

# ScheduleStream: Temporal Planning with Samplers for GPU-Accelerated Multi-Arm Task and Motion Planning & Scheduling

Caelan Garrett<sup>1</sup> and Fabio Ramos<sup>1,2</sup>

**Abstract**—Bimanual and humanoid robots are appealing because of their human-like ability to leverage multiple arms to efficiently complete tasks. However, controlling multiple arms at once is computationally challenging due to the growth in the hybrid discrete-continuous action space. Task and Motion Planning (TAMP) algorithms can efficiently plan in hybrid spaces but generally produce plans, where only one arm is moving at a time, rather than schedules that allow for parallel arm motion. In order to extend TAMP to produce schedules, we present ScheduleStream, the first general-purpose framework for planning & scheduling with sampling operations. ScheduleStream models temporal dynamics using hybrid durative actions, which can be started asynchronously and persist for a duration that's a function of their parameters. We propose domain-independent algorithms that solve ScheduleStream problems without any application-specific mechanisms. We apply ScheduleStream to Task and Motion Planning & Scheduling (TAMPAS), where we use GPU acceleration within samplers to expedite planning. We compare ScheduleStream algorithms to several ablations in simulation and find that they produce more efficient solutions. We demonstrate ScheduleStream on several real-world bimanual robot tasks at <https://schedulestream.github.io>.

## I. INTRODUCTION

There's been a spike in demand for bimanual and humanoid robots to be used to automate tasks in both industrial and home settings, in place of traditional manipulators. However, programming these robots to take full advantage of their multiple arms through simultaneous manipulation remains challenging. Imitation learning from human teleoperation demonstrations can be an effective control approach but requires an enormous amount of data, and thus human effort, even for narrowly-defined tasks [1]. In contrast, automated robot planning, such as Task and Motion Planning (TAMP) [2], is able to plan behaviors that generalize across a broad distribution of tasks through composing a set of engineered or learned primitive actions. Additionally, planning can be used to develop imitation learning systems by serving as an automated training data generator [3].

In order to safely and productively control robots, TAMP algorithms must make discrete decisions, such as which arm to use to manipulate which object and how, as well as continuous decisions in the form of the motion and other parameters that lead to executable robot actions. Consider the bimanual robot in Figure 1, which is tasked with sorting the two fruits into the two bins. In order to maximize execution efficiency, a TAMP algorithm should plan to use both arms in parallel, but critically, it must identify which arms can reach which objects by reasoning about reachability.

<sup>1</sup>NVIDIA Research Seattle Robotics Lab (SRL)

<sup>2</sup>University of Sydney, {cgarrett, ftozetoramos}@nvidia.com

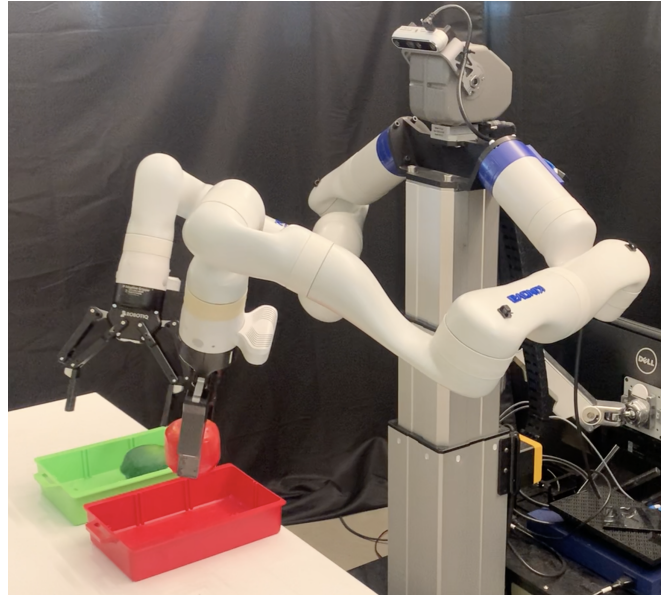


Fig. 1: **Real-World Demonstration.** Using ScheduleStream, a bimanual robot solves for a schedule to sort the apple into the red bin and the lime in the green bin. ScheduleStream algorithms automatically select which arm to use for which object based on kinematics and execute the actions asynchronously in parallel.

Most existing TAMP approaches [2] are only able to produce *plans*, serial sequences of actions, as opposed to *schedules*, sets of timed actions that can be executed in serial or parallel. As a result, they will only produce solutions that have one arm moving at a time, preventing them from using both arms in parallel to minimize schedule *makespan*, or duration. The existing work that incorporates multi-arm parallel manipulation typically is application-specific [4], [5], [6], addresses a strict subclass of TAMP [7], [8], or has limitations in its ability to leverage parallelism [9], [10], [11].

In this work, we seek to address full Task and Motion Planning & Scheduling (TAMPAS), where solutions are schedules of timed hybrid actions. Unlike all prior work, we accomplish this by first introducing ScheduleStream, a new domain-independent framework for planning & scheduling with samplers. We implement this language in the programming language Python, allowing external procedures such as GPU-accelerated collision checkers and inverse kinematics solvers to flexibly be invoked by ScheduleStream algorithms. We propose domain-independent ScheduleStream algorithms that alternate between scheduling and sampling phases, which solve each scheduling subproblem through planning the start and end of each action. In simulated experiments, we show that these algorithms have higher success rates than

strictly hierarchical algorithms that schedule *then* motion plan [4] and lower makespans than serial-only planning algorithms. **The contributions of this paper are:**

- ScheduleStream: the first domain-independent *temporal* planning language with support for procedural functions and samplers (*i.e.* streams) to enable planning & scheduling for mixed discrete-continuous systems.
- Novel ScheduleStream algorithms that *lazily* solve problems while minimizing schedule makespan.
- An application of ScheduleStream to Task and Motion Planning & Scheduling (TAMPAS), which leverages GPU acceleration for *parallelized* sampling.
- Simulated and real-world experiments and demonstrations across multiple multi-arm platforms, including a real-world bimanual robot.

## II. RELATED WORK

We build on prior work in single-arm Task and Motion Planning (TAMP) as well as multi-arm manipulation.

### A. Task and Motion Planning

There are a number of existing approaches for traditional TAMP that can efficiently produce single-arm plans but are unable to produce multi-arm schedules [2]. Several approaches apply TAMP to humanoid robots [12], [13]; however, they only address single-arm loco-manipulation rather than bimanual manipulation. cuTAMP [14] proposed leveraging GPU parallelization for single-arm sampling and gradient-based TAMP. The closest to our approach is PDDLStream [15], which also is a domain-independent planning framework for hybrid systems. Key differences from ScheduleStream are that PDDLStream describes domains using text (*i.e.* PDDL) instead of code (*i.e.* Python) and does not support temporal, functional, or GPU-batched planning.

### B. Multi-Arm Manipulation

Several prior works plan for multiple arms in automated assembly, an application of TAMP that makes various assumptions about task allocations, monotonicity, single object goals [4], [5], [6]. Many of these approaches also factor discrete and continuous planning into a strict hierarchy and will fail to solve problems that are not downward refinable [16], problems where some high-level plans aren't viable.

Several approaches address *synchronous* multi-arm manipulation [9], [17], [10], where each robot takes an action that starts and ends simultaneously according to a global clock. Such synchronicity requirements restrict the space of schedules, limiting opportunities to minimize schedule makespan. Still, many traditional TAMP systems can implement this strategy by treating the arms collectively as a single composite robot [18], [19]. However, these strategies discard factoring and often sparse interaction among robots, leading to combinatorial growth in their search spaces.

Two approaches investigate multi-arm handover tasks. Shome *et al.* [7] extend dRRT\* [20], a sampling-based multi-robot motion planner, and Wei *et al* [8] combine Linear Temporal Logic (LTL) and Graph of Convex Sets (GCS)

in a hierarchical manner. However, these planners would need to be continuously extended to address more general manipulation tasks, for example, ones where objects aren't limited to a pre-determined set of placements.

Hartmann and Toussaint [21] develop a rescheduling algorithm that greedily reschedules an inputted serial plan subject to conflicts to locally minimize its makespan. The Decomposable State Space Hypergraph (DaSH) line-of-work [22], [11] also presents several algorithms that explicitly factor problems into separate planning spaces per robot and later resolve conflicts to produce a schedule. Unlike ScheduleStream, these algorithms are application-specific and don't directly minimize schedule makespan.

## III. SCHEDULESTREAM LANGUAGE

We start by introducing ScheduleStream, a new general-purpose language for planning & scheduling with continuous sampling operations. We will use a simplified bimanual manipulation domain as a running example. Figure 2 visualizes three cartoon problems in this domain, where there's a bimanual robot with two arms, `arm1` and `arm2`, and the goal in each problem is for `arm1` to hold object `obj1` and `arm2` to hold object `obj2`. We build ScheduleStream entirely in the Python programming language, which, unlike prior works [23], [15], enables direct interfacing between the declarative and procedural language aspects.

We consider a universe of possibly infinitely many *constants*, atomic values. In our bimanual domain, there are 1) *discrete* constants such as first-class objects for the arms `arm1`, `arm2` and the objects `obj1`, `obj2` and 2) infinitely-many *continuous* constants in the form of (PyTorch) tensors for arm configurations  $q \in \mathbf{R}^d$ , object placement poses  $p \in \text{SE}(3)$ , and object grasp poses  $g \in \text{SE}(3)$ .

Similar to Functional STRIPS [24], we describe states and actions using declarative *functions*, relations between a tuple of input constants and a single output constant. This subsumes predicate-based languages, such as STRIPS [25] and Planning Domain Definition Language (PDDL) [23], which only consider Boolean functions. Each function **Function**, or predicate **Predicate**, is defined by a string of input *parameters* (e.g. "`?x1 ... ?xn`") and optionally a domain *condition* `cond` that specifies predicates that the input parameters are guaranteed to satisfy. For example, let `At` be a function from an arm "`?arm`" parameter that satisfies typing predicate `Arm(" ?arm")` to its current configuration. And let `Conf(" ?arm ?q")` be **True** if "`?q`" is a valid configuration for arm "`?arm`" in some state.

```
At = Function("?arm", cond=[Arm("?arm")])
Arm = Predicate("?arm"); Conf = Predicate("?arm ?q")
```

States are assignments to the functions in the domain. For example, the *assignment* `At (arm1) <= q1` declares that function `At (arm1)` with argument `arm1` currently evaluates to configuration `q1`. We make a version of the *closed-world* assumption in that any undeclared function and predicate evaluate to **None** and **False** respectively. We can also *test* whether a function evaluates to a particular constant using an equation `At (arm1) == q1`, which evaluates to either **True**

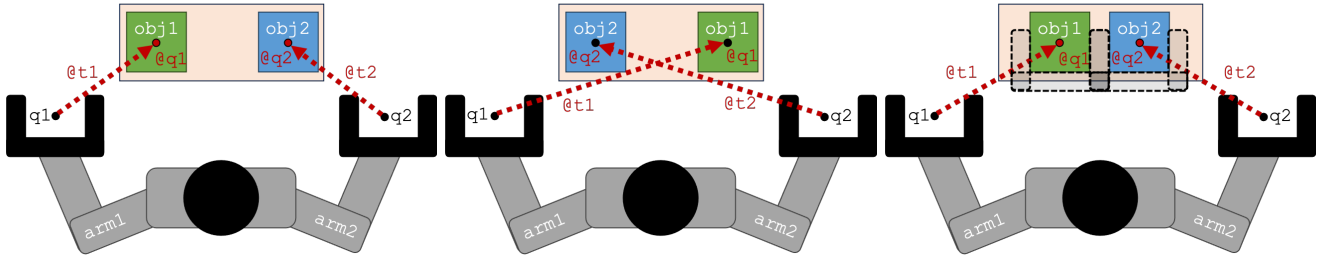


Fig. 2: **Illustrative Examples.** Three example bimanual TAMPAS problems where the goal is for arm arm1 to hold object obj1 and arm arm2 to hold object obj2. *Problem 1 (left)*: the arms can pick their objects in parallel. *Problem 2 (center)*: the arms can only pick their objects sequentially. *Problem 3 (right)*: one arm must retreat after picking its object in order to make space for the other.

or **False**. In our Python implementation, we override the `<=` and `==` operations for modeling syntactic ease.

Often, some functions are *static*, meaning that their values are fixed over time, and others are *fluent*, meaning that their values change over time as actions are applied. In our bimanual domain, the static object-related predicates are `Object("?obj")` and `Placement("?obj ?p")`, which hold if `"?obj"` is an object and `"?p"` is a valid placement pose for object `"?obj"` respectively.

```
Object = Predicate("?obj")
Placement = Predicate("?obj ?p")
```

The fluent object-related functions are `Holding("?arm")` which indicates which object the `"?arm"` is holding, `Attached("?obj")` which indicates which arm the `"?obj"` is attached to, and `Pose("?obj")` the current pose of `"?obj"`.

```
Holding = Function("?arm", cond=[Arm("?arm")])
Attached = Function("?obj", cond=[Object("?obj")])
Pose = Function("?obj", cond=[Object("?obj")])
```

Returning to the problems in Figure 2, let  $q_1, q_2 \in R^d$  be the start configurations for arms arm1, arm2 and  $p_1, p_2 \in SE(3)$  be the start placements for objects obj1, obj2. Then, the initial state function assignments are:

```
initial_state = [At(arm1)<=q1, At(arm2)<=q2,
  Holding(arm1)<=None, Holding(arm2)<=None,
  Pose(obj1)<=p1, Pose(obj2)<=p2,
  Attached(obj1)<=None, Attached(obj2)<=None, ...]
```

and the goal is the logical conjunction expressed as a list that arm1 is holding obj1 and arm2 is holding obj2.

```
goal = [Holding(arm1)==obj1, Holding(arm2)==obj2]
```

### A. Instantaneous Actions

In order to describe *actions*, operations that change the current state, we adopt a precondition/effect representation similar to that of STRIPS [25] or PDDL [23]. An action **Action** has a string of input *parameters* `params`, a list of equality *conditions* `cond`, and a list of assignment *effects* `eff`. In order to apply an action, the conditions must hold in the current state. After applying an action, the state is updated according to the effect assignments.

In our running example, we model `pick` and `place` as actions that cause an instantaneous change in the parent of object `"?obj"` in the scene graph. The `pick` action has parameters for the arm `"?arm"`, pick configuration `"?q"`, object `"?obj"`, before placement pose `"?p"`, and after grasp pose `"?g"`. Its conditions are that the parameters comprise a valid kinematic solution `Kin` (see Section III-C), `"?obj"` isn't initially attached, `"?obj"` is initially

at placement pose `"?p"`, `"?arm"` isn't initially holding anything, and `"?arm"` is currently at configuration `"?q"`.

```
pick = Action(params="?arm ?q ?obj ?g ?p",
  cond=[Kin("?arm ?q ?obj ?g ?p"),
    Attached("?obj")==None, Pose("?obj")==?p",
    Holding("?arm")==None, At("?arm")==?q"],
  eff=[Holding("?arm")<=?obj", Pose("?obj")<=?g",
    Attached("?obj")<=?arm"])
```

`place` is defined similarly, as the reverse of `pick`.

### B. Durative Actions

Existing TAMP frameworks generally only support sequential plans [15], where solutions are sequences of actions. We wish to beyond this to support *schedules* as solutions, where actions can asynchronously overlap in time. To do this, we generalize the concept of an instantaneous action to a *durative action* [26] that can be *started* at a point in time and *ended* after a certain duration. Each start and end *event* causes a change to the state. In order to correctly execute durative actions, certain conditions must hold before the start event, while the action is ongoing, and before the end event.

Formally, a durative action **DurativeAction** has a string of input *parameters* `params`, a list of *start conditions* `start_cond` that must hold prior to starting, a list of *start effects* `start_effects` that take effect after starting, a list of *ongoing conditions* `over_cond` that must remain true while the action is ongoing (exclusively), a list of *end conditions* `end_cond` that must hold prior to ending, a list of *end effects* `end_effects` that take effect after ending, and a *duration* `duration` that specifies the minimum time elapsed between starting and ending the action.

In our running bimanual domain, the key durative action is `move`, which models arm motion where the arm may or may not be holding an object. The `move` durative action has parameters for the arm `"?arm"`, start configuration `"?q1"`, trajectory `"?t"`, and end configuration `"?q2"`. Its start conditions are that `"?t"` is a valid `Motion` (see Section III-C) that connects `"?q1"` and `"?q2"` and that `"?arm"` starts at configuration `"?q1"`. Critically, its start effect is that the `"?arm"` is no longer at `"?q1"` but instead is "at" trajectory `"?t"`, meaning that it is somewhere along the trajectory. The `move` action does not have any end conditions, and its end effect is that the robot is at configuration `"?q2"`. Finally, its duration is a numeric function of the trajectory `"?t"`. Here, trajectories `"?t"` can be both long multi-second paths as well as short steering segments that are a part of a larger roadmap and executed via successive `move` actions.

```

move = DurativeAction(params="?arm ?q1 ?t ?q2",
  start_cond=[Motion("?arm ?q1 ?t ?q2"),
    At("?arm")=="?q1"],
  start_eff=[At("?arm")<="?t"],
  over_cond=collision_cond,
  end_cond=[], end_eff=[At("?arm")<="?q2"],
  duration=Duration("?t"))

```

In order to schedule with `move`, we need to evaluate the static function `Duration("?t")`. Because we can't declare its output for infinitely many possible trajectories upfront, we leverage an advantage of building off a programming language by defining `Duration` procedurally as a Python procedure rather than declaratively. Namely, we extend `Function` to also take an optional Python function `fn` that provides the unique, static value of the function.

```
Duration = Function("?t", fn=...)
```

The key conditions that ground `move` in robotics are its ongoing conditions `collision_cond`, which ensure that no collisions occur while `"?arm"` is moving along trajectory `"?t"`. We use Python to decompose `collision_cond` into negated ( $\sim$ ) nested arm-object collisions (`ObjCollision`) and arm-arm collisions (`ArmCollision`) that iterate over the movable objects `[obj1, obj2]` and robot arms `[arm1, arm2]`, consuming the current object or arm state. Both of these procedural predicates take a Python Boolean function `test` that returns `True` if, in the case of `ArmCollision`, the swept volume induced by trajectory `"?t1"` intersects with the swept volume induced by trajectory `"?t2"`.

```

ObjCollision = Predicate("?t ?p", test=...)
ArmCollision = Predicate("?t1 ?t2", test=...)
collision_cond = [ $\sim$ ObjCollision("?t", Pose(o))
  for o in [obj1, obj2]] + [ $\sim$ ArmCollision("?t", At(a))
  for a in [arm1, arm2]]

```

### C. Samplers as Streams

In order to instantiate the pick, place, and move actions in Sections III-A and III-B, we need to be able to generate possibly arbitrarily-many continuous parameter values for them. Additionally, combinations of these parameter values must satisfy the static predicates `Kin`, which holds if forward kinematics from configuration `"?q"` times grasp pose `"?g"` results in placement pose `"?p"`, and motion `Motion`, which holds if trajectory `"?t"` avoids collisions with static objects, respects dynamical joint limits, and connects configurations `"?q1"` and `"?q2"`.

```

Grasp = Predicate("?arm ?obj ?g",
  cond=[Arm("?arm"), Object("?obj")])
Kin = Predicate("?arm ?q ?obj ?g ?p",
  cond=[Conf("?arm ?q"), Placement("?obj ?p"),
  Grasp("?arm ?obj ?g")])
Motion = Predicate("?arm ?q1 ?t ?q2",
  cond=[Conf("?arm ?q1"), Conf("?arm ?q2")])

```

We build on the PDDLStream framework [15] by extending a typical finite action language [23] with *streams* `Stream`, procedural conditional generators that consume existing constants and generate new ones that satisfy specific predicates. We can derive a stream from a predicate by partitioning its arguments into inputs `inp` and outputs. This also partitions the predicate's conditions `cond` into input conditions, which the input values must satisfy to call the

stream, and output conditions, which all generated values are guaranteed to satisfy. For example, the grasps stream generates grasp poses `"?q"` for a given arm `"?arm"` and object `"?obj"`, the inverse kinematics stream `ik_qs` generates configurations `"?q"` that satisfy kinematics with input grasp and placement poses `"?g"`, `"?p"`, and the motion planner stream `motions` plans trajectories `"?t"` that connect configurations `"?q1"`, `"?q2"`. In Section V, we discuss how these samplers can be GPU accelerated.

```

grasps = Stream(Grasp, inps="?arm ?obj", gen=...)
ik_qs = Stream(Kin, inps="?arm ?obj ?g ?p", ...)
motions = Stream(Motion, inps="?arm ?q1 ?q2", ...)

```

ScheduleStream algorithms can call each stream's conditional generator `gen` to enumerate new constants and, in turn, pass these new constants to other streams and functions. A typical sequence of operations in our example is as follows, where the underlined values are generated by streams.

```

g, = next(grasps.gen(arm1))
q, = next(ik_qs.gen(arm1, obj1, g, p1))
t, = next(motions.gen(arm1, q1, q))
duration = Duration.fn(t)
collision_cond = not ObjCollision.test(t, p1)
and not ArmCollision.test(t, q2)

```

### D. ScheduleStream Problems

Finally, we formalize ScheduleStream problems.

**Definition 1.** A ScheduleStream problem  $\langle \mathcal{I}, \mathcal{G}, \mathcal{A}, \mathcal{S} \rangle$  is defined by an initial state  $\mathcal{I}$  of function assignments, a list of conjunctive goal conditions  $\mathcal{G}$ , a set of durative (and instantaneous) actions  $\mathcal{A}$ , and a set of streams  $\mathcal{S}$ .

**Definition 2.** A solution to a ScheduleStream problem is a *schedule*  $\tau = [\langle t_1, a_1(x_1), t'_1 \rangle, \dots, \langle t_n, a_n(x_n), t'_n \rangle]$  that respects action and goal conditions as well as has start and end times  $t_i, t'_i$  and action instances  $a_i(x_i)$ , where  $t_i, t'_i \in [0, \infty)$ ,  $a_i \in \mathcal{A}$ ,  $t_i + a_i(x_i).duration \leq t'_i$ , for  $i \in [n]$ .

The objective of ScheduleStream planning is to find a schedule  $\tau$  with minimal *makespan*  $t_* \equiv \max_{i \in [n]} t'_i$ , i.e. duration.

## IV. SCHEDULESTREAM ALGORITHMS

We now present algorithms that solve ScheduleStream problems without any domain-specific additional inputs. Our algorithms alternate between *scheduling* and stream phases.

### A. Eager Stream Scheduling

The simplest ScheduleStream algorithm is EAGER-STREAM (Algorithm 1), which iteratively attempts to solve a finite scheduling problem using subroutine SCHEDULE (Section IV-B) with the current set of constants in an *augmented* initial state  $I$ . Upon failing to find a schedule  $\tau$ , EAGER-STREAM instantiates streams  $s \in \mathcal{S}$  into stream instances  $s(x)$ , where  $x$  are constants in  $I$ . For each stream instance  $s(x)$ , the conditional generator of its stream  $s.gen$  is called using constants  $x$  as inputs, producing new output constants  $y$  or `None` if none exist. If  $y \neq \text{None}$ , the stream output  $s(x)(y)$  is added to *stream plan*  $\psi$ , which records a sequence of stream operations that support the ultimate solution schedule  $\tau$ . Finally, the augmented initial state  $I$  is updated to include function assignments  $p(x, y) \leftarrow \text{True}$

for the stream output predicates  $s.out\_cond$ , which induces a new finite scheduling problem on the next iteration.

---

### Algorithm 1 Eager Stream Scheduling

---

```

1: procedure EAGER-STREAM( $I, \mathcal{G}, \mathcal{A}, S$ )
2:    $I \leftarrow \text{copy}(I)$ ;  $\psi \leftarrow []$   $\triangleright$  Augmented initial state, stream plan
3:   while True do
4:      $\tau \leftarrow \text{SCHEDULE}(I, \mathcal{G}, \mathcal{A})$   $\triangleright$  Solve a scheduling subproblem
5:     if  $\tau \neq \text{None}$  then
6:       return  $\psi, \tau$   $\triangleright$  Return stream plan and schedule
7:     for  $s(x) \in \text{INSTANTIATE}(I, S)$  do
8:        $y \leftarrow \text{next}(s.gen(x))$   $\triangleright$  Generate a new stream output
9:       if  $y \neq \text{None}$  and  $s(x)(y) \notin \psi$  then
10:         $\psi \leftarrow \psi + [s(x)(y)]$   $\triangleright$  Record new stream output
11:        for  $p \in s.out\_cond$  do
12:           $I[p(x, y)] \leftarrow \text{True}$   $\triangleright$  Augment the initial state

```

---

### B. Schedule Subroutine

A key subroutine in EAGER-STREAM is SCHEDULE, which solves a finite planning & scheduling problem defined by initial state  $I$ , goal conditions  $\mathcal{G}$ , and durative actions  $\mathcal{A}$ . This requires not only selecting the action instances  $a(x)$  required but also timing them into a schedule  $\tau$ , exceeding the capabilities of classical planning algorithms [27]. Building on ideas in temporal planning [28], we reduce scheduling to sequential planning of the *start* and *end* of each applied action instance. To do this, we compile each durative action  $a$  into two instantaneous actions, a start action  $a.start$  and an end action  $a.end$ , and solve a sequential planning problem for a plan  $\pi$  for the order to start and end each action.

---

### Algorithm 2 Finite Planning & Scheduling

---

```

1: procedure SCHEDULE( $I, \mathcal{G}, \mathcal{A}$ )
2:    $A \leftarrow \{\hat{a}(x) \mid a(x) \in \text{INSTANTIATE}(I, \mathcal{A}), \hat{a} \in [a.start, a.end]\}$ 
3:    $\pi \leftarrow \text{SEARCH}(I, \mathcal{G}, A)$   $\triangleright$  Sequential search
4:   if  $\pi = \text{None}$  then return None
5:    $\pi \leftarrow \text{SEARCH}(I, \mathcal{G}, \pi)$   $\triangleright$  Reschedule initial plan  $\pi$ 
6:    $\tau \leftarrow []$ ;  $t_* \leftarrow 0$ ;  $T \leftarrow \{\}$   $\triangleright$  Schedule, current time, and starts
7:   for  $\hat{a}(x) \in \pi$  do
8:     if  $\hat{a} = a.start$  then  $\triangleright$  Start action  $a.start$ 
9:        $T[a(x)] \leftarrow t_*$   $\triangleright$  Set start time
10:    else  $\triangleright$  End action ( $\hat{a} = a.end$ )
11:       $\Delta t \leftarrow a(x).duration - (t_* - T[a(x)])$   $\triangleright$  Elapsed time
12:       $t_* \leftarrow t_* + \max(0, \Delta t)$   $\triangleright$  New current time
13:       $\tau \leftarrow \tau + [(T[a(x)], a(x), t_*)]$   $\triangleright$  Add action to schedule
14:   return  $\tau$   $\triangleright$  Return the schedule  $\tau$  derived from plan  $\pi$ 

```

---

Let action  $a$  be a durative action. First, we create a predicate `Ongoing` that is true if action  $a$  is currently being executed and a function `Ongoing` that tracks the minimum time remaining in order to terminate the action. Both have arguments corresponding to the original action parameters. We define the *start action* `start_a` to inherit the start conditions and effects of action  $a$ , add a condition that  $a$  is not currently ongoing, and add effects that  $a$  is now ongoing and its remaining time is equal to the duration of the action.

```

Ongoing = Predicate(args=a.params)
Remaining = Function(args=a.params)
start_a = Action(params=a.params,
  cond=a.start_cond + [~Ongoing(a.params),
  OverCondition("#s")],
  eff=a.start_eff + [Ongoing(a.params),
  Remaining(a.params) <= a.duration])

```

The *end action* `end_a` similarly inherits the end conditions and effects of action  $a$ , adds a condition that  $a$  is currently ongoing, and an effect that afterward  $a$  is no longer ongoing. The *cost* of `end_a` is time remaining, which can be seen as advancing time forward by this amount. Additionally, we leverage Python by defining a procedural condition `OverCondition` that checks whether the ongoing conditions `a2.over_cond` for each ongoing action  $a2$  are still satisfied as well as a procedural effect `ElapseEffect` that updates the Remaining duration for each ongoing action  $a2$  by subtracting the remaining duration for  $a$ .

```

OverCondition = Condition(args="#s", fn=...)
ElapseEffect = Effect(args="#s ?d", fn=...)
end_a = Action(params=a.params,
  cond=a.end_cond + [Ongoing(a.params),
  OverCondition("#s")],
  eff=a.end_eff + [~Ongoing(a.params),
  ElapseEffect("#s", Remaining(a.params)),
  Remaining(a.params) <= 0.],
  cost=Remaining(a.params))

```

Now, we implement SCHEDULE (Algorithm 2) by first solving a SEARCH problem with start and end actions to produce a sequential plan  $\pi$  and then extracting a schedule  $\tau$  that corresponds to that plan. First, it instantiates the start and end actions for each durative action. Then, it calls a generic search algorithm SEARCH to find a sequential plan  $\pi$ , minimizing total end action cost (i.e. time). We implement SEARCH with a lazy weighted A\* heuristic search using a modified version of the Fast-Forward (FF) heuristic [27] that ignores procedural and nested conditions. If it produces a plan, we call SEARCH again using just the action instances in  $\pi$ , with weight  $w = 1$ , in order to locally optimize the plan. For start instance  $a.start(x)$  in the plan, it records the start time as the current time  $t_*$ . For end instances  $a.start$ , it computes the remaining time  $\Delta t$  that must elapse to terminate the action. It adds this to the current time  $t_*$  or zero if  $\Delta t < 0$ . Finally, SCHEDULE returns the schedule  $\tau$ .

### C. Lazy Stream Scheduling

Although the EAGER-STREAM algorithm in Section IV-A will provably solve ScheduleStream problems if a solution exists, it will often unnecessarily call many streams not required to solve the problem, and streams for robotics domains involve, for example, expensive inverse kinematics operations. Thus, building on prior work [15], we propose a *lazy* stream scheduling algorithm that saves time by first identifying which streams could be useful for calling them. The key insight is to temporarily replace a stream's conditional generator  $s.gen$  with a version that creates unique placeholder constants (here denoted as strings with the prefix "@") and plan with these inexpensive values first. Later, an algorithm will retrace which placeholder constants are used and call the streams associated with those constants. In the running example in Figure 2, the lazy stream calls will form the following stream plan  $\psi$ , which here is a serialized computation graph of stream instances and outputs.

```

stream_plan = [grasps(arm1)("@g1"), ...
  ik_qs(arm1, obj1, "@g1", p1)("@q1"), ...
  motions(arm1, q1, "@q1")("@t1"), ...]

```

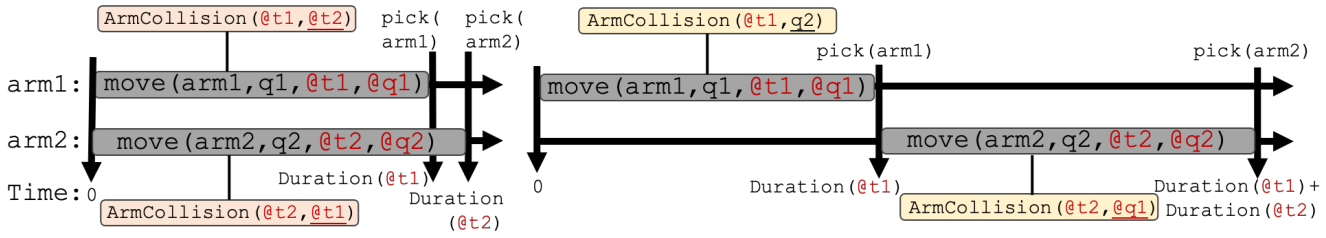


Fig. 3: **Example Schedules.** *Left:* a solution schedule for *Example 1* in Figure 2. Each arm is able to execute a move action in parallel because the trajectories " $@t1$ ", " $@t2$ " do not collide. *Right:* a solution schedule for *Example 2*. Although trajectories " $@t1$ ", " $@t2$ " collide, the trajectory-configuration pairs " $@t1$ ",  $q2$  and " $@t2$ ", " $@q1$ " do not, admitting a serial solution that has a larger makespan.

From these lazy constants, we can instantiate lazy durative action instances that will be instantiated in SCHEDULE.

```
actions = { ...,
  move(arm1, q1, "@t1", "@q1"), pick(arm1, "@q1", "@g1", p1),
  move(arm2, q2, "@t2", "@q2"), pick(arm2, "@q2", "@g2", p2) }
```

When evaluating functions involving lazy constants, we use the function's default value, which is **False** for collision predicates and 0. for numeric duration functions.

---

### Algorithm 3 Lazy Stream Scheduling

---

```
1: procedure LAZY-STREAM( $\mathcal{I}, \mathcal{G}, \mathcal{A}, \mathcal{S}$ )
2:    $P \leftarrow \emptyset$   $\triangleright$  Initialize stream plan, schedule skeleton pairs
3:   while True do
4:     for  $s \in \mathcal{S}$  do
5:        $s.gen \leftarrow$  LAZY-GENERATOR( $s, P$ )  $\triangleright$  Lazy output creation
6:        $\psi', \tau \leftarrow$  EAGER-STREAM( $\mathcal{I}, \mathcal{G}, \mathcal{A}, \mathcal{S}$ )  $\triangleright$  Reduce to eager
7:       if  $\tau \neq \text{None}$  then
8:          $\psi \leftarrow$  RETRACE-STREAMS( $\mathcal{I}, \psi', \text{PREIMAGE}(\tau, \mathcal{G})$ )
9:          $P \leftarrow P \cup \{(\psi, \tau)\}$   $\triangleright$  Add schedule skeleton
10:      for  $\langle \psi, \tau \rangle \in P$  do
11:         $Y \leftarrow \{\}$   $\triangleright$  Initialize stream output mapping
12:        for  $s(x)(y) \in \psi$  do
13:           $x' \leftarrow$  SUBSTITUTE( $x, Y$ )  $\triangleright$  Remap input values
14:           $y' \leftarrow$  next( $s.gen(x')$ )  $\triangleright$  Generate a stream output
15:          if  $y' = \text{None}$  then  $Y \leftarrow \text{None}$ ; break  $\triangleright$  Failure
16:           $Y[y] \leftarrow y'$   $\triangleright$  Update the stream output mapping
17:        if  $Y \neq \text{None}$  then
18:          return SUBSTITUTE( $\psi, Y$ ), SUBSTITUTE( $\tau, Y$ )
```

---

Algorithm 3 gives the pseudocode for the lazy stream scheduling algorithm LAZY-STREAM. It maintains a set of identified *schedule skeletons*  $P$ , schedules  $\tau$  with lazy constants along with stream plans  $\psi$  that could bind values for those constants. On each iteration, LAZY-STREAM replaces each stream conditional generator  $s.gen$  with a version that's lazy for stream instances  $s(x)$  not on a stream plan  $\psi$  in  $P$ . It then calls EAGER-STREAM (Algorithm 1) to identify a new stream plan  $\psi'$  and schedule skeleton  $\tau$ . From  $\psi'$ , it retraces a minimal subsequence  $\psi$  that's sufficient for generating values for the lazy constants on  $\tau$ . At the end of each iteration, LAZY-STREAM evaluates each stream instance  $s(x)$  on an identified stream plan  $\psi$ , attempting to replace each lazy output  $y$  with a real one  $y'$  in a map of bindings  $Y$ . If this process is successful, it replaces all lazy constants in  $\psi$  and  $\tau$  with their new values in  $Y$  and returns this pair.

#### D. Illustrated Example

Revisiting our running example in Figure 2, we now discuss the solutions to each of the three problem instances for

the task for arm1 to hold object obj1 and arm2 to hold object obj2. We will consider schedules computed by LAZY-STREAM (Section IV-C), where configurations " $@q1$ " and " $@q2$ " are lazy outputs of stream ik\_qs and trajectories " $@t1$ " and " $@t2$ " are lazy outputs of stream motions. Figure 3 (*left*) displays a schedule for *Problem 1*, where the arms are able to fully parallelize motion because trajectories " $@t1$ " and " $@t2$ " are never in collision. Figure 3 (*right*) displays a schedule for *Problem 2*, where the arms must move serially because trajectories " $@t1$ " and " $@t2$ " are in collision; however, " $@t1$ " does not collide with  $q2$  and " $@t2$ " does not collide with " $@q1$ ". This schedule uses the same durative action instances as in *Problem 1* but in a different start and end order. Finally, neither of these schedules solve *Problem 3* because pick configurations " $@q1$ " and " $@q2$ " collide. As a result, the planner must introduce an additional move action to move arm1 out of the way to make room for the other arm2.

## V. GPU-ACCELERATED STREAMS

Another challenge of performing temporal instead of sequential TAMP is the need to check collisions between pairs of moving arms instead of a single moving arm against a stationary one. Moreover, the ArmCollision procedural predicate in the move action's conditions (Section III-B) checks for collisions between all pairs of configurations along two trajectories in the event that another arm is moving, namely when  $\text{At}(\text{arm})$  is a trajectory. As an example, consider the usage of ArmCollision in the schedule for *Example 1* versus *Example 2* in Figure 3. In *Example 1*, it computes collisions between lazy trajectories " $@t1$ ", " $@t2$ " whereas in *Example 2*, it computes collisions between trajectory and configuration pairs. Sequential schedules avoid a quadratic number of collision checks; however, they have higher makespans than parallel schedules.

To ease this computational burden, we use GPU acceleration to batch together expensive collision checking, forward kinematics, inverse kinematics, and motion planning stream calls in Section III-C. Our use of GPU acceleration for multi-arm planning goes beyond that of cuTAMP [14], which only considers single-arm TAMP. Specifically, we build on the cuRobo [29] custom CUDA kernels. We represent each robot as a union of inflated inscribed spheres derived from its meshes, which we greedily sample for a given computation budget [30]. For moving object collisions, we perform asymmetric sphere-mesh collision checks where we transform the

sampled swept volume of spheres along a arm trajectory into the frame of an object at a batch of poses. For moving arm collisions, we cast pairwise arm collisions as a batch sphere-sphere self-collision check for a two arm composite robot.

## VI. RESULTS

We analyzed the performance of ScheduleStream compared to ablations representative of prior works in simulation (Section VI-A) and tested it in the real-world (Section VI-B).

### A. Simulated Experiments

We are interested in evaluating the success rate, runtime, and solution quality of ScheduleStream algorithms compared to prior ones. We compared four algorithms that consume the same ScheduleStream problem formulation.

- 1) *Sequential*: an ablation that reduces durative actions to traditional actions and thus performs sequential planning. This is representative of traditional TAMP approaches, namely PDDLStream [15] algorithms.
- 2) *Hierarchical*: an ablation that performs schedules once and attempts to satisfy the schedule in a strict hierarchy without backtracking. This is representative of multi-arm task *then* motion planning approaches [4].
- 3) *Ours*: LAZY-STREAM in Algorithm 3.
- 4) *Ours + GPU*: LAZY-STREAM with batched instead of sequential GPU-accelerated samplers in Section V.

We experimented on five tasks (Figure 4), each with parameter  $N$  that specifies the number of objects or robots, that test how the algorithms scale as  $N$  increases.

- 1) *Franka Hold Assigned  $N$* :  $N$  Franka robots must hold an assigned one of  $N$  objects on separate platforms.
- 2) *Franka Hold Any  $N$* :  $N$  Franka robots must each hold any one of  $N$  objects on separate platforms. This is more challenging than “Franka Hold  $N$ ” as algorithms must match robots to objects based on reachability.
- 3) *SO100 Hold Any  $N$* :  $N$  SO100 robots must each hold any one of  $N$  objects on the same cluttered platform.
- 4) *Franka Pack  $N$* : 2 Franka robots pack  $N$  objects randomly on the floor into a tight region.
- 5) *Franka Stack  $N$* : 2 Franka robots stack  $N$  objects randomly on the floor on top of each other.

We randomly sampled 100 problems per task, ran each algorithm in an anytime mode for 60 seconds, and recorded the discovery time and makespan of each identified solution. Table I lists the success rate (%) and runtime to produce the *first* solution (Time) per task. *Hierarchical* results in low success rates because sampled problems are not always downward refinable [16], particularly when only certain arms can reach certain objects. As a result, performing scheduling without any grounding in physical robotic constraints is generally insufficient. GPU acceleration clearly improves the success rate and decreases the runtime. Table II lists the makespan of the first solution ( $t_*^1$ ) and the best solution ( $t_*^\infty$ ) after 60 seconds. Because *Hierarchical* fails to solve many of the longer-horizon problems, their makespans are averaged across only the easier problems, resulting in a slightly lower

average makespan (denoted by \*). As expected, *Sequential* produces much longer solutions than *Ours* because it does not exploit opportunities for parallelism.

### B. Real World Demonstration

We validated ScheduleStream through 21 real-world demonstrations on the custom bimanual robot in Figure 1, which has two Kinova Gen3 arms. We deployed ScheduleStream using a strategy similar to that of Curtis *et al.* [31], where we used GroundingDINO [32] for open-world object detection and estimated object affordances from the segmented point cloud corresponding to each detection. ScheduleStream was indeed able to plan simultaneous motion that makes use of both arms to efficiently complete 10 real-world tasks involving retrieving, sorting, packing, stacking, and regrasping. See website for videos of the demonstrations as well as an open-source implementation of ScheduleStream.

## VII. CONCLUSION

We introduced ScheduleStream, a new framework for planning & scheduling with samplers that can be used to efficiently solve Task and Motion Planning & Scheduling problems. We showed that ScheduleStream algorithms result in higher success rates and lower makespans than sequential-only algorithms and demonstrated ScheduleStream on several real-world bimanual manipulation tasks.

## REFERENCES

- [1] A. Mandlekar, D. Xu, J. Wong, S. Nasiriany, C. Wang, R. Kulkarni, L. Fei-Fei, S. Savarese, Y. Zhu, and R. Martín-Martín, “What matters in learning from offline human demonstrations for robot manipulation,” in *Conference on Robot Learning (CoRL)*, 2021.
- [2] C. R. Garrett, R. Chitnis, R. Holladay, B. Kim, T. Silver, L. P. Kaelbling, and T. Lozano-Pérez, “Integrated Task and Motion Planning,” *Annual Review of Control, Robotics, and Autonomous Systems*, 2021.
- [3] M. Dalal, A. Mandlekar, C. R. Garrett, A. Handa, R. Salakhutdinov, and D. Fox, “Imitating task and motion planning with visuomotor transformers,” in *7th Annual Conference on Robot Learning*, 2023.
- [4] J. Chen, J. Li, Y. Huang, C. Garrett, D. Sun, C. Fan, A. Hofmann, C. Mueller, S. Koenig, and B. C. Williams, “Cooperative task and motion planning for multi-arm assembly systems,” *arXiv preprint arXiv:2203.02475*, 2022.
- [5] V. N. Hartmann, A. Orthey, D. Driess, O. S. Oguz, and M. Toussaint, “Long-horizon multi-robot rearrangement planning for construction assembly,” *IEEE Transactions on Robotics*, vol. 39, no. 1, pp. 239–252, 2022.
- [6] P. Huang, R. Liu, C. Liu, and J. Li, “Apex-mr: Multi-robot asynchronous planning and execution for cooperative assembly,” *arXiv preprint arXiv:2503.15836*, 2025.
- [7] R. Shome and K. E. Bekris, “Anytime multi-arm task and motion planning for pick-and-place of individual objects via handoffs,” in *2019 International Symposium on Multi-Robot and Multi-Agent Systems (MRS)*. IEEE, 2019, pp. 37–43.
- [8] Z. Wei, X. Luo, and C. Liu, “Hierarchical temporal logic task and motion planning for multi-robot systems,” *arXiv preprint arXiv:2504.18899*, 2025.
- [9] T. Pan, A. M. Wells, R. Shome, and L. E. Kavraki, “A general task and motion planning framework for multiple manipulators,” in *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2021, pp. 3168–3174.
- [10] H. Zhang, S.-H. Chan, J. Zhong, J. Li, P. Kolapo, S. Koenig, Z. Agioutantis, S. Schafrik, and S. Nikolaidis, “Multi-robot geometric task-and-motion planning for collaborative manipulation tasks,” *Autonomous Robots*, vol. 47, no. 8, pp. 1537–1558, 2023.
- [11] S. Lee, J. Motes, I. Ngui, M. Morales, and N. M. Amato, “Lazy-dash: Lazy approach for hypergraph-based multi-robot task and motion planning,” *arXiv preprint arXiv:2504.05552*, 2025.

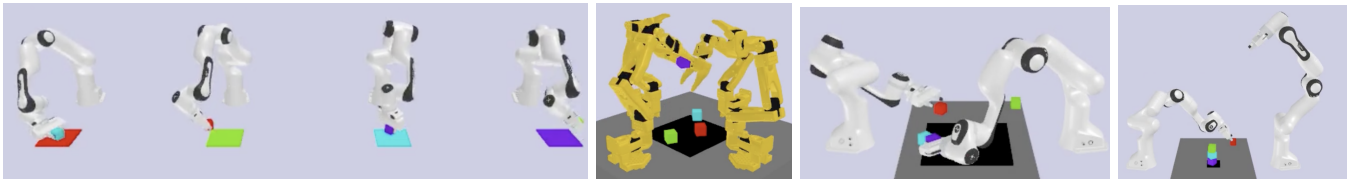


Fig. 4: **Simulated Experiments.** The “Franka Hold Any 4”, “SO100 Hold Any 4”, “Franka Pack 4”, and “Franka Stack 4” tasks.

Algorithm:	Sequential	Hierarchical	Ours	Ours+GPU
Task	% Time	% Time	% Time	% Time
Franka Assigned 1	100	0.4	100	0.4
Franka Assigned 2	100	1.7	100	1.7
Franka Assigned 3	100	3.2	100	4.9
Franka Assigned 4	100	8.2	100	14.0
Franka Any 2	100	2.4	42	35.6
Franka Any 3	100	5.7	15	51.9
Franka Any 4	100	22.1	5	57.9
SO100 Any 2	95	2.4	52	20.3
SO100 Any 3	99	5.9	24	58.1
SO100 Any 4	100	9.1	9	101.4
Franka Pack 2	100	4.0	99	6.3
Franka Pack 3	100	6.7	76	21.5
Franka Pack 4	90	12.4	41	40.2
Franka Stack 2	100	3.8	99	5.7
Franka Stack 3	100	6.4	89	12.3
Franka Stack 4	100	9.1	80	19.9
<b>Average</b>	<b>99</b>	<b>6.5</b>	<b>64</b>	<b>28.3</b>

TABLE I: **Success Rate and First Solution Time.** *Hierarchical* fails to solve several tasks because it does not backtrack, *Ours* has a lower success rate and higher runtime than its GPU-accelerated counterpart *Ours + GPU*.

[12] K. Hauser and V. Ng-Thow-Hing, “Randomized multi-modal motion planning for a humanoid robot manipulation task,” *International Journal of Robotics Research (IJRR)*, vol. 30, no. 6, pp. 676–698, 2011.

[13] M. X. Grey, C. R. Garrett, C. K. Liu, A. D. Ames, and A. L. Thomaz, “Humanoid manipulation planning using backward-forward search,” in *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2016, pp. 5467–5473.

[14] W. Shen, C. Garrett, N. Kumar, A. Goyal, T. Hermans, L. P. Kaelbling, T. Lozano-Pérez, and F. Ramos, “Differentiable gpu-parallelized task and motion planning,” in *Robotics: Science and Systems*, 2025.

[15] C. R. Garrett, T. Lozano-Pérez, and L. P. Kaelbling, “PDDLStream: Integrating Symbolic Planners and Blackbox Samplers,” in *ICAPS*, 2020.

[16] F. Bacchus and Q. Yang, “The Downward Refinement Property,” in *IJCAI*, 1991, pp. 286–293.

[17] H. Zhang, S.-H. Chan, J. Zhong, J. Li, S. Koenig, and S. Nikolaidis, “A mip-based approach for multi-robot geometric task-and-motion planning,” in *2022 IEEE 18th international conference on automation science and engineering (CASE)*. IEEE, 2022, pp. 2102–2109.

[18] M. Toussaint and M. Lopes, “Multi-bound tree search for logic-geometric programming in cooperative manipulation domains,” in *Proceedings - IEEE International Conference on Robotics and Automation*, 2017, pp. 4044–4051.

[19] D. Driess, J.-S. Ha, and M. Toussaint, “Learning to solve sequential physical reasoning problems from a scene image,” *The International Journal of Robotics Research*, vol. 40, no. 12-14, pp. 1435–1466, 2021.

[20] R. Shome, K. Solovey, A. Dobson, D. Halperin, and K. E. Bekris, “drrt\*: Scalable and informed asymptotically-optimal multi-robot motion planning,” *Autonomous Robots*, vol. 44, no. 3, pp. 443–467, 2020.

[21] V. N. Hartmann and M. Toussaint, “Towards computing low-makespan solutions for multi-arm multi-task planning problems,” *arXiv preprint arXiv:2305.17527*, 2023.

[22] J. Motes, T. Chen, T. Bretl, M. M. Aguirre, and N. M. Amato,

Algorithm:	Sequential	Hierarchical	Ours	Ours+GPU
Task	$t_*^1$	$t_*^\infty$	$t_*^1$	$t_*^\infty$
Franka Assigned 1	0.7	0.6	0.7	0.6
Franka Assigned 2	1.4	1.2	0.8	0.6
Franka Assigned 3	2.1	1.8	0.8	0.6
Franka Assigned 4	2.8	2.4	0.8	0.7
Franka Any 2	1.4	1.2	0.7	0.6
Franka Any 3	2.1	1.8	0.8	0.7
Franka Any 4	2.9	2.6	0.9	0.7
SO100 Any 2	3.9	3.8	2.0	2.0
SO100 Any 3	5.8	5.7	2.0	2.0
SO100 Any 4	7.8	7.6	2.0	2.0
Franka Pack 2	2.6	2.1	1.4	1.2
Franka Pack 3	4.0	3.3	2.7	2.3
Franka Pack 4	5.4	4.6	2.9	2.5
Franka Stack 2	2.5	2.2	1.8	1.6
Franka Stack 3	4.0	3.5	3.1	2.8
Franka Stack 4	5.3	4.9	4.4	4.0
<b>Average</b>	<b>3.4</b>	<b>3.1</b>	<b>1.7*</b>	<b>1.5*</b>

TABLE II: **First and Final Makespans.** By making use of multiple arms and parallel execution, *Ours (+ GPU)* produces schedules on average half as long as *Sequential*.

“Hypergraph-based multi-robot task and motion planning,” *IEEE Transactions on Robotics*, vol. 39, no. 5, pp. 4166–4186, 2023.

[23] D. McDermott, M. Ghallab, A. Howe, C. Knoblock, A. Ram, M. Veloso, D. Weld, and D. Wilkins, “PDDL: The Planning Domain Definition Language,” Yale Center for Computational Vision and Control, Tech. Rep., 1998.

[24] H. Geffner, “Functional strips: a more flexible language for planning and problem solving,” in *Logic-based artificial intelligence*. Springer, 2000, pp. 187–209.

[25] R. E. Fikes and N. J. Nilsson, “{STRIPS}: A New Approach to the Application of Theorem Proving to Problem Solving,” *Artificial Intelligence*, vol. 2, pp. 189–208, 1971.

[26] M. Fox and D. Long, “PDDL2.1: An extension to PDDL for expressing temporal planning domains,” *Journal of Artificial Intelligence Research (JAIR)*, vol. 20, p. 2003, 2003.

[27] J. Hoffmann and B. Nebel, “The ff planning system: Fast plan generation through heuristic search,” *Journal of Artificial Intelligence Research*, vol. 14, pp. 253–302, 2001.

[28] P. Eyerich, R. Mattmüller, and G. Röger, “Using the Context-enhanced Additive Heuristic for Temporal and Numeric Planning,” in *Proceedings of the 19th International Conference on Automated Planning and Scheduling (ICAPS)*. AAAI Press, 2009, pp. 130–137.

[29] B. Sundaralingam, S. K. S. Hari, A. Fishman, C. Garrett, K. V. Wyk, V. Blukis, A. Millane, H. Oleynikova, A. Handa, F. Ramos, N. Ratliff, and D. Fox, “curobo: Parallelized collision-free minimum-jerk robot motion generation,” 2023.

[30] M. Inui, N. Umezū, and R. Shimane, “Shrinking sphere: A parallel algorithm for computing the thickness of 3d objects,” *Computer-Aided Design and Applications*, vol. 13, no. 2, pp. 199–207, 2016.

[31] A. Curtis, X. Fang, L. P. Kaelbling, T. Lozano-Pérez, and C. R. Garrett, “Long-horizon manipulation of unknown objects via task and motion planning with estimated affordances,” in *IEEE International Conference on Robotics and Automation (ICRA)*, 2022.

[32] S. Liu, Z. Zeng, T. Ren, F. Li, H. Zhang, J. Yang, C. Li, J. Yang, H. Su, J. Zhu, *et al.*, “Grounding dino: Marrying dino with grounded pre-training for open-set object detection,” *arXiv preprint arXiv:2303.05499*, 2023.