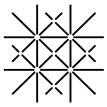


University of Basel

Automated Selection of Scheduling Techniques in OpenMP using Reinforcement Learning

Master Research Project

Natural Science Faculty of the University of Basel
Department of Mathematics and Computer Science
High Performance Parallel And Distributed Computing
https://hpc.dmi.unibas.ch

Examiner: Prof. Dr. Florina M. Ciorba
Supervisor:
Jonas Henrique Muller Korndorfer
Ali Omar Abdelazim Mohammed
Ahmed Hamdy Mohamed Eleliemy

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

13^{th} of September 2021

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 but no processor has started doing so. This results in uneven execution progress among the parallel processing units. This imbalance stems from varying application, algorithmic, and systemic characteristics. Computationally-intensive applications often represent irregular workloads and the computing systems used to run such applications often consist of heterogeneous processors and may be affected by non-uniform memory access, operating system noise, and contention due to the sharing of resources. Load imbalance can effectively be alleviated 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.

Parallel applications, such as OpenMP programs, are 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 imbalanced loads 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 runtime. As a solution we propose an extension to the existing LB4OMP library consisting of a reinforcement learning technique (named **ReinforcedSel**) as an alternative way to achieve automated selection of scheduling algorithms for OpenMP loops. **ReinforcedSel** implements a reinforcement learning agent for every parallel application loop, using two popular learning methods, Q-Learning and SARSA. **ReinforcedSel** leverages the schedule(auto) directive in OpenMP's runtime library to automatically select a scheduling algorithm during execution for automated application load balancing.

The results show that **ReinforcedSel** increases performance compared to the *static* scheduling technique by up to **15.6%**. This result is on par with the automatic load balancing methods proposed in LB4OMP using expert knowledge instead of machine learning. Additionally we showed that when using Q-Learning and SARSA as the agents learning methods, the learning rate and discount factor associated with the agent's configuration affect the experiment's results only in a minor way.
This project reveals the potential of machine learning as a promising tool to increase parallel applications' performance in an unsupervised fashion.

Table of Contents

1 Introduction

Scientific applications that require a high number of time-steps to converge toward a solution are considered time- stepping scientific applications (TSSA). These applications often involve one or several parallel loops that have a high number of loop iterations. The existence of loops with independent iterations in a program is a good resource of parallelism. While running a program on a HPC system, various factors can lead to reduced performance during execution - such as load imbalance, inter-processor communication, general communication overhead, and processes synchronization.

Load imbalance is one of the primary factor that affects the performance of an application [8] and is mainly caused by fluctuations in problem, algorithm, and system characteristics. This lets load imbalance have a complex influence on an applications performance. To lessen the effects of load imbalance during the application's runtime, scheduling of the workload among the different parallel compute units is essential. For this purpose numerous loop scheduling algorithms have been conceived [9, 11]. As one can imagine, it is a non-trivial issue to select the correct scheduling algorithm [10] for an application in advance to achieve a balanced load - as it partly depends on the characteristics of the specific application (and even its input parameters) and in other parts on the unpredictability of the environment it is executed in. The only way to know about the effectiveness of a scheduling algorithm, is to test it for a specific application and system combination. On the other hand the process of testing all available scheduling techniques is lengthy, costly and therefore highly impractical. To mitigate the selection problem, an automatic mechanism to choose the best suited scheduling algorithm at runtime is needed. Such a mechanism can employ any number of techniques and expert knowledge, as the LB4OMP [13, 14] library already implements four automated scheduling algorithm selection methods `RandomSel`, `ExhaustiveSel`, `BinarySel` and `ExpertSel`. We propose a fifth and new method called **ReinforcedSel** based on machine learning. We think that the selection process of loop scheduling algorithm can benefit from the extensive range of machine learning techniques and can further improve the performance of parallel applications. As a first step we propose the usage of *reinforcement learning*, since it is a good fit the problem space [4, 6, 16, 17].

1.1 Contributions and Research Problem

The work done in this paper is summarized best by the questions "How to select scheduling algorithms during execution efficiently?" from which the following contributions and hypotheses are derived:

C.1 Extend the LB4OMP library with two reinforcement learning algorithms, SARSA and Q-Learning as additional options for the schedule(auto) clause.

C.2 Compare the performance of different scientific applications executed on real systems using the newly implemented automatic selection method with the existing expert selection methods in Auto4OMP.

C.3 Compare the scheduling technique selection (frequency) for every automated selection approach.

H.1 The new automatic algorithm selection method using reinforcement learning (**ReinforcedSel**) improves overall parallel execution time.

H.2 **ReinforcedSel** adapts to applications' need of different scheduling algorithms across time-steps for each loop to achieve high performance.

H.3 **ReinforcedSel** adapts to applications' need of different scheduling algorithms for different loops in a single time-step (and across time-steps).

1.2 Organization

This project report is structured as follows: A summary of previous work on dynamic loop scheduling techniques, automatic selection methods and reinforcement learning is given in chapter 2. The design and implementation of the proposed extension is presented in chapter 3. The experimental setup and results with a detailed performance analysis and comparison to other scheduling algorithms and automatic selections methods are given in chapter 4. A quantitative perspective on the experimental results as well as the conclusion and future work are presented in chapter 5.

2

Background And Related Work

This chapter gives an overview of the techniques from the perspective of both scheduling algorithms for load balancing, the automatic selection of aforementioned algorithms and machine learning/reinforcement learning as well as a summary on previous work on automatic schedule selection using reinforcement learning.

2.1 Scheduling And Load Balancing

This section touches on the subject of loop scheduling as a means of load balancing the work of a parallel application and as a consequence increasing performance.

Load imbalance is a significant performance degradation factor in computationally-intensive applications. It is defined by a processors being idle while there is still work ready to be executed that has not been allocated to any processor. This results in irregular execution progress among the parallel processing units. Computationally-intensive applications such as time-stepping scientific applications, often represent uneven workloads, e.g., due to boundary conditions, non-uniform domain, convergence, conditions, and branches. Computing systems may consist of heterogeneous processors and may be affected by nonuniform memory access, operating system noise, and contention due to sharing of resources. Load imbalance can be mitigated by an efficient mapping of tasks onto processing units. Finding optimal schedules is NP-hard [10]. In addition, many scheduling heuristics with distinct characteristics have been proposed over the years [9, 11].
OpenMP is the most pervasive standard for harnessing the power of multi-threading provided by modern HPC hardware. The OpenMP standard specifies three scheduling algorithms, which are insufficient to address the load imbalance that arises during the execution of multi-threaded applications. LB4OMP [13] is a dynamic load balancing library for multi-threaded applications that extends the LLVM OpenMP runtime library. LB4OMP supports 12 dynamic and adaptive loop scheduling techniques in addition to those existing in the standard-compliant OpenMP libraries.

2.2 Automatic Selection of Scheduling Algorithms

This section examines the process of automatically selecting a scheduling algorithm from a known portfolio of algorithms to overcome choice paralysis.

The increased number of scheduling techniques in OpenMP (as well as in the LB4OMP library), leads to the difficulty of choosing the most performing scheduling algorithm for a given application-system pair. This is known as the algorithm selection problem [15]. A user needs to decide on the scheduling algorithm which yield the best performance from a set of tens of scheduling techniques on a per loop, per application, and per system configuration basis. Increasing the number of choices stifles the usability of the library and may lead to overwhelming the user [7] and sub-optimal decisions.
To avoid analysis paralysis, the OpenMP standard provides the **auto** keyword as an argument to the schedule clause. This option delegates the scheduling decision to the compiler/runtime implementation. Currently, most widespread and successful OpenMP implementations do not take full advantage of the OpenMP **auto** scheduling option. For example, the GNU OpenMP [1] runtime library maps auto $\rightarrow$ static, which equally divides the number of loop iterations over the threads. The LLVM OpenMP runtime library, maps auto to an optimized implementation of guided scheduling. This displays that the state of practice implementations of automatic schedule selection has not yet matured enough to be of any significance for end users and indicates that more research in this area is needed.
Recently a paper with the title *Automated Load Balancing in OpenMP* (from now on referred to as Auto4OMP [14]), made strides towards an automatic loop scheduling system that requires no user interaction and no prior knowledge about the application by implementing four distinct schedule selection mechanisms that effectively unburden the user of the scheduling algorithm and chunk selection problem. They showed that their methods could dynamically and adaptively refine their selection during application execution, yielding decreased application performance variation and achieving performance that is better than any state of the practice method [14].

2.3 Concepts Of Reinforcement Learning

This section describes the basic concepts on reinforcement learning techniques which will be employed in the design of the agent for the stated problem of this project.

In computer science the area of machine learning concerns itself with the exploration of algorithms that grow with learning which produces intelligent programs. The basic concept is the interaction between an intelligent system, the agent, and the environment in which the agent operates. Machine learning can be generally classified into supervised, unsupervised and reinforcement learning (RL).
In supervised learning, the agent is told how to behave while with unsupervised learning, the agent learns to reduce the problem size but may learn the correct outputs only with an associated error margin. As a middle ground between supervised and unsupervised learning there is reinforcement learning. An agent receives feedback for performing an action, which

guides it to the correct solution. The interaction between the agent and the environment is illustrated in figure 2.1.

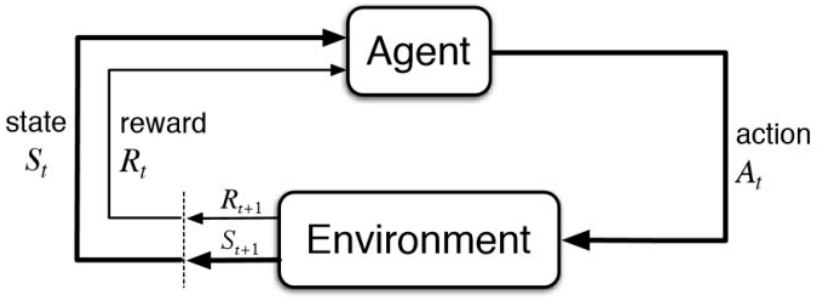


Figure 2.1: Basic components of a reinforcement learning setup. [18]

Reinforcement learning is of particular interest for our implementation because of its adaptability to both accessible and inaccessible environments. RL focuses on a goal-driven approach for solving learning problems by interacting with the complex and uncertain environments (such as large-scale heterogeneous computer systems). Reinforcement learning involves an agent, which learns the behavior of a dynamic environment through trial and error. The agent is given an immediate reward for taking an action and the environment is transferred into the state following the action. The agent learns the optimal path that will lead to the goal (e.g. minimizing parallel execution time or load imbalance) by learning through the experience gained about the states, actions, and rewards. The trial and error learning mechanism and the concept of reward makes the reinforcement learning distinct from other learning techniques. A challenging problem and a key aspect in reinforcement learning is the trade-off between exploration and exploitation. To exploit is to use the best experienced action, and to explore, the agent has to try new actions to discover better action selections for the future.
The components of the reinforcement learning system consist of distinct states, actions, rewards, a policy, the environment and in some cases a model of the environment. In chapter 3 we will define how these components map to the problem of automatic schedule selection.

All reinforcement learning problems are based on value functions to determine the return for being in a particular state. Two important concepts involved here are the policy and the value. A policy defines the learning agent's way of behaving at a given time. The policy is a mapping from the the environment's states to actions to be taken when in those states. Given a policy, the expected return can be obtained from a given state. The value is the expected return that can be achieved by being in a particular state when following a given policy. The optimal value function assigned to each state is the largest return achievable by a given policy. The policy whose value functions are optimal, is the optimal policy. We can find the optimal policy through an optimal value function through value iteration. The policy iteration finds the optimal policy directly without the optimal value function. There are two designs for learning optimal policy - one is through a model-free learning and the other is through a model-based learning (see figure 2.2 for an overview of different

reinforcement learning categories).

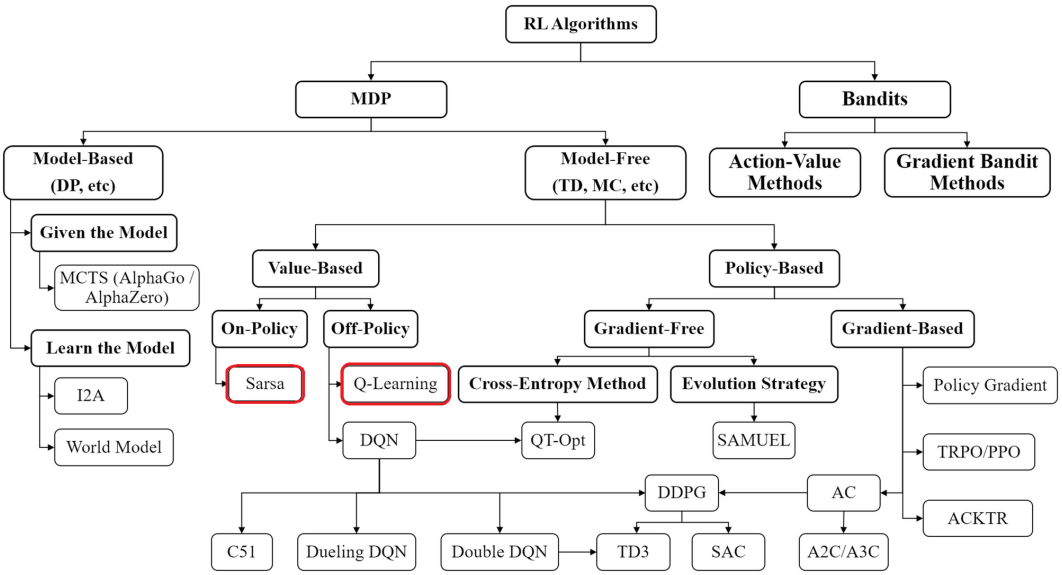


Figure 2.2: Categorisation of different reinforcement learning techniques. [20]

The model-based method uses a model and its utility function. The model-free approach uses a action-value function (Q) and does not require a model for the learning process. Temporal Difference (TD) learning is a model free approach of reinforcement learning. The TD methods are classified in on-policy and off policy methods. The State-Action-Reward-State-Action (SARSA) learning method is an on-policy method, while the Q-Learning method is an off-policy method [18]. The SARSA learning algorithm learns the transitions from a state-action pair to another state-action pair and finds the policy by using a greedy approach. The Q-function directly approximates to $Q*$ (optimal value) independent of the policy being followed. In Q-Learning, the agent chooses the action with the maximum Q-value from a particular state. The pseudocode for both algorithm is listed in figure 2.3.

Initialize $Q(s,a)$ arbitrarily
Repeat (for each episode):
  Initialize s
  Choose a from s using policy derived from Q (e.g., ε-greedy)
  Repeat (for each step of episode):
    Take action a, observe r, s'
    Choose a' from s' using policy derived from Q (e.g., ε-greedy)
    $Q(s,a) \leftarrow Q(s,a) + \alpha\big[r + \gamma Q(s',a') - Q(s,a)\big]$
    $s \leftarrow s'$; $a \leftarrow a'$;
  until s is terminal

(a) SARSA: On-policy TD control.

Initialize $Q(s,a)$ arbitrarily
Repeat (for each episode):
  Initialize s
  Repeat (for each step of episode):
    Choose a from s using policy derived from Q (e.g., ε-greedy)
    Take action a, observe r, s'
    $Q(s,a) \leftarrow Q(s,a) + \alpha\big[r + \gamma \max_{a'} Q(s',a') - Q(s,a)\big]$
    $s \leftarrow s'$;
  until s is terminal

(b) Q-learning: Off-policy TD control.

Figure 2.3: Examples of temporal difference learning algorithms [18].

The takeaway from figure 2.3 and the key difference between both learning methods is, that SARSA learns action values relative to the policy it follows, while Q-Learning does it relative to the greedy policy. Under some common conditions, both converge to the optimal value function, but at different rates. Q-Learning tends to converge slower, but has the capability to continue learning while changing policies.

2.4 Previous Work On Reinforcement Learning

Previous work proposed and evaluated the usage of several reinforcement learning algorithms to solve the algorithm selection problem in the context of scheduling. In the following we give a short summary for the most important work in order of their publication date.

[6], Dhandayuthapani, S., 2005 This work arose from the need for an automatic selection of scheduling algorithm for scientific applications in a environment using machine learning. The design and implementation of an integrated (into the scientific application) reinforcement learning agent, which improves the performance of large applications with parallel loops. Embedding the integrated technique into the scientific applications improves the performance by cost minimization and efficient utilization of computational resources. The obtained results were gathered from a wave-packet simulation algorithm using quantum trajectory method (large number of time steps).

[16], Rashid, M., et al., 2008 This paper investigates the performance of a dynamic loop scheduling with reinforcement learning (DLS-with-RL) approach to load balancing. The RL agent was designed for the class of large-scale time-stepping applications that have one or more computationally intensive parallel loops with nonuniform iteration execution times. Each parallel loop is assigned an RL agent which dynamically chooses from a library a loop scheduling algorithm to minimize the loop completion time. Analysis of the results showed that the application performance is insensitive to the RL technique used and to a particular combination of the learning parameters.

[17], Sukhija, N, et al., 2014 A portfolio-based approach for selecting the most robust DLS algorithm was proposed in this paper. A wide range of supervised machine learning techniques were investigated to obtain an empirical hardness model which predicts the robustness of DLS algorithms with better accuracy than previous models. The prediction model enables selection of the most robust DLS algorithm for a given simulated problem instance. A shortcoming of our approach, common to supervised machine learning, is that the learned models do not generalize well to completely new populations but still remained competitive with other strategies because the predictions still were able to often separate robust algorithms from non-robust ones.

[4], Boulmier, A., et al., 2017 This paper presents an autonomic computing approach for performance optimization of the time-stepping scientific application (TSSA) via an autonomic selection of robust DLS techniques. This approach employs a RL agent for each parallel loop of a TSSA. The present work proposes a modified version of the flexibility metric to reward the choices of the RL agents. This paper uses a generic simulation framework to study the effectiveness of the modified flexibility metric as a RL reward. The results show that the RL agent was unable to select the optimal DLS technique during the execution of a TSSA facing extreme perturbations. The robust-DLS-with-RL performed either as good as, or worse than, the fixed-DLS approach.

3
ReinforcedSel

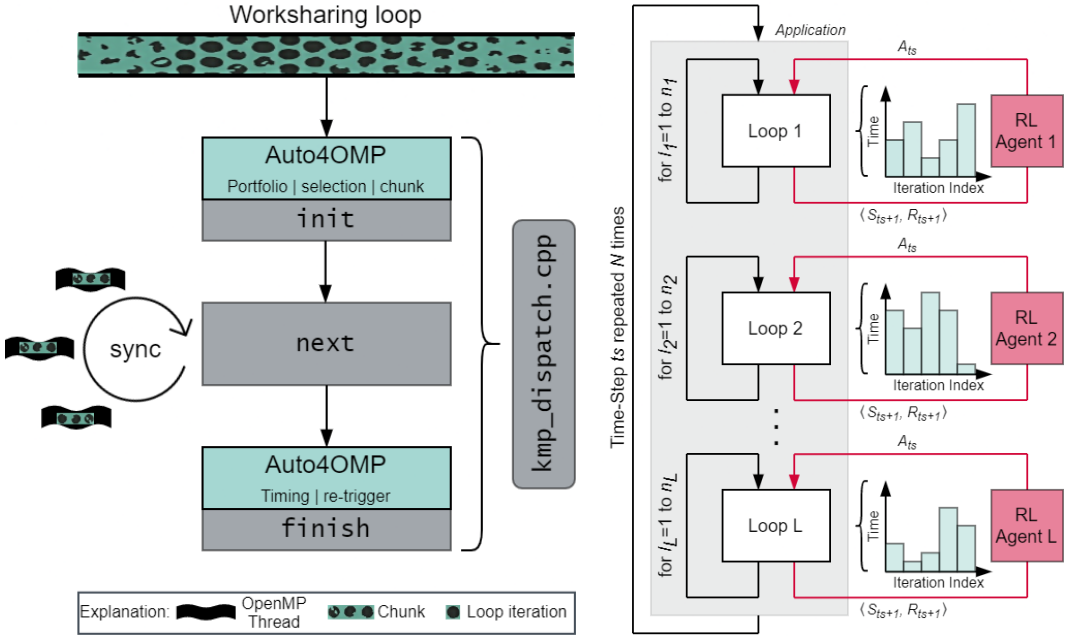
This chapter presents the design and implementation details of a reinforcement learning agent for automatic selection of dynamic scheduling algorithms based on the LB4OMP library reported in [13] and as an extension to the already existing portfolio of automatic selection methods from Auto4OMP [14]. This chapter is divided into two sections. The first section describes the design of the reinforcement learning system with regards to the theory provided in the previous chapter. The second section provides a short technical explanation of the implementation.

3.1 Design And Implementation

A time-stepping scientific application (for example a modified Mandelbrot program) which requires dynamic load balancing, provides an excellent environment for an reinforcement learning agent's operation. This section describes the integration of our reinforcement learning algorithm selection method into a time-stepping scientific application for performance improvements on the basis of the LB4OMP library with the extension from the Auto4OMP paper. Such performance gains can be achieved by overcoming the scheduling algorithm selection problem in OpenMP. We take advantage of the *auto* keyword, present in the LLVM OpenMP runtime library as a scheduling option, to extend the existing implementation with our new reinforcement learning selection method. Figure 3.1b illustrates a high-level view of a general time-stepping application with L number of computationally intensive sections containing parallel loops.

Such an application evolves over N time steps. Within a single time-step, L parallel loop sections are executed. The full design is obtained by integrating the reinforcement learning logic for each loop into the overall time-step loop. The goal of the reinforcement learning agent is to minimize the total time spent by the application in each loop. Due to the lack of any prior knowledge, the agent selects each algorithm in the portfolio set sequentially during the initial $N_{explore}$ time-steps of the loop - this is called the exploration phase. The exploration phase will take exactly **card**(Portfolio)^2 time-steps, since the agent will explore every state-action transition that is possible. For our use case, we map the environment's states to the different scheduling algorithms that can be employed and the actions to switching to

any other algorithm that is not currently in use. After knowledge about the environment is first obtained during this initial learning period (exploration), the agent applies the learned policy for the remaining $N_{exploit}$ time-steps ($N_{exploit} = N - N_{explore}$). The loop completion time for the selected algorithm (S_{ts}) is the basis for the reward R_{ts+1} given to the agent for taking action A_{ts}.



(a) Loop scheduling workflow of the standard LLVM OpenMP RTL augmented with Auto4OMP (adaption from [13, 14]).

(b) Anatomy of a time-stepping application with reinforcement learning agents (adaptation from [16]).

Figure 3.1: Architecture

The LLVM OpenMP runtime and LB4OMP libraries collect information about the application's loops and make them available as data structures in `kmp_dispatch.cpp` (refer to figure 3.1a). We take advantage of this potential by deriving the needed knowledge about changes in the environment and in turn reward the agent to continuously improve the learned policy and select the best actions. As depicted in figure 3.1b the reinforcement learning agent and the associated data structure (which stores the agent's data such as e.g., the Q-values) is created and updated separately for every parallel loop present in the application. Thusly state, action and reward information for each parallel loop is handled by the appropriate agent. This favors a more fine-grained control for selecting a scheduling algorithm and consequently also allows for larger performance gains (due to large differences in loop characteristics as we can observe from figure 4.2) instead of just using one agent for the entire application.

3.2 Usage

This section will provide a short technical overview on how the reinforcement learning selection method was implemented in the OpenMP+LB4OMP library and how to use it 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 figure 3.2. 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 (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 MHz. Refer to the example jobfile in the appendix (section A.4) for more details.

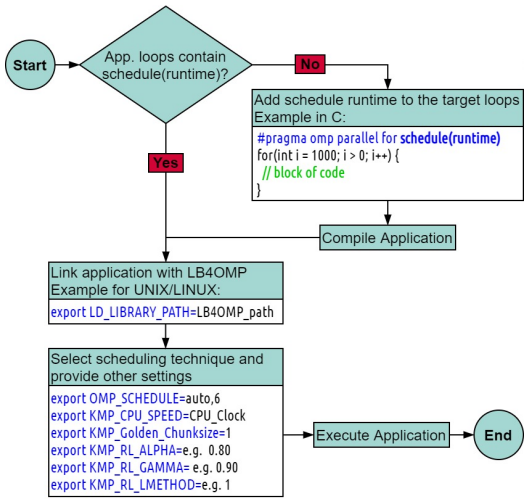


Figure 3.2: Workflow using automatic selection of scheduling algorithms in LB4OMP.

Based on the previous work of the Auto4OMP paper, the same method to enable our automatic selection method for scheduling algorithms is used. The environment variable `OMP_Schedule` needs to be exported with a value of `auto,<int>` where `<int>` can take a value from range 1 to 6. The number 1 refers to the standard implementation by OpenMP, values 2 to 5 denote the automatic selection methods implemented by Auto4OMP and the value 6 identifies our new automatic selection method via reinforcement learning. Further there are 3 additional environment variables that can be exported to control the behaviour of the reinforcement learning agent during runtime (all three variables have defaults - setting them is not required to use this selection method).

- `KMP_RL_ALPHA=<value>` (Learning Rate, Range 0.1 .. 0.9): The learning rate sets the ratio of importance between old and new information. A factor of 0 makes the agent learn nothing, while a factor of 1 makes the agent consider only the most recent information.
- `KMP_RL_GAMMA=<value>` (Discount Factor, Range 0.1 .. 0.9): The discount factor determines the importance of future rewards. A factor of 0 will make the agent short-sighted, while a factor close to 1 will make it look for a long-term high reward.
- `KMP_RL_LMETHOD=<value>` (Learning Method, Selection 0/1): Values-based model-free temporal difference learning support different learning methods. In this work two methods have been implemented: SARSA (0) and Q-Learning (1).

As described in the Auto4OMP paper, the portfolio of scheduling algorithms from which the automatic selection methods can choose from, has been compiled according to three inclusion characteristics and ordered in ascending order of their scheduling overhead and load balancing capacity. Our reinforcement learning selection method will also select algorithms from the same portfolio to make the comparison of the results as meaningful as possible. Therefore, the ordered set from which to choose the loop scheduling algorithms from is composed of: `STATIC`, `SS`, `GSS`, `GAC`[1], `TSS`, `Static Steal`, `mFAC2`, `AWF-B`, `AWF-C`, `AWF-D`, `AWF-E`, `mAF`.

[1] GAC refers to LLVM's implementation of schedule(auto) for the OpenMP runtime library. The acronym is short for guided_analytical_chunked (a special variation of GSS). We will refer to this scheduling algorithm with *Auto(LLVM)*.

4

Performance Evaluation

This chapter presents the experiments and analysis of the results to verify and validate the hypotheses stated below. The main objective of this project, to provide an automatic selection mechanism for finding the optimal scheduling algorithm in a dynamic environment without prior execution using reinforcement learning, was realized by implementing the design from the previous chapter.

The performance experiments are conducted using a modified version of the parallel Mandelbrot [12] application, to be compatible with our time-stepping approach. The application was compiled using the Intel compiler version 2019/a. If not stated otherwise, we use the following input for Mandelbrot (`maxiter, [x0, y0, size]`): ⟨ `1000, 512, 0, 0, 0.5` ⟩.
All of the experiments were conducted on the same operating system (CentOS 7) on the *miniHPC-Broadwell* cluster at the Department of Mathematics and Computer Science of the University of Basel. The exact specifications of the system are listed in table 4.1. The table also shows the set of performance evaluation experiments we designed to test the following hypotheses that motivated this project:

H.1 The new automatic algorithm selection method **ReinforcedSel**, using reinforcement learning, improves overall parallel execution time.

H.2 **ReinforcedSel** adapts to applications' need of different scheduling algorithms across time-steps for each loop to achieve high performance.

H.3 **ReinforcedSel** adapts to applications' need of different scheduling algorithms for different loops in a single time-step (and across time-steps).

Each experiment was repeated 5 times and the time values in the results denote the average of those 5 runs. We collect the execution time of every modified loop (latest finishing thread), every application thread, and the total duration of the execution of the applications. The number of time-steps within the Mandelbrot application is denoted as T, the number of loops we annotated with the *schedule(runtime)* as L. To keep the amount of experiment to a manageable size, we have split the execution into two different categories:

Table 4.1: Design of factorial experiments for the performance evaluation of the automatic schedule selection method using reinforcement learning

Factors		Values	Properties
Applications		Mandelbrot	$N = 262'144$, $T = 200$, Total loops=3, Modified loops=3
Computing nodes		Type A (miniHPC-Broadwell)	Intel Broadwell E5-2640 v4 (2 sockets, 10 cores each), $P = 20$ without hyperthreading, $P = 40$ with hyperthreading Pinning: OMP_PLACES=cores OMP_PROC_BIND=close
Scheduling techniques	OpenMP standard	STATIC	Straightforward parallelization
		GUIDED (GSS), DYNAMIC (SS)	Dynamic and non-adaptive self-scheduling techniques
	OpenMP non-standard (LB4OMP)	mFAC2	
		mAF	Dynamic and adaptive self-scheduling techniques
Selection methods	Expert based (Auto4OMP)	RandomSel, ExhaustiveSel, BinarySel, ExpertSel	Automated DLS algorithm selection across application loops and time-steps
	ML based	ReinforcedSel	Automated DLS algorithm selection across application loops and time-steps
Chunk parameter	State of the practice	Default	Chunk size = N/P for static and 1 for all other scheduling algorithms
	Auto4OMP	Expert chunk	A point at 0.618 on the curve between $N/(xP)$ and 1, with x increasing in steps of 2
Computing nodes		Type A (miniHPC-Broadwell)	Intel Broadwell E5-2640 v4 (2 sockets, 10 cores each), $P = 20$ without hyperthreading, $P = 40$ with hyperthreading Pinning: OMP_PLACES=cores OMP_PROC_BIND=close

Metrics	Performance per loop	Parallel loop execution time T loop
	Schedule selection	Counter (in case of auto methods)

1. Comparing the performance of our new `ReinforcedSel` method to itself with different configurations.
2. Compare the `ReinforcedSel` method to fixed scheduling algorithms and other automatic selection methods.

The pseudocode in listing 1 and 2 explain the design and partitioning of the factorial experiments according to the two aforementioned categories. The results are presented in section 4.1 for listing 1 and sections 4.2, 4.3 for listing 2.

Algorithm 1: Factorial experiment showing α, γ sensitivity of Mandelbrot

input : `DLS` ← ReinforcedSel, `GOLDEN` ← TRUE

for `RL` **in** { `Q-Learning, SARSA` } **do**
  `/* Denotes the crossprodcut */`
  for α, γ **in** { `0.1, 0.2, .., 0.9` } **do**
    repeat the following 5 times:
    • execute Mandelbrot with ⟨ `DLS`, `GOLDEN`, `RL`, α, γ ⟩;
    • record frequency of scheduling algorithm per loop;
    • record T_p;
  end
end

Algorithm 2: Factorial experiment comparing parallel execution times of Mandelbrot

input : RL, α, γ

```
DLS_METHODS = { STATIC, GUIDED, DYNAMIC, ... }
AUTO_METHODS = { RandomSel, ExhaustiveSel, BinarySel, ... }
PORTFOLIO = DLS_METHODS ∪ AUTO_METHODS
```

for DLS **in** PORTFOLIO **do**
  for GOLDEN **in** { FALSE, TRUE } **do**
    repeat the following 5 times:
    • execute Mandelbrot with ⟨ DLS, GOLDEN, RL, α, γ ⟩;
    • record frequency of scheduling algorithm per loop;
    • record T_p;
  end
end

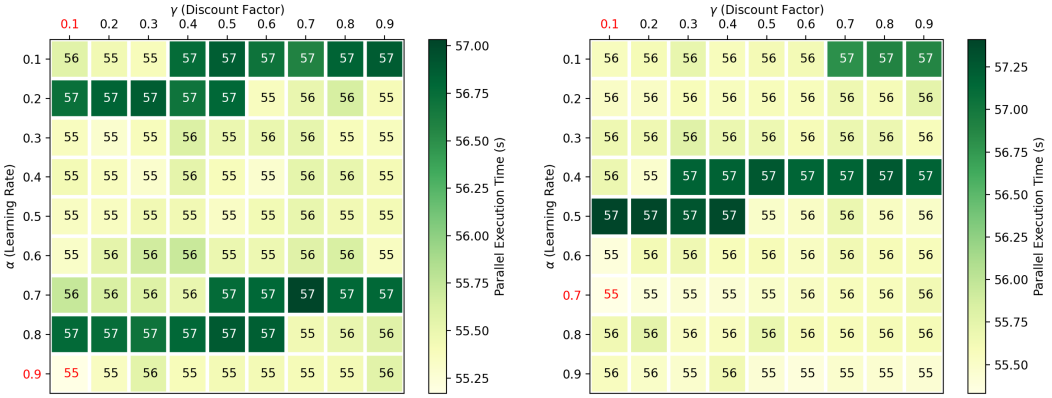
4.1 Comparison Of Reinforcement Learning Parameters

Figure 4.1 illustrates all the results generated from the 162 experiments specified by listing 1. The figure consists of 6 sub-figures depicting the different combinations of learning method and *maxiter* parameter. For one sub-figure we see the different α and γ values on the vertical and horizontal axis respectively. To the right the heatbar indicates the execution time of the application (lighter colors are better). At the intersection of two corresponding α and γ values, the related execution time is denoted. The heatmap itself illustrates the distribution of the execution times in that particular plot. The best combination of learning rate α and discount factor γ for each sub-figure are highlighted in red.

Table 4.2 shows a summary of the best performing α,γ-value combination given one of the two learning methods, Q-Learning and SARSA. The experiment is also conducted for 3 different input sizes for the *maxiter* parameter of the Mandelbrot application.

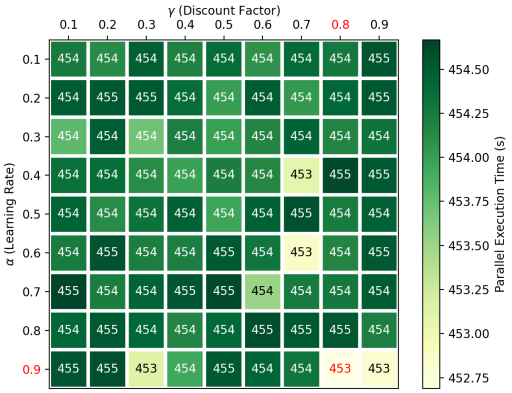
Firstly we observe that the execution time increases in a linear fashion proportional to the *maxiter* parameter. Secondly we see that the learning method has no significant impact on the overall execution time of the application. The difference in execution time is less than 1% across all the experiments. Furthermore we discover that regardless of the *maxiter* parameter a higher value for the learning rate α is preferred, while for the discount factor γ a lower value for less iterations, increasing shortsightedness of the agent, is favored. For longer running experiments a higher discount factor seems to perform better. This result can be explained through the fact that for longer running experiments the true characteristic of the associated loop has a larger effect on the execution time that decides the reward and in turn also the next action. This should in theory lead to better choices regarding to the actions taken by the agent.

In practice, however the overall application performance is not sensitive to any of the three configuration parameters for the agent, α, γ and learning method. In the following experiments the parameters for the agent are nevertheless chosen according to table 4.2. Since the next set of experiments is run with *maxiter*=100'000, the following values for the learning method, α and γ are chosen: ⟨ SARSA, 0.8, 0.9 ⟩. For completeness' sake the rest of the experiments are also conducted with the parameters ⟨ Q-Learning, 0.7, 0.4 ⟩, however the results will only be mentioned in the appendix.

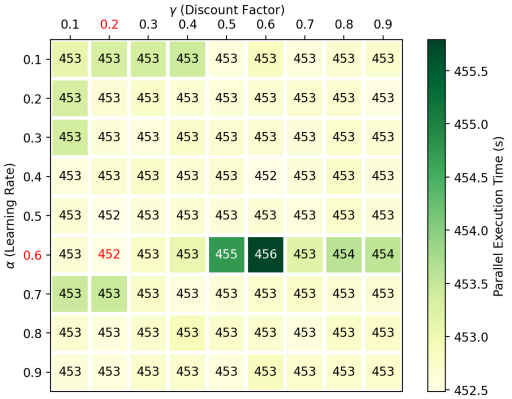


(a) Method: `SARSA`, maxiter: `1'000`

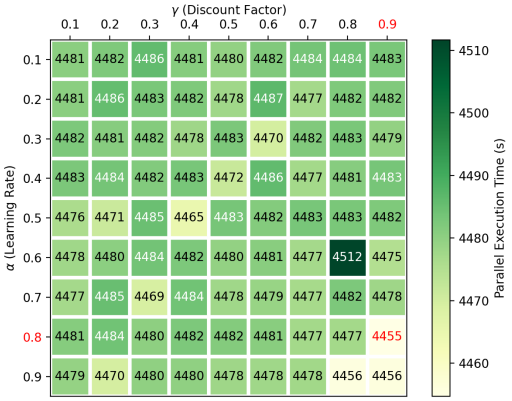
(b) Method: `Q-Learning`, maxiter: `1'000`



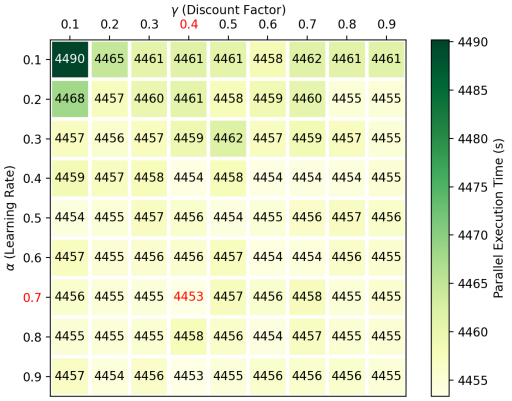
(c) Method: `SARSA`, maxiter: `10'000`



(d) Method: `Q-Learning`, maxiter: `10'000`



(e) Method: `SARSA`, maxiter: `100'000`



(f) Method: `Q-Learning`, maxiter: `100'000`

Figure 4.1: Comparing the performance for the Mandelbrot application with different *max-iter* parameters and α,γ-values (lighter colors are better).

Table 4.2: Best performing α,γ-values with respect to the overall parallel execution time.

Iterations	SARSA				Q-Learning				Δ
	α	γ	Time (s)		α	γ	Time (s)		(Min. Time of
			Min	Avg			Min	Avg	Learning Met)
1'000	0.9	0.1	55.33	55.85	0.7	0.1	55.17	55.83	+ 0.29%
10'000	0.9	0.8	452.48	452.82	0.6	0.2	452.69	454.24	− 0.05%
100'000	0.8	0.9	4'453.34	4457.21	0.7	0.4	4'454.71	4479.65	− 0.03%

Comparing the distributions of the execution times for every row in the 4.1, we realize that for a larger *maxiter* parameter the hotpots for good or bad α,γ-value combinations disappear. This becomes even more apparent when looking at the density plots in figure A.1 in the appendix and emphasizes the insensitivity of the execution time toward the agent's configuration parameters.

4.2 Comparison Of Automatic Selection Methods

The evaluation in this section is comprised of two different plots. The first plot shows the loop execution times and scheduling algorithm selection per time-step and application loop (rows) for 5 different automatic scheduling technique selection methods (columns). The x and y axis on the subplots are labeled with the time-step and the loop execution time. The colors represent the 12 different loop scheduling algorithms implemented in the portfolio. The second plot shows how often a scheduling algorithm for a specific loop (rows) has been chosen from the portfolio by the given selection method (columns). The difference between figure 4.2 and figure A.4 lies in the `KMP_GOLDEN_CHUNK` parameter, which has been left enabled (as per default) for the first set of experiments and deliberately disabled for the second set of experiments. Generally it can be stated that there is no difference in the relative performance between the automatic selection methods weather the `KMP_GOLDEN_CHUNK` parameter was set to **true** or **false**. But as shown in [13], the absolute value of the parallel execution time decreases significantly with expert chunks enabled for any of the loop scheduling algorithms.

For the first plot we observe at first glance that loop $L0$ of the Mandelbrot application does not experience any speedup, regardless of which scheduling technique is used. Further we establish the the plots in the column **ReinforcedSel** clearly show the difference between the exploration and exploitation phase of the agent. It is also observable that in the current implementation of this method, the exploration phase lasts for almost 75% of the total 200 time-steps present in the Mandelbrot application. It is also interesting to note that the agent quickly settles on one particular scheduling algorithm once the exploitation phase has begun. This indeed leads to a reduction in the execution time for for loops $L1$ and $L2$. However as we will reveal In the following section, the selected scheduling algorithms by **ReinforcedSel** did not yield the best overall result.

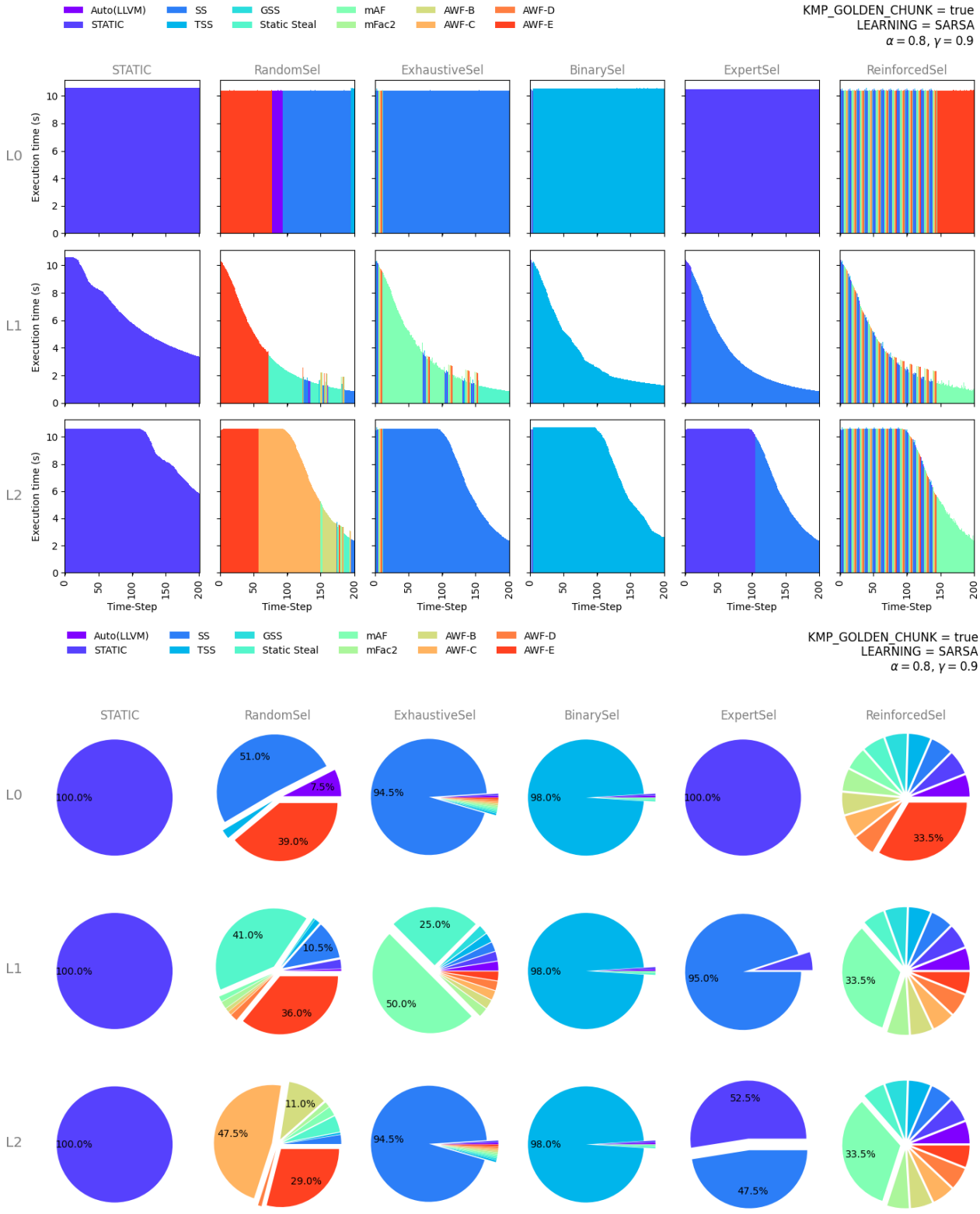


Figure 4.2: Comparing loop execution times and frequency of DLS algorithm selection for AUTO methods (`KMP_GOLDEN_CHUNK=true`, `LEARNING_METHOD=SARSA`).

4.3 Comparison Of Automatic Selection Methods And Fixed Scheduling Algorithms

Figure A.5 shows the results from the factorial experiments constructed by listing 2 in detail. The different bars represent the scheduling techniques selected for the performance evaluation. On the y axis we find the corresponding parallel execution time. The bars also show how much every loop in the parallel application contributed to the parallel execution time. The three horizontal lines represent the highest (red), baseline (dashed) and lowest (black) measured parallel execution time. Here the highest and baseline lines coincide.

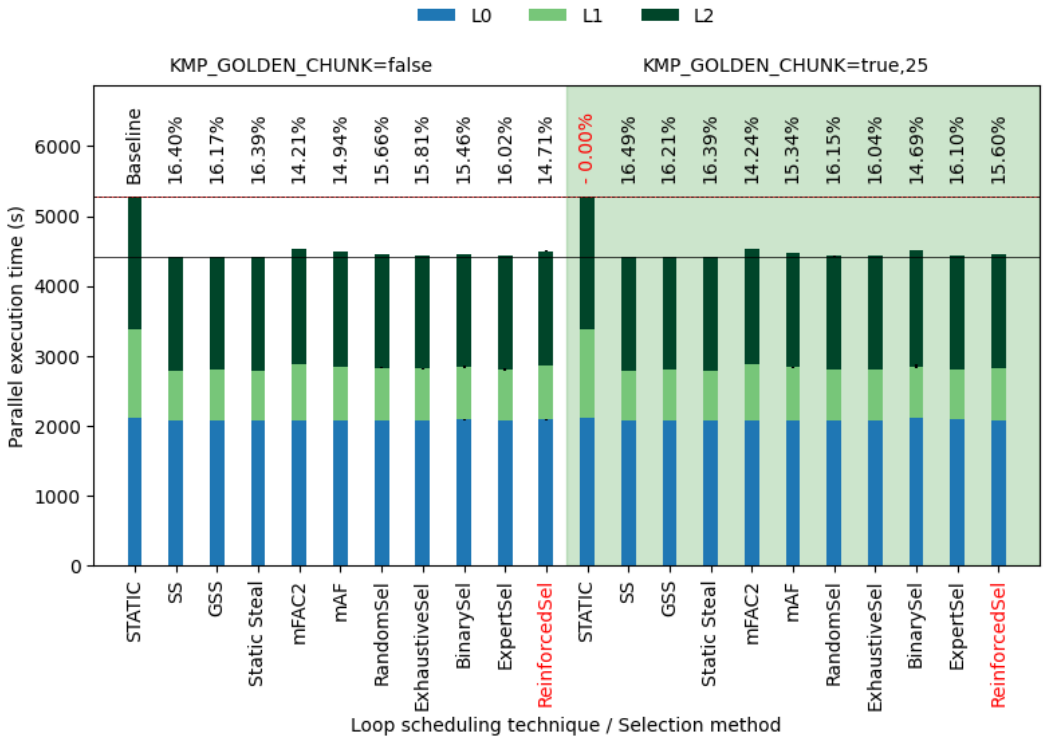


Figure 4.3: Parallel execution time for each loop in Mandelbrot executing on node Type A. The percentages denote performance improvements for the selected scheduling method vs. the baseline (`STATIC` scheduling method) with learning method SARSA.

Table 4.3 presents an overview of the performance achieved by the available scheduling algorithms in LB4OMP and by the automatic selection methods in Auto4OMP and the proposed reinforcement learning selection method. As a comparison baseline we use the `STATIC` scheduling method provided by the OpenMP library. The percentage of performance degradation for the scheduling methods are computed by using the parallel execution time of the `STATIC` method ($T_{par,static}$) as reference value. The remaining values are calculated according to the following formula: $P_{schedule} = \mathbf{abs}(\frac{T_{par,static}}{T_{par,schedule}} * 100 - 100)$.

Table 4.3: Comparison between state of the practice scheduling algorithms vs. automatic selection methods in terms of performance (execution time) degradation relative to the baseline (STATIC schedule) for the learning method SARSA.

Selection / App-Sys Pair	Fixed Scheduling Algorithms						Automatic Selection Methods				
	STATIC	SS	GSS	Static Steal	mFAC2	mAF	RandomSel	ExhaustiveSel	BinarySel	ExpertSel	ReinforcedSel
Mandelbrot-Type A (Expert Chunk **false**)	±0.00%	16.40%	16.17%	16.39%	14.21%	14.94%	15.66%	15.81%	15.46%	16.02%	14.71%
Mandelbrot-Type A (Expert Chunk **true**)	±0.00%	16.49%	16.21%	16.39%	14.24%	15.34%	16.15%	16.04%	14.96%	16.10%	15.60%

Table cells with red and green backgrounds, highlight the lowest and the highest performance gain relative to the baseline for every selection method in the set of state of practice scheduling algorithms and in the set of automatic selection methods separately. From the these results we observe that the proposed automatic selection method **ReinforcedSel** is on par with the other automatic selection methods from Auto4OMP, but can never outperform them.

When we take figure 4.2, A.5 and table 4.3 into consideration and remember the 3 hypotheses stated at the beginning of this chapter, we can confirm hypotheses **H.1**, **H.2** and **H.3**. Even though we never outperform all of the selection methods from the Auto4OMP paper, our method can compete in terms of performance optimization. In chapter 5 we suggest several improvements and further experiments that might give deeper insight into the benefits of using reinforcement learning as a automatic selection method.

5
Conclusion and Future Work

This chapter summarizes the issues that were addressed and the contributions made by this project. It is followed by an outline of the conclusions and the possible directions for future research work.

This project introduces an automatic method (named **ReinforcedSel**) for selecting loop scheduling algorithms from a portfolio using reinforcement learning. The agent has been implemented alongside the work of Auto4OMP as an effort to overcome the algorithm selection problem for massively parallel OpenMP scientific applications and to improve the overall performance via load balancing. We evaluated the performance for **ReinforcedSel** for a parallel Mandelbrot time-stepping application on a single hardware system and tested three hypothesis. We compared the performance achieved by **ReinforcedSel** with the set of automatic selection methods from Auto4OMP and a reduced set of fixed loop scheduling algorithm from the LB4OMP portfolio. The recorded parallel execution time for each loop was used to calculate the performance gain compared to the *static* scheduling technique, provided by the LLVM OpenMP runtime library, as a baseline.

The evaluation of the results show, the proposed automatic algorithm selection method quickly settles on a scheduling algorithm after its exploration phase and as a result yields increased application performance by up to **15.60%** better than the selected baseline scheduling algorithm. This puts our method on par with the ones from LB4OMP. The best performing method from their portfolio, RandomSel, achieved a performance gain of **16.15%**, while the worst achieved **14.96%**.
The advantage of an automatic schedule selection method based on reinforcement learning (or machine learning in general) is that it will always learn to act in its environment to the best of its abilities. To results from this paper show that this method is only on par with the methods based on expert knowledge presented in LB4OMP. However improving on the expert knowledge in the future might be difficult as to where a machine learning agent can be tuned and adapted in an easier fashion. In case of the reinforcement learning method, more research has to be made on the metric used to derive the reward and on the other techniques employed by the agent.

5.1 Future Work

In this section we are proposing changes and extensions to implementation provided with this paper based on the shortcomings of our method we uncovered during the performance evaluation and insights provided by related work.

Verification Most importantly we need a method to reliably and reproducibly test the behaviour of the reinforcement learning agent. Without this addition we cannot be certain that our implementation fulfills the correctness criterion. Therefore an interface for testing the reinforcement learning logic outside time-stepping applications should be created. There has been recent research [3, 19] which is looking into the verification of trained machine learning models via policy extraction.

Applications & Hardware Further we should extend the range of time-stepping application and system types we use to obtain the performance evaluation data. This project only relied on a parallel Mandelbrot time-stepping program and a single hardware type to measure parallel execution time. This can be problematic since this particular implementation of Mandelbrot is known to produce a constant amount of load imbalance which is well controlled by any dynamic loop scheduling algorithm. Instead we should employ other time-stepping applications (e.g., SPHYNX [5]) for performance analysis as well. Another drawback of Mandelbrot for this performance evaluation is the small number of time-steps present in the application. Compared to the amount of time-steps required for the reinforcement learning agent to complete its exploration phase, the ensuing exploitation phase is rather short.

Metrics Additionally other performance metrics to determine the reward for the reinforcement learning agent should be considered as well. Here we compared the execution time of the loops for two consecutive time-steps to reward or punish the agent's actions. In future research one could consider load imbalance, robustness [17], flexibility [4] or a rolling average of loop execution times as an input for the reward function. In general any metric used as a decision criteria for loop scheduling algorithm selection needs to be able to encode the environments state well in terms of load imbalance and absorbed perturbations.

Learning Methods As discussed in section 2.3, we can leverage an abundance of learning methods that we can interchangeably use in our reinforcement learning setup. So far this project uses the well known Q-Learning and SARSA algorithms. As an extension we should also look at more advanced learning methods such as Expected-SARSA, Double-Q-Learning, QV-Learning and even Deep-Q-Learning which eliminate some of the shortcomings of Q-Learning and SARSA (overhead, may become trapped in local minima).

Agent Policy For the on- and off-policy temporal difference learning methods, the agent usually follows a certain policy that dictates (in accordance with the learning method) which action should be chosen next. In this work we implemented a greedy policy that simply chooses the action which promises the most reward by exploiting the agent's current action-value estimates. Epsilon-Greedy is a simple alternative policy to balance exploration and exploitation by choosing between exploration and exploitation randomly. In epsilon-greedy, the epsilon refers to the probability of choosing to explore, otherwise the current best action is exploited. This could serve as a valuable improvement over the current implementation where exploration only happens at the beginning.

Bibliography

[1] openmp - GCC Wiki. URL https://gcc.gnu.org/wiki/openmp.

[2] Mandelbrot-Menge, August 2021. URL https://de.wikipedia.org/w/index.php?title=Mandelbrot-Menge&oldid=214955674. Page Version ID: 214955674.

[3] Osbert Bastani, Yewen Pu, and Armando Solar-Lezama. *Verifiable Reinforcement Learning via Policy Extraction*. May 2018.

[4] 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, 2017. doi: 10.1109/ISPDC.2017.9.

[5] Ruben Cabezon, Domingo García-Senz, and J. Figueira. Sphynx: An accurate density-based sph method for astrophysical applications. *Astronomy & Astrophysics*, 606, 07 2017. doi: 10.1051/0004-6361/201630208.

[6] Sumithra Dhandayuthapani, Ioana Banicescu, R.L. Carino, Eric Hansen, J.R. Pabico, and Mahbubur Rashid. Automatic selection of loop scheduling algorithms using reinforcement learning. volume 2005, pages 87 – 94, 08 2005. ISBN 0-7803-9043-1. doi: 10.1109/CLADE.2005.1520907.

[7] Randy L. Haupt and Amy J. Shockley. Ethically speaking: Analysis paralysis. *URSI Radio Science Bulletin*, 2018(366):23–24, 2018. doi: 10.23919/URSIRSB.2018.8627428.

[8] Susan Flynn Hummel, Ioana Banicescu, Chui-Tzu Wang, and Joel Wein. Load Balancing and Data Locality Via Fractiling: An Experimental Study. In Boleslaw K. Szymanski and Balaram Sinharoy, editors, *Languages, Compilers and Run-Time Systems for Scalable Computers*, pages 85–98. Springer US, Boston, MA, 1996. ISBN 978-1-4615-2315-4. doi: 10.1007/978-1-4615-2315-4_7. URL https://doi.org/10.1007/978-1-4615-2315-4_7.

[9] Hesam Izakian, Ajith Abraham, and Václav Snasel. Comparison of Heuristics for Scheduling Independent Tasks on Heterogeneous Distributed Environments. In *2009 International Joint Conference on Computational Sciences and Optimization*, volume 1, pages 8–12, April 2009. doi: 10.1109/CSO.2009.487.

[10] David S. Johnson. The NP-completeness column: an ongoing guide. *Journal of Algorithms*, 6(3):434–451, 1985. ISSN 0196-6774. doi: https://doi.org/10.

1016/0196-6774(85)90012-4. URL https://www.sciencedirect.com/science/article/pii/0196677485900124.

[11] A. A. Khan, C. L. McCreary, and M. S. Jones. A Comparison of Multiprocessor Scheduling Heuristics. In *In Proceedings of the 1994 International Conference on Parallel Processing, volume II*, pages 243–250, 1994.

[12] Benoit B. Mandelbrot. Fractal aspects of the iteration of z $\rightarrow$ λz(1-z) for complex λ and z. *Annals of the New York Academy of Sciences*, 357(1):249–259, 1980. doi: https://doi.org/10.1111/j.1749-6632.1980.tb29690.x. URL https://nyaspubs.onlinelibrary.wiley.com/doi/abs/10.1111/j.1749-6632.1980.tb29690.x.

[13] Jonas H. Müller Korndörfer, Ahmed Eleliemy, Ali Mohammed, and Florina M. Monica Ciorba. Lb4omp: A dynamic load balancing library for multithreaded applications. *IEEE Transactions on Parallel and Distributed Systems*, pages 1–1, 2021. doi: 10.1109/TPDS.2021.3107775.

[14] Authors omitted for blind review. Automated Load Balancing in OpenMP. *Under review.*

[15] Naren Ramakrishnan, John Rice, and Elias Houstis. Knowledge discovery in computational science: A case study in algorithm selection. 04 1997.

[16] Mahbubur Rashid, Ioana Banicescu, and Ricolindo L. Carino. Investigating a dynamic loop scheduling with reinforcement learning approach to load balancing in scientific applications. In *2008 International Symposium on Parallel and Distributed Computing*, pages 123–130, 2008. doi: 10.1109/ISPDC.2008.25.

[17] Nitin Sukhija, Brandon Malone, Srishti Srivastava, Ioana Banicescu, and Florina M. Ciorba. Portfolio-based selection of robust dynamic loop scheduling algorithms using machine learning. In *2014 IEEE International Parallel Distributed Processing Symposium Workshops*, pages 1638–1647, 2014. doi: 10.1109/IPDPSW.2014.183.

[18] R.S. Sutton and A.G. Barto. Reinforcement learning: An introduction. *IEEE Transactions on Neural Networks*, 9(5):1054–1054, 1998. doi: 10.1109/TNN.1998.712192.

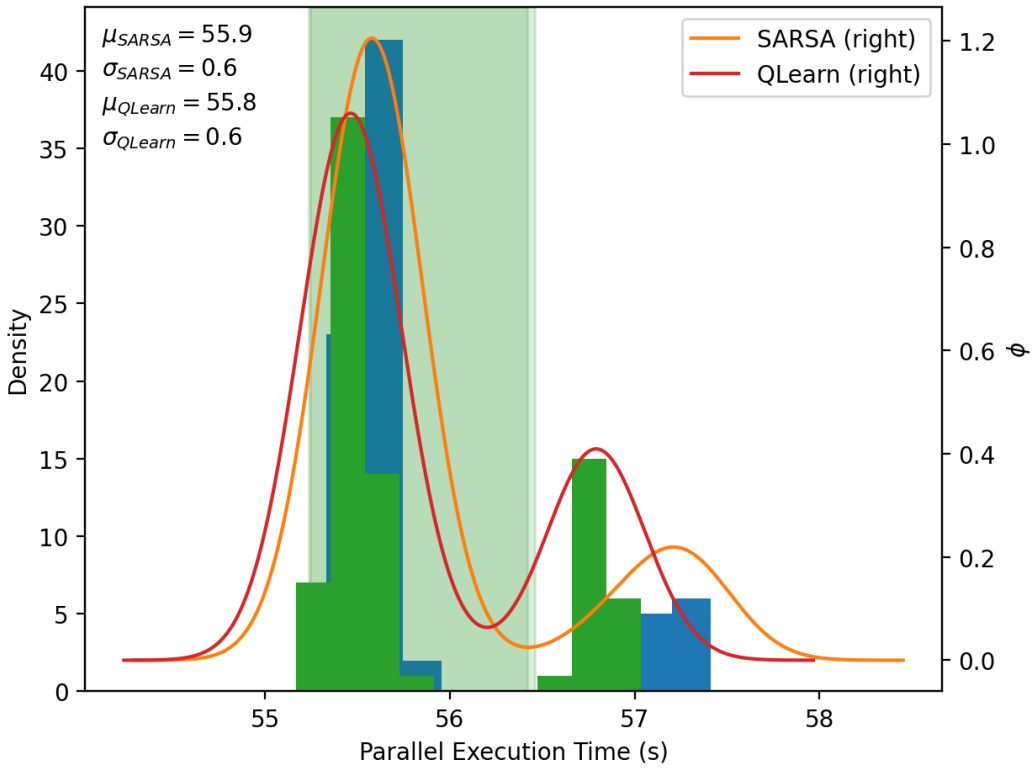
[19] Perry van Wesel and A. Goodloe. Challenges in the Verification of Reinforcement Learning Algorithms. *undefined*, 2017. URL https://www.semanticscholar.org/paper/Challenges-in-the-Verification-of-Reinforcement-Wesel-Goodloe/0cd691675e100008085317156efce978aa9446ea.

[20] Hongming Zhang and Tianyang Yu. Taxonomy of Reinforcement Learning Algorithms. In Hao Dong, Zihan Ding, and Shanghang Zhang, editors, *Deep Reinforcement Learning: Fundamentals, Research and Applications*, pages 125–133. Springer, Singapore, 2020. ISBN 9789811540950. doi: 10.1007/978-981-15-4095-0_3. URL https://doi.org/10.1007/978-981-15-4095-0_3.

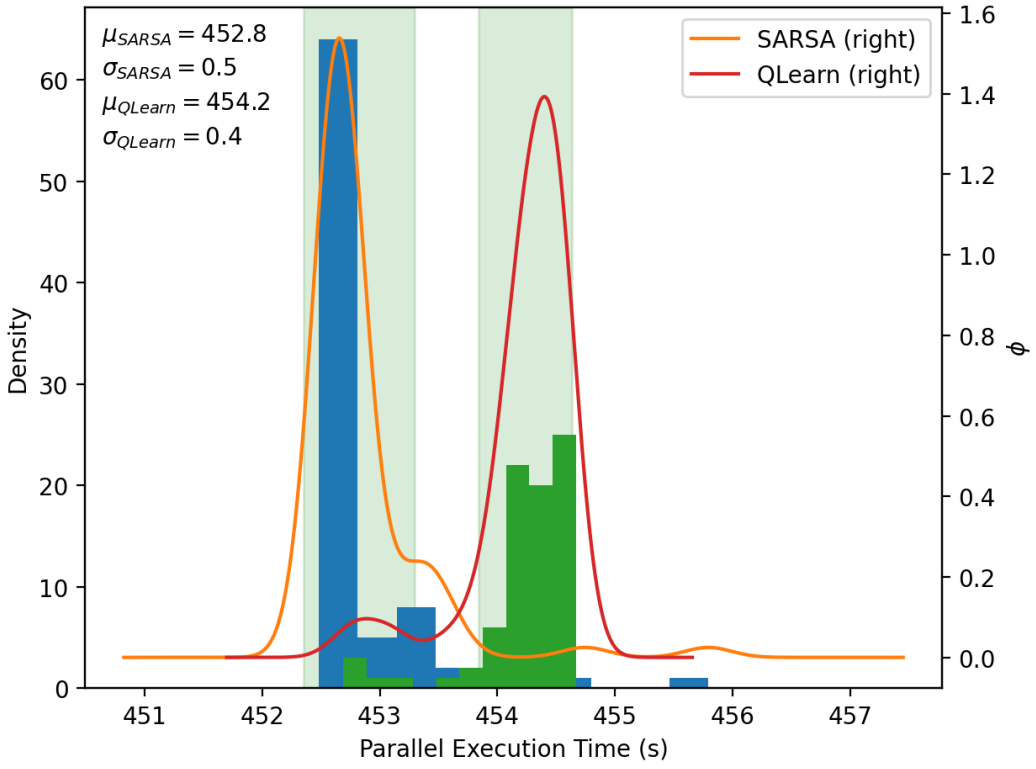
A
Appendix

For completeness, this appendix lists figures for the performance evaluation which where no featured in the main part of this report due to the similarity in the results between the Q-Learning and SARSA learning methods.

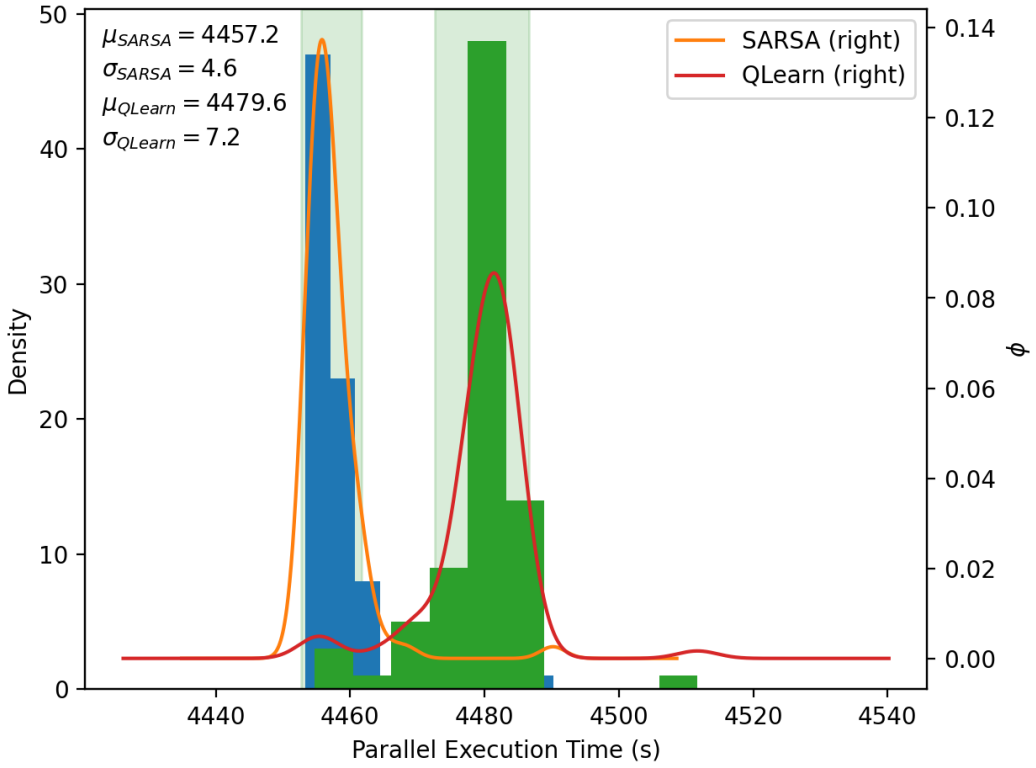
A.1 Distribution Of Parallel Execution Times



(a) Iterations: 1'000



(b) Iterations: 10'000



(c) Iterations: 100'000

Figure A.1: Distribution of parallel execution times when varying α, γ-values.

A.2 Comparison Of Automatic Selection Methods

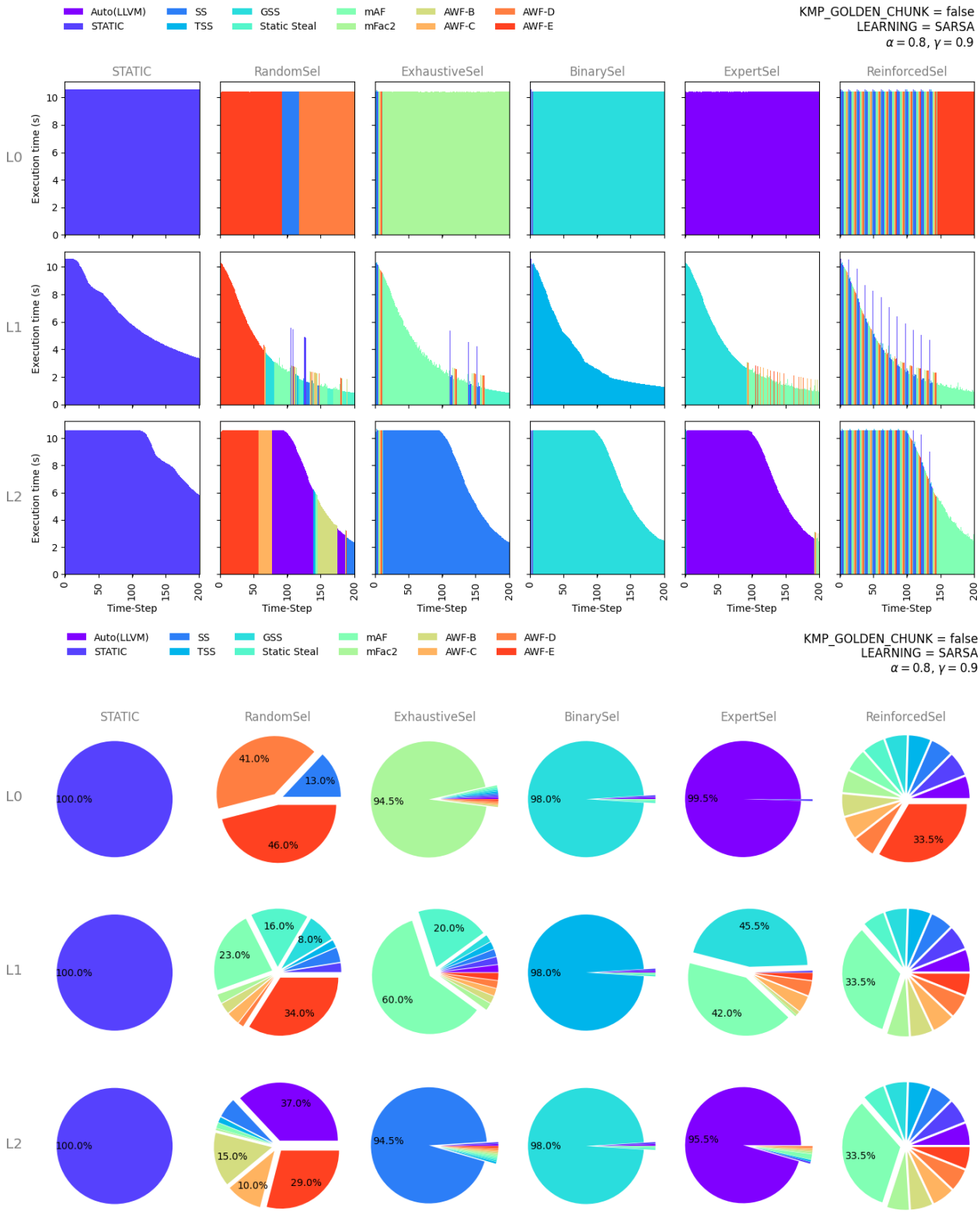


Figure A.2: Comparing loop execution times and frequency of DLS algorithm selection for AUTO methods (`KMP_GOLDEN_CHUNK=false`, `LEARNING_METHOD=SARSA`).

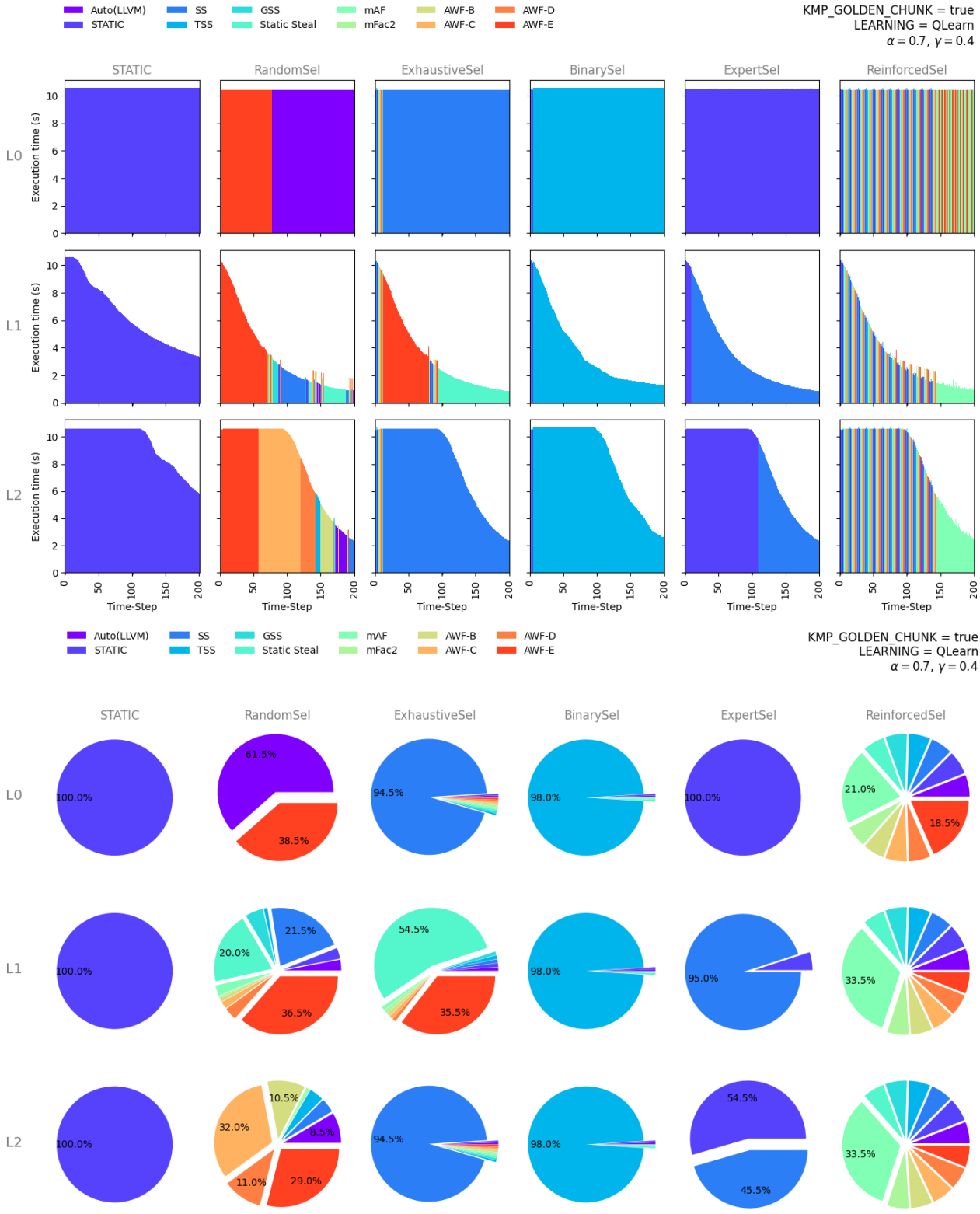


Figure A.3: Comparing loop execution times and frequency of DLS algorithm selection for AUTO methods (`KMP_GOLDEN_CHUNK=true`, `LEARNING_METHOD=Q-Learning`).

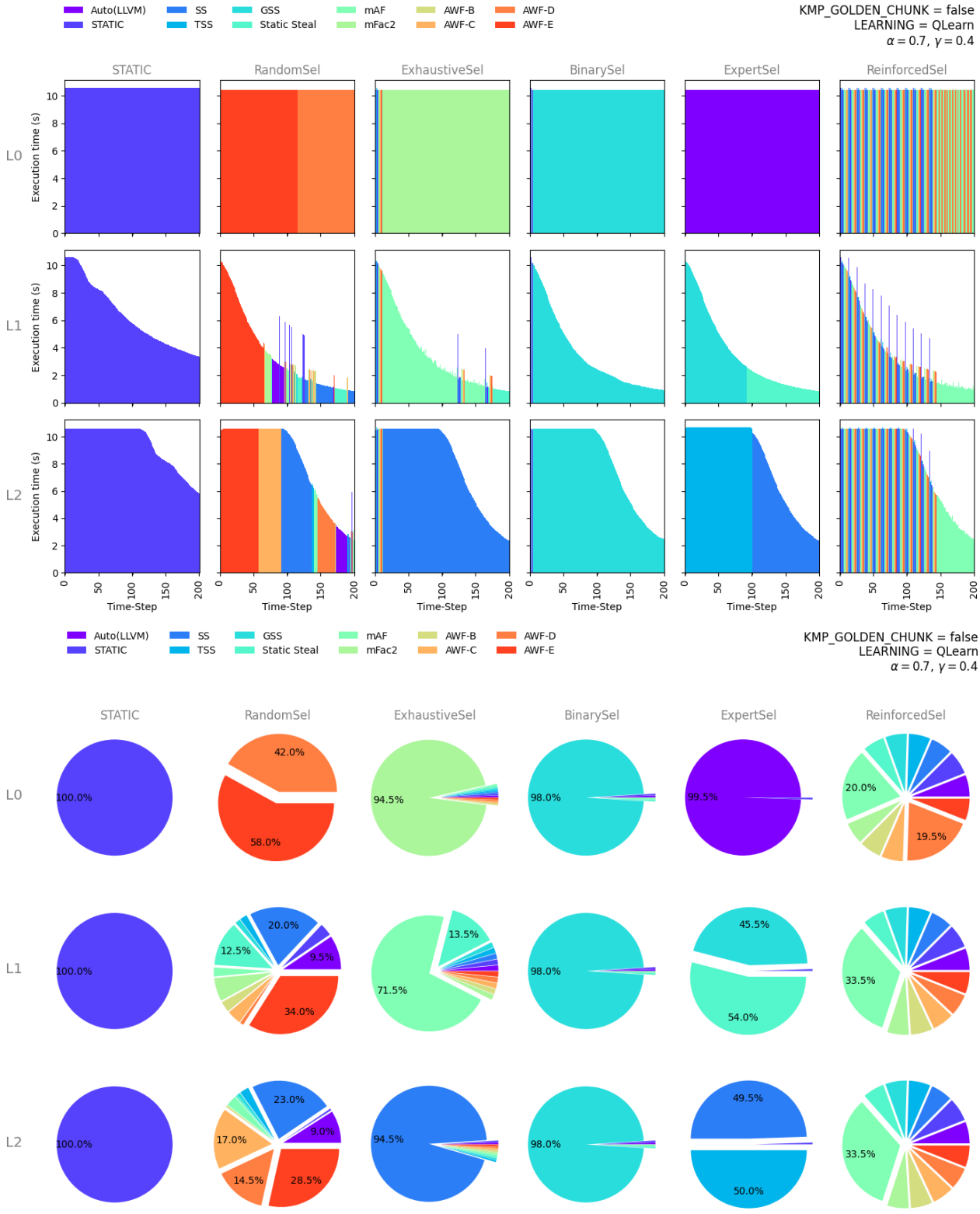


Figure A.4: Comparing loop execution times and frequency of DLS algorithm selection for AUTO methods (`KMP_GOLDEN_CHUNK=false`, `LEARNING_METHOD=Q-Learning`).

A.3 Comparison Of Automatic Selection Methods And Fixed Scheduling Algorithms

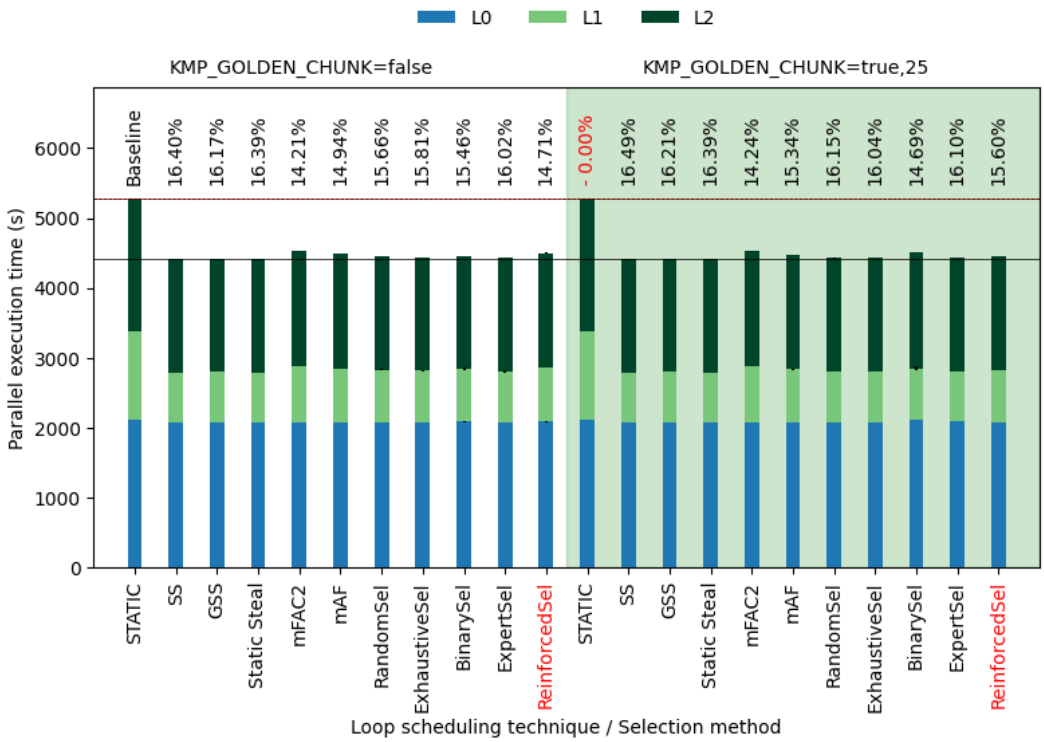


Figure A.5: Parallel execution time for each loop in Mandelbrot executing on node Type A. The percentages denote performance improvements for the selected scheduling method vs. the baseline (`STATIC` schedule) with learning method Q-Learning.

A.4 Sample Jobfile

Listing A.1: Running the distributed time-stepping Mandelbrot application with the SLURM workload manager with appropriate parameters on the miniHPC-Broadwell system

```
#!/bin/bash

#SBATCH --job-name mandel
#SBATCH --time=03:00:00
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=1
#SBATCH --cpus-per-task=20
#SBATCH --partition=xeon
#SBATCH --exclude=cl-node001,cl-node002,cl-node004,cl-node007,cl-node006,cl-node010
#SBATCH --hint=nomultithread
#SBATCH --output=/storage/shared/msc/luc-kury/kury-msc-project/code/mandel/output/
    mandel_schedule-auto,6_golden-1_run-1.txt

module load intel/2019a

export LD_LIBRARY_PATH=/storage/shared/msc/luc-kury/kury-msc-project/code/lb4omp/intel/
    runtime/src/:$LD_LIBRARY_PATH
export OMP_NUM_THREADS=20
export OMP_PROC_BIND=close
export OMP_PLACES=cores
export OMP_SCHEDULE=auto,6
export KMP_ALPHA=1
export KMP_CPU_SPEED=2600
export KMP_Golden_Chunksize=1
export KMP_TIME_LOOPS=/storage/shared/msc/luc-kury/kury-msc-project/code/mandel/output/
    looptimes/looptime_mandel_schedule-auto,6_golden-1_run-1.txt
export KMP_RL_ALPHA=0.80
export KMP_RL_GAMMA=0.90
export KMP_RL_LMETHOD=1

srun /storage/shared/msc/luc-kury/kury-msc-project/code/mandel/mandel.o 100000 512 0 0 0.5
```

Declaration on Scientific Integrity
Erklärung zur wissenschaftlichen Redlichkeit

includes Declaration on Plagiarism and Fraud
beinhaltet Erklärung zu Plagiat und Betrug

Author — Autor
Luc Kury

Matriculation number — Matrikelnummer
10-462-687

Title of work — Titel der Arbeit
Automated Selection of Scheduling Techniques in OpenMP using Reinforcement Learning

Type of work — Typ der Arbeit
Master Research Project

Declaration — Erklärung
I hereby 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.

Hiermit erkläre ich, dass mir bei der Abfassung dieser Arbeit nur die darin angegebene Hilfe zuteil wurde und dass ich sie nur mit den in der Arbeit angegebenen Hilfsmitteln verfasst habe. Ich habe sämtliche verwendeten Quellen erwähnt und gemäss anerkannten wissenschaftlichen Regeln zitiert.

Basel, 13^{th} of September 2021

Signature — Unterschrift