A novel algorithm for parallelizing actions of a sequential plan

Sofia Santilli,[1] **Alessandro Trapasso,**[1] **Luca Iocchi,**[1] **Fabio Patrizi**[1]

[1]DIAG, Università degli Studi di Roma "La Sapienza", Italy
e-mail: sofiasantilli1998@gmail.com {trapasso, iocchi, patrizi}@diag.uniroma1.it

Abstract

The execution of robot plans often offers the possibility of executing actions in parallel in order to achieve better performance. However, this feature is rarely present in robotic applications using automated planning techniques. In this paper, we present a novel algorithm that, given as input a sequential plan computed by a classical planning engine, outputs a plan in which actions are parallelized in order to reduce the plan execution time. The proposed algorithm uses the planning domain specification, but not a planning engine, thus it is extremely efficient with respect to the plan length. Moreover, it overcomes some limitations of current multi-agent planning engines that offer limited support to the generation of parallel plans. The proposed algorithm has been developed and integrated within the AIPlan4EU Unified Planning framework and experimented in a real industrial robotic use-case.

1 Introduction

Automated planning and scheduling have become increasingly important in robotics, particularly in applications where time is a limiting factor, such as industrial automation or autonomous driving. One of the challenges in this field is minimizing the execution time of plans, especially in a multi-agent planning (MAP) context.

There are two main variants of MAP: cooperative, where agents work together towards a common goal while maintaining privacy about specific information, and competitive, where each agent acts towards their own private goal. MAP has two main solution approaches: centralized and distributed. In the centralized approach, a central planner creates a plan that assigns actions to agents, requiring access to information from all agents. In the distributed approach, agents plan separately to contribute to a common goal or to reach their private goals. This paper focuses on agents acting collaboratively and centralized planning.

We use a formalization of multi-agent planning with explicit agent representation (Trapasso et al. 2023) to model our multi-agent problems. This formalism is integrated into the Unified Planning framework (UP) of the AIPlan4EU project[1]. Within this formalism, agents are explicitly represented as individual entities with specific private and public fluents and specific actions. These agents act within a common environment, which retains the information available to all of them.

Figure 1: Robot manipulation task

In this article, we present a novel algorithm for action parallelization aimed at minimizing the total execution time of a given plan. The algorithm works on plans generated by classical planners (used as a centralized planner for a MAP problem) and can handle sequential and partially parallelizable actions. We consider an industrial use case (Figure 1) where a robotic manipulator performs qualitative tests on laundry pouches. In this task, the robot places these pouches on measurement instruments for quality tests. We assess the performance of the proposed algorithm in terms of planning time and plan execution time, showing efficiency and effectiveness in reducing plan execution time through the execution of actions in parallel.

Our algorithm improves plan execution efficiency, taking as input a classical sequential plan and producing a plan in which some actions are parallelized with a sequence of atomic actions. The resulting parallel plan significantly reduces the execution time.

The experimental results reported in this paper show that our approach effectively reduces the total execution time of plans and can obtain parallelizations that are not possible

[1]https://www.aiplan4eu-project.eu/

with multi-agent planning methods. Overall, the proposed algorithm represents a promising solution for optimizing plan execution times in industrial automation and other applications where time is critical.

2 Related work

In recent years, the planning community has shown an increasing interest in cooperative planning. The International Planning Competition (ICP), with its first 1998 edition, has played a crucial role in advancing the field. In particular, in 2015, the first IPC track on Distributed and Multi-Agent Planning (CoDMAP) (Komenda, Štolba, and Kovács 2016) established MA-PDDL (MA-PDDL) (Kovács 2012) as the de-facto standard language for MAP. Some of the different planning engines participating in that competition generated parallel plans. Competing MAP solvers that allow for parallel plans are CMAP (Borrajo and Fernandez 2015), MAPR (Borrajo and Fernandez 2015), PMR (Luis and Borrajo 2015), and FMAP (Torreño, Sapena, and Onaindia 2018). We briefly review the main features of these planners.

MAPR (Planning by Reuse) and CMAP (Cooperative MAP) were developed by the same authors and integrated into the same MAP framework. These algorithms generate totally ordered (sequential) plans. However, in MAP, plans are executed by multiple agents, thus making the use of parallel plans preferable to sequential ones. To address this, the authors implemented an algorithm that transforms the sequential plan into a partially ordered plan (POP), similarly to (Veloso, Perez, and Carbonell 2008), from which a parallel plan is finally extracted by an independent algorithm. The algorithm, however, does not optimize the overall plan duration.

PMR is a centralized, single-threaded planner, which combines two MAP techniques: plan merging (Foulser, Li, and Yang 1992) and plan reuse (Fox et al. 2006). PMR allows individual agents to build their plans independently and combines them into a single plan with parallel actions. The obtained plans, however, are not guaranteed to be valid, especially in tightly coupled domains. In such cases, PMR applies a re-planning approach, taking the invalid plan as input, to generate a sound plan via planning-by-reuse. Once a valid plan is obtained, PMR parallelizes it by transforming a totally ordered plan into a POP. This allows multiple agents to execute actions in the same step.

FMAP adopts the POP paradigm, which enables handling plans with parallel actions and imposes an ordering on actions only when strictly required. FMAP parallelizes only atomic actions, as it does not explicitly deal with time constraints or durative actions.

The approach proposed in this paper has similar features to the ones described in this section, but with a different perspective combining an efficient classical planner generating a centralized sequential plan with a greedy algorithm computing the parallel plan. The approach in this paper considers some specific aspects arising in robotic applications and, although not in an optimal way as for other MAP planners, it allows for significantly reducing plan execution time, which is an important objective of using planning technologies for robotic applications. Moreover, the development of a solution based only on classical planning allows for exploiting the large development of planning engines with many features that may not be available for other specialized planners, like MAP planners. For example, in our formulation of the problem, we use conditional effects that are not available in some MAP planners.

3 Problem definition

In this paper, we consider problems modelled with a planning language and the use of automated planning engines to find solutions to such problems. We denote with $\langle \mathcal{D}, \mathcal{P} \rangle$ a planning problem, where the domain description $\mathcal{D}$ includes formal specification of actions and the problem description $\mathcal{P}$ contains specification of initial state and goals. The set of actions is denoted with $A = \{a_1, \ldots, a_n, d_1, \ldots, d_m\}$, in which a_i are *atomic actions* and d_i are *parallelizable actions* that can be executed in parallel with atomic actions. We assume the designer to provide information about which actions should be considered as *parallelizable actions*, depending on the specific application.

In this paper, we do not provide an explicit representation of action duration, and we assume that the system designer is able to identify those actions that our algorithm should parallelize. In general, these *parallelizable actions* are actions that take a substantial amount of time during execution, while other actions could be executed in parallel. In robotic applications described as multi-agent planning problems, *parallelizable actions* are typically actions performed by other agents while the robot could execute its own actions in parallel.

Actions A are described with a classical planning language (e.g., PDDL or AIPlan4EU Unified Planning formalism) and a classical planning engine (e.g., FastDownward) is used to generate a plan (i.e., a sequence of actions) to achieve the specified goal. At this stage, the difference between atomic actions and parallelizable actions is not considered. The sequential plan generated by the planner is denoted with $\Pi := [\alpha_1, \ldots, \alpha_L]$, $\alpha_i \in A$

The goal of the algorithm presented in this paper is to transform a sequential plan Π into a Simple Parallel Plan, where parallelizable actions are performed in parallel with a sequence of atomic actions, while maintaining the correctness of the plan in achieving the goal.

Simple Parallel Plan: $SP := [\sigma_1, \ldots, \sigma_K]$, with

$$\sigma_i := a_i \mid d_i \mid d_i \,||\, \Pi'$$

with $\Pi' := [\alpha_1, \ldots, \alpha_N]$, $\alpha_i \in A$.

SP is a sequence of terms where each term can be either an action (atomic or parallelizable) or the parallel execution of a parallelizable action with a sequence Π' of actions. This plan format is denoted *simple* since it considers only one level of parallelism. The generalization of the approach is left as future work.

The problem we are considering in this paper is the following.

Problem Definition. Given a classical planning problem $\langle \mathcal{D}, \mathcal{P} \rangle$, and a sequential plan Π generated by a classical

planning engine as a solution of the problem, generate a simple parallel plan SP that is equivalent to Π in solving the planning problem.

The solution that is presented in this paper is based on a transformation of the plan Π by using the action specification of the planning domain, but without using a planning engine.

4 Algorithm

The method for solving the problem defined in the previous section is divided in two steps: i) pre-processing of the input sequential plan, ii) generation of the simple parallel plan.

Plan actions pre-processing

Pre-processing of the sequential plan is needed to adapt the action specifications in the planning domain to their actual implementation. Two pre-processing steps are needed.

The first step is to divide the parallelizable actions d_i into two actions that are denoted by $d_i.start$ and $d_i.end$, where $d_i.start$ is the action that is parallelized to a sequence of actions and $d_i.end$ is needed to make the termination of d_i explicit. A new fluent is also introduced $d_i.inprogress$ to denote that the action d_i is currently running. $d_i.start$ has the same preconditions of d_i and as effect it only sets $d_i.inprogress$ to true. $d_i.end$ has precondition $d_i.inprogress$ to be true and the same effects of d_i. Replacing the action d_i with the sequence $[d_i.start; d_i.end]$ in a plan solution of a planning problem preserves its correctness. In the actual implementation of d_i, $d_i.start$ is an instantaneous action (i.e., the fluent $d_i.inprogress$ is immediately set to true) representing the beginning of a parallelizable action, while $d_i.end$ blocks plan execution until the action is actually completed and the effects are achieved.

The second pre-processing step is needed to mask some parameters of actions that are used in the planning specification, but not in the action implementation. This is a typical situation arising when formalizing planning domains for robots using a classical planning language. In fact, the position of a mobile robot or a robot arm is usually denoted with a predicate (for example, `at(location)`) and actions denoting robot motion are usually described with two parameters: `from` and `to` (for example, `goto(from, to)`). In particular, the `from` parameter is used in the effects of the action to make the value of `at(from)` false. However, in most robotic applications, the implementation of such actions does not require the parameter related to the starting position of the robot, which is usually stored in a different format (e.g., within the localization module), and, in general, navigation modules are able to achieve the target position from any current position. In other words, there is a mismatch in the definition of the action specification at planning level `goto(from,to)` with respect to its actual implementation not using the parameter `from`. The two representations need to be aligned before the execution of the algorithm proposed in this paper, in order to find effective parallel plans. Consequently, in this paper, we make the following assumption and the following transformation of the actions in the plan.

Assumption. If an action is specified at planning level with a parameter that is not used in its actual implementation, the action is assumed to achieve its effects for any value of such a parameter.

For any action specified with parameters at planning time that are not used at execution time, those parameters are replaced by a special symbol `_` denoting any value. In the above example, the action `goto(from,to)` is replaced with `goto(_,to)`.

After the two pre-processing steps described above, the original sequential plan Π has been transformed into an equivalent plan P, in which parallelizable actions are decomposed in two atomic actions (start and end) and parameters not used in the action implementation are masked with an any-value symbol.

Simple parallel plan generation

The general structure of the proposed algorithm is presented in *Algorithm 1*.

This algorithm takes as input a sequential plan P that was previously pre-processed after being generated by a planner and that is characterised by a length equal to L (computed at line 6), the number of actions that compose the plan. The other input to the algorithm is a number N, representing the length of the maximum sequence Π' of actions that we want to parallelize to a parallelizable action d_i. The output of the algorithm is a simple parallelized plan SP, initialized as empty (line 8). At line 9, the variable cs, denoting the current state (i.e., values of all the fluents) that is updated during the execution of the algorithm, is set to the initial state of the problem $\mathcal{P}$.

The algorithm consists in a main loop over all the actions composing P. Here at line 12, an empty list γ is initialized: it will memorize one or more actions that at the end of the current iteration of the loop will be added to SP. For each action act_i (computed at line 13), there are three possibilities:

- if act_i has already been inserted in SP (lines 14-17), the algorithm directly skips to the next action of the loop. Otherwise, the current loop continues;
- the second possibility is that the action is any action of the plan, except for the $d_i.start$ ones (lines 18-22). In this case, the *UpdateState* algorithm (*Algorithm 2*) is first called, which computes the event associated to the action, by instantiating its parameters (*Algorithm 2*, line 7). If the action is one that presents one or more unspecified parameters (so it was subjected to the second step of our preprocessing), those parameters are replaced by the value of their corresponding fluent from $curr\_state$ (*Algorithm 2*, line 5). After computing the event, it is checked if the event can be applied to the current state; if so, it is simulated and the current state is updated (*Algorithm 2*, line 9).

  Finally, the action is added to the list γ, in order to be memorized and later added to SP.
- the last possibility consists in act_i being an action $d_i.start$ (lines 23-65), that points out that the execution of a parallelizable action begins. The event relative to the

Algorithm 1 Action parallelization algorithm

```
1: input: ⟨D, P⟩ planning problem
2: input: P pre-processed sequential plan
3: input: N maximum length of the sequence to parallelize
4: output: SP simple parallelized plan
5:
6: L = len(P)
7: i = 0  // C-style index
8: SP = []
9: cs = P.initialState  // current state
10:
11: while i < L do
12:     γ = []
13:     act_i = P[i]
14:     if act_i already in SP then
15:         i+=1
16:         continue
17:     end if
18:     if act_i atomic action then
19:         // Build an atomic action term
20:         UpdateState(cs, act_i, D)
21:         γ.add(act_i)
22:         i+=1
23:     else
24:         // act_i is the start of a parallelizable action
25:         // Build a parallel term
26:         applicability = False
27:         act_start = act_i ; act_end = P[i+1]
28:         UpdateState(cs, act_start, D)
29:         γ.add(act_start)
30:         j = i+2
31:         // Search for first applicable action
32:         while j ≤ L and not(applicability) do
33:             if applicable(cs,P[j]) then
34:                 cs' = cs  // copy of current state
35:                 UpdateState(cs', P[j], D)
36:                 γ.add(P[j])
37:                 applicability = True
38:             end if
39:             j+=1
40:         end while
41:         if applicability then
42:             // Apply up to N-1 more actions in parallel
43:             w = 0
44:             while w < N-1 and applicability do
45:                 if applicable(cs,P[j+w]) then
46:                     UpdateState(cs', P[j+w], D)
47:                     γ.add(P[j+w])
48:                     w+=1
49:                 else
50:                     applicability = False
51:                 end if
52:             end while
53:             // Check if remaining part is valid
54:             P_r = [P[i+2], , , , P[j−1], P[j+w], ..., p[L−1]]
55:             if valid(cs', P_r)  then
56:                 cs = cs'
57:             else
58:                 γ = [act_start, act_end]
59:             end if
60:         end if
61:         UpdateState(cs, act_end, D)
62:         γ.add(act_end)
63:         i+=2
64:         UpdateSP(SP,γ)
65:     end if
66: end while
```

Algorithm 2 UpdateState

```
1: input: curr_state, action, , domain D
2: output: updated curr_state (if applicable)
3:
4: if action has unspecified_params then
5:     assign unspecified parameters from curr_state
6: end if
7: event = sim.get_events(curr_state, action, , D)
8: if applicable(curr_state, event) then
9:     curr_state = sim.apply(event, curr_state)
10: end if
```

start action is computed and applied to the state in the UpdateState function; act_{start} is added to γ (lines 28-29). Whereupon, a first inner loop (lines 32-40) is exploited in order to slide the remaining sequential plan until we find a first action that can be parallelized to the current act_{start}: each action act_j = P[j], is tested as applicable in the current state. If the first inner loop does not find an action to parallelize, the algorithm does not enter the condition at line 41 and directly update the state according to act_{end} and adds act_{end} to γ (lines 61-62). This accounts for the case in which d_i is not parallelizable with the actions in the plan. Instead, if an applicable action is found, applicability is set to true (line 37) and the algorithm enters the second inner loop at lines 44-52, in order to find up to other N-1 actions to complete the sequence to be parallelized (w at line 43 is used to count the number of parallelized actions). At each iteration of the second inner loop, the action P[j+w] is tested for applicability in the current state (line 45) through the *applicable* function that, given the current state and an action, generates the relative event and checks its applicability. If so, we add P[j+w] to γ and continue the loop. This is repeated until an action cannot be parallelized or until the maximum length of the sequence to parallelize is reached.

A final check (lines 54-59) is needed to verify that the current state is valid for the remaining part of the plan, excluding the actions added in γ in the first inner loop. This validity check is performed by simulating the execution of the actions for cs', but it does not change the value of cs'. If the remaining part of the plan is valid, we update the current state cs with the copy cs' used so far, otherwise this sequence is not parallelizable and we roll back to adding in the final plan only the action d_i denoted with $[act_{start}, act_{end}]$ (line 58), without modifyng cs. Next, the current state is updated with the effects of act_{end}, which is also added to γ (lines 61-62). Finally, γ is used to update the output plan SP (line 58) and the main loop can continue.

The $UpdateSP$ function (presented in *Algorithm 3*) is used to insert the actions memorized in the γ list into SP. This list will contain at least one action, always inserted in SP. If γ contains two elements, it can only mean that the first one is an act_{start}, and the second is the relative act_{end}, which are reported in SP in two different inputs. Instead,

if γ contains more than two elements, it means that the algorithm found a sequence AP of at least one action to parallelize to act_{start}. So we add in SP first the parallel term with act_{start} and the sequence $AP = \gamma[1], \ldots, \gamma[n-2]$ (with n being the length of γ). Then the action act_{end} (i.e., the last instance of γ) is also added to SP to close the parallel execution.

Algorithm 3 UpdateSP

```
input: SP, list of actions γ
output: updated SP

n = len(γ)
if len(γ) = 1 then
    SP.push_back(γ[0])
else if len(γ) = 2 then
    SP.push_back(γ[0])
    SP.push_back(γ[1])
else
    σ = γ[0] || [γ[1], ..., γ[n − 2]]
    SP.push_back(σ)
    SP.push_back(γ[n − 1])
end if
```

The pre-processing step and the algorithm described in this section compute a parallel plan that is equivalent to the original plan in terms of correctness in solving the planning problem, while reducing the plan execution time.

The overall computational cost of this process is $O(L^2)$, with L being the size of the sequential plan.

5 Implementation and experimental results

The algorithm has been implemented within the AIPlan4EU UP framework. Although it is possible to implement this algorithm in other frameworks (for example, by using plain PDDL specifications), using the UP framework provides many useful functionalities that make this implementation easy and effective. In particular, in the UP framework, the sequential plan generated by a planner is a sequence of action instances and a simulator is available to compute state evolution when applying actions. Therefore the steps of the algorithm to check the applicability of an action in a given state and the applicability of parallel execution of actions were easily and effectively computed by using such functionalities.

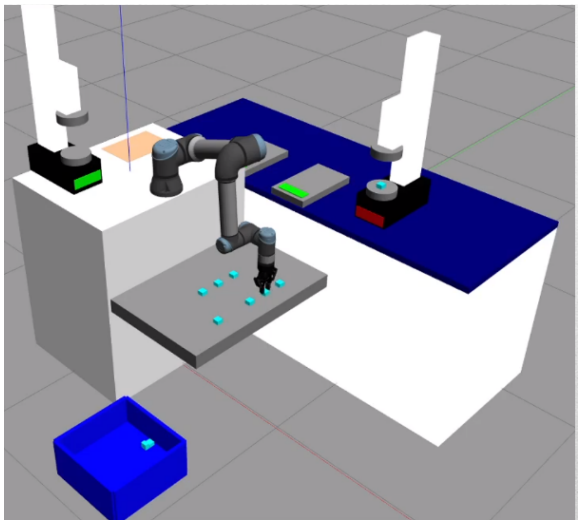

Figure 2: Gazebo simulation environment

The implemented algorithm has been tested on a Gazebo simulator (Figure 2) used to replicate the real industrial use case and is going to be deployed in the real robotic scenario in the near future. In the simulator, the actions of the measurement instruments are simulated through green/red lights denoting the time of operation of the instrument.

Some results obtained through the use of the proposed algorithm for an industrial robot task are described below. In the considered task, a robotic manipulator has to perform some qualitative tests on laundry pouches. The robot performs pick & place actions to move pouches on measurement instruments that perform the required measures. This domain is modelled as a multi-agent problem in which the robot and all measurement instruments are modelled as agents. In this domain, we considered measurement actions as parallelizable actions, since they require a substantial amount of time, during which the robot arm can perform other placement actions.

As described in the previous section, a centralized classical planner has been used to compute a sequential plan satisfying the goals of the presented problem. In the case study shown here, it was required to test a number of pouches into two instruments: a scale and a tightness meter.

A first example of the application of the algorithm to the sequential plan is illustrated here. Let us consider the following portion of a sequential plan solution of a problem.

```
[ ...
movegripper(open, _, _)
measure(scaleB pouch1).start;
measure(scaleB, pouch1).end;
goto_grasp(_, scaleUpB, pouch1);
movegripper(close, _, _);
... ]
```

In this portion of the plan, the robot opens its gripper to place a pouch on the scale, then waits for its measure, moves the arm in the grasp position and closes its gripper to pick the pouch up for the next operation. The algorithm presented

in this paper, with input $N = 1$, can parallelize the portion of this plan as follows.

```
[ ...
movegripper(open, _, _)
measure(scaleB pouch1).start ||
  [ goto_grasp(_, scaleUpB, pouch1) ];
measure(scaleB, pouch1).end;
movegripper(close, _, _);
... ]
```

At execution time, the parallel plan is faster since the action to position the robot arm to grasp the pouch after the measurement is performed in parallel to the measure. Notice also that the action `measure(scaleB, pouch1).end` is a blocking action, thus guaranteeing that the robot picks the pouch only after the measurement is completed. Notice also that in this case we would obtain the same parallel plan for any $N \geq 1$, since grasping the object is not a valid parallel action with respect to measuring. Thus this case is not sensitive to the input value N.

The second example considers a longer operation performed by another instrument. In this case, while a pouch is measured on such an instrument, the robot can operate on a different pouch and complete another measure using a different instrument. The portion of plan below shows the parallel term built by our algorithm with $N \geq 4$.

```
[ ...
measure(markt10, pouch1).start ||
  [ goto_grasp(_, drawer, pouch3);
    movegripper(close, _, _);
    goto(_, scaleUpA);
    movegripper(open, _, _);
  };
measure(markt10, pouch1).end;
goto(_, markt10);
... ]
```

In this parallel plan, while a pouch is measured by the instrument `markt10`, another one is picked and placed on `scaleA`. This is the situation shown in Figure 2, where the red label of the rightmost instrument indicates that the measure is in progress, while the robot arm is manipulating another pouch.

A relevant factor to be taken into account, when deciding which parallelization method to use, is the computational time employed for figuring out the final plan. For the proposed approach, the overall planning is given by the sum of the time t_P required by the planner for computing a sequential plan and of the time t_A required by Algorithm 1.

The planning times t_P and t_A for different sizes of the problem are shown in *Table 1*. As expected, with the increase in the number of pouches to be tested, the plan length increases linearly, the planning time increases exponentially, while Algorithm 1 time increases linearly. The proposed approach can thus exploit the benefit of using very performing classical planners (such as fast downward), in contrast with the use of more complex MAP planners. Comparing computational times of the proposed approach with other MAP planners is left as future work.

# pouches	plan length	t_P	t_A
1	21	3-5	2
2	39	10-12	3
3	57	34-41	5

Table 1: Planning times in seconds of fast-downward (min-max on different platforms) and Algorithm 1 to generate an overall parallel plan with respect to the number of pouches in the problem.

The main goal of the proposed approach is however to reduce plan execution time. Algorithm 1 applied with N=1 returns parallel plans with only one atomic action in parallel with parallelizable actions, being equivalent to planners able to parallelize only atomic actions, such as FMAP. While these parallel plans are better (in terms of plan execution time) with respect to the sequential plan, parallel execution is not fully exploited. On the other hand, the advantage of using our algorithm is the possibility of increasing the size of the parallel sequence N to achieve higher parallelism.

plan	t_e avg %	t_e stddev %
Sequential	100 %	-
FMAP / Alg. 1 ($N = 1$)	95.9 %	0.81 %
Alg. 1 ($N = 4$)	79.7 %	2.70 %

Table 2: Average and standard deviation execution times in percentage with respect to the sequential plan.

Table 2 shows the results of 24 experiments covering different situations in the simulated environment. The execution time is reduced in average to 95.9% when using single atomic actions in parallel, and to 79.7% when using sequences of 4 actions in parallel.

6 Discussion

The proposed approach guarantees the correctness of the plan, is fast, and may significantly reduce plan execution time. It is especially useful in robotic applications with complex tasks executed by different machines when automated planning techniques are used to generate plans of action to achieve specific goals.

Nonetheless, this approach has some limitations. First, the parallelizable actions must be given in input. This is usually not a difficult choice in application domains where different operations are taken by different machines and we aim at parallelizing those operations with the ones of a robot. Second, the maximum length of the parallel sequence must be provided as input. The sensitivity of the final results to this input decreases as the value of N increases. Third, Algorithm 1 generates a simple parallel plan not considering the opportunity of more levels of parallelism. This can be extended by applying Alg. 1 recursively on the plan subsequence computed to be parallelized. Finally and most importantly, the proposed approach does not guarantee to return the best parallel plan. Indeed, it acts as a greedy approach in searching for parallel terms to a parallelizable ac-

tion.

Despite these limitations, we believe that the proposed approach may be very useful in practical robot applications using planning technology. Moreover, some of these limitations can be addressed in a future extension of the approach.

7 Conclusions

In this paper, we have presented a novel algorithm exploited in order to obtain, starting from sequential plans, parallel plans in which parallelizable actions are executed simultaneously to a sequence of N (parameter that can be chosen) actions. We described the algorithm structure in detail, explaining that this method does not use a planning engine for parallelization, but exploits the planning domain specifications and a simulator computing states evolution and action applicability. We have shown the results obtained from an industrial robotic use-case, demonstrating that the proposed approach allows to reduce the plan execution time in an efficient way. We also discussed the limitations of this approach and its practical applicability in a set of real robotic problems.

The current solution will be soon deployed in the real industrial use case to actually measure its impact in this application. As future work, we intend to work to reduce the current limitations and to extend the proposed approach to other application domains, such as service robots and human-robot cooperative tasks, to parallelize robot actions with human operations.

References

Borrajo, D.; and Fernandez, S. 2015. Mapr and cmap. *Proceedings of the Competition of Distributed and Multi-Agent Planners (CoDMAP-15)*, 1–3.

Foulser, D. E.; Li, M.; and Yang, Q. 1992. Theory and algorithms for plan merging. *Artificial Intelligence*, 57(2-3): 143–181.

Fox, M.; Gerevini, A.; Long, D.; and Serina, I. 2006. Plan Stability: Replanning versus Plan Repair. In *ICAPs*, volume 6, 212–221.

Komenda, A.; Štolba, M.; and Kovács, D. 2016. The International Competition of Distributed and Multiagent Planners (CoDMAP). *AI Magazine*, 37: 109–115.

Kovács, D. L. 2012. A multi-agent extension of PDDL3.1. In *ICAPS-2012 Proc. of the 3rd Workshop on Distributed and Multi-Agent Planning*, 19–27.

Luis, N.; and Borrajo, D. 2015. PMR: Plan merging by reuse. *Competition of Distributed and Multi-Agent Planners (CoDMAP-15)*, 11.

Torreño, A.; Sapena, O.; and Onaindia, E. 2018. FMAP: A Platform for the Development of Distributed Multi-Agent Planning Systems. *Know. Based Syst.*, 145(C): 166–168.

Trapasso, A.; Santilli, S.; Iocchi, L.; and Patrizi, F. 2023. A formalization of multi-agent planning, with explicit agent representation. In *Proceedings of the 38th ACM/SIGAPP Symposium on Applied Computing*, SAC '23, 816–823. New York, NY, USA: Association for Computing Machinery. ISBN 9781450395175.

Veloso, M. M.; Perez, A.; and Carbonell, J. G. 2008. Nonlinear Planning with Parallel Resource Allocation.