E-MAPP: Efficient Multi-Agent Reinforcement Learning with Parallel Program Guidance

Can Chang[1,2], **Ni Mu**[3], **Jiajun Wu**[4], **Ling Pan**[5], , **Huazhe Xu**[1,2]
[1] IIIS, Tsinghua University [2] Shanghai Qi Zhi Institute [3] Southeast University
[4] Stanford University [5] Mila, Université de Montréal
cc22@mails.tsinghua.edu.cn, xuhuazhe12@gmail.com

Abstract

A critical challenge in multi-agent reinforcement learning (MARL) is for multiple agents to efficiently accomplish complex, long-horizon tasks. The agents often have difficulties in cooperating on common goals, dividing complex tasks, and planning through several stages to make progress. We propose to address these challenges by guiding agents with programs designed for parallelization, since programs as a representation contain rich structural and semantic information, and are widely used as abstractions for long-horizon tasks. Specifically, we introduce **E**fficient **M**ulti-**A**gent Reinforcement Learning with **P**arallel **P**rogram Guidance (E-MAPP), a novel framework that leverages parallel programs to guide multiple agents to efficiently accomplish goals that require planning over $10+$ stages. E-MAPP integrates the structural information from a parallel program, promotes the cooperative behaviors grounded in program semantics, and improves the time efficiency via a task allocator. We conduct extensive experiments on a series of challenging, long-horizon cooperative tasks in the *Overcooked* environment. Results show that E-MAPP outperforms strong baselines in terms of the completion rate, time efficiency, and zero-shot generalization ability by a large margin.

1 Introduction

Multi-agent reinforcement learning (MARL) has achieved significant progress by advancing the cooperation of multiple agents to accomplish complex tasks, e.g., multi-robot control [20], autonomous driving [48, 54], and video games [50, 4]. Most recent advances in MARL focus on the tasks that feature behavior coordination [39] or joint motion planning [41]. However, for *long-horizon tasks* such as preparing a dish, existing methods often suffer from the inability to understand the task compositionality and subtasks' dependencies, resulting in inefficient cooperation and frequent conflicts. Therefore, a natural question to ask here is how we can solve long-horizon tasks in MARL, in the face of large state/action spaces and sparse feedback.

Long-horizon tasks are usually blessed with rich structure, and thus can be divided into a sequence of subtasks that can be resolved separately. Previous work [45, 52] has introduced programs as instructions to help a single agent understand the task hierarchy and accomplish the task. Inspired by them, we develop a general multi-agent framework, where agents can leverage programs for accomplishing long-horizon tasks together. This is a challenging problem, and three substantial issues will emerge if multiple agents are naively enforced to follow sequential programs: first, sequential programs do not explicitly express the dependencies among subtasks, thus hindering the division of jobs among agents; second, different agents might have different abilities to accomplish certain

36th Conference on Neural Information Processing Systems (NeurIPS 2022).

subtasks or lines of programs; third, when assigned to a subtask together, multiple agents might need to collaborate without blocking resources with each other.

As modern CPUs dispatch instructions to parallel processors, we propose a new multi-agent framework, **E**fficient **M**ulti-**A**gent Reinforcement Learning with **P**arallel **P**rogram Guidance (E-MAPP), guiding cooperation and execution by automatically inferring the structure of parallelism from programs. Specifically, we first design a domain-specific language (DSL) for multi-agent cooperation, and use multi-stage learning to ground subroutines of a given program into the agents' policy. Then, we learn feasibility functions, which entail the ability of agents to complete specific subroutines in the program in the status quo. Finally, we leverage the learned task structure to automatically enforce cooperation and division of labor among agents.

We conduct experiments on gradually more difficult challenges in the *Overcooked* [50] environment. The *Overcooked* environment requires the agents to cooperate on very long-horizon tasks, such as preparing dishes, while avoiding conflicting behaviors. Our method significantly outperforms other strong baselines in completion rates and efficiency. The program structure also enables E-MAPP to deliver superior compositional generalization to novel scenes.

Our main contributions can be summarized as follows:

- We formulate a novel task of learning multi-agent cooperation via the guidance of parallel programs.
- We present a novel framework for program grounding and long-horizon planning and instantiate the framework into practical multi-agent reinforcement learning algorithms.
- We demonstrate the effectiveness of E-MAPP in completion rates and generalization ability over existing strong baselines and provide empirical analysis in long-horizon tasks.

2 Related Work

Cooperative Multi-Agent Reinforcement Learning. In a multi-agent cooperative game, agents collaborate with each other on a common goal [33]. Researchers have investigated many ways to facilitate agent coordination [21, 12, 53, 34, 19]. Value-based MARL algorithms engage in discovering the relationship between global value function and local value functions [46, 38], while policy-based MARL algorithms use a centralized critic [51, 26] to coordinate agent behaviors. More specifically, MAPPO [51] and MADDPG [26] leverage a fully-observable central critic to solve the issue of non-stationarity [33]. Value factorization approaches [38, 46, 37] decompose the global value function into a combination of local agent-wise utilities to cope with scalability, while policy factorization approaches [18] factorize the joint action space to coordinate marginal policies.

Reinforcement Learning for Long-horizon Tasks. Reinforcement learning agents usually lack the ability to plan and reason over a long time horizon due to sparse rewards [49, 14, 40, 15, 16, 30]. Goal-conditioned reinforcement learning [36] is one of the popular paradigms to address the sparse supervision problem. Imitation learning [17, 16] is another approach to solve the sparse reward problem. Another line of work has presented automatic goal generation and selection algorithms [3, 35, 24, 9]; however, this introduces new challenges to design a suitable goal space that enjoys rich semantic meanings [13]. By contrast, our work uses the subtasks corresponding to possible subroutines in a program as goals, which are associated with domain knowledge. Hierarchical reinforcement learning [HRL; 47, 42, 28, 43, 29] is another path to solve long-horizon tasks, using a high-level policy for long-term planning and low-level policies for motion planning or specific behaviors. While our work is related to multi-agent hierarchical reinforcement learning [27], which explicitly provides the directed acyclic task graph and the necessity of cooperation of each subtask, we focus on learning task structure such as subtask dependencies, loops, and branchings from the program and judging the necessity of cooperation without additional information.

Instruction-Guided Agents. Many recent advances have testified the advantages of leveraging structured prior knowledge such as task graphs [1, 22], natural languages [2, 6, 23] and programs [45, 52] to promote efficient policy learning. In contrast to other structured priors, programs stand out because of their strictly formatted and composable subroutines [5, 7]. Previous works leverage programs to enable a single agent to learn complex tasks by following programs [45, 52]. However, a plethora of new challenges have been introduced including task dependencies, collaboration schemes, etc; hence, it is difficult to trivially extend the existing works in the MARL settings. To enable

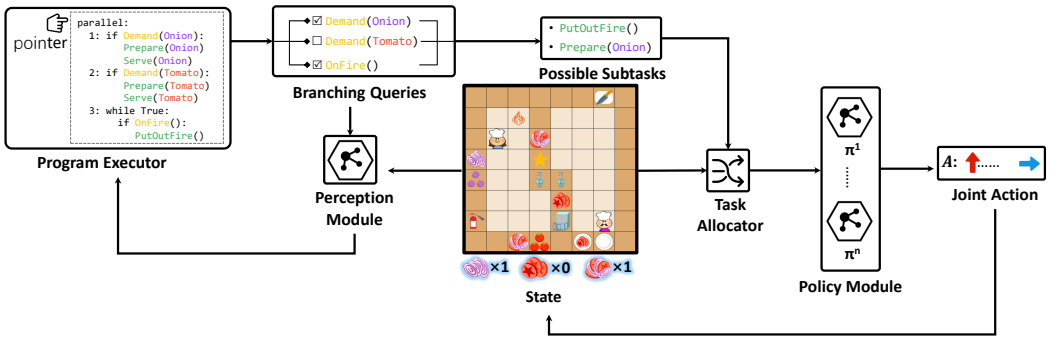


Figure 1: **The overall framework of E-MAPP.** E-MAPP includes four components: 1) A perception module that maps a query q and the current state s to boolean responses. 2) A program executor that maintains a pool of possible subtasks and updates them according to the perceptive results. 3) A task allocator that chooses proper subtasks from the subtask pool and assigns those to agents. 4) A policy module that instructs agents in taking actions to accomplish specific subtasks.

multi-thread orderless policy execution, we leverage "plug-and-play" auxiliary functions to infer the relationships among subtasks.

3 Problem Statement

3.1 Program Guided Cooperative Markov Game

An infinite-horizon Markov Game is defined by a tuple $(\mathcal{N}, \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}, \gamma)$, where $\mathcal{N} = \{1, \ldots, N\}$ denotes the set of N engaging agents, $\mathcal{S}$ denotes the state space, $\mathcal{A} = \mathcal{A}^1 \times \cdots \times \mathcal{A}^N$ denotes the Cartesian product of all the N agents' action space, $\mathcal{T}$: $\mathcal{S} \times \mathcal{A} \to \mathcal{S}$ denotes the state transition function from current state s to the next state s' for the joint action $a = (a^1, \ldots, a^N)$, $\mathcal{R} = \mathcal{R}^1 \times \cdots \times \mathcal{R}^N$ denotes the Cartesian product of all the N agents' reward functions, where each $\mathcal{R}^i$ determines the immediate reward for the i-th agent from the current state s and joint action a, and γ is the discount factor.

At each time t, each agent i observes the current state s_t, makes the decision a_t^i, and receives the reward r_t^i. In a cooperative game, the collective goal is to optimize the joint policy $\pi : \mathcal{S} \to A$ to maximize the sum of each agent's expected cumulative discounted rewards $\mathbb{E}_{\mathbf{a}_t^i \sim \pi^i(\cdot|\mathbf{s}_t), \mathbf{s}_t \sim \mathcal{P}} \left[\sum_{t=1}^{\infty} \sum_{i=1}^{N} \gamma^t r_t^i \left(\mathbf{s}_t, \mathbf{a}_t^i \right) \right]$.

A program-guided Markov game is a Markov game with a factorized state space. Specifically, state $\mathcal{S} = \mathcal{S}_e \times \mathcal{S}_p$, where $\mathcal{S}_e$ is the common state space of the environment and $\mathcal{S}_p$ is the multi-pointer program space (see Section 3.2). Accordingly, the state transition function takes as input the current compounded states (s_e, s_p) and joint actions a and then returns the next joint state (s'_e, s'_p). In this study, the transition function of the program space is based on predefined rules.

3.2 Parallel Programs

The program space consists of three components: a domain-specific language (DSL) that contains all the possible subroutines, a set of pointers that point to relevant subroutines, and a control flow that manages the pointers. A complete specification of the DSL used by our framework is in the Appendix A.1. Inside this DSL, a subroutine is a minimal executable unit in the program that corresponds to a subtask in the domain (e.g., *Chop(Tomato)*). As in previous works [45, 52], we consider two types of subroutines: perception primitives (e.g., *IsOnFire()*), which query about the status of the environment; and behavior primitives (e.g., *Chop(Tomato)*, which issue an instruction. Control flow involves branching statements (*if/else*), loops (*for/while*), and **parallelism indicators** (*repeat/parallel*). The parallelism indicators are designed for multi-thread execution. The subroutines in a *parallel* block are possible but not guaranteed to run concurrently, i.e., the agents should reason about what subroutines in a *parallel* block can be executed concurrently. The subroutine in a *repeat* block can be executed many times simultaneously. We summarize all the

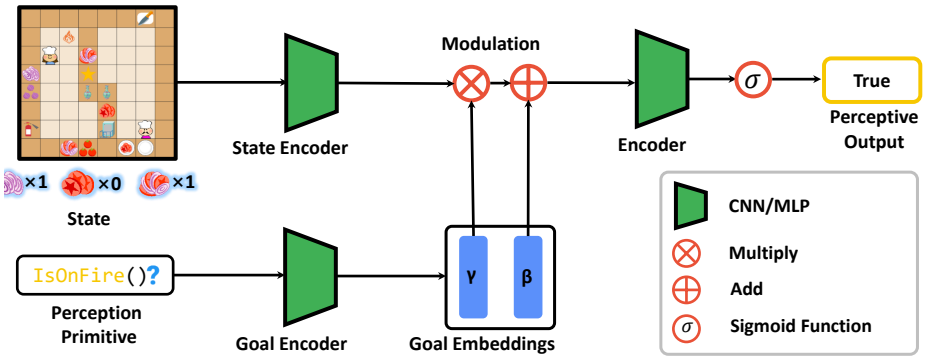


Figure 2: **The perception module**. The perception module encodes the perception primitive into modulation parameters, which operate on the state embeddings to get the goal-conditioned embeddings. These embeddings are then fed into another encoder, followed by a sigmoid function. The output is a real number in $[0, 1]$, and will be binarized for branching selection by the program executor.

current subroutines that are possible and executable as the *Possible Subroutine Set*. The set of pointers points to all the subroutines in this set.

3.3 Multi-Agent RL with Parallel Programs

In our study, we aim at developing a framework for optimizing the joint policy in a parallel program-guided cooperative Markov game. To this end, the agents must keep track of the pointers in the program, learn to reason the primitives to choose the right branches, and perform the action either collaboratively or individually to pursue high efficiency.

4 Method

Our goal is to enable multiple agents to cooperate to solve long-horizon tasks guided by parallel programs. There are three important factors we should consider: first, the agents should learn subtask-conditioned policies that can be composed together to accomplish long-horizon tasks and further generalize compositionally to unseen tasks; second, the agents should reason about task dependencies so that they can automatically parallelize tasks among them; and third, the agents should distinguish between cooperative tasks and non-cooperative tasks that can be achieved by a single agent to avoid competition for common resources.

We propose **E**fficient **M**ulti-**A**gent Reinforcement Learning with **P**arallel **P**rograms (E-MAPP), a multi-agent reinforcement learning framework where agents can cooperate to solve long-horizon tasks by following program guidance. As shown in Figure 1, E-MAPP includes four components: a perception module that is able to judge whether the queried status exists in the state; a parallel program executor that keeps track of subroutines and updates them based on the perception module output; a task allocator, which extracts queries of interest and obtains the corresponding set of possible subtasks from the program executor, and assigns tasks to agents; and a multi-agent policy module, which agents use to make their decisions based on the input subtask. The following of this section will introduce these four key components of E-MAPP. The complete algorithm is shown in Appendix A.3.

4.1 Parallel Program Executor

The program executor keeps a set of pointers pointing to possible subroutines in a domain-specific *Possible Subroutine Set*. There are four types of control flows: *if*-routine, *while*-routine, *parallel*-routine, and *repeat*-routine. Meanwhile, there are two types of subroutines in our program: behavior primitives and perception primitives.

Control flows. An *if*-routine contains a condition statement (usually a perception primitive) and subroutines in the corresponding blocks. A *while*-routine contains a condition statement and a looping block of subroutines. A *parallel*-routine contains parallel blocks of subroutines that are possible to be

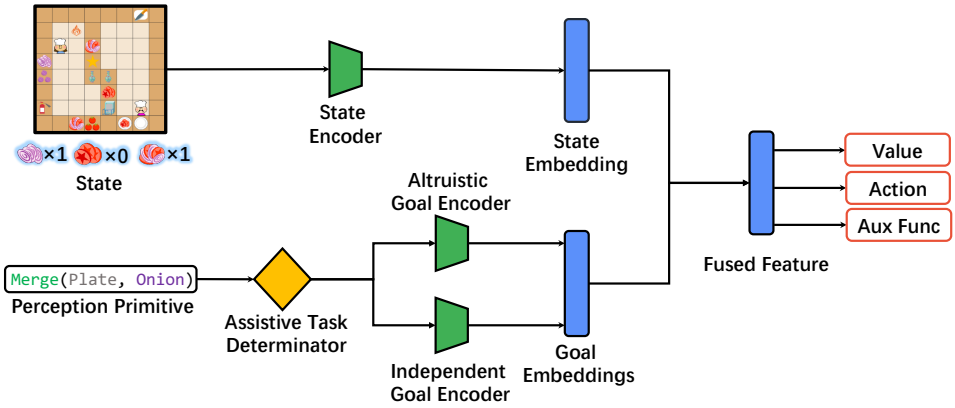


Figure 3: **The policy module**. We adopt a goal-conditioned reinforcement learning framework to determine the policy. We encode goals with a goal encoder and fuse them into the state features. If an agent is assistive, the goal is encoded with an altruistic goal encoder instead. The network has three outputs including the value function, the action distribution and the auxiliary functions respectively.

executed simultaneously. A *repeat*-routine contains an unconditioned block of subroutines that can be executed multiple times by different agents.

Subroutines. A behavior primitive (e.g., *Chop(Tomato)*) corresponds to a subtask that must be completed by the agents. As for a perception primitive (e.g., *IsOnFire()*), it is a query that requires a boolean response.

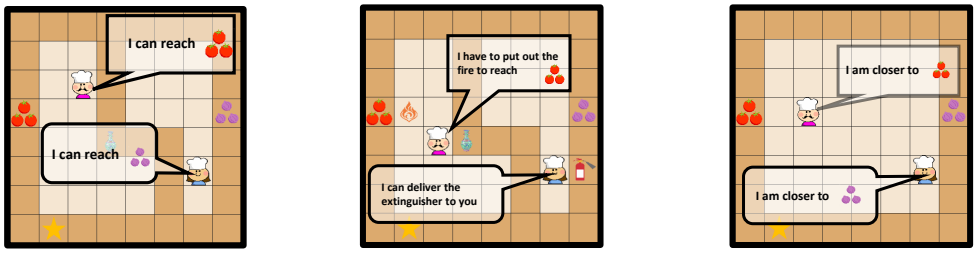
After an action is performed or a response to a perceptive query is received, the program executor updates its pointers and the Possible Subroutine Set. Detailed updating rules are shown in Appendix A.2. We note that if an achieved subtask does not correspond to any subroutine in the Possible Subroutine Set, the program executor will terminate the program immediately. This guarantees that no exceptions will occur due to violating the instructed order of subtasks.

4.2 Perception Module

The perception module learns to map a query q and the current state s_t to a boolean answer $h = \phi(q, s_t)$. For example, when a perception primitive *IsOnFire()* is passed to the perception module, it learns to check the existence of fire in the environment and returns true/false. The architecture is shown in Figure 2. Specifically, we randomly sample states, queries, and the ground-truth perception h_{gt} as the training dataset and train the network ϕ in a supervised manner. We use binary cross entropy (BCE) loss $\mathcal{L}_{\text{perception}} = BCELoss(h_{\text{pred}}, h_{\text{gt}})$ as the objective, where h_{pred} denotes the perception output. In terms of architecture, we encode the perception primitive s_p and the common state s_e with a neural encoder. The encoded primitives are represented as γ and β to modulate the encoded common state. At the last layer, we use a sigmoid function to obtain binary output. Training details and detailed architecture descriptions can be found in Appendix A.5. In this way, this module can determine whether the queried primitive exists in the state to aid the agents' decision-making.

4.3 Policy Module

The policy module grounds agents' actions with the programs and encourages cooperation for completing the tasks. The overall policy learning procedure advocates a subtask-conditioned reinforcement learning framework as shown in Figure 3. Our algorithm backbone is MAPPO [51] and consists of two stages. More concretely, we firstly learn a policy for each agent where other agents' policies are fixed. The input is the state and the encoded subtask. The reward signal is based on whether a subroutine is executed correctly. Then we enable multiple agents to coordinate by learning a joint policy for accomplishing collective goals cooperatively and efficiently, with the help of auxiliary functions and a task allocator. We use the same architecture to fuse state and goal features as that in the perception module in Figure 2. We also leverage self-imitation learning to tackle the challenge of sparse rewards, which is shown in Appendix A.4



(a) reachability (b) feasibility (c) cost-to-go

Figure 4: **The auxiliary functions**. Figure 4a shows the reachability that indicates if an agent can reach an object. Figure 4b shows the feasibility that indicates whether a subtask can be achieved by the agents collectively. Figure 4c shows the cost-to-go function that indicates time consumption.

Learning to cooperate on a subtask. After obtaining single agent policies, we then encourage agents to accomplish harder tasks that require cooperation. When assigned a cooperative subtask, one of the agents is in charge of finalizing the task. The rest of the agents are assistive. For example, the assistive agents might pass an onion to the leading agents to chop. In practice, we randomly appoint one agent as the leading agent and the other ones as the assistive agents. The leading agent is rewarded if this subtask is completed. The reward function for the assistive agents is calculated based on the *reachability* improvement of the leading agent. We defer the formal definition of this reward function to Section 4.4. Such reward function encourages the assistive agents to help the leading agents to obtain rewards. Then, we use a MAPPO-style algorithm to obtain cooperation policy. We also encode the leading agent's goal into the observation space of the assistive agents to enhance information sharing.

4.4 Task Allocator

The task allocator assigns subtasks to each agent to accomplish a long-horizon task together. Specifically, we design the task allocator for efficient coordination based on the following principles: 1) It only assigns subtasks that are feasible for the agent(s) without additional prerequisite subtasks. 2) If a subtask is cooperative, the task allocator would assign the subtask to a number of agents. 3) It assigns a subtask to the agent that has the lowest cost in terms of execution time.

To facilitate the task allocation, we propose to learn three auxiliary functions: a *reachability function*, a *feasibility function*, and a *cost-to-go function* as illustrated in Figure 4. Training details of the auxiliary functions are in Appendix A.6

Reachability. The reachability function $f_{reach} = f_{reach}(s_t, i, \tau)$ is defined as a boolean value to indicate if agent i can complete task τ alone at the state s_t. For a cooperative subtask τ and a selected leading agent i, an extrinsic reward $f_{reach}(s_t^{'}, i, \tau) - f_{reach}(s_t, i, \tau)$ is provided to the assistive agents for learning altruistic behaviors. To train this reachability function, we randomly sample triplets (s_t, i, τ) and obtain the ground-truth value f_{reach}^{gt} through running the pre-trained non-cooperative policy. Then we optimize the network by minimizing the binary cross entropy BCE$(f_{reach}^{pred}, f_{reach}^{gt})$.

Feasibility. The feasibility function $f_{feas} = f_{feas}(s_t, i, \tau)$ is defined as a boolean variable indicating if an agent i can complete a subtask τ with others' assistance at the state s_t. The training procedure of the feasibility function is similar to that of the reachability function, except that we leverage the cooperative policy instead of the single agent policy to collect training data.

Cost-to-go. The cost-to-go function $f_{cost} = f_{cost\text{-}to\text{-}go}(s_t, i, \tau)$ denotes the remaining timesteps for agent i to accomplish the subtask from the state s_t. We leverage trajectories generated by a pre-trained intermediate cooperative policy from E-MAPP to train this cost function. Specifically, we randomly sample triplets (s_t, i, τ) and execute the pre-trained cooperative policy in Section 4.3

to obtain the ground-truth timesteps $f^{gt}_{cost\text{-}to\text{-}go}$ to complete the subtask τ. Then we optimize the network by reducing the Mean Squared Error (MSE) $\mathcal{L}_{cost\text{-}to\text{-}go} = \text{MSE}(f^{pred}_{cost\text{-}to\text{-}go}, f^{gt}_{cost\text{-}to\text{-}go})$.

Criteria for subtask allocation. For a specific Possible Subroutine Set $\{\tau_1, \ldots, \tau_m\}$, we denote a legal subtask allocation as $\{T_1, T_2, \ldots, T_m\}$ such that

- T_i is a list of n_i agents where $n_i = |T_i|$. The agents are those in the agent set $\{1, \ldots, N\}$ who aim at completing the subtask τ_i.
- For all $i \in \{1, \ldots, m\}$, if n_i is larger than one, then the agent T_i^1 is selected as the leading agent and the agent(s) $T_i^2, \ldots, T_i^{n_i}$ are selected as the assistive agent(s).
- For $1 \leq i < j \leq m$, we have that $T_i \cap T_j = \emptyset$, which means no agent is assigned two subtasks simultaneously.

We compute the cost of each possible subtask allocation $\{T_1, T_2, \ldots, T_m\}$ as the sum of three terms

$$c_{\text{total}} = c_{\text{feas}} + c_{\text{cost-to-go}} + c_{\text{reach}} \tag{1}$$

,where

$$c_{\text{feas}} = \sum_{i=1}^{m} -n_i w_{\text{feas}} \log f_{\text{feas}}(s, T_i[1], \tau_i) \tag{2}$$

$$c_{\text{cost-to-go}} = \sum_{i=1}^{m} -n_i w_{\text{cost}} f_{\text{cost}}(s, T_i[1], \tau_i) \tag{3}$$

$$c_{\text{reach}} = \sum_{i=1}^{m} \mathbb{I}[n_i = 1] w_{\text{reach}} \log f_{\text{reach}}(s, T_i[1], \tau_i) \tag{4}$$

w_{feas}, w_{reach}, and w_{cost} are tunable hyperparameters and $\mathbb{I}[\cdot]$ is the indicator function. The *feasibility* term encourages the agents to choose the feasible subtasks, the *cost-to-go* term encourages the agents to choose the less costly subtasks, and the *reachability* term guarantees that the subtask assigned to only one agent is non-cooperative. The log operator operating on a value approximating to 1 will induce a huge cost, thus preventing the allocation of an infeasible or unreachable subtask to an agent. We search from the possible allocations $\{T_1, T_2, \ldots, T_m\}$ and apply the one with minimal cost as final allocation.

In practice, we also use two hyperparameters: c_r to be subtracted from the cost-to-go function to encourage the agents to finish their ongoing subtasks, and c_i to be added to the cost-to-go function to avoid allocating subtasks to an agent who can never accomplish a task within a timeout threshold t_o.

4.5 Complexity Analysis

In an environment with M subtasks and N agents, the brute force search for an optimal allocation indeed has a complexity of $O(M^N)$. However, the practical complexity is much smaller than it. The reasons are as follows:

1. In a certain stage of a long-horizon task, only a small amount of subtasks are feasible. Thus, the subtask amount M can be pruned into a smaller number L by checking the feasibility function $O(M \times N)$ times and ignoring the subtasks whose feasibility functions are less than a given threshold.
2. The engaging N agents can be classified into C roles. The agents sharing the same role have the same reachability functions. C is often a property of the task that does not scale with N. For example, in the overcooked environment, C can be the number of connected components of the map. Note that, in E-MAPP, the assistive agents aim to increase the reachability of the leading agents. We define $C \times L$ new subtasks (τ, c), where τ comes from the L feasible subtasks and c comes from the C roles. The goal of each new subtask (τ, c) is to help agents with role c to gain reachability on subtask τ. We can obtain a new subtask set of size $O(C \times L)$ by extending the original subtask set with these newly defined subtasks. Assume that the number of agents is smaller than the number of feasible subtasks (otherwise, idle agents will inevitably emerge). Under this assumption, each agent will choose to either complete a subtask alone or assist a certain group of agents with the same role, and each subtask in the new subtask set is allocated to at most one agent to avoid

conflict. Then the task allocation problem turns into finding the best matching of N agents and $O(C \times L)$ subtasks with the smallest total cost, which can be solved by the Hungarian algorithm. The computational complexity is $O((N + CL)^3) \leq O((N + CM)^3)$ that scales well.

5 Experiments

In this section, we aim to investigate the following key questions. First, is the program guidance helpful for agents to understand long-horizon tasks in comparison with other structured information guidance? Second, are the parallel structures in the programs bring forth cooperative behaviors among the agents? Third, does the task allocator improve the completion rates and the time efficiency of the long-horizon tasks by virtue of the auxiliary functions?

5.1 Environment Description

To evaluate the proposed framework, we adapted the previous environment [50]mimicking the video game to a more challenging one "Overcooked v2". Concretely, we extend temporally for the horizon length of a task by adding additional behaviors such as "wash dishes" and "put out fire". As shown in Figure 5, agents can navigate through the grid world, interact with objects (e.g., tomatoes, knives), and deliver the dishes to the customers (the yellow star). The goal of the agents is to serve dishes according to the given recipes that can be divided into subtasks. More details about the environment can be found in Appendix A.7.

Figure 5: A sample from the *Overcooked* environment.

5.2 Setup

Tasks. We test the agents on a variety of tasks with different difficulty levels. The **easy** tasks contain only one subroutine for verification. The **medium** tasks contain two or three subroutines. The **hard** task requires the agents to complete multiple dishes that require at least two subroutines each while putting out the randomly appearing fire. We conduct evaluation in two different patterns, based on seen or unseen tasks. Note that both evaluation scheme are conducted on novel maps, which makes the task more challenging. The unseen tasks share the subroutines with the seen tasks but is never used as a goal during training. The unseen tasks are used to test whether the methods can generalize compositionally. We also note that for the tasks with a *repeat* command (e.g., repeatedly pick an onion from the supply), a targeted repeat number is preset.

Metrics. We use the completion rates and the average scores as the metrics. The completion rate is defined as the percentage of tasks that can be completed in an episode. The average score is the discounted cumulative rewards in an episode across all testing tasks. Agents will receive a reward of 1 when the final goal is achieved and receive a reward of 0.2 when a correct subtask is completed. Each algorithm is tested in 1000 environments and we report their average in the tables.

Baselines. We compare our model with two baselines: MAPPO [51] and a natural language-guided agent. Detailed descriptions are in Appendix A.9.

5.3 Results

Results with seen tasks in novel maps. Table 1 shows the completion rate and average scores on seen tasks in novel maps, demonstrating that E-MAPP excels at understanding the structure of complex tasks. As expected, the end-to-end MAPPO model performs well on short-horizon tasks, but suffers from a significant performance drop when the task becomes complex. The natural language–guided model also has a large performance drop when the horizon of the tasks becomes longer. We attribute this to that the agents fail to understand the complex task structure described in natural language without explicit structure. By contrast, E-MAPP performs well even when the tasks have a long horizon with various accidental events happening.

Table 1: **Results on the seen tasks in novel maps.** Completion rates and average scores of seen tasks in the *Overcooked* environment with three difficulty levels.

	Completion Rates			Average Scores		
Methods	Easy	Medium	Hard	Easy	Medium	Hard
E-MAPP (ours)	98.0%	**97.5**%	**56.3**%	1.06±0.17	**1.12**±0.22	**1.58**±0.60
Natural language guidance	81.9%	48.1%	1.01%	0.87±0.43	0.63±0.51	0.82 ± 0.31
MAPPO [51]	**100.0**%	65.7%	0.00%	**1.11**±0.04	0.79±0.31	0.59± 0.27

Generalization with unseen tasks in novel maps. Table 2 shows the results on the zero-shot generalization scheme. The scenarios and tasks are very different from the training domain, posing extra challenge to all the algorithms including the auxiliary functions of E-MAPP. We find that E-MAPP is significantly better than both the previous multi-agent RL algorithms and a language-guided agent. We attribute the success of E-MAPP to the fact that programs have better compositionality and less ambiguity. In the training time, E-MAPP learns certain subroutines from other tasks; it applies the behavior compositionally to achieve goals that is never seen during training. More visualized results are in Appendix A.8.

Table 2: **Results on the unseen tasks in novel maps.** Completion rates and average scores of unseen tasks with two difficulty levels.

	Completion Rates		Average Scores	
Methods	Medium	Hard	Medium	Hard
E-MAPP (ours)	**100.0%**	**43.7%**	**1.13** ± 0.10	**0.99** ± 0.22
Natural language–guided model	58.4%	0.0%	0.58 ± 0.50	0.48 ± 0.21

Ablation of auxiliary functions. We analyze the significance of the proposed key components in our model by comparing our model with the variants that removes 1) the feasibility predictor 2) the reachability predictor 3) the cost-to-go predictor. The completion rates and the average scores are shown in Table 3. We find that all the components are important to E-MAPP. Specifically, we find that removing the reachability predictor from E-MAPP leads to a significantly increased average timesteps, and this is the component that provides the largest performance gain. We attribute this to that the reachability function pointed out the subtasks that can be completed individually, providing an impetus for agents to parallelize.

Ablation of parallelism in the programs. We compare the proposed parallel programs with sequential programs in the same environments. In Table 3, we find that the agents with parallel programs use 15% fewer time steps to accomplish the same goal as those with sequential programs. This results show that the parallel programs are central to E-MAPP in terms of time efficiency.

Table 3: **Ablation study.** Completion rates, average scores and timesteps when one of the key components of E-MAPP is removed.

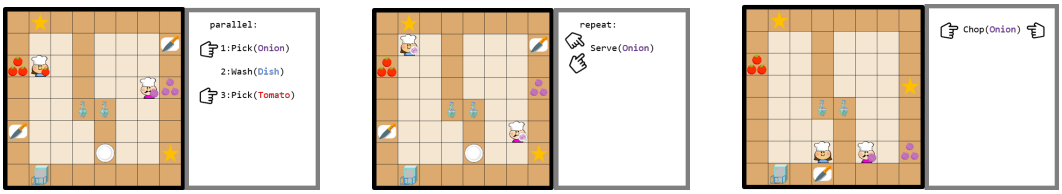
	Completion Rates ↑	Average Scores ↑	Average Timesteps ↓
E-MAPP	**56.3%**	**1.58** ± 0.60	**17.6**
w/o feasibility predictor	38.5%	1.42 ± 0.56	21.24
w/o reachability predictor	43.8%	1.45 ± 0.50	23.37
w/o cost-to-go predictor	52.0%	1.46 ± 0.45	20.42
sequential program	48.8%	1.51 ± 0.47	20.33

Partially observable environments. We conduct an experiment to show that E-MAPP can still outperform other methods in a partially observable environment. In the new setting, the observation of each agent is only the part of the map within reach. Table 4 shows the results. Under the new setting, E-MAPP can still learn to allocate sub-tasks to agents and accomplish the tasks efficiently.

Table 4: **Additional experiment in partially observable environments** we conduct an experiment to show that E-MAPP can still outperform other methods in a partially observable environment.

model	score	completion rate
E-Mapp(partial obs)	1.01±0.38	27.1%
E-MAPP(original)	1.58±0.60	56.3%
MAPPO	0.59± 0.27	0.0%

5.4 Visualization of Learned Behaviors



(a) parallel (b) repeat (c) assist

Figure 6: **Learned behaviors of E-MAPP**. Figure 6a shows that two agents are assigned two subtasks concurrently. Figure 6b shows that both of the agents are trying to perform a repeated task. Figure 6c shows that one agent is passing an onion to the agent who can chop it.

In Figure 6, we show the representative learned behaviors of E-MAPP. In Figure 6(a), the two subroutines are allocated to two agents separately. This demonstrates the effectiveness of the *feasibility* function, with which the task allocator successfully identified the parallelizable subtasks that can be executed concurrently. We note that the *parallel* indicator in the program only suggests the agent trying to identify what subtasks are parallelizable rather than providing a ground-truth task structure. In Figure 6(b), we find that the two agents are performing the same task in a *repeat* subroutine independently. We show the allocator can assign tasks to multiple agents that are reachable to the goal. In Figure 6(c), we show that an assistive agent is passing an onion to the leading agent to chop. This shows how the agents learn to cooperate with each other. More visualizations can be found in Appendix A.8.

5.5 Scalability to New Domains

We also investigate whether E-MAPP can be effective on tasks with continuous action space based on the *Stacking* environment. More details and demos in *Stacking* can be found in Appendix A.10. We discuss some potential parallel-program synthesis approaches when applying E-MAPP to a new domain in Appendix A.14

6 Conclusion

In this paper, we first formulate the problem of program guided multi-agent tasks. We propose Efficient Multi-Agent Reinforcement Learning with Parallel Programs (E-MAPP), an effective framework that uses a type of parallelism-aware program for multi-agents' collaboration and a task allocation strategy via a set of learnable auxiliary functions. The results show that our algorithm can infer the task structure and significantly boost completion rates, efficiency, and generalization ability on long-horizon tasks.

Limitations. Our current framework does not consider more challenging scenarios such as dynamics scenes or generalization to novel objects. We believe that this work opens a welcoming avenue to this research direction and more future works will address additional challenges.

7 Acknowledgement

We thank Yuping Luo and Zhecheng Yuan for their careful proofreading and writing suggestions.

References

[1] Christopher Agia, Krishna Murthy Jatavallabhula, Mohamed Khodeir, Ondrej Miksik, Vibhav Vineet, Mustafa Mukadam, Liam Paull, and Florian Shkurti. Taskography: Evaluating robot task planning over large 3d scene graphs. In Aleksandra Faust, David Hsu, and Gerhard Neumann, editors, *Proceedings of the 5th Conference on Robot Learning*, volume 164 of *Proceedings of Machine Learning Research*, pages 46–58. PMLR, 08–11 Nov 2022.

[2] Michael Ahn, Anthony Brohan, Noah Brown, Yevgen Chebotar, Omar Cortes, Byron David, Chelsea Finn, Keerthana Gopalakrishnan, Karol Hausman, Alex Herzog, Daniel Ho, Jasmine Hsu, Julian Ibarz, Brian Ichter, Alex Irpan, Eric Jang, Rosario Jauregui Ruano, Kyle Jeffrey, Sally Jesmonth, Nikhil Joshi, Ryan Julian, Dmitry Kalashnikov, Yuheng Kuang, Kuang-Huei Lee, Sergey Levine, Yao Lu, Linda Luu, Carolina Parada, Peter Pastor, Jornell Quiambao, Kanishka Rao, Jarek Rettinghouse, Diego Reyes, Pierre Sermanet, Nicolas Sievers, Clayton Tan, Alexander Toshev, Vincent Vanhoucke, Fei Xia, Ted Xiao, Peng Xu, Sichun Xu, and Mengyuan Yan. Do as i can and not as i say: Grounding language in robotic affordances. In *arXiv preprint arXiv:2204.01691*, 2022.

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

[4] Kai Arulkumaran, Antoine Cully, and Julian Togelius. Alphastar: An evolutionary computation perspective. In *Proceedings of the genetic and evolutionary computation conference companion*, pages 314–315, 2019.

[5] Rudy Bunel, Matthew Hausknecht, Jacob Devlin, Rishabh Singh, and Pushmeet Kohli. Leveraging grammar and reinforcement learning for neural program synthesis. *arXiv preprint arXiv:1805.04276*, 2018.

[6] David Chen and Raymond Mooney. Learning to interpret natural language navigation instructions from observations. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 25, pages 859–865, 2011.

[7] Xinyun Chen, Chang Liu, and Dawn Song. Execution-guided neural program synthesis. In *International Conference on Learning Representations*, 2018.

[8] Xinyun Chen, Dawn Song, and Yuandong Tian. Latent execution for neural program synthesis, 2021.

[9] Todor Davchev, Oleg Sushkov, Jean-Baptiste Regli, Stefan Schaal, Yusuf Aytar, Markus Wulfmeier, and Jon Scholz. Wish you were here: Hindsight goal selection for long-horizon dexterous manipulation. In *Deep RL Workshop NeurIPS 2021*, 2021.

[10] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[11] Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. Robustfill: Neural program learning under noisy i/o. In *International conference on machine learning*, pages 990–998. PMLR, 2017.

[12] Ziluo Ding, Tiejun Huang, and Zongqing Lu. Learning individually inferred communication for multi-agent cooperation. *Advances in Neural Information Processing Systems*, 33:22069–22079, 2020.

[13] Zach Dwiel, Madhavun Candadai, Mariano Phielipp, and Arjun K Bansal. Hierarchical policy learning is sensitive to goal space design. *arXiv preprint arXiv:1905.01537*, 2019.

[14] Scott Emmons, Ajay Jain, Misha Laskin, Thanard Kurutach, Pieter Abbeel, and Deepak Pathak. Sparse graphical memory for robust planning. *Advances in Neural Information Processing Systems*, 33:5251–5262, 2020.

[15] Ben Eysenbach, Russ R Salakhutdinov, and Sergey Levine. Search on the replay buffer: Bridging planning and reinforcement learning. *Advances in Neural Information Processing Systems*, 32, 2019.

[16] Abhishek Gupta, Vikash Kumar, Corey Lynch, Sergey Levine, and Karol Hausman. Relay policy learning: Solving long-horizon tasks via imitation and reinforcement learning. In *Conference on Robot Learning*, pages 1025–1037. PMLR, 2020.

[17] Jonathan Ho and Stefano Ermon. Generative adversarial imitation learning. *Advances in neural information processing systems*, 29, 2016.

[18] Unnat Jain, Luca Weihs, Eric Kolve, Ali Farhadi, Svetlana Lazebnik, Aniruddha Kembhavi, and Alexander Schwing. A cordial sync: Going beyond marginal policies for multi-agent embodied tasks. In *European Conference on Computer Vision*, pages 471–490. Springer, 2020.

[19] Natasha Jaques, Angeliki Lazaridou, Edward Hughes, Caglar Gulcehre, Pedro Ortega, DJ Strouse, Joel Z Leibo, and Nando De Freitas. Social influence as intrinsic motivation for multi-agent deep reinforcement learning. In *International Conference on Machine Learning*, pages 3040–3049. PMLR, 2019.

[20] Christian Jestel, Harmtmut Surmann, Jonas Stenzel, Oliver Urbann, and Marius Brehler. Obtaining robust control and navigation policies for multi-robot navigation via deep reinforcement learning. In *2021 7th International Conference on Automation, Robotics and Applications (ICARA)*, pages 48–54. IEEE, 2021.

[21] Jiechuan Jiang and Zongqing Lu. Learning attentional communication for multi-agent cooperation. *Advances in neural information processing systems*, 31, 2018.

[22] Kishor Jothimurugan, Suguman Bansal, Osbert Bastani, and Rajeev Alur. Compositional reinforcement learning from logical specifications. *Advances in Neural Information Processing Systems*, 34, 2021.

[23] Russell Kaplan, Christopher Sauer, and Alexander Sosa. Beating atari with natural language guided reinforcement learning. *arXiv preprint arXiv:1704.05539*, 2017.

[24] Yunfei Li, Yilin Wu, Huazhe Xu, Xiaolong Wang, and Yi Wu. Solving compositional reinforcement learning problems via task reduction. In *International Conference on Learning Representations*, 2020.

[25] Yunchao Liu and Zheng Wu. Learning to describe scenes with programs. In *International conference on learning representations*, 2019.

[26] Ryan Lowe, Yi I Wu, Aviv Tamar, Jean Harb, OpenAI Pieter Abbeel, and Igor Mordatch. Multi-agent actor-critic for mixed cooperative-competitive environments. *Advances in neural information processing systems*, 30, 2017.

[27] Rajbala Makar, Sridhar Mahadevan, and Mohammad Ghavamzadeh. Hierarchical multi-agent reinforcement learning. In *Proceedings of the fifth international conference on Autonomous agents*, pages 246–253, 2001.

[28] Shie Mannor, Ishai Menache, Amit Hoze, and Uri Klein. Dynamic abstraction in reinforcement learning via clustering. In *Proceedings of the twenty-first international conference on Machine learning*, page 71, 2004.

[29] Ofir Nachum, Shixiang (Shane) Gu, Honglak Lee, and Sergey Levine. Data-efficient hierarchical reinforcement learning. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018.

[30] Soroush Nasiriany, Huihan Liu, and Yuke Zhu. Augmenting reinforcement learning with behavior primitives for diverse manipulation tasks. In *IEEE International Conference on Robotics and Automation (ICRA)*, 2022.

[31] Junhyuk Oh, Yijie Guo, Satinder Singh, and Honglak Lee. Self-imitation learning. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 3878–3887. PMLR, 10–15 Jul 2018.

[32] Junhyuk Oh, Yijie Guo, Satinder Singh, and Honglak Lee. Self-imitation learning. In *International Conference on Machine Learning*, pages 3878–3887. PMLR, 2018.

[33] Afshin Oroojlooyjadid and Davood Hajinezhad. A review of cooperative multi-agent deep reinforcement learning. *CoRR*, abs/1908.03963, 2019.

[34] Peng Peng, Ying Wen, Yaodong Yang, Quan Yuan, Zhenkun Tang, Haitao Long, and Jun Wang. Multiagent bidirectionally-coordinated nets: Emergence of human-level coordination in learning to play starcraft combat games. *arXiv preprint arXiv:1703.10069*, 2017.

[35] Silviu Pitis, Harris Chan, Stephen Zhao, Bradly Stadie, and Jimmy Ba. Maximum entropy gain exploration for long horizon multi-goal reinforcement learning. In *International Conference on Machine Learning*, pages 7750–7761. PMLR, 2020.

[36] Matthias Plappert, Marcin Andrychowicz, Alex Ray, Bob Mcgrew, Bowen Baker, Glenn Powell, Jonas Schneider, Josh Tobin, Maciek Chociej, Peter Welinder, et al. Multi-goal reinforcement learning: Challenging robotics environments and request for research. 2018. *URL https://arxiv.org/pdf/1802.09464. pdf*, 2018.

[37] Tabish Rashid, Gregory Farquhar, Bei Peng, and Shimon Whiteson. Weighted qmix: Expanding monotonic value function factorisation for deep multi-agent reinforcement learning. *Advances in neural information processing systems*, 33:10199–10210, 2020.

[38] Tabish Rashid, Mikayel Samvelyan, Christian Schroeder, Gregory Farquhar, Jakob Foerster, and Shimon Whiteson. Qmix: Monotonic value function factorisation for deep multi-agent reinforcement learning. In *International Conference on Machine Learning*, pages 4295–4304. PMLR, 2018.

[39] Mikayel Samvelyan, Tabish Rashid, Christian Schroeder De Witt, Gregory Farquhar, Nantas Nardelli, Tim GJ Rudner, Chia-Man Hung, Philip HS Torr, Jakob Foerster, and Shimon Whiteson. The starcraft multi-agent challenge. *arXiv preprint arXiv:1902.04043*, 2019.

[40] Nikolay Savinov, Alexey Dosovitskiy, and Vladlen Koltun. Semi-parametric topological memory for navigation. In *International Conference on Learning Representations*, 2018.

[41] Samaneh Hosseini Semnani, Hugh Liu, Michael Everett, Anton De Ruiter, and Jonathan P How. Multi-agent motion planning for dense and dynamic environments via deep reinforcement learning. *IEEE Robotics and Automation Letters*, 5(2):3221–3226, 2020.

[42] Satinder Singh, Richard L Lewis, Andrew G Barto, and Jonathan Sorg. Intrinsically motivated reinforcement learning: An evolutionary perspective. *IEEE Transactions on Autonomous Mental Development*, 2(2):70–82, 2010.

[43] Martin Stolle and Doina Precup. Learning options in reinforcement learning. In *International Symposium on abstraction, reformulation, and approximation*, pages 212–223. Springer, 2002.

[44] Shao-Hua Sun, Hyeonwoo Noh, Sriram Somasundaram, and Joseph Lim. Neural program synthesis from diverse demonstration videos. In *International Conference on Machine Learning*, pages 4790–4799. PMLR, 2018.

[45] Shao-Hua Sun, Te-Lin Wu, and Joseph J Lim. Program guided agent. In *International Conference on Learning Representations*, 2019.

[46] Peter Sunehag, Guy Lever, Audrunas Gruslys, Wojciech Marian Czarnecki, Vinícius Flores Zambaldi, Max Jaderberg, Marc Lanctot, Nicolas Sonnerat, Joel Z Leibo, Karl Tuyls, et al. Value-decomposition networks for cooperative multi-agent learning based on team reward. In *AAMAS*, 2018.

[47] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018.

[48] Eugene Vinitsky, Aboudy Kreidieh, Luc Le Flem, Nishant Kheterpal, Kathy Jang, Cathy Wu, Fangyu Wu, Richard Liaw, Eric Liang, and Alexandre M Bayen. Benchmarks for reinforcement learning in mixed-autonomy traffic. In *Conference on robot learning*, pages 399–409. PMLR, 2018.

[49] Ruosong Wang, Simon S Du, Lin F Yang, and Sham M Kakade. Is long horizon reinforcement learning more difficult than short horizon reinforcement learning? *arXiv preprint arXiv:2005.00527*, 2020.

[50] Sarah A Wu, Rose E Wang, James A Evans, Joshua B Tenenbaum, David C Parkes, and Max Kleiman-Weiner. Too many cooks: Bayesian inference for coordinating multi-agent collaboration. *Topics in Cognitive Science*, 13(2):414–432, 2021.

[51] Chao Yu, Akash Velu, Eugene Vinitsky, Yu Wang, Alexandre Bayen, and Yi Wu. The surprising effectiveness of ppo in cooperative, multi-agent games. *arXiv preprint arXiv:2103.01955*, 2021.

[52] Zelin Zhao, Karan Samel, Binghong Chen, et al. Proto: Program-guided transformer for program-guided tasks. *Advances in Neural Information Processing Systems*, 34, 2021.

[53] Ming Zhou, Jun Luo, Julian Villella, Yaodong Yang, David Rusu, Jiayu Miao, Weinan Zhang, Montgomery Alban, Iman Fadakar, Zheng Chen, et al. Smarts: Scalable multi-agent reinforcement learning training school for autonomous driving. *arXiv preprint arXiv:2010.09776*, 2020.

[54] Ming Zhou, Jun Luo, Julian Villella, Yaodong Yang, David Rusu, Jiayu Miao, Weinan Zhang, Montgomery Alban, IMAN FADAKAR, Zheng Chen, Chongxi Huang, Ying Wen, Kimia Hassanzadeh, Daniel Graves, Zhengbang Zhu, Yihan Ni, Nhat Nguyen, Mohamed Elsayed, Haitham Ammar, Alexander Cowen-Rivers, Sanjeevan Ahilan, Zheng Tian, Daniel Palenicek, Kasra Rezaee, Peyman Yadmellat, Kun Shao, dong chen, Baokuan Zhang, Hongbo Zhang, Jianye Hao, Wulong Liu, and Jun Wang. Smarts: An open-source scalable multi-agent rl training school for autonomous driving. In Jens Kober, Fabio Ramos, and Claire Tomlin, editors, *Proceedings of the 2020 Conference on Robot Learning*, volume 155 of *Proceedings of Machine Learning Research*, pages 264–285. PMLR, 16–18 Nov 2021.

Checklist

1. For all authors...
   (a) Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope? [Yes]
   (b) Did you describe the limitations of your work? [Yes]
   (c) Did you discuss any potential negative societal impacts of your work? [N/A] Our work focus on designing an algorithm for boosting the performance and generalization ability, rather than a specific real-world application.
   (d) Have you read the ethics review guidelines and ensured that your paper conforms to them? [Yes]
2. If you are including theoretical results...
   (a) Did you state the full set of assumptions of all theoretical results? [N/A]
   (b) Did you include complete proofs of all theoretical results? [N/A]
3. If you ran experiments...
   (a) Did you include the code, data, and instructions needed to reproduce the main experimental results (either in the supplemental material or as a URL)? [Yes] In the supplementary material
   (b) Did you specify all the training details (e.g., data splits, hyperparameters, how they were chosen)? [Yes] In the supplementary material
   (c) Did you report error bars (e.g., with respect to the random seed after running experiments multiple times)? [Yes]
   (d) Did you include the total amount of compute and the type of resources used (e.g., type of GPUs, internal cluster, or cloud provider)? [Yes] In the supplementary material
4. If you are using existing assets (e.g., code, data, models) or curating/releasing new assets...
   (a) If your work uses existing assets, did you cite the creators? [Yes]
   (b) Did you mention the license of the assets? [Yes]
   (c) Did you include any new assets either in the supplemental material or as a URL? [Yes]
   (d) Did you discuss whether and how consent was obtained from people whose data you're using/curating? [N/A]
   (e) Did you discuss whether the data you are using/curating contains personally identifiable information or offensive content? [N/A]
5. If you used crowdsourcing or conducted research with human subjects...
   (a) Did you include the full text of instructions given to participants and screenshots, if applicable? [N/A]
   (b) Did you describe any potential participant risks, with links to Institutional Review Board (IRB) approvals, if applicable? [N/A]
   (c) Did you include the estimated hourly wage paid to participants and the total amount spent on participant compensation? [N/A]

A Appendix

The project website can be viewed at https://sites.google.com/view/e-mapp.

A.1 Domain Specific Language (DSL) Specifications

Table 5 shows the domain-specific language (DSL) designed for E-MAPP in the *Overcooked* environment.

Table 5: **DSL for the Overcooked Environment.** We list types of constituent elements of the DSL. We provide the context-free grammar for all the legal programs.

Type	Instances
Program p	`def main():s`
Item t	`FreshOnion` | `FreshTomato` | `Plate` | `ChoppedOnion` | `ChoppedTomato` | `ChoppedOnion+Plate` | `ChoppedTomato+Plate` | `ChoppedOnion+ChoppedTomato` | `ChoppedOnion+ChoppedTomato+Plate`
Behavior b	`Chop(t)` | `Pick(t)` | `Merge(t,t)` | `Serve(t)` | `WashDirtyPlate()` `PutOutFire()`
Conditions c	`h` | `tautology` | `contradictory`
Statement s	`While(c):(s)` | `Parallel:(s_1, s_2, ···)` | `b` | `Repeat(s,i)` | `If(c):(s)` | `else:(s)` |
Perception h	`is_ordered[t]` | `is_there[t]`

A.2 Multi-Pointer Updater

Algorithm 1 shows the execution rules of parallel programs. The program executor maintains a set of pointers $\mathbb{P}$ pointing to different subroutines. After the agents take the actions, we update each pointer p in $\mathbb{P}$ according to the type of the subroutine that p was pointing to.

Algorithm 1 Multi-Pointer Updater

input : a set of pointers $\mathbb{P}$, a required repetition number M
for *each pointer p in $\mathbb{P}$* **do**
  if *p points to an* if-*routine* **then**
    Point p to the routine inside/outside the block of primitives in the if-routine if the response to its condition is True/False.
  end
  else if *p points to an* while-*routine* **then**
    Point p to the routine inside/outside the block of primitives in the while-routine if the response to its condition is True/False.
  end
  else if *p points to a* parallel-*routine* **then**
    Split p into multiple pointers, each pointing to a block of subroutines in the parallel-routine.
  end
  else if *p points to a* repeat-*routine* **then**
    Split p into M pointers, each pointing to a copy of the block of subroutines.
  end
  else if *p points to a behavior primitive and the corresponding subtask is completed* **then**
    if *p points to the end of a block of subroutines in a* while-*routine* **then**
      Point p to the while-routine.
    end
    else if *p points to the end of a block of subroutines in a group of blocks of subroutines spawned from a* parallel-*routine*/repeat-*routine* **then**
      if *the pointed block is not the last remaining one* **then**
        remove the pointer
      end
      else
        remove the pointer and generate a new pointer q pointing to the subroutine subsequent to the *parallel*-routine/*repeat*-routine
      end
    end
    else
      Point p to the subsequent subroutine. Terminate the program if no subsequent subroutine exists.
    end
  end
end

A.3 Full Illustration of the E-MAPP Algorithm

We describe the whole process of E-MAPP in Algorithm 2, which corresponds to the inference stage of E-MAPP.

Algorithm 2 The E-MAPP Algorithm during test time

input : Environment env and its guiding program, a set of pointers $\mathbb{P}$ pointing to the possible subroutine set, a policy module f_{policy}, a perception module $f_{perception}$

while *the task is not completed* **do**

  while *there are pending perception primitives in the possible subroutine set* **do**

    Run $f_{perception}$ on the perception primitives and get the results o_{perc}. Update P according to o_{perc} and the rule in Algorithm 1.

  end

  Run f_{policy} to obtain the auxiliary functions on each behavior primitive p points to.

  Compute the cost of each possible allocation based on the auxiliary functions.

  Run f_{policy} on the subtasks with minimal cost and obtain the joint action a

  Environment steps forward with a

  while *There are completed behavior primitives in the possible subroutine set* **do**

    Update p accourding to the rule in Algorithm 1

  end

end

A.4 Training Details for the Policy Module

Architecture details. The policy module takes as input a goal vector g and a state s and outputs an action distribution $\tilde{a}$. A goal vector g has a size of 20 where the first 10 elements are one-hot encoded behavior type (e.g., *Chop*) and the latter 10 elements are one-hot encoded behavior arguments (e.g., *Tomato*). A state s is comprised of a map state s_{map} and an inventory state s_{inv}. The map state s_{map} has a size of $20 \times 8 \times 8$ where 8×8 is the resolution of the map and 20 is the number of object types. The inventory state s_{inv} has a size of 6 where the first two entries denote the location of the agent, the third entry denotes whether the agent is holding objects, and the last three entries denote whether a certain dish is ordered.

The map state s_{map} is encoded by a four-layer convolutional neural network (CNN) with channel sizes of 32,64,64, and 64. For each convolutional layer, we use 1 as the stride size and 1 as the "same" padding size. Each convolutional layer has a kernel size of 3 except for the first one that has a kernel size of 5. A ReLU nonlinearity is applied to every convolutional layer. The output is finally flattened into a feature vector and fed into a linear layer, producing a 128-dim map feature vector, denoted as f_{map}.

The inventory state s_{inv} is encoded by a three-layer MLP with hidden size 128 for all layers. The output feature vector has a dimension of 32, denoted as f_{inv}.

The map feature f_{map} and the inventory feature f_{inv} are then concatenated, producing the 160-dim state feature f^1_{state}.

The goal vector g is encoded by a three-layer MLP with a hidden size of 128 for all the layers. The output goal feature f_{goal} is a 640-dim feature vector. f_{goal} can be viewed as the concatenation of four 160-dim feature vectors,denoted as γ_1, β_1, γ_2, and γ_2 respectively.

The state feature is modulated by γ_1 and β_1 as $f^2_{\text{state}} = f^1_{\text{state}}\dot{\gamma}_1 + \beta_1$. Next, the modulated feature f^2_{state} is encoded by a two-layer MLP with both hidden size and output size equals 128. The output is then modulated by γ_2 and β_2, producing the goal-conditioned state feature f_{state}.

Finally, the goal-conditioned state feature f_{state} is encoded by two linear layers to produce the 24-dim action distribution and the 1-dim value function.

In practice, the parameters of the convolutional layers are shared by the policy net and the value net, while other parameters are separate.

Table 6: **Hyperparameters for training of the policy module.** We use MAPPO as the backbone algorithm. The common hyperparameters are listed below.

Name	Value
learning rate	3e-4
training steps	10M
update batch size	256
number of rollout threads	8
rollout buffer size	4096$\times$ 8
weight of value loss	0.1
weight of policy loss	1
weight of entropy loss	0.01

Table 7: **Hyperparameters for training of the perception module.** We model the perception module as a binary classifier and use cross-entropy loss as the training objective.

Name	Value
learning rate	3e-4
update batch size	128

In cooperative settings, the goal input of the assistive agent is the leading agent's goal. We use a separate assistive goal encoder for it. The architecture of the altruistic goal encoder is the same as the independent goal encoder mentioned above.

Self-imitation learning. In our task domain, it is important to address the challenge of sparse rewards, which is also a key issue for goal-conditioned reinforcement learning [31]. To tackle this, we propose to better utilize the successful trajectories of each agent inspired by self-imitation [32]. For each agent, we select the state-action pairs from the replay buffer with empirical returns that are larger than a threshold r_{thresh}. The self-imitation learning objective of the agent i is $\mathcal{L}_{sil} = -r_i(s,t)\log\pi_i(s_t)$. This loss is added directly to the reinforcement learning loss.

Hyperparameters. Table 6 shows the hyperparameters used in the policy module.

A.5 Architecture and Training Details for the Perception Module.

Architecture of the perception module. The architecture of the perception module can be obtained from that of the policy module in Appendix A.4 by replacing the goal encoders with perceptive query encoders. The details of the encoders remain unchanged.

Dataset collection. We randomly sample 10k environments as the training dataset. To augment the data, we label the ground-truth of the perceptive queries after the agents randomly take 10-15 steps. We leave 10% of the dataset as the evaluation dataset and terminate the training process when the accuracy on the evaluation dataset is larger than 99% for 5 times.

Hyperparameters. Table 7 shows the hyperparameters used in training the perception module.

A.6 Training Details for the Auxiliary Functions

We randomly sample 10k environments as the training environments for each of the three auxiliary functions: f_{reach}, f_{feas}, and $f_{\textit{cost-to-go}}$.

To train the reachability function, we run the pre-trained single-agent policy on the training environments to collect the training data. The agent is required to fulfill a specific subtask within an episode of 128 timesteps. If the agent violates the program by completing the wrong subtask or exceeding the time limit, we will label every state in the trajectory as $False$. In contrast, if the agent successfully completes the subtask, we will label the states as $True$. We train the reachability function by alternatively collecting (state, goal, label) triplets and train on the collected data.

To train the feasibility function, we run the pre-trained multi-agent cooperative policy in the training environments to collect the training data. The agents are required to complete a specific subtask within an episode of 128 timesteps. Although both of the agents are responsible for the subtask, we only take account of the leading agent's trajectories. The states in the trajectories are labeled True/False according to whether the subtask is successfully completed during an episode.

To alleviate the problem of mistakenly labeling a feasible (state, goal) as *False* due to the imperfection of the pre-trained policy, we run multiple times on the environment that is labeled *False* and correct the label if there is a successful case.

A.7 Detailed Environment Description

In the *Overcooked* environment, the goal of the agents is to complete long-horizon tasks, such as preparing a dish. A typical dish would require first picking up ingredients from certain supplies, then processing the ingredients (e.g., chop, merge together, or put on a plate), and finally delivering the dish (and washing the plates if another dish is still needed). We also introduce "on-fire" as an accident for the agents to handle with a fire extinguisher. The subtasks correspond to the subroutines in the domain-specific language (DSL). Different subroutines together with the control flow form the guiding parallel programs for the agents.

For each agent, the state is composed of a map state and an inventory state. Both the maps and the programs are one-hot encoded as object-centric representations. The agents' action space is discrete. There are 24 possible actions, including 6 operations (move, pick, place, serve, merge, and interact) in 4 directions. In multi-agent settings, we follow the original game and assume full observability.

A.8 More Visualizations for Generalization

Figure 7 shows more detailed visualizations of two emergent behaviors.

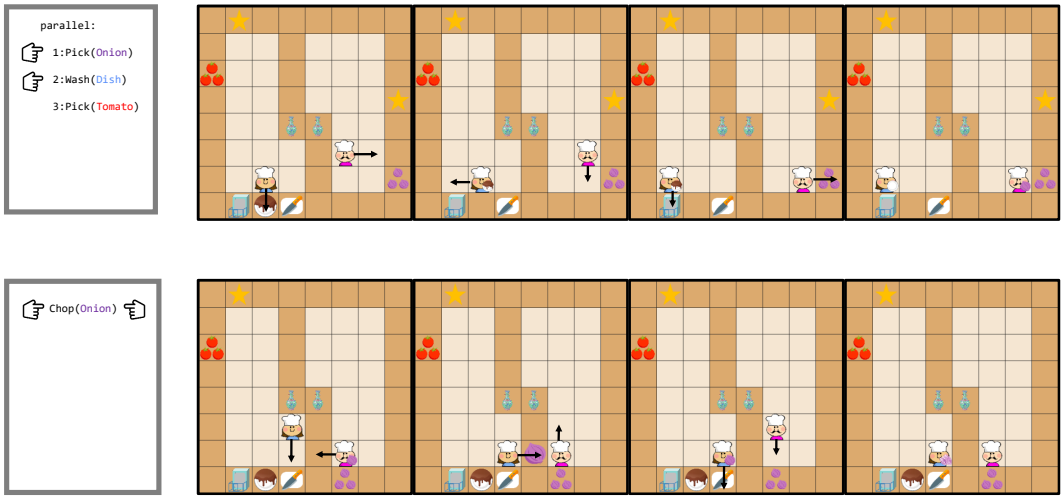


Figure 7: **Visualizations of parallel executions and cooperative behaviors.** In the first example, there are three feasible subtasks. The blue agent chooses the subtask *Wash(Dish)* because the dirty plate is both reachable from and close to him. The magenta agent chooses the subtask *Pick(Onion)* because the onion supply is reachable from him. In the second example, the two agents cooperate on the subtask *Chop(Onion)*. The magenta agent picks the onion and passes it to the blue agent, while the blue agent finalizes the subtask by chopping the onion at the chopping block.

A.9 Baseline Description

In our study, we mainly consider two types of baselines: state-of-the-art MARL algorithms and the natural language guided agents.

- State-of-the-art multi-agent reinforcement learning. We directly use MAPPO [51] to train joint policies without the guiding program, while still providing the dense rewards to the agents when any correct subtask is completed;
- Natural language-guided agent. We train a goal-conditioned policy where the goal is encoded from the natural language description with a pre-trained BERT model in the pytorch package *transformer* [10]. The encoded features of the tokens are average pooled and frozen during the training process. Then we use a learnable MLP to encode the frozen features into goal features. The MLP has three layers connected by the ReLU activation and the hidden sizes of the MLP are 128,128,128 respectively. We note that here we do not provide the program-like structure for the language-guided agents.

The baseline models have the same policy and value network architecture as those of E-MAPP. When training the end-to-end baseline models, the agents are rewarded 0.2 if they complete a correct subtask and 1 if the final task is completed. We also use self-imitation learning in baseline algorithms to address the sparse reward problem.

The computation costs of E-MAPP and baselines are shown in Table 8

Table 8: **Computation Cost.** The number of paramerters and the running time in E-MAPP and baselines.

Model	Parameters	Running Time
E-MAPP	5.6M	around 72h
natural language guided agent	2.3M	around 36h
MAPPO	1.8M	around 24h

A.10 Results in the Stacking Environment

To evaluate the ability of E-MAPP to scale to more complex control tasks, we also demonstrate how E-MAPP works on a parallel stacking task set. Figure 8 is a demonstration of the task completion process. The baselines considered above fail to achieve the goal in any episode, while E-MAPP can achieve 46% completion rate.

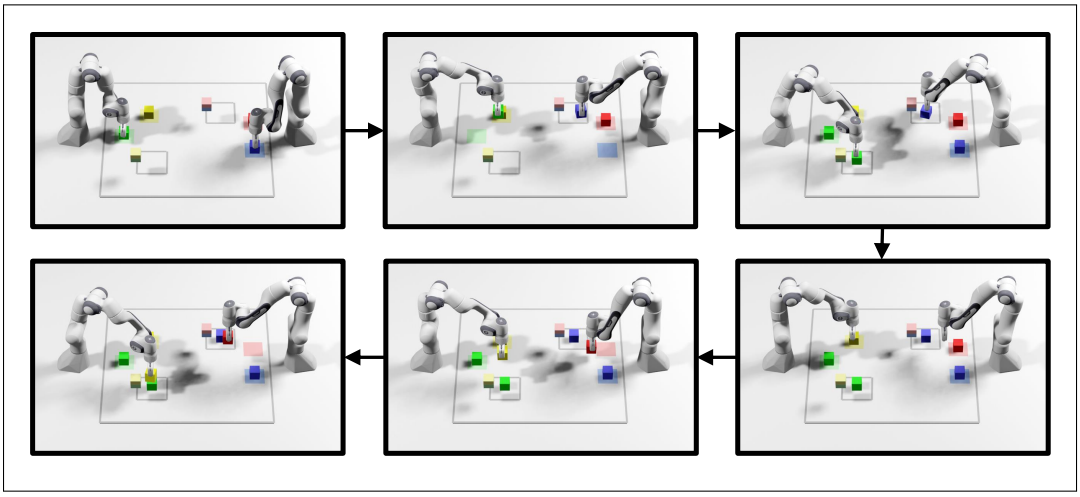


Figure 8: **A demonstration of the completion process of stacking tasks.** The pictures show the process of picking up boxes and stacking them.

A.11 Computational Resources

We train our model on a single Nvidia TITAN-X GPU, in a 16 core Ubuntu 18.04 Linux server.

A.12 Examples of Programs Used in Evaluation

A.12.1 Easy tasks

```
if IsOnFire():
    PutOutFire()

if is_ordered(ChoppedTomato):
    Serve(ChoppledTomato+Plate)

if is_ordered(ChoppedOnion):
    Pick(FreshOnion)
```

A.12.2 Medium tasks

```
repeat:
    Pick(FreshTomato)

parallel:
    1. Pick(FreshOnion)
    2. Pick(FreshTomato)
    3. WashDirtyPlate()

if is_ordered(ChoppedTomato):
    Chop(FreshTomato)
    Merge(ChoppedTomato,Plate)
    Serve(ChoppedTomato+Plate)
```

A.12.3 Hard tasks

```
parallel:
    1: if is_ordered(‘‘Onion’’):
            Merge(‘‘ChoppedOnion’’,’’Plate’’)
            Serve(‘‘ChoppedOnion’’)
    2: if is_ordered(‘‘Onion’’):
            Merge(‘‘ChoppedTomato’’,’’Plate’’)
            Serve(‘‘ChoppedTomato’’)
    3: while (True):
        If (IsOnFire()):
             PutOffFire()

parallel:
        1:
            Pick(FreshOnion)
            Chop(FreshOnion)
        2:
            Pick(FreshTomato)
            Chop(FreshTomato)
        3:
            WashDirtyPlate()
        4:
            Merge(ChoppedOnion,Plate)
            Serve(ChoppedOnion)
        5:
            Merge(ChoppedTomato,Plate)
            Serve(ChoppedTomato)
```

A.13 More experiments

A.13.1 Overcooked with more agents

We conduct an experiment with a doubled number of agents to evaluate our algorithm. Table 9 shows the results. The results indicate that E-MAPP can scale to environments with more agents and further boost the time efficiency by parallelization.

Table 9: **Additional experiment in Overcooked involving four agents.** E-MAPP can scale to environments with more agents and further boost the time efficiency by parallelization.

model	score	completion rate
E-MAPP(original)	0.99±0.22	43.7%
E-Mapp(larger)	1.13± 0.31	46.3%

A.13.2 Comparison with other centralized execution agents

We also compare E-MAPP with a centralized execution approach. We implement a centralized PPO where joint policy is directly produced by a centralized network. Table 10 shows the results. The results indicate that the centralized PPO suffers from the high-dimensionality of the joint action space and fails to learn the cooperation and the coordination.

Table 10: **Comparison of E-MAPP, centralized algorithm PPO and decentralized algorithm MAPPO.** We additionally compare E-MAPP with a centralized PPO that directly output the joint policy.

model	score
E-MAPP	1.58±0.60
MAPPO(decentralized)	0.59± 0.27
MAPPO(centralized)	0.48 ± 0.21

A.14 Potential parallel-program synthesis approaches

The guiding program in our work can be obtained with program sysnthesis approaches. When it comes to a new domain, we can devise new perception primitives and behavior primitives based on object properties and interactions among objects [25]. These primitives along with the branching and parallelization keywords compose the DSL. Previous approaches on program synthesis [44, 11, 7, 8] can be applied to synthesize programs for tasks. For example, we can synthesize programs from diverse video demonstrations. The activities (subtasks) of a tasks in a video can be segmented out as a subroutine for program extraction [44]. By summarizing the chronological order of subtask completions, we can obtain the dependence of subtasks and put the possibly parallelizable subtask in one parallel subroutine.