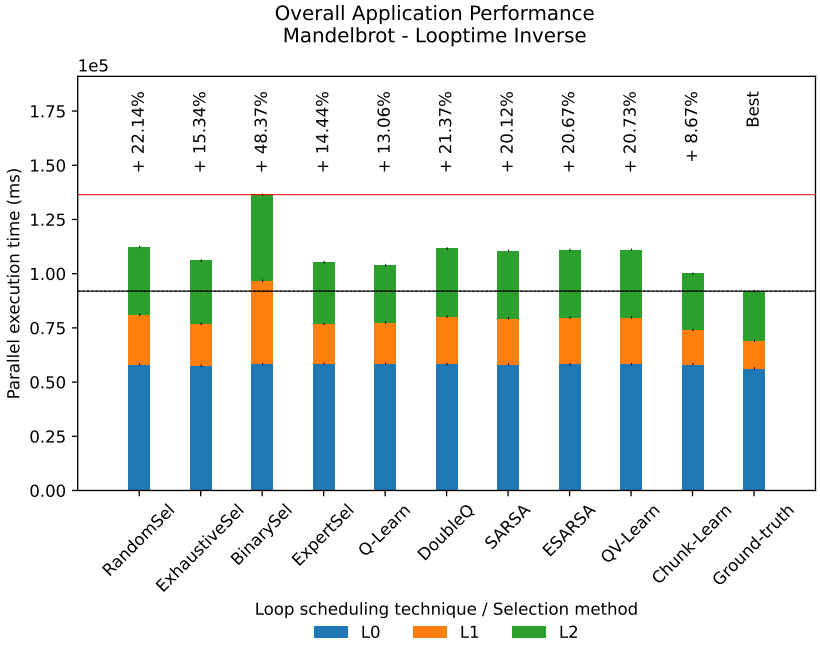

Figure A.3: Mandelbrot Overall Application Performance with *Looptime Inverse* configuration.

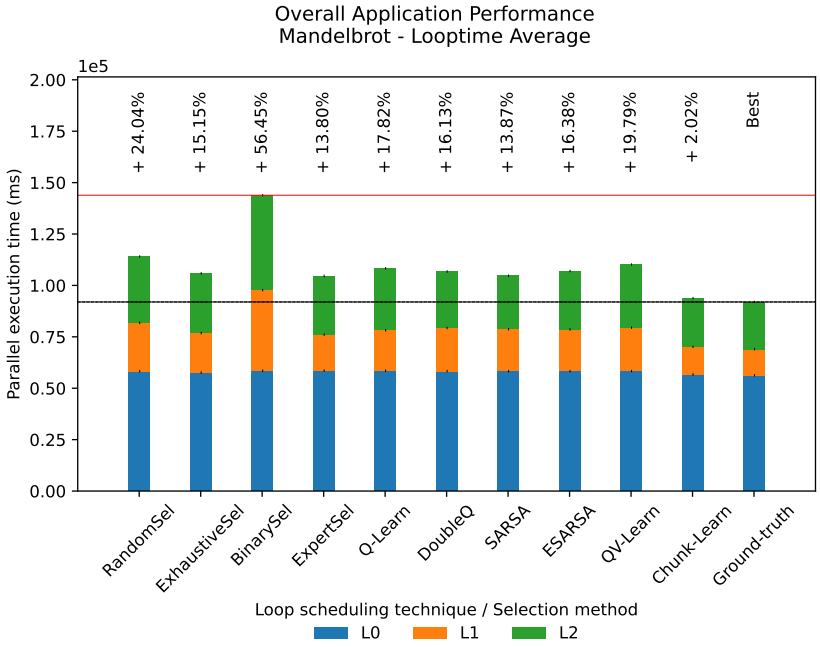

Figure A.4: Mandelbrot Overall Application Performance with *Looptime Average* configuration.

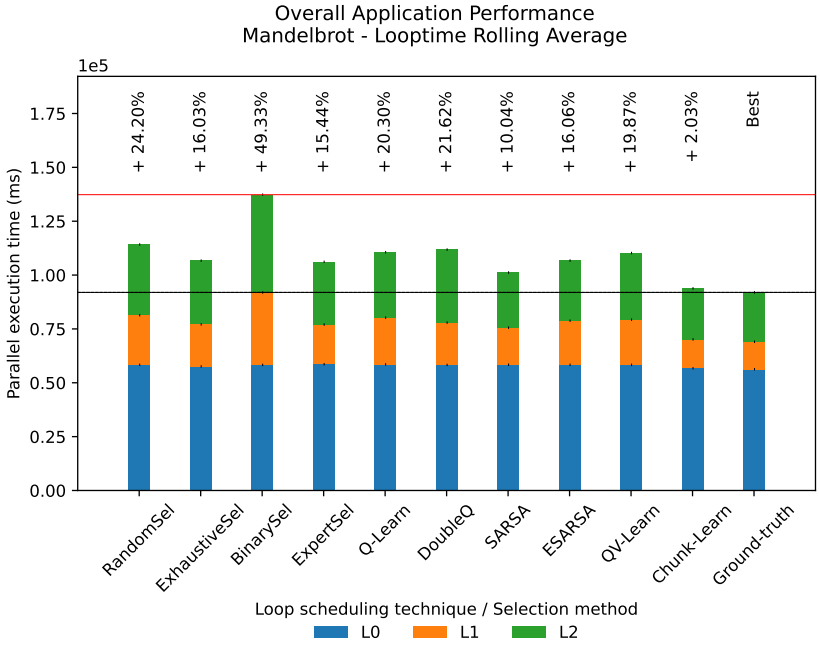

Figure A.5: Mandelbrot Overall Application Performance with *Looptime Rolling* configuration.

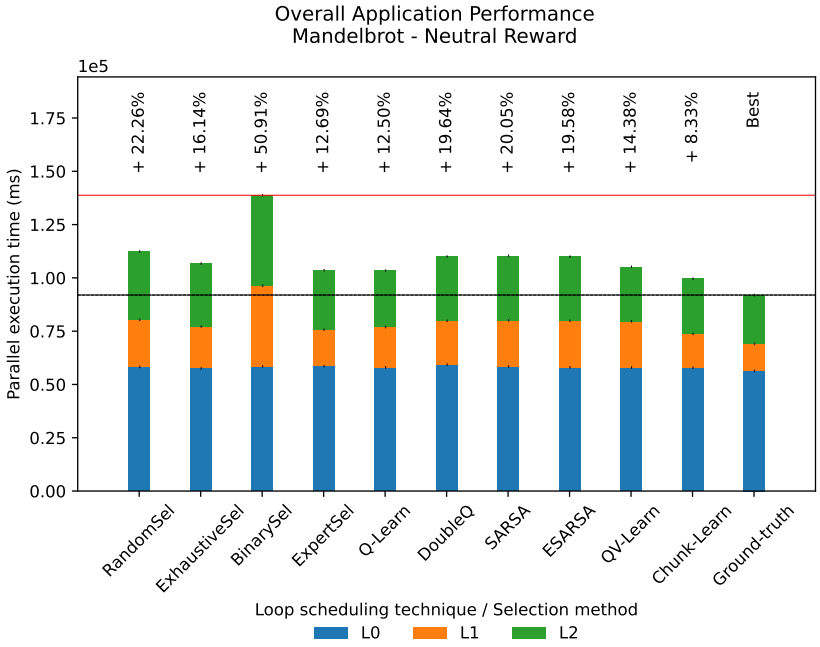

Figure A.6: Mandelbrot Overall Application Performance with *Neutral Reward* configuration.

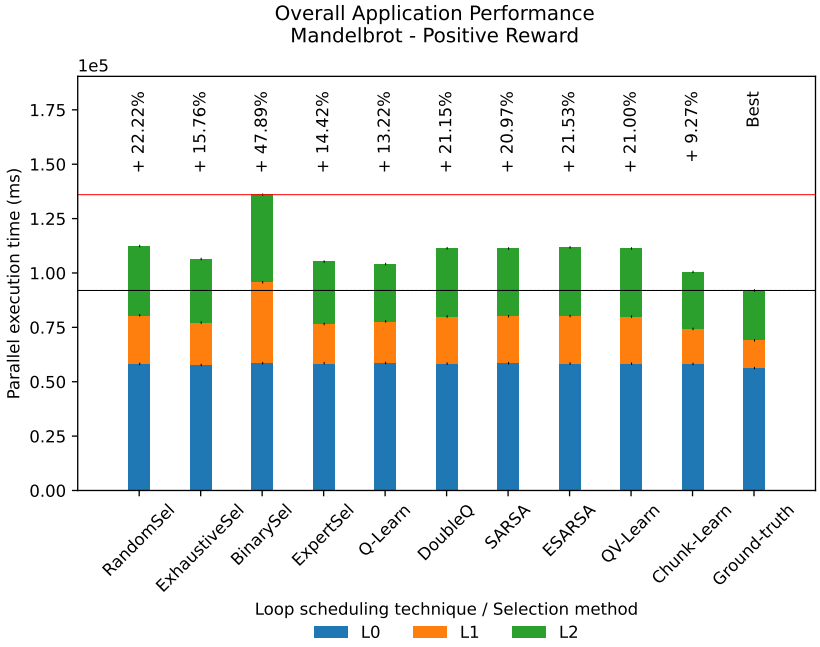

Figure A.7: Mandelbrot Overall Application Performance with *Positive Reward* configuration.

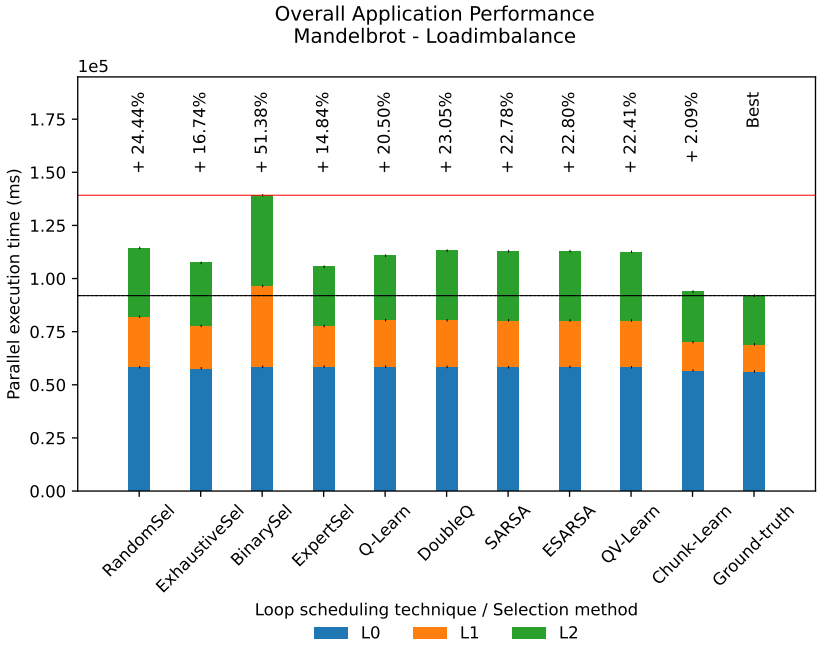

Figure A.8: Mandelbrot Overall Application Performance with *Loadimbalance* configuration.

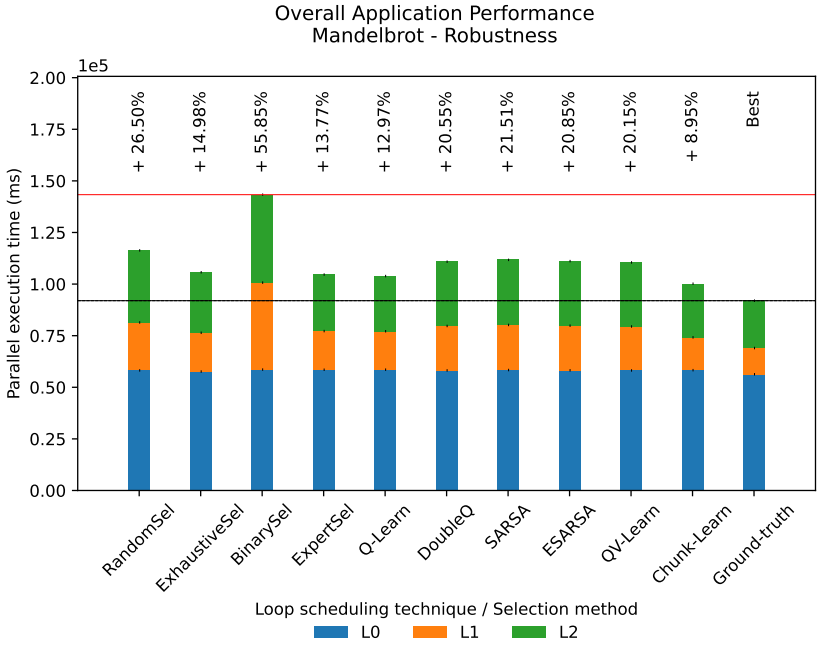

Figure A.9: Mandelbrot Overall Application Performance with *Robustness* configuration.

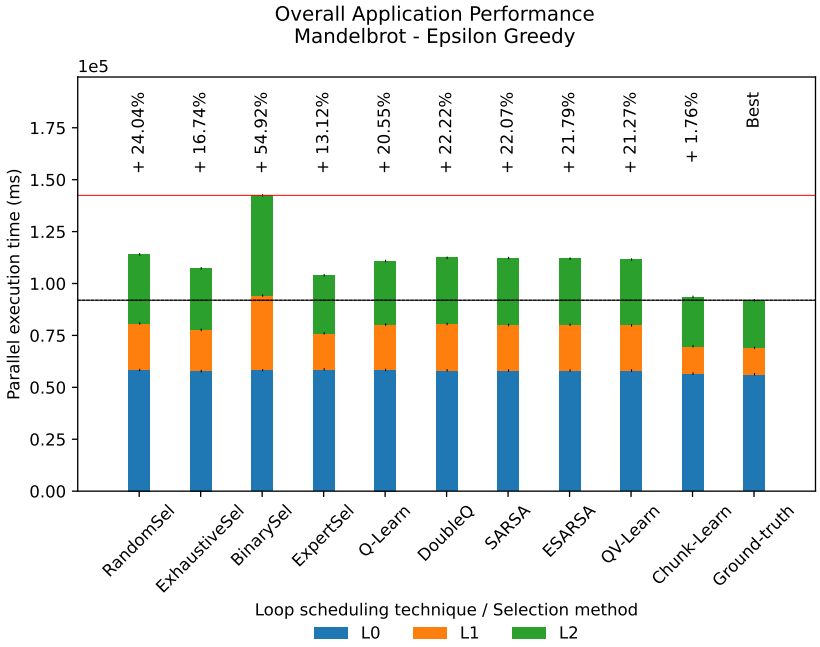

Figure A.10: Mandelbrot Overall Application Performance with *Epsilon Greedy* configuration.

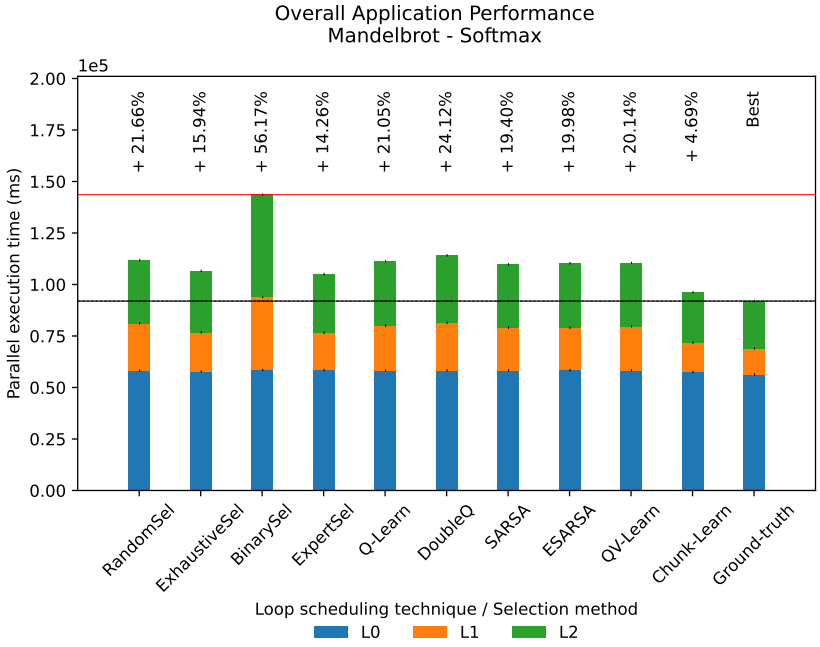

Figure A.11: Mandelbrot Overall Application Performance with *Softmax* configuration.

A.2.2 DLS Selection Sequence

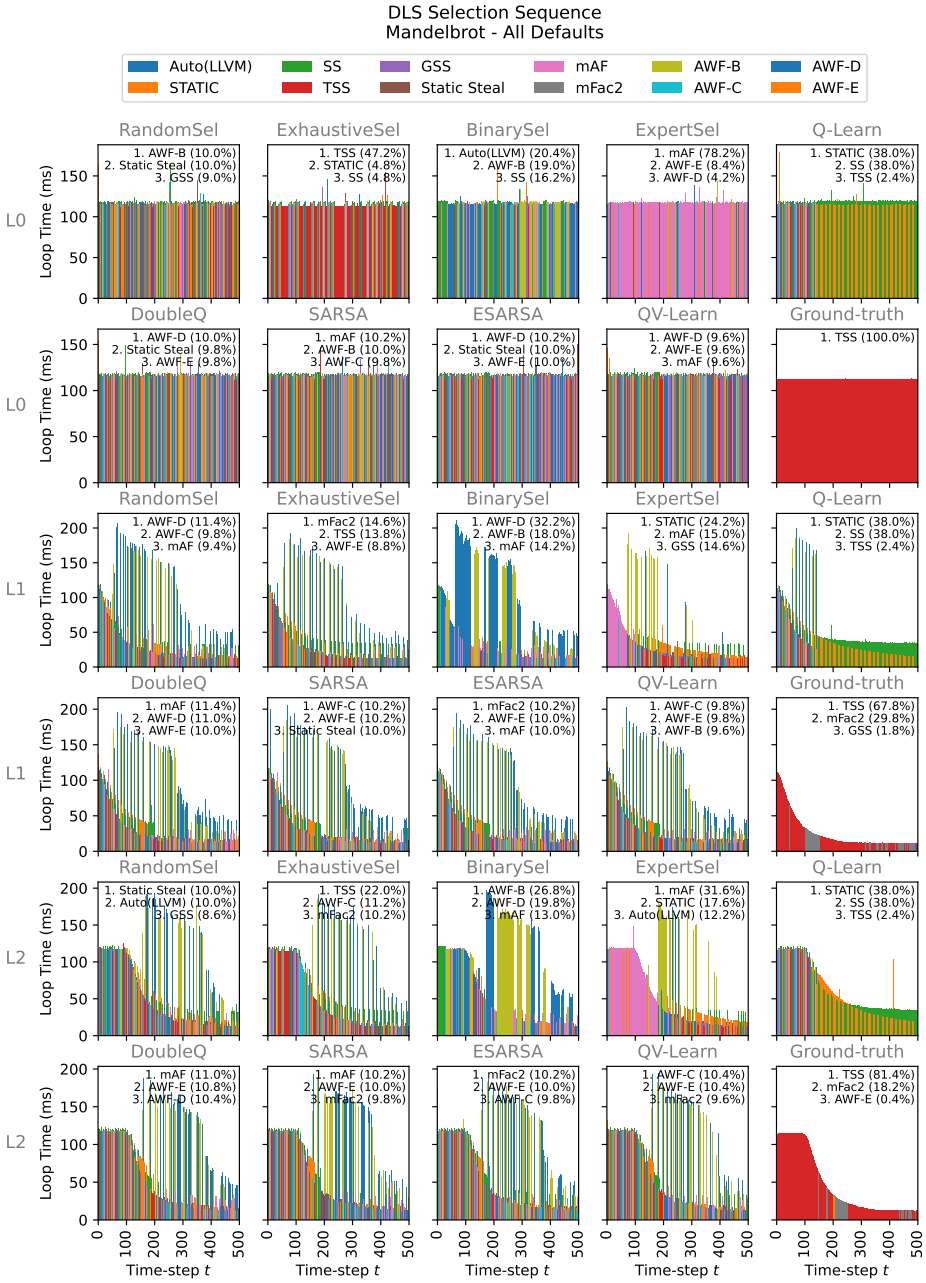

Figure A.12: Mandelbrot DLS Selection Sequence with *All Defaults* configuration.

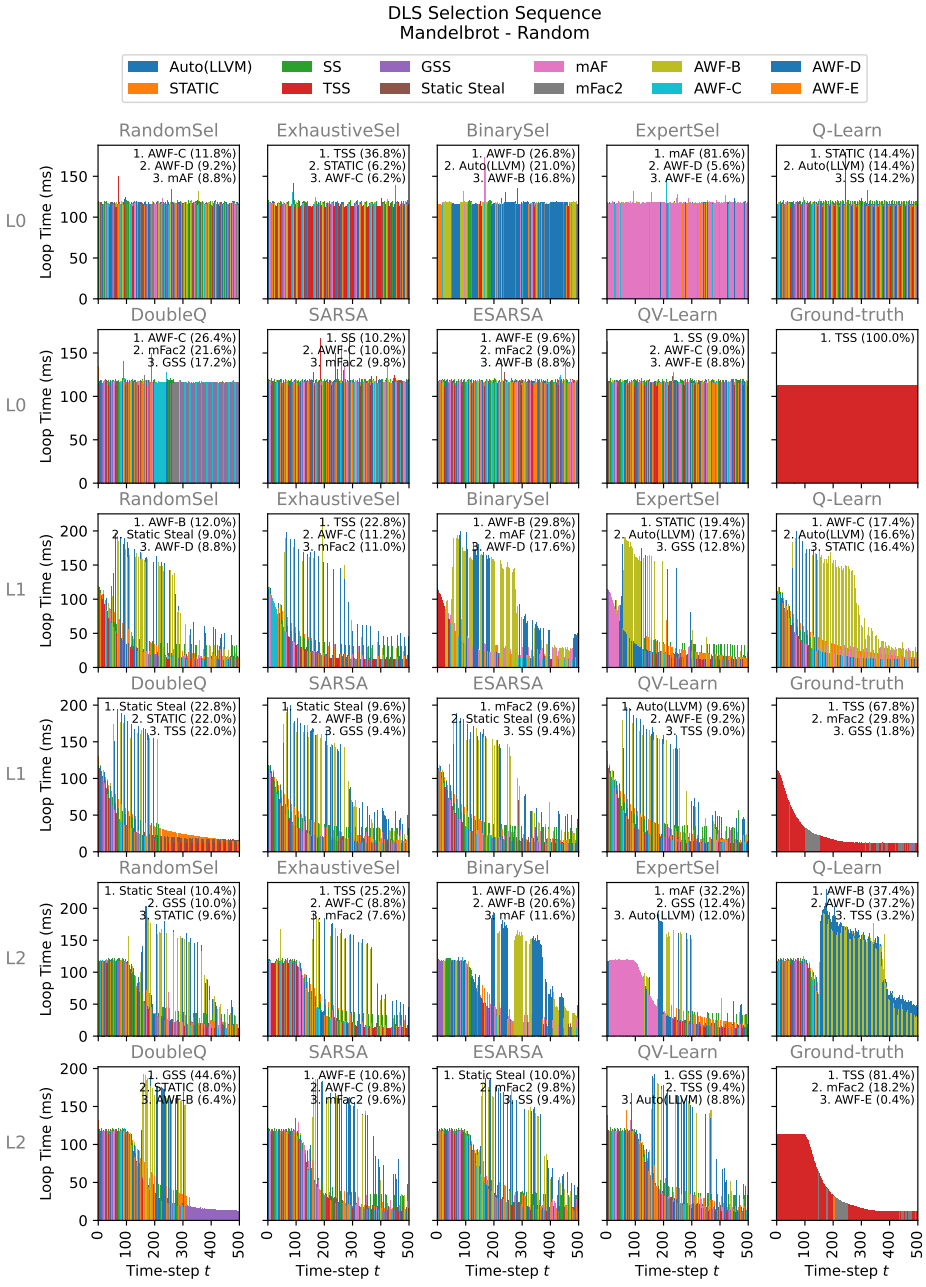

Figure A.13: Mandelbrot DLS Selection Sequence with *Random* configuration.

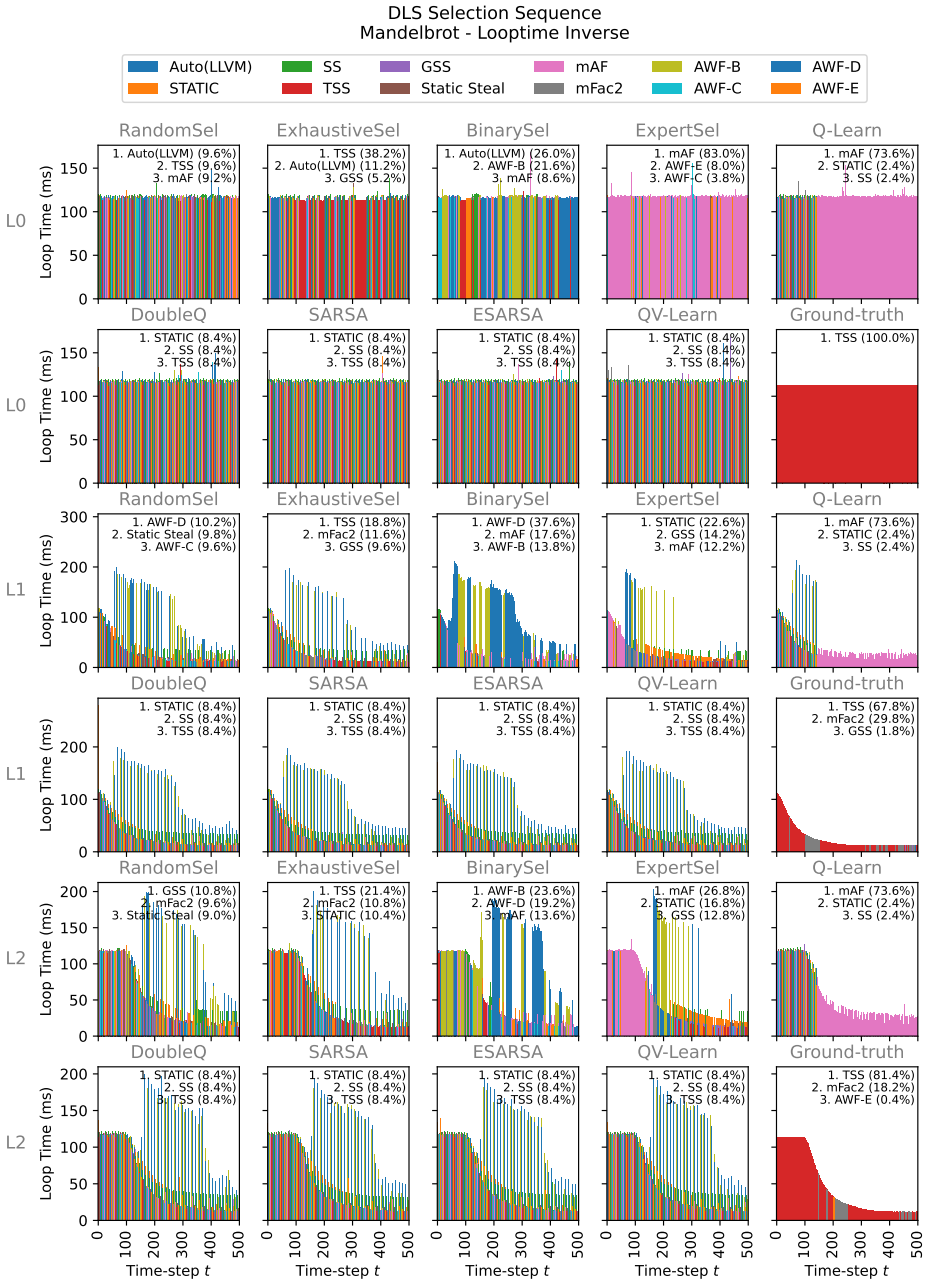

Figure A.14: Mandelbrot DLS Selection Sequence with *Looptime Inverse* configuration.

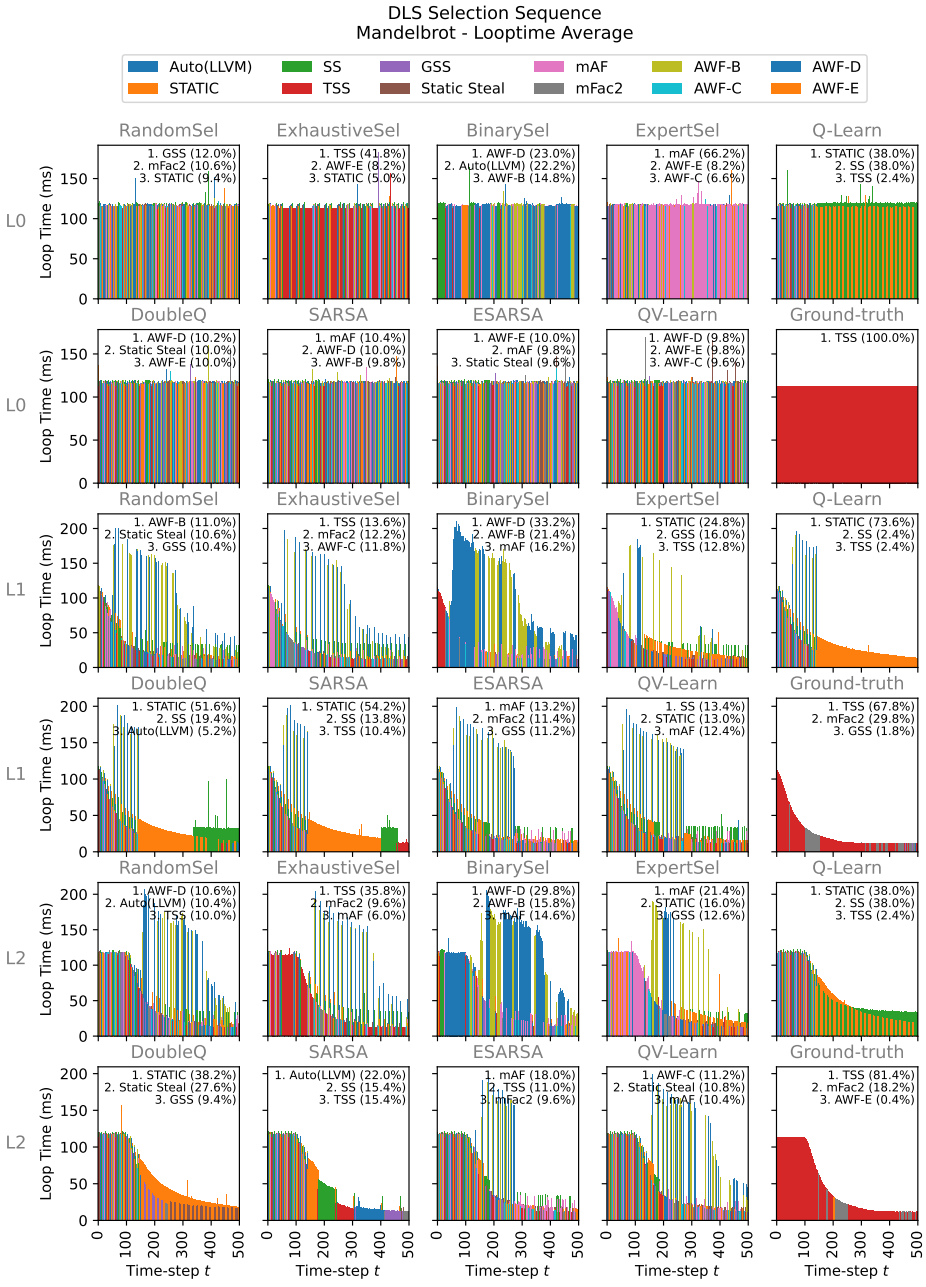

Figure A.15: Mandelbrot DLS Selection Sequence with *Looptime Average* configuration.

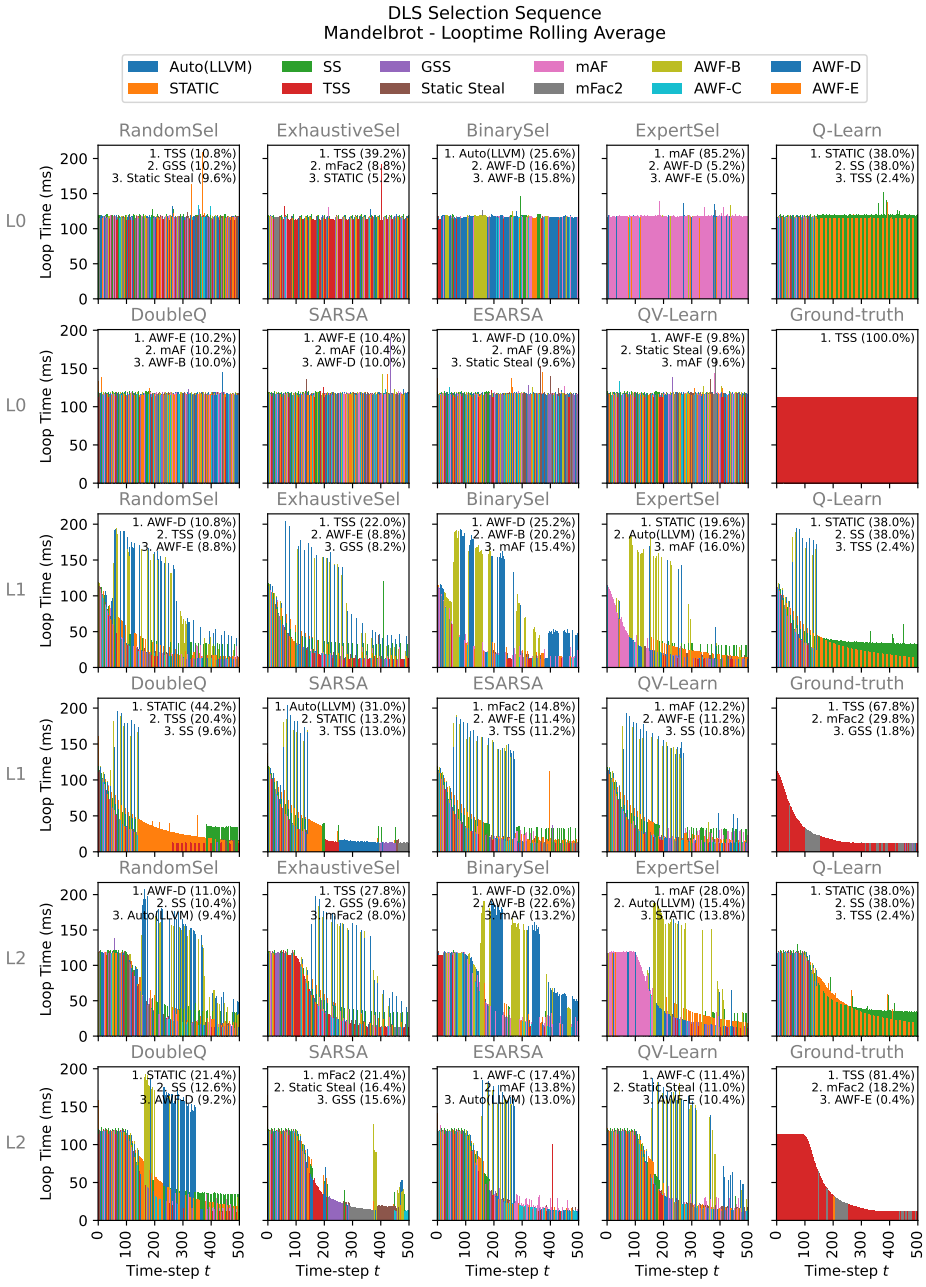

Figure A.16: Mandelbrot DLS Selection Sequence with *Looptime Rolling* configuration.

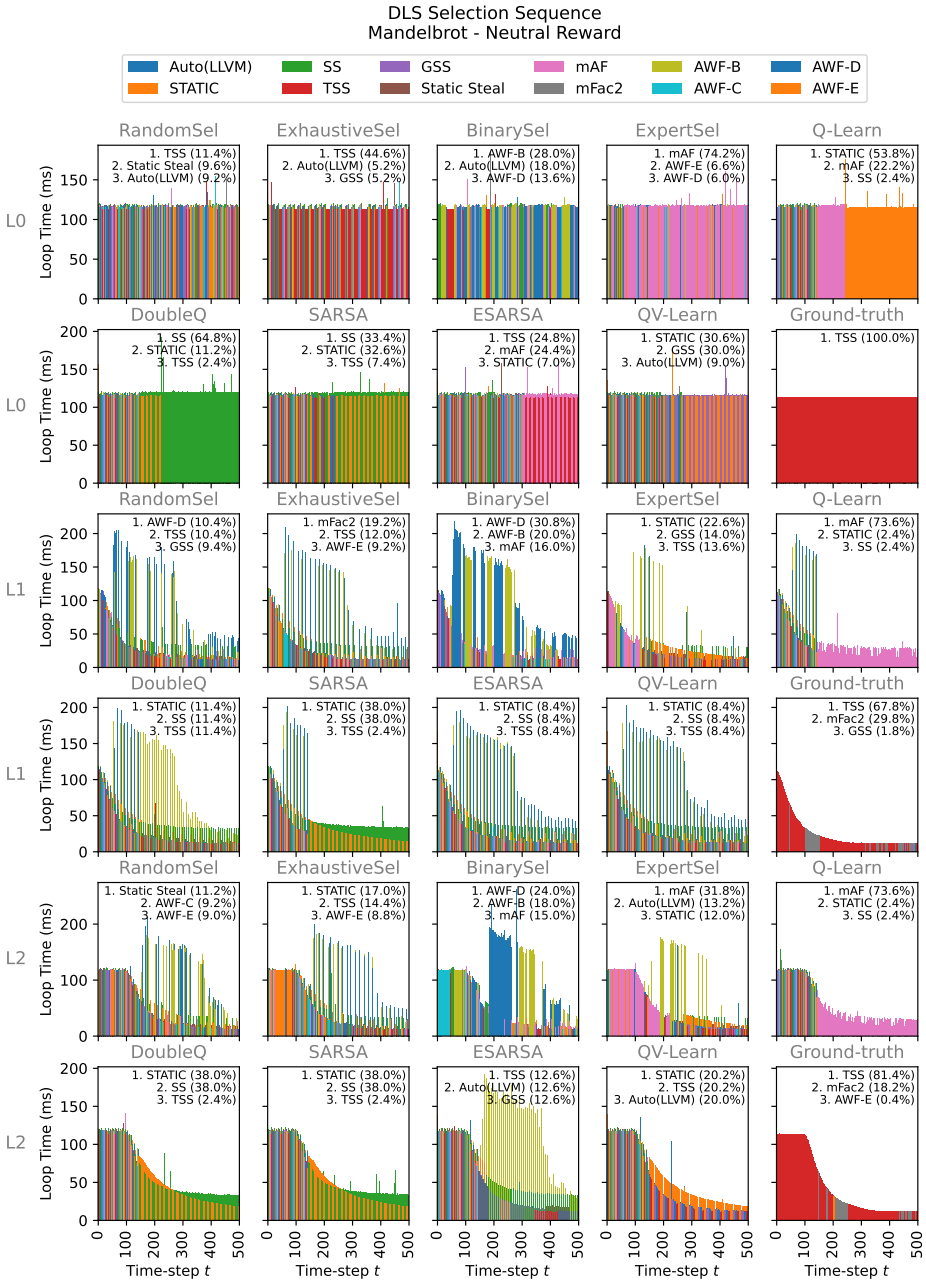

Figure A.17: Mandelbrot DLS Selection Sequence with *Neutral Reward* configuration.

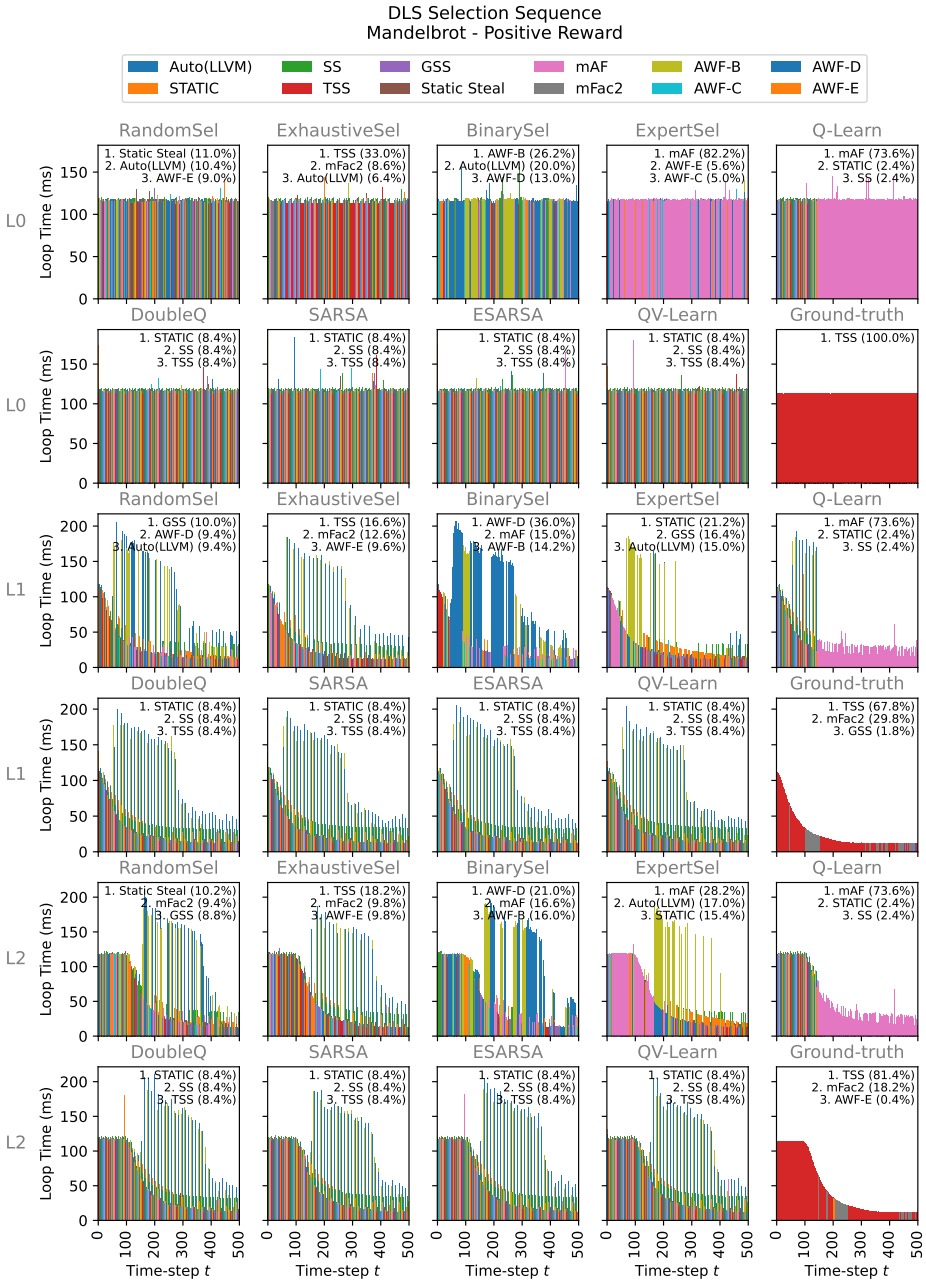

Figure A.18: Mandelbrot DLS Selection Sequence with *Positive Reward* configuration.

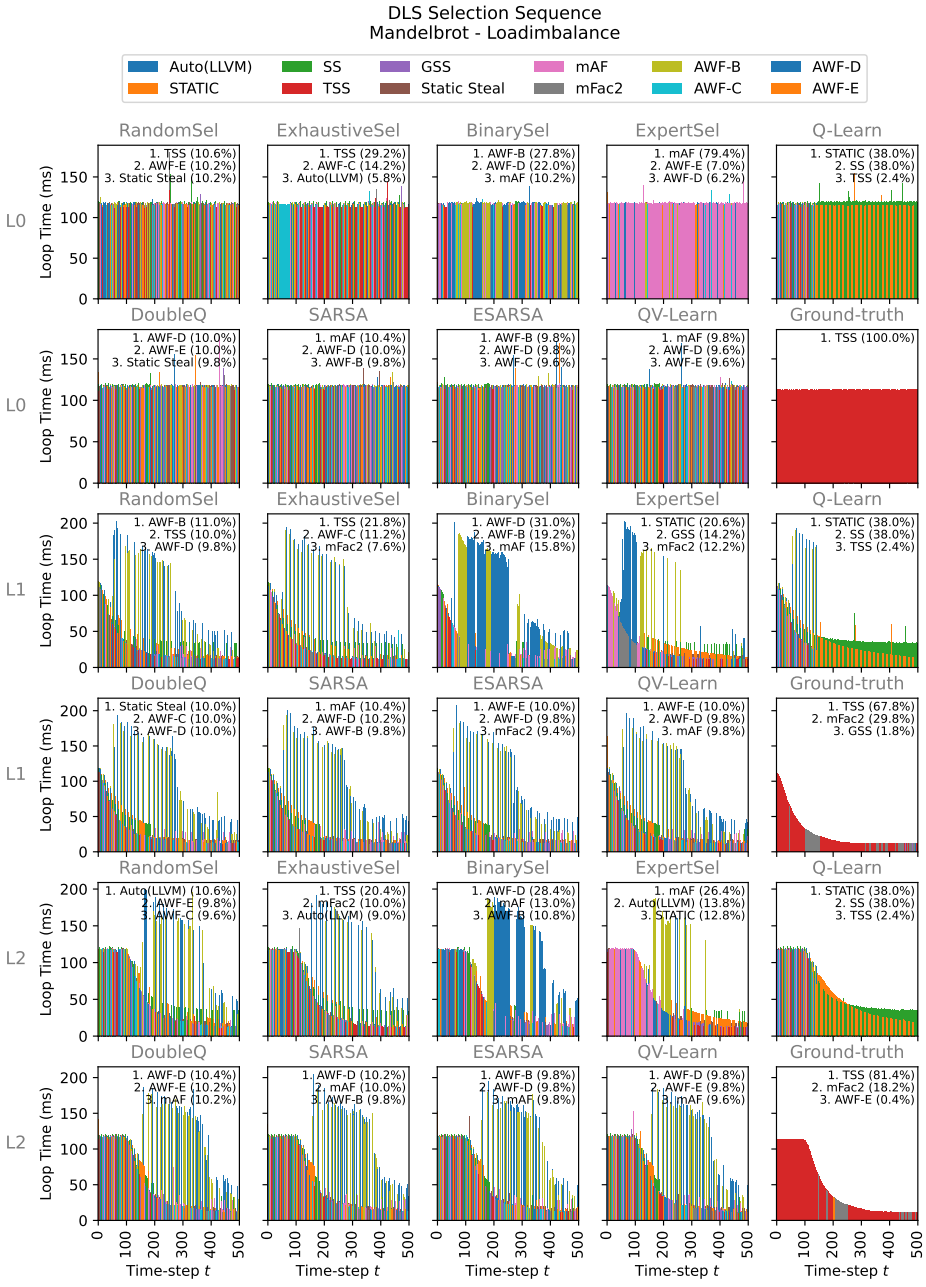

Figure A.19: Mandelbrot DLS Selection Sequence with *Loadimbalance* configuration.

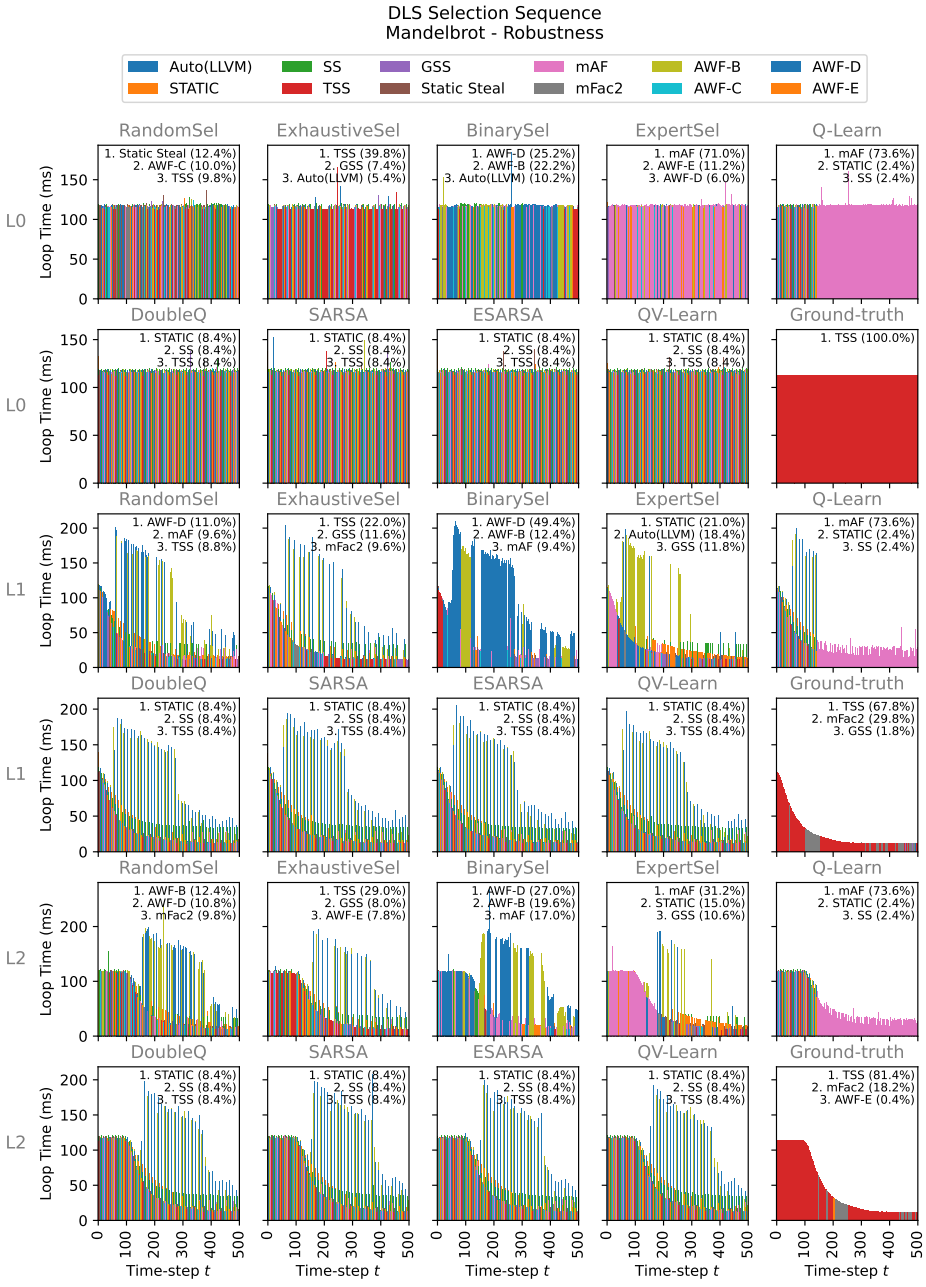

Figure A.20: Mandelbrot DLS Selection Sequence with *Robustness* configuration.

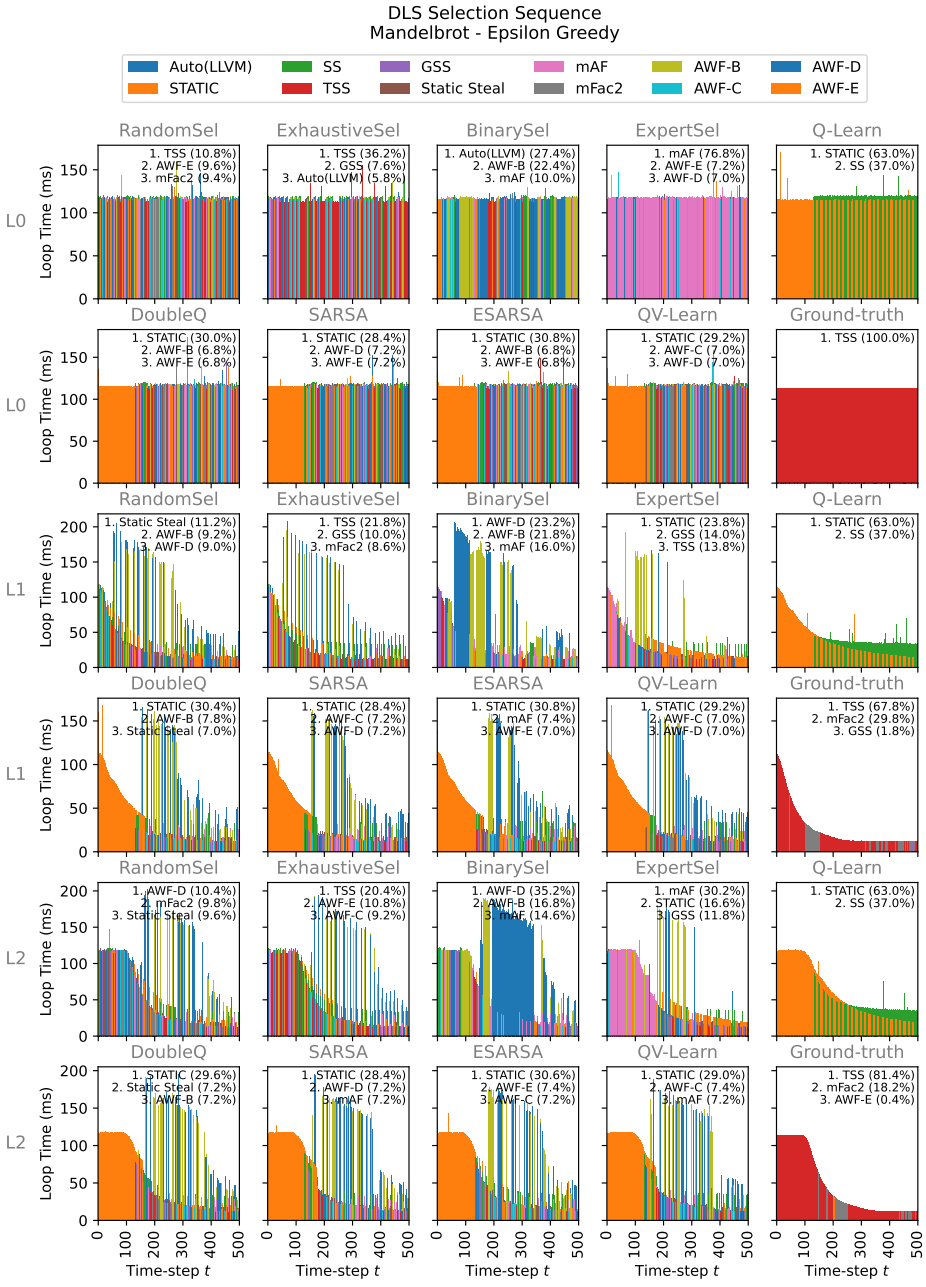

Figure A.21: Mandelbrot DLS Selection Sequence with *Epsilon Greedy* configuration.

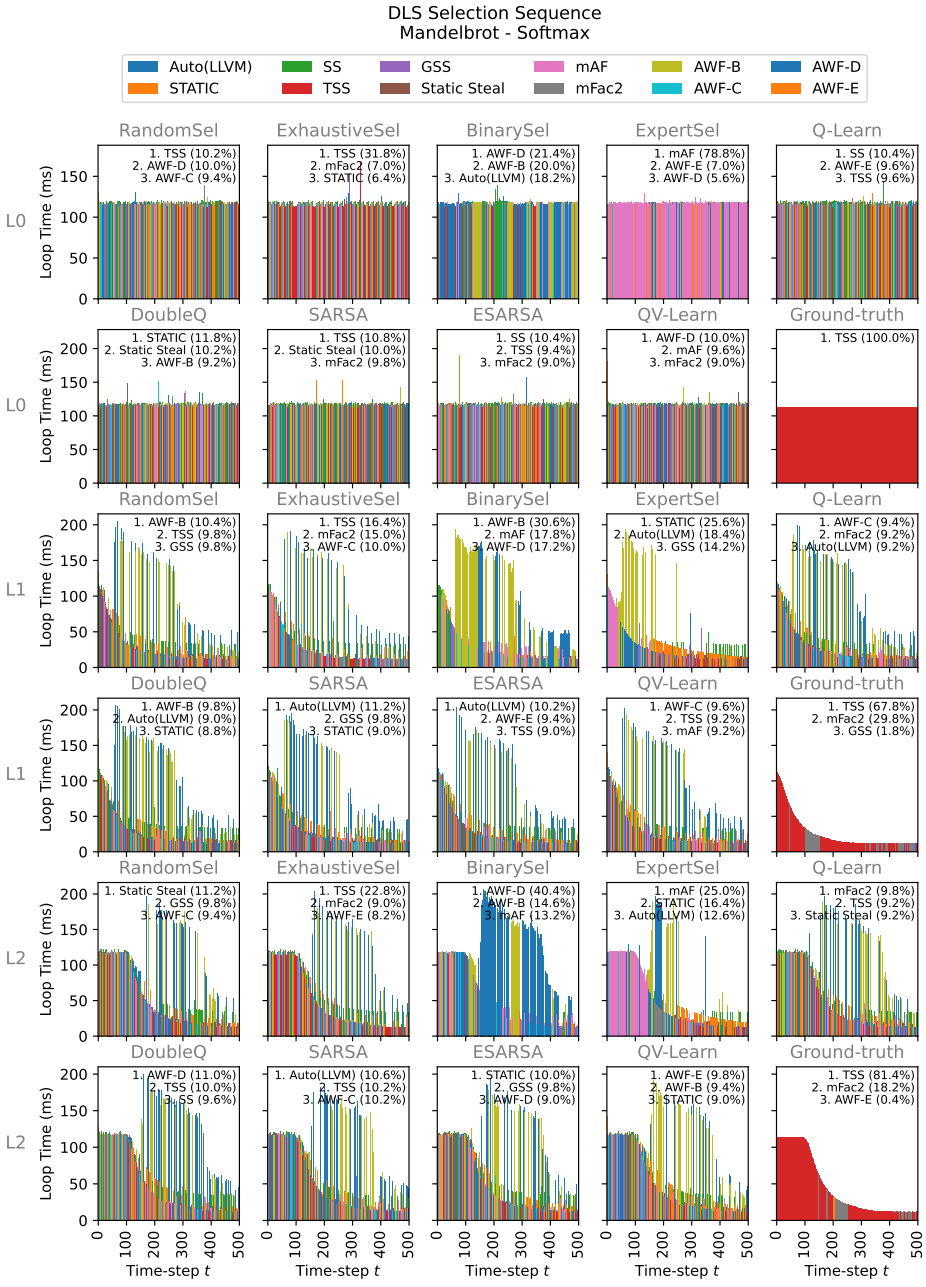

Figure A.22: Mandelbrot DLS Selection Sequence with *Softmax* configuration.

A.3 SPHYNX - Extended Results

A.3.1 Overall Application Performance

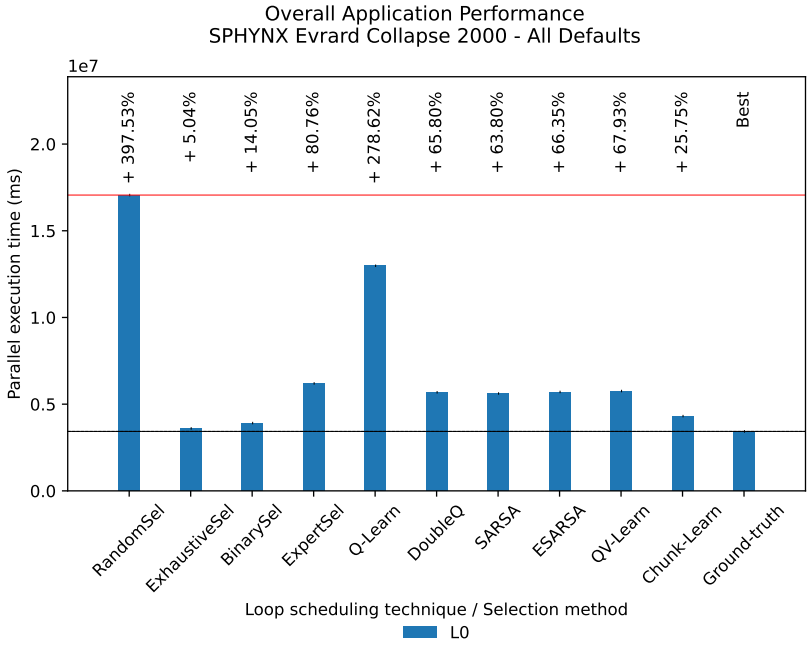

Figure A.23: SPHYNX Evrard Collapse 2000 Overall Application Performance with *All Defaults* configuration.

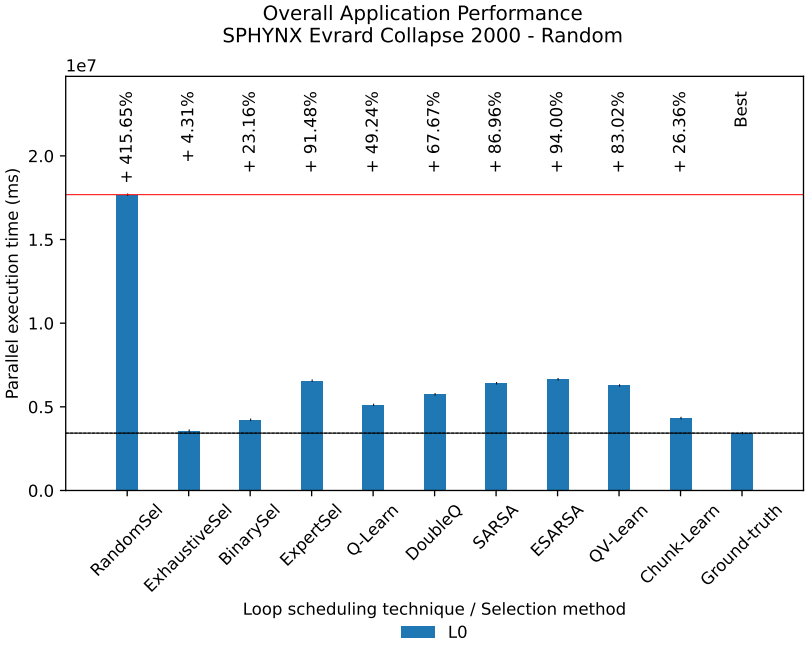

Figure A.24: SPHYNX Evrard Collapse 2000 Overall Application Performance with *Random* configuration.

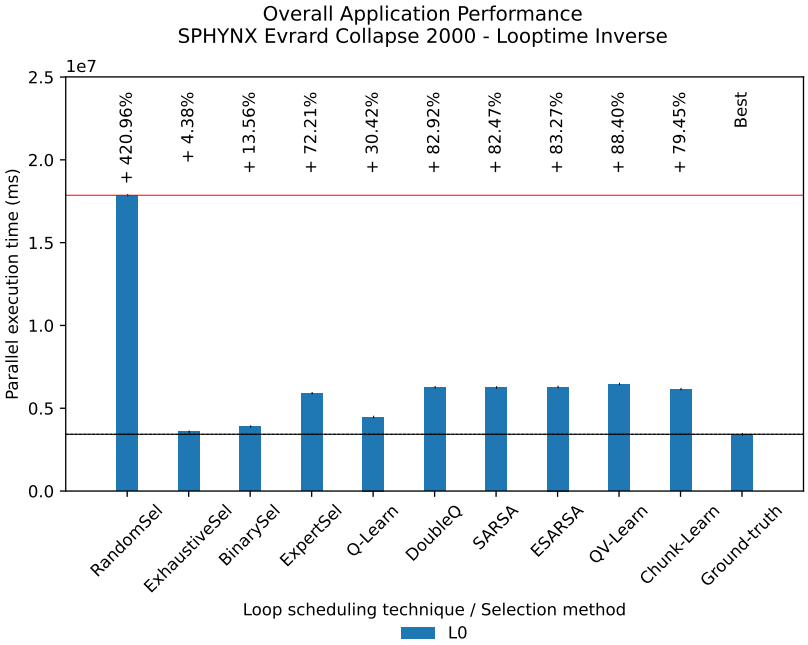

Figure A.25: SPHYNX Evrard Collapse 2000 Overall Application Performance with *Looptime Inverse* configuration.

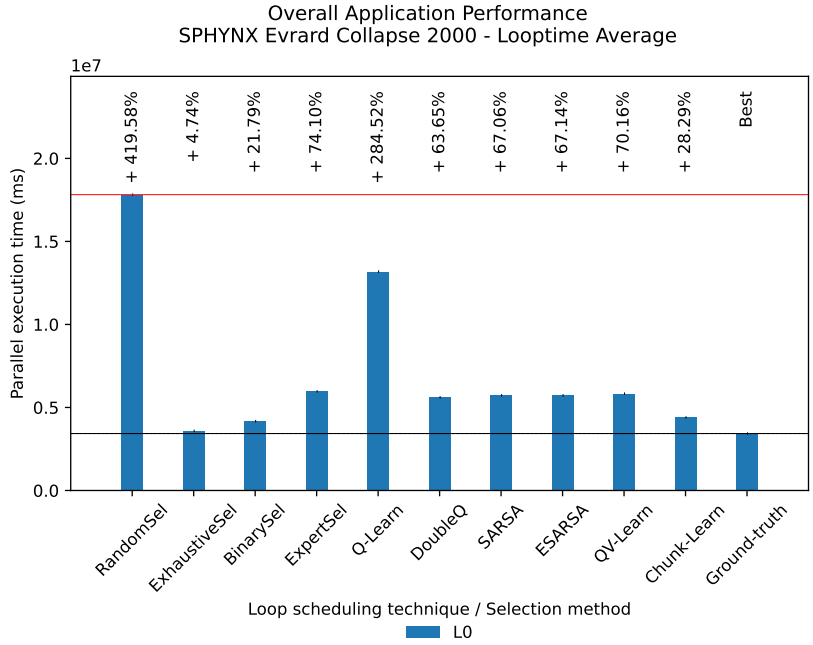

Figure A.26: SPHYNX Evrard Collapse 2000 Overall Application Performance with *Looptime Average* configuration.

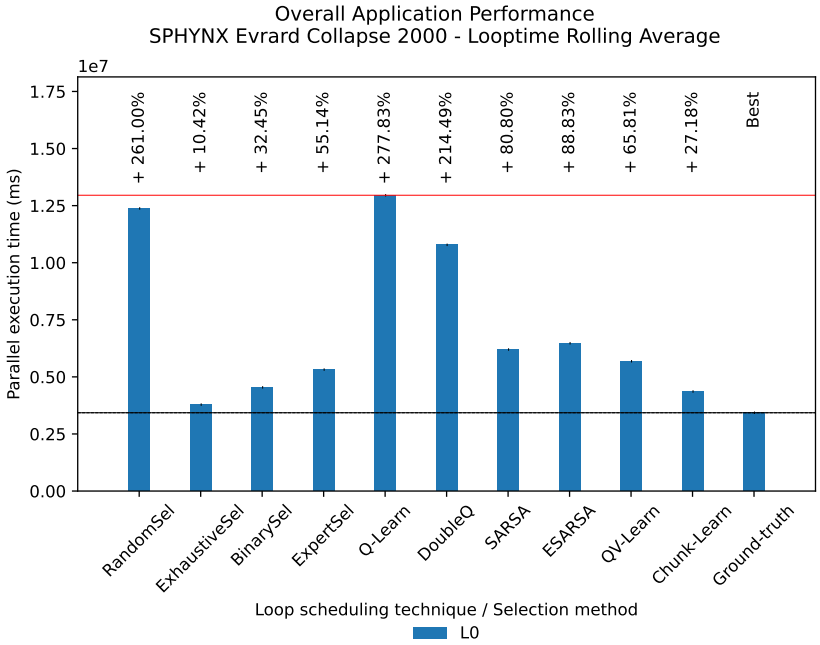

Figure A.27: SPHYNX Evrard Collapse 2000 Overall Application Performance with *Looptime Rolling* configuration.

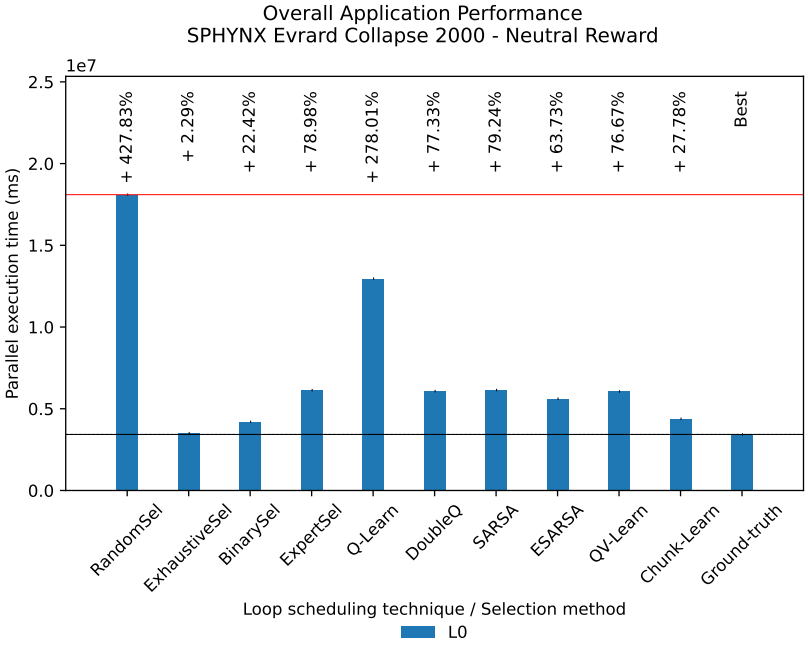

Figure A.28: SPHYNX Evrard Collapse 2000 Overall Application Performance with *Neutral Reward* configuration.

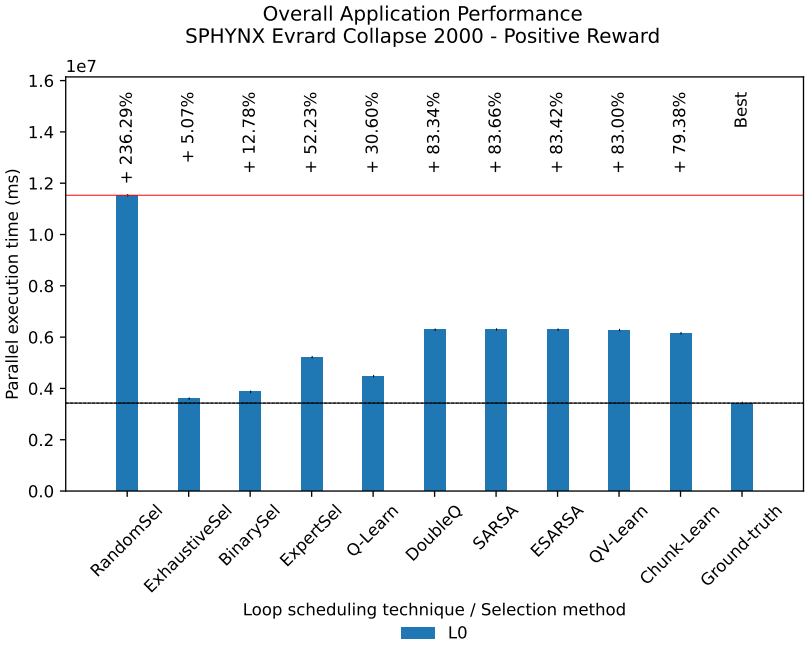

Figure A.29: SPHYNX Evrard Collapse 2000 Overall Application Performance with *Positive Reward* configuration.

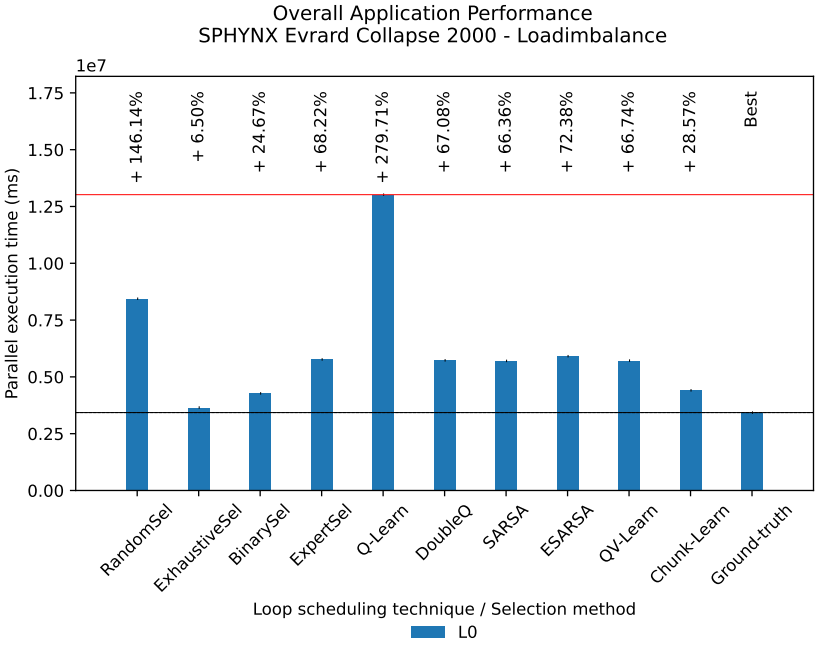

Figure A.30: SPHYNX Evrard Collapse 2000 Overall Application Performance with *Loadimbalance* configuration.

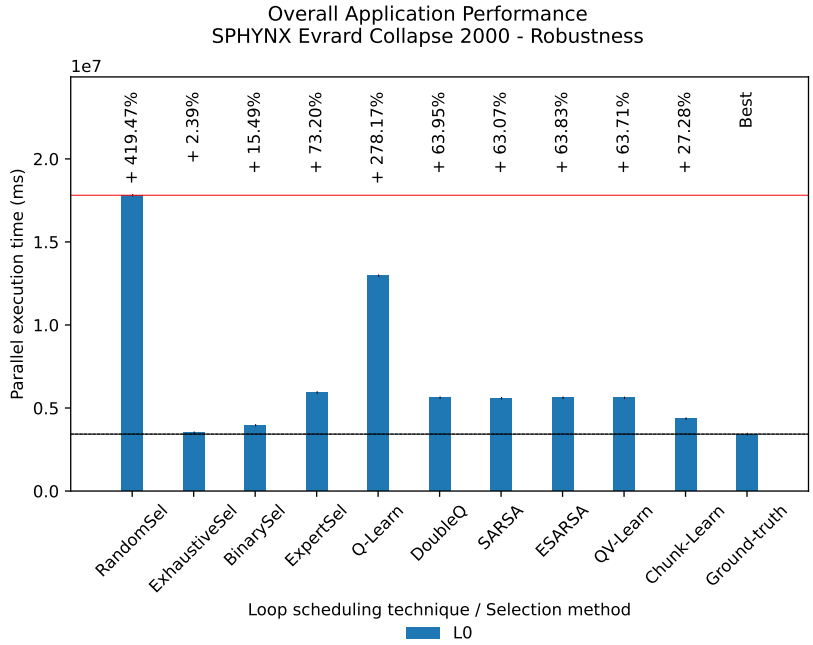

Figure A.31: SPHYNX Evrard Collapse 2000 Overall Application Performance with *Robustness* configuration.

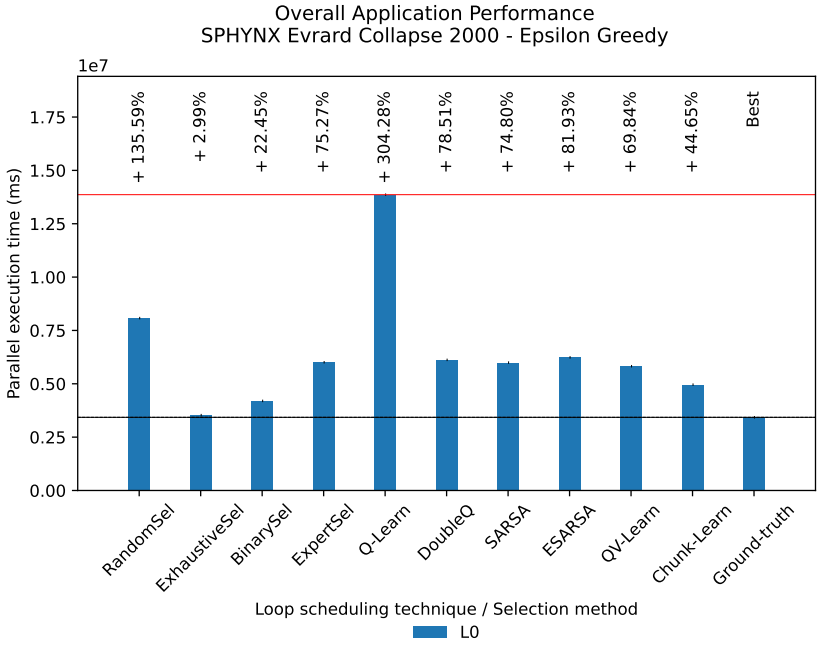

Figure A.32: SPHYNX Evrard Collapse 2000 Overall Application Performance with *Epsilon Greedy* configuration.

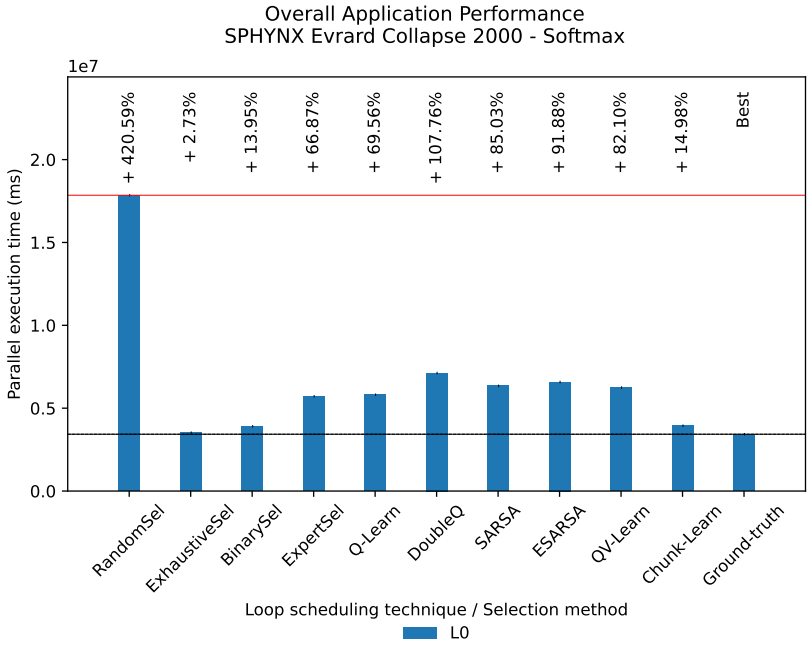

Figure A.33: SPHYNX Evrard Collapse 2000 Overall Application Performance with *Softmax* configuration.

A.3.2 DLS Selection Sequence

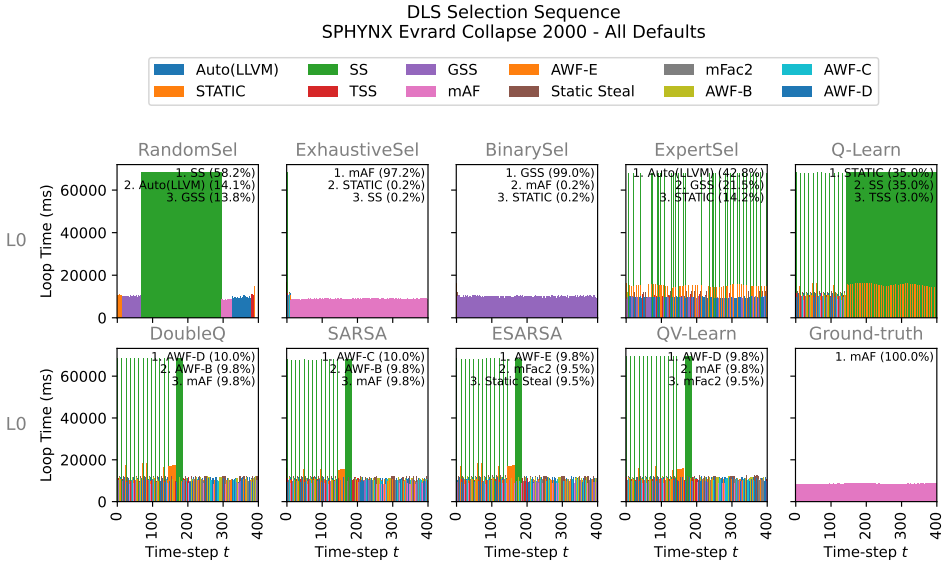

Figure A.34: SPHYNX Evrard Collapse 2000 DLS Selection Sequence with *All Defaults* configuration.

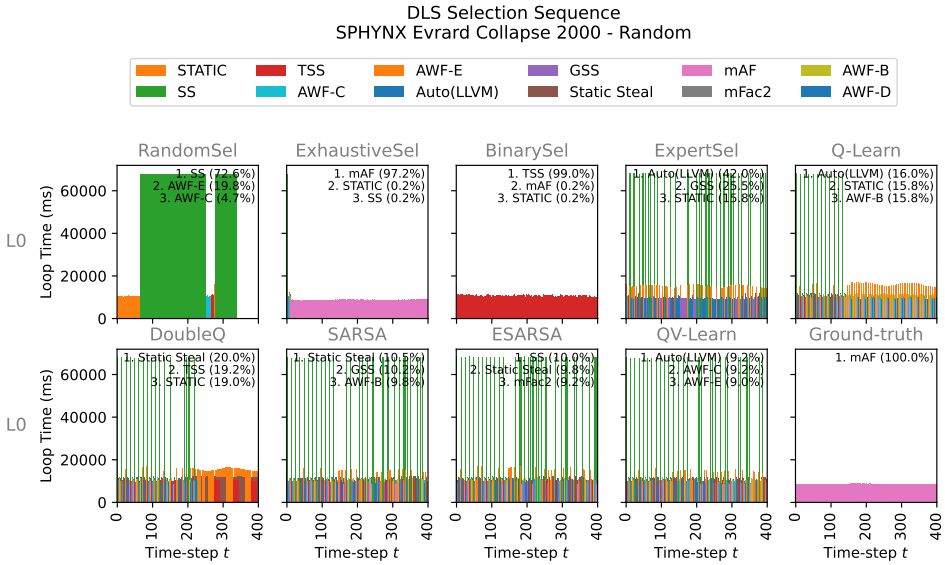

Figure A.35: SPHYNX Evrard Collapse 2000 DLS Selection Sequence with *Random* configuration.

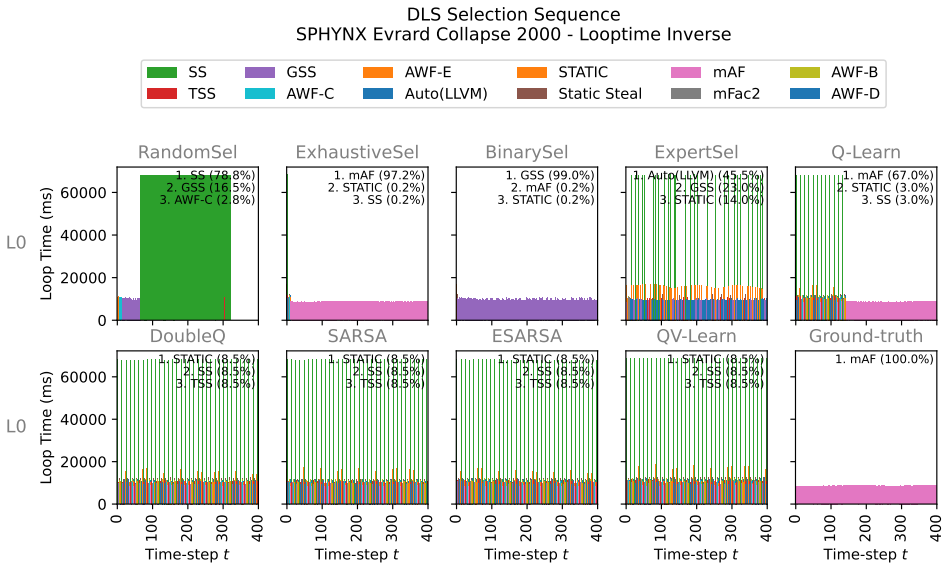

Figure A.36: SPHYNX Evrard Collapse 2000 DLS Selection Sequence with *Looptime Inverse* configuration.

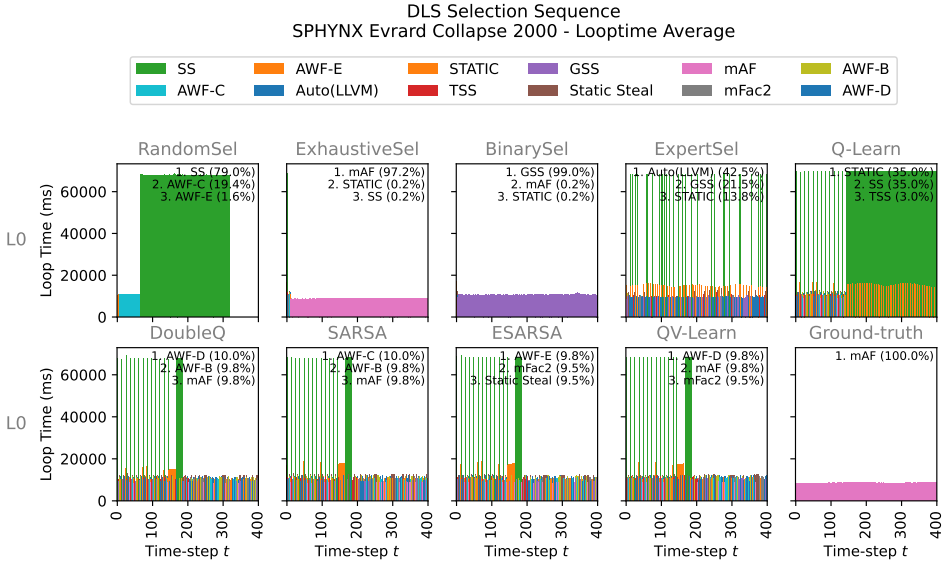

Figure A.37: SPHYNX Evrard Collapse 2000 DLS Selection Sequence with *Looptime Average* configuration.

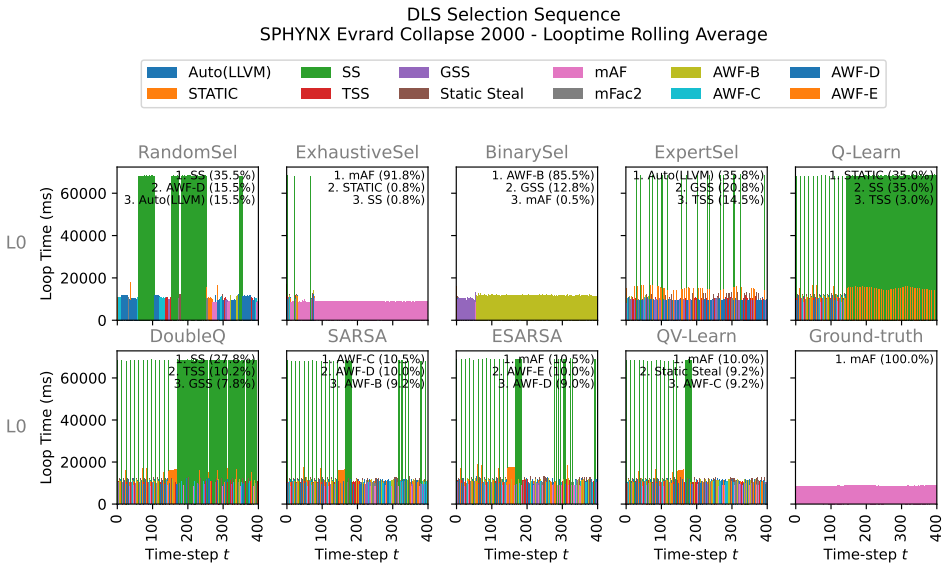

Figure A.38: SPHYNX Evrard Collapse 2000 DLS Selection Sequence with *Looptime Rolling* configuration.

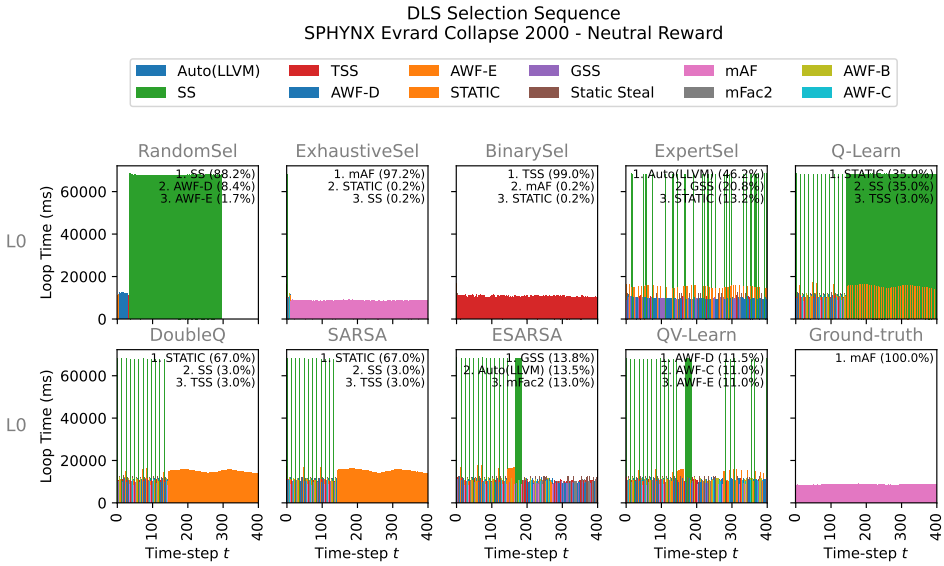

Figure A.39: SPHYNX Evrard Collapse 2000 DLS Selection Sequence with *Neutral Reward* configuration.

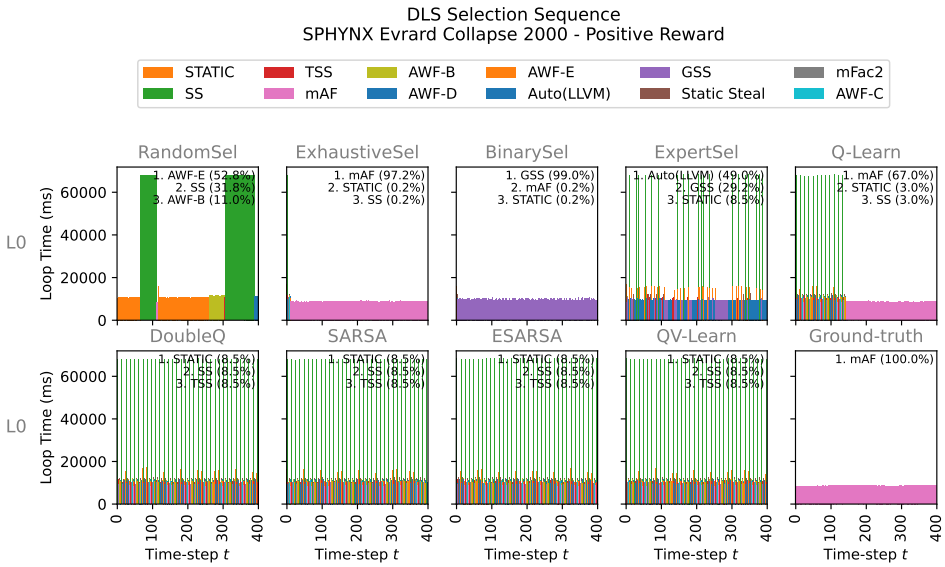

Figure A.40: SPHYNX Evrard Collapse 2000 DLS Selection Sequence with *Positive Reward* configuration.

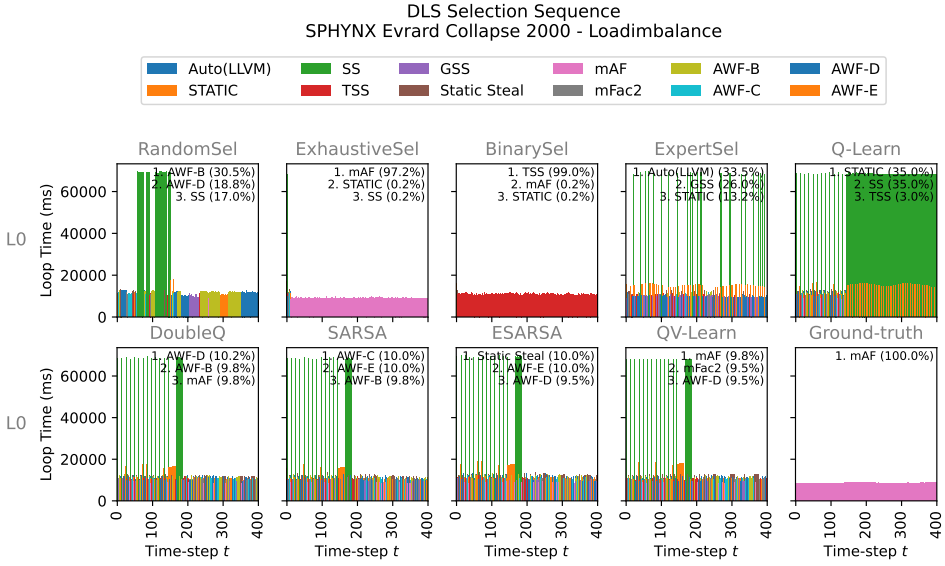

Figure A.41: SPHYNX Evrard Collapse 2000 DLS Selection Sequence with *Loadimbalance* configuration.

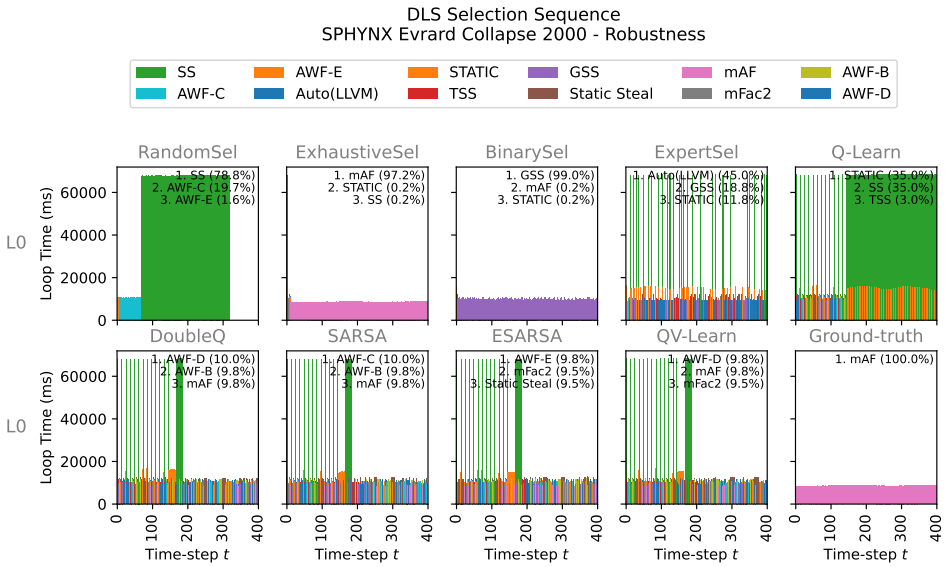

Figure A.42: SPHYNX Evrard Collapse 2000 DLS Selection Sequence with *Robustness* configuration.

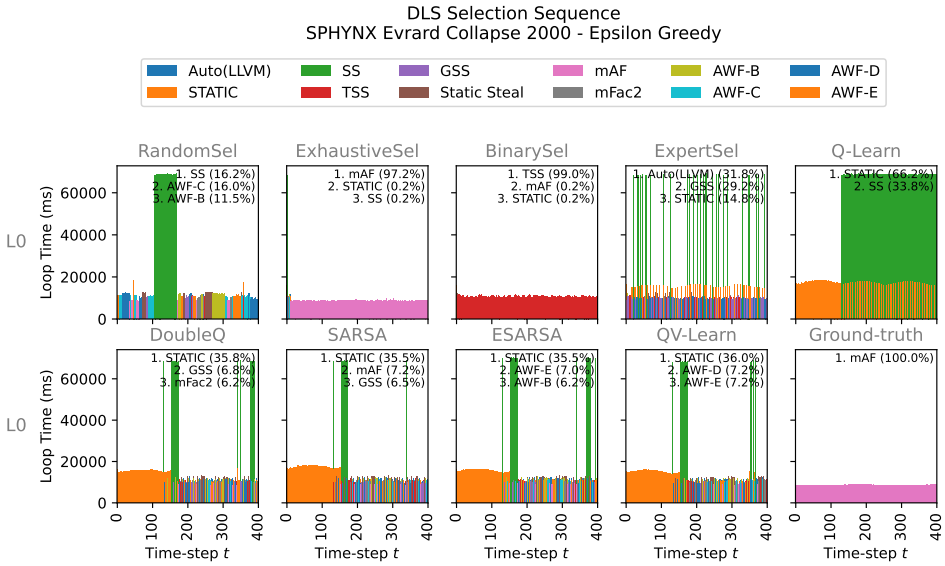

Figure A.43: SPHYNX Evrard Collapse 2000 DLS Selection Sequence with *Epsilon Greedy* configuration.

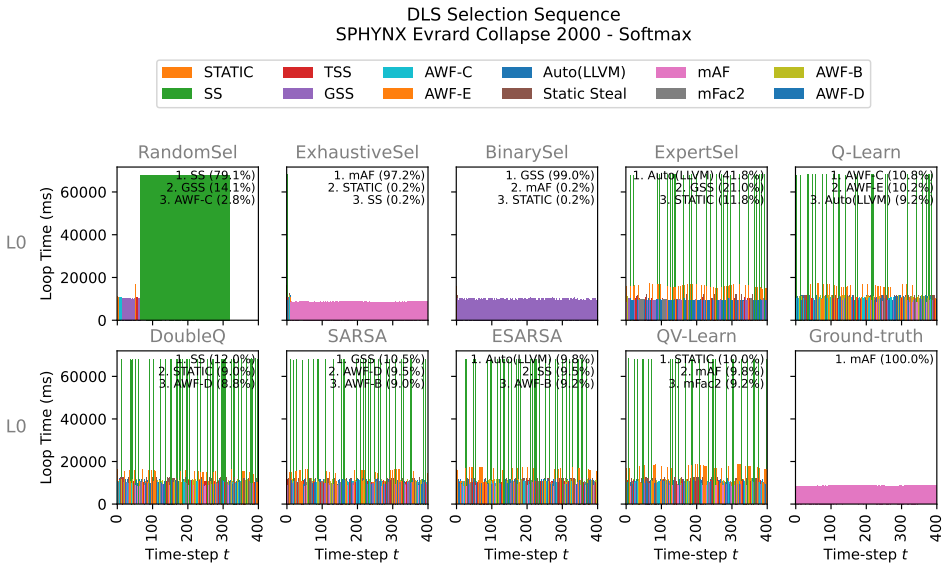

Figure A.44: SPHYNX Evrard Collapse 2000 DLS Selection Sequence with *Softmax* configuration.

University of Basel

Faculty of Science

Declaration on Scientific Integrity

(including a Declaration on Plagiarism and Fraud)
Translation from German original

Title of Thesis: **Automated Selection of Scheduling Algorithms for Parallel Scientific Applications using Reinforcement Learning with OpenMP**

Name Assesor: _____

Name Student: _____

Matriculation No.: _____

With my signature I declare that this submission is my own work and that I have fully acknowledged the assistance received in completing this work and that it contains no material that has not been formally acknowledged. I have mentioned all source materials used and have cited these in accordance with recognised scientific rules.

Place, Date: _____ Student: _____

Will this work be published?

☐ No

☐ Yes. With my signature I confirm that I agree to a publication of the work (print/digital) in the library, on the research database of the University of Basel and/or on the document server of the department. Likewise, I agree to the bibliographic reference in the catalog SLSP (Swiss Library Service Platform). (cross out as applicable)

Publication as of: _____

Place, Date: _____ Student: _____

Place, Date: _____ Assessor: _____

Please enclose a completed and signed copy of this declaration in your Bachelor's or Master's thesis .