Decentralized scheduling through an adaptive, trading-based multi-agent system

Michael Kölle[1], Lennart Rietdorf[1] and Kyrill Schmid[1]

[1]LMU Munich, Germany
{michael.koelle, kyrill.schmid}@ifi.lmu.de, len.rietdorf@campus.lmu.de

Abstract

In multi-agent reinforcement learning systems, the actions of one agent can have a negative impact on the rewards of other agents. One way to combat this problem is to let agents trade their rewards amongst each other. Motivated by this, this work applies a trading approach to a simulated scheduling environment, where the agents are responsible for the assignment of incoming jobs to compute cores. In this environment, reinforcement learning agents learn to trade successfully. The agents can trade the usage right of computational cores to process high-priority, high-reward jobs faster than low-priority, low-reward jobs. However, due to combinatorial effects, the action and observation spaces of a simple reinforcement learning agent in this environment scale exponentially with key parameters of the problem size. However, the exponential scaling behavior can be transformed into a linear one if the agent is split into several independent subunits. We further improve this distributed architecture using agent-internal parameter sharing. Moreover, it can be extended to set the exchange prices autonomously. We show that in our scheduling environment, the advantages of a distributed agent architecture clearly outweigh more aggregated approaches. We demonstrate that the distributed agent architecture becomes even more performant using agent-internal parameter sharing. Finally, we investigate how two different reward functions affect autonomous pricing and the corresponding scheduling.

Introduction

In the field of cooperative AI, we seek methods to establish cooperative behavior amongst independent and autonomous agents (Dafoe et al., 2020). Many situations, like autonomous driving, require autonomous agents working together, which makes the ability to act cooperatively a key part in integrating artificial intelligence into our daily lifes. Research in reinforcement learning (RL) has shown successful applications for single agent (Mnih et al., 2015; Silver et al., 2017) and multi-agent systems (Leibo et al., 2017; Phan et al., 2018; Vinyals et al., 2019). While fully cooperative tasks, where all agents receive the same reward and pursue the same goal, can be solved by a centralized training approach, this is not the case if agents have independent rewards and goals. Furthermore, the behavior of purely self-interested agents in multi-agent systems with common, shared resources often results in sequential social dilemmas which expose tension between individual and collective rationality (Rapoport, 1974), especially when the resources are scarce (Leibo et al., 2017). Exploration in this field has led to various approaches on how to influence the self-interested actions of independent, decentralized training agents towards a higher, emergent common good. The approaches range from game theory (Lerer and Peysakhovich, 2018), modeling of social preferences (Busoniu et al., 2010), to ones where agents incentivise each other to cooperate (Yang et al., 2020; Schmid et al., 2018; Lupu and Precup, 2020; Schmid et al., 2021).

In this work, we build upon the action market approach in Schmid et al. (2018, 2021), specifically we introduce a step between making and accepting an offer. This allows agents to observe the offers first and then make a decision about those offers whereas in Schmid et al. (2018) the agents had to guess the demand based on past transitions. Because of this extra step, the emergent cooperation tends to be more stable than in Schmid et al. (2018). We introduce a multi-agent scheduling environment where mutual benefits can be realized by trading with each other. In this environment highly multi-dimensional actions are necessary to trade and allocate jobs on compute cores. Using neural networks (NNs) as the decision making entities, an important problem emerges: Each multi-dimensional action has to be translated into one of exponentially many one-dimensional actions. This renders decision making exponentially difficult with a linearly increasing number of compute cores and queue lengths. Therefore, we evaluate different agent architectures that are designed to address or even circumvent the problem of exponential action spaces. Thus, we answer the question which agent architecture is most successful in mastering the highly multi-dimensional action space of the trading-based scheduling environment. Additionally, we evaluate the implications if one of these agent architectures is enabled to set trading prices freely on its own. All code and parameters of the experiments can be found here[1].

[1]https://github.com/lr40/marl-scheduling.git

Related Work

In this work, the approach to coordinate the multiple agents is related to the Action Market introduced in Schmid et al. (2018, 2021). Yet there is an important difference in the time dimension: While the market mechanism in Schmid et al. (2018) is based on a simultaneous matching of supply and demand, the market mechanism of this work always has a time step between the making of an offer and its acceptance. Agents observe first and then decide which ones to accept and which ones not to accept. The "guessing" of a demand, learned by past rewards as in Schmid et al. (2018), which then leads to behavioral changes, does not exist here. Rather an explicit offer, which will be observed, has to be accepted to conclude the trade. This circumvents the problem that agents in Schmid et al. (2018) learn over time to take advantage of the cooperative behavior of the counter party: They do so by suddenly stopping the costly demand action - even though demand is still present. The deceived agent then delivers the desired action - expecting the traded reward - without receiving the reward. Learned cooperation can be unlearned by such breaches of trust. On the other hand, the approach presented in this paper is closer to a direct, explicit communication of the agents. Thus, it is accompanied by an increased overhead.

Approach

Scheduling environment

Scheduling comes into play when the demand for resources is higher than the available processing capacity (Tanenbaum, 2009). In this context we use scheduling for the assignment of computational jobs to computational cores. A computational job i can be described by its arrival time t_i^{AT}, burst time t_i^{BT} and priority p_i. The burst time is the duration it takes for the job to be executed. This time cannot be influenced by the scheduler. After the jobs have completed we measure the turnaround time t_i^{TAT} (Eq. 1) for each job. The turnaround time is the total duration that the job remained in the system. In addition to the burst time, it also includes the waiting time t_i^{WT} during which the job waited inactively for its allocation to a core.

$$t_i^{TAT} = t_i^{BT} + t_i^{WT} \qquad t_i^{NTAT} = \frac{t_i^{TAT}}{t_i^{BT}} \quad (1)$$

In order to evaluate a scheduling method, we use the burst time and turnaround time to measure the normalized turnaround time t_i^{NTAT} (Eq. 1) of a computational job. The scheduling process controls the waiting time. The closer the normalized turnaround time approaches its optimal value 1, the smaller the included waiting time has been.

We implemented a scheduling environment for multiple RL agents. In this environment a common resource - *M* computing cores - ought to be used efficiently by *N* agents, each with *K* job slots. The agents want to compute renewing jobs from the slots on the cores. Jobs are of a certain type whose combination of priority value, length in timesteps and spawn probability are specified as an environment parameter. The trading mechanism allows the agents to enhance the individual rewards as well as the overall scheduling performance. In contrast to traditional, centralized scheduling concepts, the scheduling of this work is partially decentralized. The agents have the chance to exchange access to compute cores amongst each other if the current owner (i.e. user) of a core accepts an explicit, observed offer directed at this core. This offer is a 2-tuple consisting of the reward payment and the necessary timesteps until payment. In scenarios with fixed prices, the reward payments are given as an environment parameter corresponding to the job type's priority. In scenarios with free prices, the reward payment can be chosen by the offering agent. No scheduling strategy is predefined; the scheduling results from the learned actions of the agents. An exception to this is the hard coded auctioneer, which manages currently idle cores (agents lose ownership of idle cores by design) and grants access to the highest bidding RL agent. The pure behavior of the auctioneer - without any intra-agent trading - implements a variant of first-come-first-serve (FCFS) where jobs with higher priority values are preferred. Figure 1 gives an overview of the scheduling environment and the RL loop that the environment implements.

Rewards

In the scheduling environment, the rewarding scheme for the individual agents follows an egoistic principle. This is plausible in the context of independent learners that want to get their jobs computed or want to receive traded rewards.

Reward for terminating jobs and trading access The responsible owner of a core - i.e. the agent currently running a job on it - gets a reward as soon as its own job has terminated. This reward has the same value as the priority of the job. The owner of a core can receive a reward in yet another way: through a trade. If offers are accepted, the promised reward payments are first stored by the environment in a chronological order with respect to a core as long as no job has terminated yet. When a job terminates all stored reward claims and liabilities of the preceding trades are settled and each participant in the reward chain receives its net payout. These reward chains initially start with the auctioneer as the owner of all cores and bring no additional net rewards into the system when a job terminates. They are only a redistribution of the generated reward.

Reward for making offers For offering, on the other hand, the rewards are received directly in the next time step if the offer has been successfully accepted by the counterparty. The generated reward is equal to the priority of the mediated job.

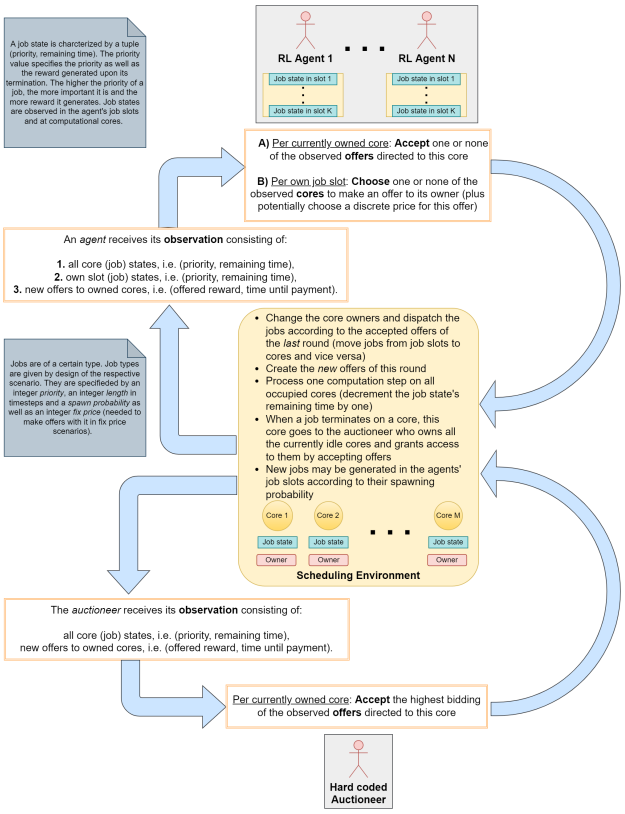


Figure 1: The general structure of the RL environment. Agents own cores when they are currently computing on them. If no compute job is currently running on a core, the auctioneer owns it. Agents make offers to the owner of a core to let a job compute on that core if the offer is accepted.

Price setter reward In free price scenarios, the agents will be enabled to freely set prices for their offers. The rewarding scheme of the price setting action can be motivated by a commercial and a non-commercial motive. In the former case the agent wants to bid an amount as small as possible that is just on the brink of being accepted. Mainly, the paid out reward is the difference between the priority of the facilitated job p_i and the chosen price x which can also be negative if the price is set too high (Eq. 2).

$$R_1(p_i, x) = \begin{cases} p_i - x & \text{if } p_i \neq x \\ 0.5 & \text{otherwise} \end{cases} \tag{2}$$

In the non-commercial case, the reward equals the priority of the the facilitated job p_i as long as the set price x was not higher than the job priority (Eq. 3). In this case the price setter can increase the bids for free as long as it does not overshoot.

$$R_2(p_i, x) = \begin{cases} p_i & \text{if } p_i \geq x \\ p_i - x & \text{otherwise} \end{cases} \tag{3}$$

Agent architectures

Multi-agent systems consist of multiple agents that share a common environment. An agent is an autonomous entity with two main capabilities: observing and acting. The observation of the current state of the environment allows the agent to choose an appropriate action out of a given action set. The chosen action depends on an agent's policy. In this work, we used different RL agent architectures based on the PPO algorithm (Schulman et al., 2017) to learn a good policy for the proposed environment. All NNs used in this work contain one hidden layer of neurons. An important constraint to consider in this regard is that RL algorithms usually allow one NN to output only one one-dimensional (1D) action at a time. A multi-dimensional action consisting of d sub actions is only attainable if the environment translates the chosen 1D-action a back into one of the $\mathcal{O}(2^d)$ d-dimensional actions. In this manner, the resulting amount of actions grows exponentially as the dimensionality of the multi-dimensional action increases with the problem size or scaled setup of the environment.

Fully aggregated agent Disregarding this important constraint, one can naively construct an agent composed of only one neural net that has one hidden layer of 64 neurons. The action of this single neural net is responsible for accepting or declining offers for up to M cores (if this agent owns all the cores), and making up to K offers for all of its associated K job slots at the same time. A single offer is made by choosing a target core for a job of an own slot. The prices are predefined per job type as an environment parameter. The dimensionality of this aggregated action vector sums up to overwhelming $M + K$ which translates to $\mathcal{O}(2^{M+K})$ many actions. The agent type implementing this design will henceforth be called *fully aggregated agent*.

Semi-aggregated agent The first step to reduce the exponential scaling behavior of the action space is to split the agent up in two neural networks (each with a hidden layer of 32 neurons) which can be regarded as two independent, complementary sub agents: one part **A** for accepting resp. declining offers (cf. fig. 1 action part **A**) and one part **B** for making offers (cf. fig. 1 action part **B**). Core (job) states plus an ownership flag and offers to this core are observed by part **A**. Core (job) states and slot (job) states are observed by part **B**. The two networks generate two 1D-actions which will be translated back into an M-dimensional action (one dimension for each core) resp. K-dimensional action (one dimension for each job slot). This results in a less intense scaling behavior of the action space which is nevertheless still exponential. The constructed agent type will be called *semi-aggregated agent*.

Distributed agent The idea to split the agent up in several, independent, complementary neural networks can be taken

one step further by dividing the semi-aggregated accepting side **A** up into M-many acceptor networks and dividing the offer side **B** up into K-many offer networks (in both cases each with a hidden layer of 16 neurons). Each acceptor network is responsible for managing the incoming offers of *one* core if the agent owns the core. Each core chooser network is responsible for making the offers of *one* job slot. Although the amount of the neural networks increases sharply by this, one big advantage arises: The $M + K$-many 1D-actions of the $M + K$ networks do not need to be translated into a multi-dimensional action vector. This is because these actions are already intrinsically one-dimensional since just one offer index respectively one core index has to be selected per action.

Distributed agent with parameter sharing For the distributed agent, a possible optimization is local parameter sharing. Instead of sustaining $M + K$ independent networks which constitute the distributed agent, we implement a local parameter sharing between the M-many accepting networks and between the K-many offering networks. Consequently, only two neural networks that output in total $M + K$ intrinsic 1D-actions will make up the agent. By doing so, a faster training process and less overhead arises. Thus, the observation and action spaces correspond to atomic actions rather than aggregated actions which had to be translated to the atomic level.

Distributed agent with free price setting So far, the considered agent types are able to make offers and to accept offers, but they are not able to set the prices for the offers on their own. Instead, they have to rely on fixed prices specified as an environment parameter. To be able to set prices freely the distributed Agent is extended by a third type of neural network: a price setter network. The chosen price has to be an integer from [0,*max_Prio*] where *max_Prio* denotes the maximum priority of all job types in the run scenario.

Experiments[2]

The effect of intra-agent trading

We evaluate two types of jobs in this scenario: Long-running, frequently occurring, low-priority jobs that block the scarce compute cores and much rarer, shorter, high-priority jobs. The exchange prices are given as an environment parameter according to the job's priority. The intra-agent trading mechanism enables the agents to trade computing core access amongst each other. Figure 2 shows that intra-agent trading significantly reduced the normalized turnaround time of the highly prioritized job type. Trades were used to let the high priority jobs get access to the cores. The intra-agent trading thus outperformed the priority oriented FCFS approach of the auctioneer.

[2]The used hyperparameters and environment parameters can be found in the README file of the repository.

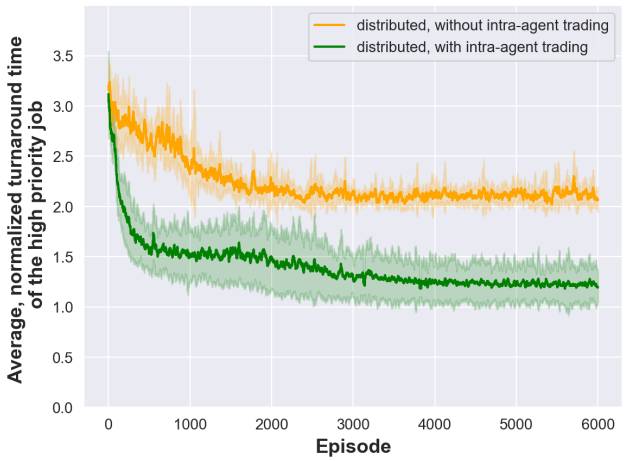


Figure 2: Mean and standard deviation of 10 independent runs. Intra-agent trading significantly reduces the normalized turnaround time of the highly prioritized job type. The distributed agent architecture was used.

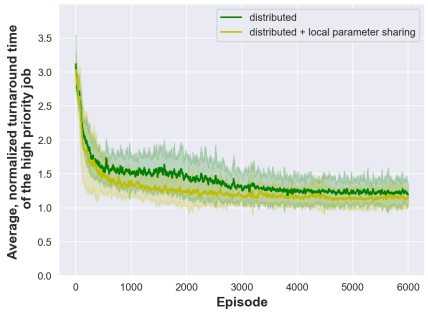
Agent architecture and scheduling performance

In this section the same scenario of the previous section is evaluated. Here, we used 2 agents each with 3 self-refilling job slots competing for 2 cores. Different agent architectures lead to different performances. This is due to their different sizes of action spaces. It is shown in figure 3a that the distributed agent architecture yields the lowest turnaround time of the high priority job type. Interestingly, the fully aggregated architecture yields the second lowest, normalized turnaround time although it has the largest action space. The better performance is achieved because the neural network learns to trade with itself since it makes the decision for accepting as well as making offers. Since the used scaling is still manageable it can derive an advantage compared to the semi-aggregated agent type (cf. table 1).

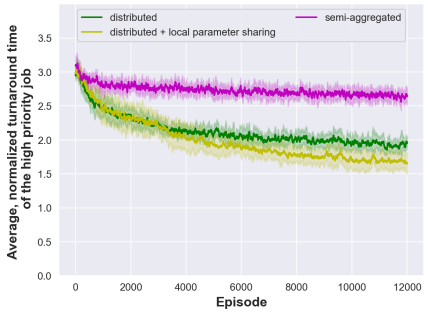
Figure 3b shows the same data for the distributed agent type as in figure 3a but compares those to the performance of the distributed agent type with local parameter sharing. It can be seen that the employment of local parameter sharing improves the overall performance as well as the time necessary for the adaption.

The scenario can also be scaled up to 4 agents each with 3 job slots who compete for 4 cores. Remarkably, this scaling is already some orders of magnitude too high for the fully aggregated agent architecture since its action space cardinality is already out of scope (cf. table 1). So only the performance of the distributed architecture with and without parameter sharing as well as the semi-aggregated architecture can be compared. Figure 3c shows this. Again, the distributed agent architecture significantly outperforms the semi-aggregated architecture. Employing local parameter sharing yields another performance boost in addition to that.

(a) 2 agents with self-refilling queues of 3 jobs compete for 2 cores.



(b) 2 agents with self-refilling queues of 3 jobs compete for 2 cores.

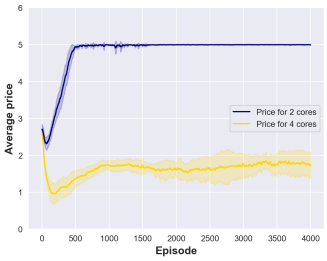


(c) 4 agents with self-refilling queues of 3 jobs compete for 4 cores.

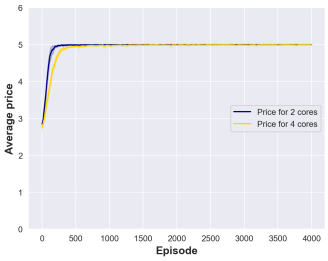
Figure 3: Mean and standard deviation of 10 independent runs. The average, normalized, turnaround times of the high priority job type when using the distributed (green), semi-aggregated (magenta), fully aggregated (blue) and distributed with local parameter sharing (yellow) agent architecture. The lower the curve, the better the effective scheduling performance.

Acting unit	**2 agents, 2 cores**	**4 agents, 4 cores**
distributed offer unit	3	5
distributed acceptor unit	7	13
semi-aggregated offer unit	9	125
semi-aggregated acceptor unit	49	28 561
fully aggregated unit	441	35 701 125

Table 1: Cardinality of the action spaces of the acting (sub) units for two scalings of the scenario.



(a) The price development under **commercial** reward when the cores are scarce (blue) or less scarce (gold).



(b) The price development under **non-commercial** reward when the cores are scarce (blue) or less scarce (gold).

Figure 4: Mean and standard deviation of 10 independent runs are shown.

Price level and scarcity

In real life, prices are expressions of relative scarcities. This experiment investigates to what extent in the scheduling environment scarcer computational cores lead to higher prices if free price setting is enabled. For the price setters, two reward regimes are compared. To keep complexity low, there is only a single job type in this scenario. Its priority and length equals 5. 2 agents each with 3 job slots - i.e. a total of 6 jobs - compete for 2 cores in one scenario and for 4 cores in the other scenario.

In Figure 4a, the price development in the commercial reward regime is shown with scarce 2 compute cores and more abundant 4 cores. For 2 compute cores the price rises to the maximum priority of the scenario. The 6 price setters learn to bid each other up to this level because access to the few computational cores is so contested. On the other hand, when the number of cores is doubled to 4, the commercial price setters learn a significantly lower price level. So using the commercial reward function, the found price level increases with the scarcity of computational cores.

Figure 4b, on the other hand, shows the price development in the non-commercial reward regime in the same scenario with 2 and 4 cores. In both cases, the price level rises to the maximum value. In contrast to the commercial reward regime, the price also becomes maximum when more abundant 4 cores are available. This is due to the fact that price increases are not associated with any cost for the price setter but increase the chances for an offer to be accepted. Thus the high price becomes the dominant strategy. Using the non-commercial reward function, prices reflect the priority of the associated jobs rather than the scarcity of the compute cores.

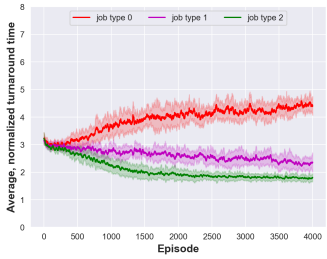
Price level and scheduling

How the free price setting influences the realized scheduling will be analysed in this section. There are 2 agents, each with a queue length of 3 jobs that want to compute three, equally frequent job types with the priorities 2, 4, and 8 on two cores.

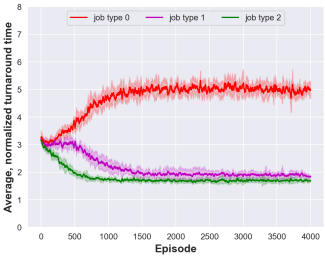
How prices evolve under the commercial reward regime is shown in Fig. 5a. In the beginning, there is a sharp initial drop in the price levels of all job types. At a certain point, however, the commercial price setters learn that outbidding competitors is a decisive factor for success. As a result,

(a) **Commercial** reward: average prices

(b) **Non-commercial** reward: average prices



(c) **Commercial** reward: average, normalized turnaround times



(d) **Non-commercial** reward: average, normalized turnaround times

Figure 5: Mean and standard deviation of 10 independent runs are shown. The priorities of the job types are given as [2 (red), 4 (magenta), 8 (green)]. The subfigures **(a)** and **(b)** only include realized prices of accepted offers.

prices for all types of jobs rise again but split up according to their priority. The transitive order of prices then corresponds to the transitive order of job type priorities. Overall, the spectrum of prices spreads less upward than for the non-commercial rewards (see subfig. 5b). This is because in the case of commercial rewards there is an incentive for the individual price setter to outbid the lower job types with an amount as low as possible. The price of the lowest priority job type 0 is permanently very close to its priority of 2. At the lower end of the possible priorities, the prices have to be at their maximum to have any chance of competing with the same or other job types.

How prices evolve under the non-commercial reward regime is shown in Figure 5c. Here, too, we see an initial drop in all price levels in the beginning. Compared to the commercial reward regime, however, the drop is less deep because there is less incentive for prices to fall. After that, prices also split up according to the respective job priorities when the price setters learn to outbid each other. However, the splitting leads to significantly higher prices for job types 1 and 2. This is because the price setters under this reward function get the same reward for all prices that are not too high. Hence a strong incentive for price raises is given. In the emerging system the low and medium prioritized jobs types 0 and 1 have to be priced at their maximum in order to have a chance against the arbitrarily high prices of job type 2.

The evolution of prices is reflected in the evolution of turnaround times, shown in Figures 5c and 5d. Under both reward functions, as the prices of higher-priority jobs increase, the corresponding turnaround times decrease and vice versa. The greater spread of price levels under non-commercial rewards causes the turnaround times to spread apart more sharply and quickly. For commercial rewards, on the other hand, the normalized turnaround times are closer together and change only more slowly. This slower and weaker adaptation is caused by the less differentiated prices.

Conclusion

This work addressed the question which agent architecture is most successful in mastering the highly multi-discrete action space of the trading-based scheduling environment. Success depends on how the agents' action and observation spaces scale, which in turn is largely determined by the agent architecture. Semi- and fully aggregated agent types exhibit an exponential scaling behavior of their observation and action spaces with the scenario size. This exponential scaling behavior can be transformed into a linear one if the agent is split up in multiple neural networks. We show that agents of the distributed architecture adapt best and fastest to a scenario where high priority jobs are to be prioritized over low priority ones via intra-agent trading. Furthermore, it became evident that the action space of the fully aggregated agent type increased so strongly even at small problem sizes that it is unsuitable for practical use. In addition, we show that the performance of the distributed architecture further improved when agent-level parameter sharing was implemented.

Finally, we examined the results when agents set the trading prices freely by themselves. We compared the effects of two, different reward functions for the price setter. Using the commercial reward function, the emergent prices can reflect the scarcity conditions in the environment, whereas the non-commercial reward function causes prices to be a neutral carrier of information about job priorities. For scheduling different job types it was found that adaption succeeded under both reward functions. The higher the priority of a job type, the lower its mean normalized turnaround time and the higher its found price. Compared to the commercial reward, the non-commercial reward led to more differentiated prices, lower turnaround times of high priority jobs and faster adaption.

In conclusion, splitting the RL agent into different submodules led to crucial performance improvements in the trading-based scheduling environment compared to more aggregated approaches. The reduced cardinality of the action spaces even allowed for the successful introduction of free price setting by the agents.

References

Busoniu, L., Babuska, R., and Schutter, B. D. (2010). Multi-agent reinforcement learning: An overview. *Chapter 7 in Innovations in Multi-Agent Systems and Applications – 1 (D. Srinivasan and L.C. Jain, eds.), vol. 310 of Studies in Computational Intelligence*, pages 183–221.

Dafoe, A., Hughes, E., Bachrach, Y., Collins, T., McKee, K. R., Leibo, J. Z., Larson, K., and Graepel, T. (2020). Open problems in cooperative AI. *CoRR*, abs/2012.08630.

Leibo, J. Z., Zambaldi, V., Lanctot, M., Marecki, J., and Graepel, T. (2017). Multi-agent reinforcement learning in sequential social dilemmas. *Proceedings of the 16th Conference on Autonomous Agents and MultiAgent Systems (AA-MAS 2017)*, pages 464–473.

Lerer, A. and Peysakhovich, A. (2018). Maintaining cooperation in complex social dilemmas using deep reinforcement learning. *arXiv:1707.01068 [cs]*.

Lupu, A. and Precup, D. (2020). Gifting in multi-agent reinforcement learning. *Proceedings of the 19th International Conference on autonomous agents and multiagent systems*, pages 789–797.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A., Veness, J., Bellemare, M., Graves, A., Riedmiller, M., Fidjeland, A., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., and Hassabis, D. (2015). Human-level control through deep reinforcement learning. *Nature*, 518:529–33.

Phan, T., Belzner, L., Gabor, T., and Schmid, K. (2018). Leveraging Statistical Multi-Agent Online Planning with Emergent Value Function Approximation. *arXiv:1804.06311 [cs]*.

Rapoport, A. (1974). Prisoner's Dilemma — Recollections and Observations. In Rapoport, A., editor, *Game Theory as a Theory of a Conflict Resolution*, Theory and Decision Library, pages 17–34. Springer Netherlands, Dordrecht.

Schmid, K., Belzner, L., Gabor, T., and Phan, T. (2018). Action markets in deep multi-agent reinforcement learning. *Artificial Neural Networks and Machine Learning – ICANN 2018. ICANN 2018. Lecture Notes in Computer Science*, 11140:240–249.

Schmid, K., Belzner, L., Müller, R., Tochtermann, J., and Linnhoff-Popien, C. (2021). Stochastic Market Games. In Zhou, Z.-H., editor, *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI-21*, pages 384–390. International Joint Conferences on Artificial Intelligence Organization.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. (2017). Proximal policy optimization algorithms. *CoRR*, abs/1707.06347.

Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., Chen, Y., Lillicrap, T., Hui, F., Sifre, L., van den Driessche, G., Graepel, T., and Hassabis, D. (2017). Mastering the game of Go without human knowledge. *Nature*, 550(7676):354–359.

Tanenbaum, A. S. (2009). *Moderne Betriebssysteme*. Pearson Studium, 3. aktualisierte Auflage edition.

Vinyals, O., Babuschkin, I., Czarnecki, W. M., Mathieu, M., Dudzik, A., Chung, J., Choi, D. H., Powell, R., Ewalds, T., Georgiev, P., Oh, J., Horgan, D., Kroiss, M., Danihelka, I., Huang, A., Sifre, L., Cai, T., Agapiou, J. P., Jaderberg, M., Vezhnevets, A. S., Leblond, R., Pohlen, T., Dalibard, V., Budden, D., Sulsky, Y., Molloy, J., Paine, T. L., Gulcehre, C., Wang, Z., Pfaff, T., Wu, Y., Ring, R., Yogatama, D., Wünsch, D., McKinney, K., Smith, O., Schaul, T., Lillicrap, T., Kavukcuoglu, K., Hassabis, D., Apps, C., and Silver, D. (2019). Grandmaster level in StarCraft II using multi-agent reinforcement learning. *Nature*, 575(7782):350–354.

Yang, J., Farajtabar, M., Sunehag, P., Hughes, E., and Zha, H. (2020). Learning to incentivize other learning agents. *34th Conference on Neural Information Processing Systems (NeurIPS 2020)*.