2024 IEEE International Conference on Robotics and Automation (ICRA 2024)
May 13-17, 2024. Yokohama, Japan

A Method for Multi-Robot Asynchronous Trajectory Execution in MoveIt2

Pascal Stoop[1], Tharaka Ratnayake[2] and Giovanni Toffetti[3]

***Abstract*— This paper introduces a method that enables the parallel independent execution of trajectories for multi-robot / multi-arm systems in a shared workspace in MoveIt2. The proposed method leverages a centralized scheduler in a distributed set up to prevent collisions while the robots move independently. We argue that this approach is better suited than the state of the art (i.e., synchronous execution) for flexible/adaptive robotic tasks where the actions to be performed may vary in planning and execution time depending on sensor data (e.g., pick and place with inspection, assembly) as it is able to reduce the total execution time w.r.t. current approaches leveraging a single arm or multiple arms with synchronous motion planning.**

I. Introduction

In the context of multi-arm robots, *synchronous* and *asynchronous* execution refer to different ways of coordinating and controlling the movements and actions of robot arms. In **synchronous execution**, the robot arms perform their actions *simultaneously and in a coordinated manner*. In **asynchronous execution**, *each robot arm operates independently* and performs its actions without strict synchronization with other arms. That means that each arm can work independently, so that multiple arms can manage multiple uncoordinated tasks in parallel.

The most common library used for robotic arm motion planning and execution in ROS[1] is MoveIt[2] [1]. In this work we are focusing on the ROS2 Version of MoveIt, MoveIt2. The current implementation of MoveIt2 *supports multi-arm motion planning and execution out of the box*, exclusively with *synchronous execution*, meaning all robots collaboratively execute a common motion. In fact, in multi-arm manipulation tasks synchronous execution is the typical go-to solution, due to its deterministic behavior, simplified coordination, and ease of error handling.

Even if it is possible to plan and execute each robot motion independently, MoveIt is *not able to guarantee the collision-free execution of the trajectories of the different robots*, as each arm is unaware of the planned trajectories of others. To allow this, a shared entity should know about the other motion trajectories *while planning*, or run collision checks between different trajectories *while executing* them.

This work takes inspiration from the experience and ideas recounted by Felix von Drigalski in robotic assembly challenges [2] [3] and implemented in a thesis in 2022 [4].

[1]ILT, OST, Rapperswil SG, Switzerland pascal.stoop@ost.ch
[2]InIT, ZHAW, Winterthur, Switzerland raty@zhaw.ch
[3]InIT, ZHAW, Winterthur, Switzerland toff@zhaw.ch
[1]The Robot Operating System https://www.ros.org/
[2]https://moveit.picknik.ai

In this paper we argue, that despite years of research in optimization of the synchronous execution of multi-arm tasks, asynchronous execution offers several advantages in modern robotics. In fact, *synchronous execution is inherently less efficient than asynchronous for parallel tasks with different duration*, simply because the need of synchronized motion implies that the duration of a synchronized multi-arm task is always equal to the duration of its slowest single-arm task. This results in limited parallelism, hence not employing a multi-arm system to its full capacity.

The trivial objection to this line of reasoning is asking: why not use multiple arms independently without sharing the workspace? That would indeed just multiply the theoretical task throughput of a single arm N times in case of N arms. You'd have N arms working in parallel and by all means that is the maximum task throughput you could theoretically achieve.

Still, sharing the workspace for multiple arms can have advantages. The most obvious one is in terms of space occupation (i.e., more robots per square meter), but much more importantly, a multi-arm shared workspace allows the *flexibility of being able to execute both asynchronous and synchronous tasks when needed*. An example of when this is useful is for instance robotic assembly (i.e., some task require arms synchronously holding and fitting parts), or bin picking (e.g., some objects might need to be lifted by more than one grasping point simultaneously).

In this work, we propose a simple framework that allows reduced complexity and non-deterministic behavior in asynchronous execution to harness the advantages of combining synchronous and asynchronous execution.

The contributions of this paper can be summarized as follows:

- The implementation of a MoveIt2 extension to allow the execution of multi-arm asynchronous tasks;
- The experimental validation of the collision detection implementation enabling safe asynchronous execution in a shared workspace;
- The experimental validation of increased efficiency of asynchronous execution w.r.t. synchronous for an example use case requiring flexible behavior.

II. State of the Art

As tasks grow more complex, single-arm robots face limitations in handling large rigid objects, multi-object interactions in assemblies, non-rigid object deformations, and tool manipulation relative to other objects [5]. Multiple-arm robots, such as the PR2 and YuMi, have been developed to

address these challenges, along with the utilization of parallel individual robot arms. There are numerous works in literature on multiple-arm robots and we focus on planning, execution and collision checking.

A. Collision checking

Collision checking is crucial for manipulators to prevent collisions with obstacles in the workspace, avoid self-collisions between the manipulators links and joints, prevents collisions with tools or manipulated objects, optimizes workspace utilization and ensures human safety in collaborative scenarios. Collision detection algorithms aim to achieve output sensitivity in real-world scenarios where only a small number of polygons collide. They strive to minimize the number of O(n^2) primitive tests by quickly excluding (filtering) large portions of objects that cannot possibly collide. [6] Modern physics simulation libraries such as PhysX [7], Bullet [8], and ODE [9] incorporate various levels of filtering in their collision detection pipelines to enhance performance and accuracy.

Discrete Collision Detection (DCD) focuses on detecting collisions between objects at specific points in time or discrete time steps. It checks for collisions between objects' current positions or bounding volumes without considering their motion between those positions. Continuous Collision Detection (CCD) is concerned with detecting collisions between objects during their continuous motion. It handles cases where objects are in motion and may pass through each other within a single frame or time step by considering the objects' trajectories and sweeping their positions between discrete time steps/robot states. [10].

Conservative advancement proposed by Mirtich [11] involves iteratively advancing the objects by a fixed time-step, ensuring that a non-penetration constraint is maintained throughout the simulation. This approach helps prevent objects from intersecting or penetrating each other during their motion.

Another approach in collision detection is to enclose the objects with bounding volumes at the start and end of each motion step. This is achieved by using a swept volume, which efficiently represents the space occupied by the object during its motion. Different types of bounding volumes can be used, such as Axis-Aligned Bounding Boxes (AABBs) [12], velocity-aligned Discrete Oriented Polytopes (DOPs) [13], Oriented Bounding Boxes (OBBs) [14], sphere swept convex hulls [14], or even ellipsoids [15]. These bounding volumes serve as approximations of the object's shape and help in quickly identifying potential collisions during the motion.

Furthermore, collision detection can be categorized based on whether it is performed online or offline. *Online collision detection* involves detecting collisions in real-time during the simulation or execution of a system. It continuously checks for collisions and responds to them immediately as they occur. It is commonly used in interactive applications, video games, and robotics where real-time responsiveness is crucial.

Offline collision detection refers to the process of pre-computing collision information or performing collision detection as a separate step before the actual simulation or execution. It is commonly used when the environment or objects are static or change infrequently. The pre-computed collision information is then used during the simulation to improve performance by avoiding redundant collision checks.

B. Multi-arm motion planning

Multi-arm motion planning refers to the problem of planning the motions of multiple robotic arms in order to achieve specific tasks or objectives. This is a challenging problem because it involves coordinating the motions of multiple arms while considering various constraints such as collision avoidance, joint limits, and task-specific requirements.

There are two main approaches of planning and execution: synchronous and asynchronous. In synchronous planning, all arms are planned and controlled simultaneously, ensuring coordination and synchronization. Synchronous movement can be achieved by planning or execution [16]. It works well for tasks requiring close collaboration but can be complex and limiting. Kimmel and Bekris [17] proposes a method of dual-arm concurrent motion where time coordination is enforced between the arms during movement. This approach involves using a velocity tuning technique where each arm independently solves its motion path and coordinates their movements by finding a velocity that enables them to occupy the same space without colliding.

Buhl *et al.* [18] proposes a collaborative dual-arm robot platform which considers the environment as static. This results in only one arm planning and executing at a given time. Even if this guarantees the safe operation of arms, it cannot harness the full potential of dual arms.

Another approach proposed by Wang *et al.* [19] aims to synchronize the movements of two arms, ensuring collision-free motion by using a motion planner to generate plans for both arms at the same time. While this allows for concurrent movements, it restricts flexibility as the arms must remain synchronized, leading to possible waiting times if one arm completes its motion before the other.

Asynchronous planning involves independent planning and control of each arm, providing flexibility, but requiring collision avoidance mechanisms and potentially sacrificing coordination. Salehian el al. [16] proposed a method that employed a centralized inverse kinematics solver with self-collision avoidance constraints. These constraints were determined by learning a continuous and differentiable function from a dataset of collided and non-collided multi-arm configurations. This function determined the feasibility of robot configurations and was used to identify potential collisions.

Moving two arms within the same workspace can be approached as a multi-agent problem, where each arm acts as an independent agent. This relates to the concept of asynchronous motion planning for multiple agents with safety guarantees, as explored by Grady *et al.* [20]. In their approach, agents operate in cycles of fixed duration, where

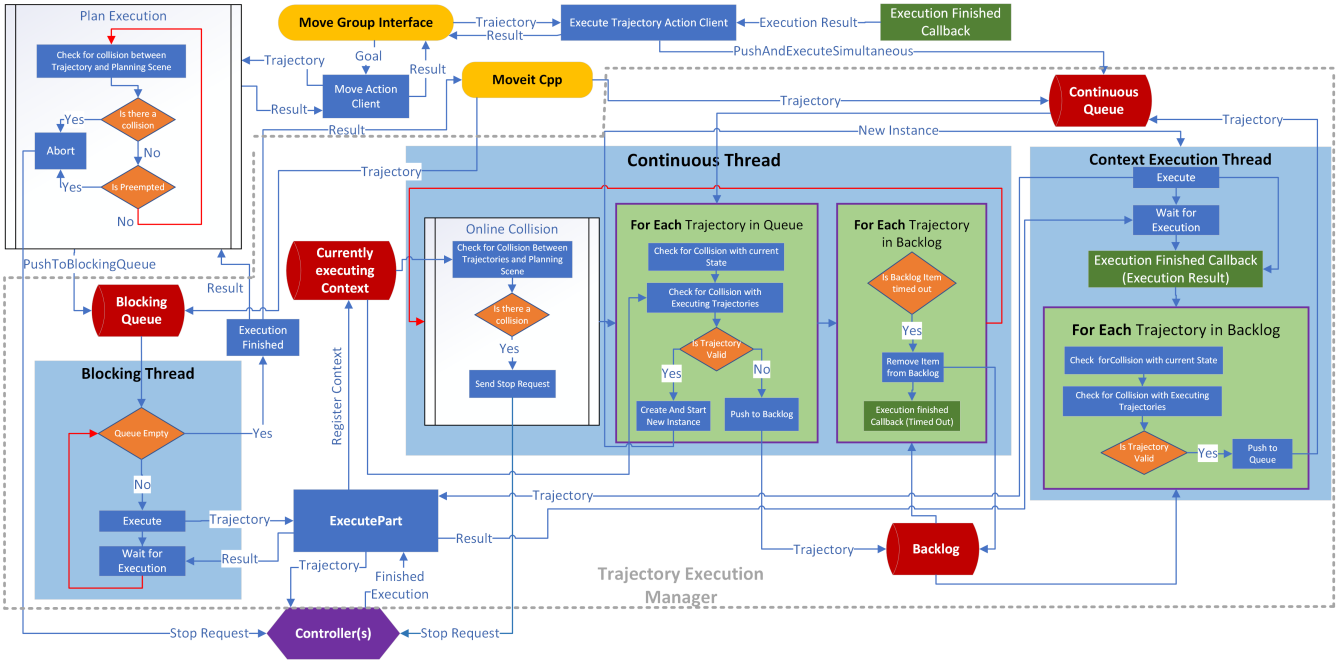

Fig. 1. Proposed Trajectory Execution Manager.

each agent plans multiple potential trajectories and listens for broadcasts from neighboring agents. As the cycle nears completion, an agent selects a conflict-free trajectory based on received information and broadcasts its chosen trajectory to reserve the corresponding space.

Meehan *et al.* [21] proposes an approach for asynchronous motion execution utilizing a trajectory reservation component along with off-the-shelf motion planning to prevent collisions between multiple arms. By creating collision objects in the environment corresponding to planned trajectories, the projected space for each arm is reserved. This however limits the reachable space for other arms, while the method we propose uses an online collision detector to prevent the execution of colliding trajectories.

Pecora *et al.* [22] propose a multi-arm collision-free coordinated motion using pure trajectory planning. The idea being that expensive multi-arm path planning can be replaced with planning for collision-free movement by adjusting the timing of path execution without changing it.

III. Approach

The main interface in MoveIt the *Move Group*, which represents a subset of joints and links of the robot that the planner will use to generate plans. While the planner focuses on the specified move group, it considers other parts of the robot not in the same group to be static during planning. The *Planning Scene* collects the current robot states as well as the states of other collision objects and provides collision checking between the robots and other objects that are registered in the planning scene itself.

The *Trajectory Execution Manager* executes generated plans. Unlike the planner, which is exclusively selected and executed for each move group, the execution manager is a shared entity, which allows only a single trajectory to be executed at the same time. It is responsible for validating the trajectory, splitting trajectory plans into sub-trajectories for each controller and sending them to those controllers.

With the current implementation of MoveIt2, it is only possible to do synchronized movements as, for safety reasons, only a single trajectory can be executed at the same time. In order to support asynchronous execution, we introduced the following changes to MoveIt:

- Collision detection strategies for multiple trajectories
- Central scheduler ensuring collision free execution
- Online collision detection

A. Central scheduler

To allow the trajectory execution manager to handle multiple trajectories, the blocking execution that was previously present has been replaced with a continuous execution queue (top right in Figure 1). Any new trajectory t_n added to the queue will be collision checked against all currently running trajectories $t_r \in T$. If no collision is expected, t_n will be scheduled for immediate execution and the execution manager will store its information for future collision avoidance. Instead, in case of expected collision, t_n is added to a backlog queue (bottom right in figure), similarly to the implementation suggested by Felix von Drigalski in [2]. As soon as the running trajectory t_r that was causing a collision with t_n is finished, t_n will be moved from the backlog into the continuous queue and checked against collisions to be executed again. To reduce deadlock chances and minimize execution duration, all trajectories are assigned a backlog timeout. If the execution manager is not able to schedule a

trajectory within said timeout, the trajectory execution aborts, allowing the user to re-plan.

B. Collision Detection

Rather than limiting the motion space available for planning as done in 'sweeping" methods for collision avoidance (as done in e.g., [21]), we chose to check for collision in a time dependent manner in the execution context.

The default collision checking library in MoveIt2 is FCL (Flexible Collision Library), which allows for discrete collision detection for a single robot state. To be able to check for collision in a time-variant manner, we will have to discretize the trajectory and check for every discrete time step. At first we tried evaluating collisions between a trajectory's discrete robot state and the interpolated state of another trajectory using a fixed default time step, but this led to undetected collisions, so we resorted to use a configurable parameter for the time step used to break trajectories into discrete positions. This allows to tune the trade-off between performance and operational safety.

This approach is still of course sub-optimal, as it won't detect collisions between discrete time-steps. To solve this issue, Continuous Collision Detection (CCD) could be used in a future implementation.

C. Online collision detection

MoveIt incorporates real-time trajectory validity assessment using collision checking for single trajectories, yet collision checks only consider a single trajectory as well as the continuously updated planning scene. This design, meant to account for a single executing trajectory, may incur in a high execution overhead due to extensive collision evaluations. A more efficient approach is sought to accommodate multiple trajectories while expanding collision checks between executing trajectories and the planning scene. Our proposed solution consolidates all robot states from concurrent trajectories at discrete intervals, subjecting this combined state to self-collision checks and evaluations against the planning scene. Discretization strategies to balance efficiency and safety vary, with the preferred approach involving periodic collision checks with planning scene updates.

IV. Discussion

The proposed implementation allows to flexibly combine both synchronous and asynchronous motion execution to perform robotic tasks. While synchronous execution is the current go-to approach, we'd like to discuss under which conditions asynchronous execution brings advantages.

First of all, one has to consider that the asynchronous approach requires individual planning per each arm resulting in additional computational costs (e.g., in a dual-arm system it would incur in twice as many planner invocations as sync). Further computational cost is due to collision avoidance which might trigger re-planning. Finally, albeit timeouts, independent motion execution might still result in arm configurations (e.g., one extended arm "on top" of another) requiring resolution through a *recovery policy*.

On the other hand, when individual arm tasks have different execution duration, the async approach reduces total execution time due to independent parallel execution and "load balancing" across arms (i.e., an arm executing "shorter" tasks will perform more than one performing "longer" tasks).

Such scenarios are quite common, for example in multi-arm assembly tasks, garbage sorting with specific treatments for different materials, bin picking with conditional visual inspection based on sensor data. For a more concrete example, and as validation, we introduce such a scenario in the following subsection.

A. Example Use Case

Let's assume we have a *dual-arm* bin picking scenario where N objects in the bin are randomly stacked on top of each other, and only the few topmost ones can be seen and picked. Objects are detected by a camera and, based on some visual criteria, may require just picking and placing in other bins or picking with further inspection (i.e., rotating them in front of a camera before placing).

To simplify things, let's assume objects are just red (R) and blue (B) cubes, blue ones require inspection, and the probability of a cube being red is α (see Figure 2). Pick and place of R cubes takes t_1 seconds, while B cubes require t_2 seconds, δ represents the "task difference" due to the inspection step (i.e., $t_2 = t_1 + \delta$).

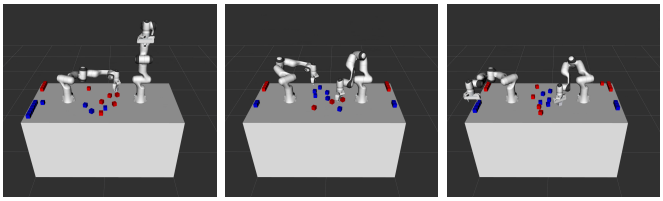

Fig. 2. baseline with only one arm *[left]*, two arms synchronous movements *[middle]* and two arms asynchronous movements *[right]*

B. Asynchronous vs Synchronous Approach

In an asynchronous picking approach, each arm would work independently and in parallel, so we can approximate the total execution time for one bin as:

$$T_{async} \simeq \frac{N}{2}(\alpha t_1 + (1-\alpha)t_2) + \lambda(N) \qquad (1)$$

Where λ represents the delay due to collision checking and added planner invocations over all the pick and place operations (also a function of N), and we assume that each arm roughly picks half of the objects.

In the synchronous approach, each pick and place will use both arms and pick two objects. Depending on which objects are visible and reachable at each point in time, we have four possible combinations of dual picks: RR, RB, BR, BB. The RR combination will require t_1 execution time, while all others will need t_2. We can approximate the total bin processing time as:

$$T_{sync} \simeq \frac{N}{2}(\alpha^2 t_1 + (1-\alpha^2)t_2) \quad (2)$$

In this simplified scenario, the time saved from using asynchronous execution would be:

$$T_{sync} - T_{async} \simeq \frac{N}{2}(\alpha - \alpha^2)\delta - \lambda(N) \quad (3)$$

Although this is a simplified analysis of an example use case, the following general principles will hold independently of actual distributions and combinations of picks: the time saved with async grows with the number of picks N and the task difference δ, while the delay $\lambda(N)$ due to collision checking, added planner invocations for re-planning and retries for each arm independently will reduce the advantage.

C. Generalized Approach

We can generalize to multi-arms: Assume we have M robotics arms and we need to complete N tasks from a set of tasks T, where t_i represent the time it would take to complete task i for a single robot in non-shared workspace.

Due to the independent execution and load balancing effects, for the asynchronous execution we could write total execution time as

$$T_{async} \simeq \frac{N}{M} \times \{Average\_task\_time\} + total\_delay \quad (4)$$

$$T_{async} \simeq \frac{N}{M} \times \{\frac{1}{N} * \sum_{i=0,j=0}^{N,M} t_i\} + \lambda(T, M) \quad (5)$$

where $\lambda(T, M)$, represents added delay due to collision avoidance, planning invocations, and re-planning. λ is a function of the tasks set and the number of arms, however it is hard to characterize further due to the fact it's highly application-specific. As a rule of thumb, we should expect λ to grow with the number of arms and, more specifically, with the workspace volume shared by multiple arms while executing the expected tasks as well as the application-specific deadlock recovery policy.

Beyond the contributions of this paper, which is just providing an implementation to support asynchronous execution in MoveIt, the choice between synchronous and asynchronous execution (or their combination) should be subject to higher level considerations involving task planning optimization, treated thoroughly and more in general in works such as [22]. The expectation here is that for pure automation (i.e., no / low uncertainty) total execution time minimization can be obtained by global task order optimization (e.g., using tools such as MoveIt Task Constructor) accommodating all M arms. In other terms, MTC could be used to choose the combination of synchronous and asynchronous motions (and the number of arms involved) to minimize execution time.

On the other hand, when uncertainty / conditional behavior is present (like in our use case example), a naive approach of independent arm planning, collision detection and eventual re-planning with a recovery policy to prevent deadlocks, could yield better results then a purely synchronous approach.

V. Validation

We devised experiments to: 1) validate that the implemented solution prevents collisions and deadlocks; 2) demonstrate the expected advantage of using the asynchronous approach vs synchronous in an example use case; 3) estimate $\lambda(N)$ for our use case and hence assess how close we are to a theoretically "optimal" result, which would involve employing two arms working independently in separate workspaces.

A. Experimental Setup

The proposed implementation was evaluated using two 7DoF PANDA robot arms in rviz. The experimental setup is shown in Figure 2. Arms are placed on opposite sides of a workbench each with access to the entire workspace. Both robotic arms are given predefined home configuration at the start in all cases. The robot arms pick cubes from the center of the workspace and place them in color ordered trays on the sides. The code for the experiments is available online[3].

To simulate objects "appearing" at the top of the bin while other are removed, we used the following algorithm: At the beginning, the workspace will include 6 cubes of random colors. Afterwards, it will spawn 5 cubes in a random position once the batch is picked until a selected amount of objects have been spawned. Different amounts of objects have been evaluated in order to gauge how the total execution time varies in function of the number of picks.

We used pseudo-random numbers to provide the same objects appearance patterns and repeated the same experiments for the following scenarios: *a) one arm*, this is our "baseline" scenario; *b) two arms synchronous movements* synchronous planning and execution of both arms; *c) two arms asynchronous movements*.

Each pick and place operation has six atomic steps, with each step triggering a plan and execute sequence: pre-grasp, grasp, move, put down, and post-move and a inspection. The mean planning and execution time for the pick and place operation of red cubes is 25s.

Since we could arbitrarily choose both the number of items to be picked N and the additional time δ required for the inspection task, to ease repeatability, we decided to keep N small (i.e., 12, 20 picks) and chose values for δ around the break-even point between synchronous and asynchronous execution (i.e., 6, 7.5, and 10 seconds).

The system's performance was evaluated based on task completion time, and the number of collisions in asynchronous execution against the two other cases.

B. Results

The experimental data in Table I shows the average total execution time in seconds and standard deviation over 10 runs for the given pick and place operations.

As we mentioned in Section I a theoretical "best-case" scenario for our use case would be to use two independent arms each with its own workspace, thus avoiding any risk

[3]https://github.com/stooppas/moveit2_asynchronous_trajectory_execution

TABLE I

TASK DIFFERENCE VS TOTAL EXECUTION TIME

			12 picks			20 picks		
			Baseline	***Sync***	***Async (Proposed)***	***Baseline***	***Sync***	***Async (Proposed)***
Task Difference (δ)	6s	Mean	278.58392	**209.900**	211.943	517.31543	396.454	**352.746**
		SD	5.117	**3.249**	3.694	3.541	3.774	**3.695**
	7.5s	Mean	289.212	229.360	**216.455**	537.537	401.699	**374.731**
		SD	3.550	1.070	**3.305**	4.588	9.051	**8.334**
	10s	Mean	306.21013	242.41	**222.802**	555.126	414.305	**378.101**
		SD	3.845	6.157	**4.772**	8.282	5.064	**3.921**

TABLE II

TIME PER PICK (WITH $\delta = 6s$)

number of picks		12	20
baseline (s)	Mean	23.215	25.866
	SD	0.426	0.177
synchronous (s)	Mean	17.492	19.823
	SD	0.271	0.189
asynchronous (s)	Mean	17.662	17.637
	SD	0.308	0.185

of collision and need for synchronization. To approximate that scenario we can use the baseline case and divide total execution time by two, this gives us a lower bound for an optimal total execution time.

For 12 picks the lower-bound execution time would be roughly $139s$ (i.e., 278/2), while for sync we have $209s$ and for async $212s$. It is important to understand the reasons why there is such a large difference between the lower bound and the measured execution times.

Our experimental logs show that the main source of delay in the sync policy is due to controllers synchronization. To ensure synchronous behavior, the move group controller waits for all arm controllers to return successfully after sending a goal to each ros controller. As expected, when task duration differ for the two arms the move group controller sits waiting for the straddler. Moreover, in our setup, we noticed a fixed time cost per controller invocation of roughly between 1 and 2 seconds, this also gets compounded when doing synchronized motions.

For async instead, extra execution time is what we called λ, the delay of re-planning due to collision avoidance and the recovery policy employed to avoid deadlocks. From our experimental logs, the primary factor impacting on delay was our recovery policy. We kept it intentionally simple for implementation purposes: in case of deadlock (both arms not able to execute any queued motion) we tried re-planning one arm three times and if that didn't work we'd move the arm to initial state. Our results show that both re-planning and moving to home configurations are time consuming. Smarter (e.g., see [22]), possibly application-specific, recovery policy schemes could be adopted to further reduce λ.

Finally, we can compare the performance of sync vs async for our experimental runs. We cross the break-even point of T_{sync} and T_{async} between values of δ of 6s and 7.5s for 12 picks. For longer pick sequences (e.g., 20 picks) we have async consistently achieving lower total execution time than sync. This tells us that, even with our simple recovery policy, $\lambda(N)$ in Equation 3 is asymptotically dominated (at least for this use case and implementation) by the term accounting for task time difference δ. Our explanation for this is that (full) recoveries are relatively infrequent, and with longer picking runs the work-balancing effect of independent arm motion becomes more apparent.

C. Threats to Validity

The discrete collision checking approach, although efficient, still leaves room for potential errors in ensuring collision-free execution. To achieve greater reliability for industrial applications, implementing continuous collision detection (CCD) for self-collision becomes crucial. This would enhance the robustness of the system and provide a more solid foundation for safe trajectory execution.

VI. CONCLUSION AND FUTURE WORK

This work introduced a method to enable the parallel independent execution of trajectories for multi-robot / multi-arm systems in a shared workspace in MoveIt2. We discussed potential advantages of combining synchronous and asynchronous trajectory executions enabled by the proposed method. We provided an application use case example and used to validate the implementation and provide a preliminary performance evaluation.

Future work will focus on including CCD, optimizing collision checking for new trajectories inserted in the execution queue to only look for collisions among trajectories that are to be executed in parallel, and providing a more extensive experimental evaluation using real hardware.

REFERENCES

[1] D. Coleman, I. A. Sucan, S. Chitta, and N. Correll, "Reducing the barrier to entry of complex robotic software: a moveit! case study," *CoRR*, vol. abs/1404.3785, 2014. [Online]. Available: http://arxiv.org/abs/1404.3785

[2] F. von Drigalski, C. Schlette, M. Rudorfer, N. Correll, J. C. Triyonoputro, W. Wan, T. Tsuji, and T. Watanabe, "Robots assembling machines: learning from the world robot summit 2018 assembly challenge," *Advanced Robotics*, vol. 34, 2020.

[3] F. von Drigalski, C. Nakashima, Y. Shibata, Y. Konishi, J. C. Triyonoputro, K. Nie, D. Petit, T. Ueshiba, R. Takase, Y. Domae, T. Yoshioka, Y. Ijiri, I. G. Ramirez-Alpizar, W. Wan, and K. Harada, "Team o2as at the world robot summit 2018: an approach to robotic kitting and assembly tasks using general purpose grippers and tools," *Advanced Robotics*, vol. 34, 2020.

[4] P. Stoop. (2022) Multi-arm robotic tasks progamming. [Online]. Available: https://digitalcollection.zhaw.ch/

[5] P. Brett and K. Khodabandehloo, "Definition and application of multi-arm robots," in *IEE Colloquium on Multi-Arm Robotics*, 1992.

[6] R. Weller, *A Brief Overview of Collision Detection*. Heidelberg: Springer International Publishing, 2013, pp. 9–46.

[7] ageia. (2022) Ageia-physx. [Online]. Available: https://ageia-physx.software.informer.com/

[8] E. Coumans. (2012) Bullet physics library. [Online]. Available: https://pybullet.org/wordpress/

[9] R. Smith. (2014) Open dynamics engine. [Online]. Available: http://www.ode.org

[10] M. Held, J. T. Klosowski, and J. B. M. Mitchell, "Collision detection for fly-throughs in virtual environments," in *SCG '96*, 1996.

[11] B. Mirtich, "Timewarp rigid body simulation," in *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH '00, 2000, p. 193–200.

[12] G. Zachmann, *Collision Detection as a Fundamental Technology in VR Based Product Engineering*, 2008, pp. 195–230.

[13] D. S. Coming and O. G. Staadt, "Velocity-aligned discrete oriented polytopes for dynamic collision detection," *IEEE Transactions on Visualization and Computer Graphics*, vol. 14, no. 1, pp. 1–12, 2008.

[14] S. Redon, A. Kheddar, and S. Coquillart, "Fast continuous collision detection between rigid bodies," *Computer Graphics Forum*, vol. 21, 05 2002.

[15] X. Jia, Y.-K. Choi, B. Mourrain, and W. Wang, "An algebraic approach to continuous collision detection for ellipsoids," *Computer Aided Geometric Design*, vol. 28, no. 3, pp. 164–176, 2011.

[16] S. S. M. Salehian, N. Figueroa, and A. Billard, "A unified framework for coordinated multi-arm motion planning," *The International Journal of Robotics Research*, vol. 37, no. 10, pp. 1205–1232, 2018.

[17] A. Kimmel and K. E. Bekris, "Scheduling pick-and-place tasks for dual-arm manipulators using incremental search on coordination diagrams," 06/2016 2016.

[18] "A dual-arm collaborative robot system for the smart factories of the future," *Procedia Manufacturing*, vol. 38, pp. 333–340, 2019, 29th International Conference on Flexible Automation and Intelligent Manufacturing (FAIM 2019), Beyond Industry 4.0: Industrial Advances, Engineering Education and Intelligent Manufacturing.

[19] J. Wang, S. Liu, B. Zhang, and C. Yu, "Inverse kinematics-based motion planning for dual-arm robot with orientation constraints," *International Journal of Advanced Robotic Systems*, vol. 16, no. 2, 2019.

[20] D. Grady, K. Bekris, and L. Kavraki, "Asynchronous distributed motion planning with safety guarantees under second-order dynamics," vol. 68, 01 2010, pp. 53–70.

[21] M. R. Charles A. Meehan and L. M. Hiatt. (2022) Asynchronous motion planning and execution for a dual-arm robot. [Online]. Available: https://icaps22.icaps-conference.org/workshops/PlanRob/PlanRob/2022/paper_11.pdf

[22] S. Zhang and F. Pecora, "Online and scalable motion coordination for multiple robot manipulators in shared workspaces," *IEEE Transactions on Automation Science and Engineering*, pp. 1–20, 2023.