Fog Robotics Algorithms for Distributed Motion Planning Using Lambda Serverless Computing

Jeffrey Ichnowski[1], William Lee[2], Victor Murta[2], Samuel Paradis[1], Ron Alterovitz[2], Joseph E Gonzalez[1], Ion Stoica[1], Ken Goldberg[1]

***Abstract*— For robots using motion planning algorithms such as RRT and RRT*, the computational load can vary by orders of magnitude as the complexity of the local environment changes. To adaptively provide such computation, we propose Fog Robotics algorithms in which cloud-based serverless lambda computing provides parallel computation on demand. To use this parallelism, we propose novel motion planning algorithms that scale effectively with an increasing number of serverless computers. However, given that the allocation of computing is typically bounded by both monetary and time constraints, we show how prior learning can be used to efficiently allocate resources at runtime. We demonstrate the algorithms and application of learned parallel allocation in both simulation and with the Fetch commercial mobile manipulator using Amazon Lambda to complete a sequence of sporadically computationally intensive motion planning tasks.**

I. Introduction

When planning motions for robots in complex environments or facing high-dimensional problems, finding the necessary computational resources can be costly. Motion planning is computationally demanding [1] and can at times require substantially costly parallel compute resources to run at interactive rates. However, the computational cost of planning can also be highly variable. Consider a mobile manipulator robot tasked with decluttering an office space; the low-dimensional motion planning problem of wheeling around the office requires little computation relative to the occasional demands of computing a collision-free motion for a multi-link robot arm to move an object from a desk to a shelf. Having an always-on high-end computer, whether local or in the cloud, would result in an over-allocation of resources and result in unnecessary costs. This paper proposes methods for robot motion planning that make use of on-demand parallelism via serverless computing.

This research was performed at the AUTOLAB at UC Berkeley in affiliation with the Berkeley AI Research (BAIR) Lab, Berkeley Deep Drive (BDD), the Real-Time Intelligent Secure Execution (RISE) Lab, and the CITRIS "People and Robots" (CPAR) Initiative. Authors were also supported by the Scalable Collaborative Human-Robot Learning (SCHooL) Project, a NSF National Robotics Initiative Award 1734633, and in part by donations from Google and Toyota Research Institute. This research was also performed at UNC, supported by the U.S. National Science Foundation (NSF) under Award CCF-1533844. We thank our colleagues who provided helpful feedback and suggestions, in particular Vikram Sreekanti and Chenggang Wu. This article solely reflects the opinions and conclusions of its authors and do not reflect the views of the sponsors or their associated entities. [1]University of California, Berkeley, CA 94720, USA {jeffi,samparadis, jegonzal,istoica,goldberg}@berkeley.edu,
[2]University of North Carolina at Chapel Hill, Chapel Hill, NC 27599 USA, wlee221@live.unc.edu, victor.murta97@gmail.com, ron@cs.unc.edu

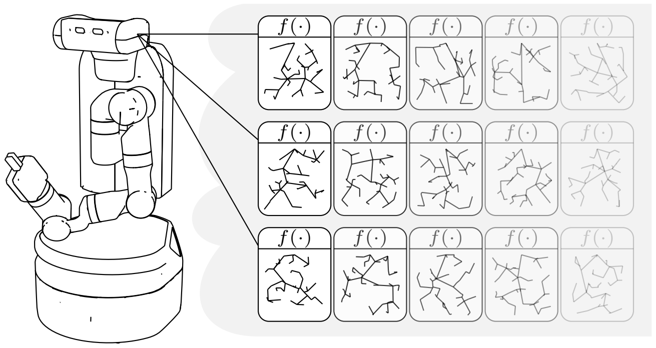

Fig. 1. Robots computing parallel versions of RRT and RRT* for complex motion-planning problems using functions-as-a-service (FaaS) serverless computing can scale to an arbitrary number of parallel processes. Under the proposed system, the robot spawns "functions" (shown here as $f(\cdot)$) to compute a solution in parallel. Each function runs a sampling-based motion planner to generate its a graph of motions, and coordinates with the robot to produce a solution to the motion planning problem. More computing can solve the problem faster but at higher cost.

In a serverless computing environment, cloud- and fog-based service providers charge for units of compute time in quanta on the order of 100 ms [2]. To achieve tight latencies and rapid scaling, existing serverless platforms separate compute and storage and expose a Functions-as-a-Service (FaaS) compute model. In the FaaS model, user logic is encapsulated in stateless functions or *lambdas* that manipulate cloud storage services and respond to web requests. While each lambda running on a FaaS has limitations, FaaS provides a significant computing benefit: the number of lambdas that can compute in parallel is unbounded.

In this paper, we propose using FaaS as a scalable parallel-processing platform on which robots can compute motion plans. We present two novel motion planners targeted towards the parallel processing offered by FaaS. These motion planning algorithms parallelize both within a lambda execution, and across multiple concurrently running lambda executions. This "multi-lambda" approach to parallel processing inherits properties from both shared-memory and distributed-memory approaches. Within the lambda execution, the planners we propose benefit from shared-memory parallelism which allows each lambda to compute faster with additional hardware threads. The planners we propose also use a distributed-memory parallelism approach that communicates updates to concurrently running lambda executions (Fig. 1) so that each lambda can either terminate sooner or converge to a better result faster.

This paper makes the following contributions:

1) probabilistically-complete and asymptotically-optimal sampling-based motion planners for mixed-parallel (shared/distributed-memory) model,
2) an implemented system for mixed-parallel motion planning using lambdas on a FaaS platform,
3) an approach to predicting solution probability and convergence rates of similar motion planning problems,
4) budget-based allocation of parallel resources for motion planning computation, and
5) experiments with simulated robots and the Fetch mobile manipulator using Amazon's AWS Lambdas for motion planning that suggest the proposed algorithms and system can significantly speed up motion planning computation in a deployed robot system.

II. Related Work

Sampling-based motion planners, such as probabilistic roadmaps (PRM) [3], rapidly-exploring random trees (RRT) [4], and asymptotically-optimal variants thereof, e.g., RRT* [5], BIT* [6], and SST* [7], tackle complex motion planning problems by generating random robot configuration samples and connecting them into a graph of valid motions. These planners are probabilistically complete, and with proper attention to how samples are connected (or "rewired") in the graph, these planners can be asymptotically optimal. Both probabilistically-complete and asymptotically-optimal planners are presented in this paper.

Parallelized sampling-based motion planners find solutions and converge towards optimality faster through the use of parallel computing techniques [8], [9]. We consider two parallel computing techniques: shared-memory and distributed-memory parallelism. In shared-memory parallel motion planners, multiple processors and cores independently sample and contribute to the construction of a single shared graph, through locks (e.g., KPIECE [10]) or lock-free atomic operations (e.g., PRRT, and PRRT* [11]). In distributed-memory parallel motion planners, such as SRT [12], Bulk-Synchronous RRT [13], Blind RRT [14], RRT-OR [15], [16], [17], and C-FOREST [18], independent single-threaded motion planners collaborate to compute motion plans by communicating progress towards a solution. While other forms of computational parallelism, such as those based upon GPU computing (e.g., [19], [20]), can speed up the computation of motion plans, these computing platforms are as yet not readily available on FaaS platforms. The proposed motion planners combine the shared-memory parallelism (within a lambda) and distributed-memory parallelism (between lambdas) of PRRT and RRT-OR, and PRRT* and C-FOREST, to speed up motion planning when running on multiple multi-core lambdas.

Prior work on gaining additional cloud-based computation often focuses on *servers*. Kehoe et al. [21] presented an overview of cloud-robotics in application and approaches. Bekris et al. [22] showed the benefit of cloud-based computation in a warehouse environment. Tian et al. [23] offload grasp-analysis to the cloud. Tanwani et al. [24] propose a fog-robotics approach to deep learning for surface decluttering. Cloud-based motion planning can also adapt to dynamic environments [25] has the potential to speed up motion planning while being economically viable [26]. While this prior work focuses on *server*-based computation, in this work, we instead explore using *serverless* computing.

Cloud and fog-based service providers are still evolving their serverless computing offerings. Hellerstein et al. [27] identify limitations in using serverless in both computing and data-intensive algorithms. Jonas et al. [28] provide a history of cloud computing and the evolution of serverless computing, as well as potential directions for future evolutions of serverless computing. In this paper, we make use of serverless computing in its present offering, and envision that the benefits of using the approach we present will grow as serverless computing evolves (e.g., [29], [30], [31], [32]).

A common thread of modern cloud-based computing is determining which virtual machine (VM) best fits the demands of the application. Yadwadkar et al. [33] identify this as a problem and propose a solution for selecting an optimal VM. Chung et al. [34] explicitly optimize a cluster setup for budget. Kröhnert et al. [35] propose a method that adapts computing resource allocation to a motion planning algorithm as a single instance of the algorithm runs. In this paper we codify the selection of serverless VM type and amount of serverless allocation as a minimization problem on a per-problem basis and explicitly solve it in the context of the motion planning algorithms we present.

III. Problem Definition

Serverless motion planning requires addressing two problems detailed here: parallelized motion planning, and how much parallelism to allocate to its computation.

A. Motion planning problem

Let $\mathbf{q}$ be a robot's configuration—the complete specification of a robot's degrees of freedom. Let $\mathcal{C}$ be the set of all possible configurations, thus $\mathbf{q} \in \mathcal{C}$. Let $\mathcal{C}_{\text{free}} \subseteq \mathcal{C}$ be the set of *valid* configurations, e.g., ones that are not in collision with any obstacle and do not violate any task-specific constraints. Let $L : \mathcal{C} \times \mathcal{C} \to \{0, 1\}$ be a robot-specific predicate function that is 1 if the path between two configurations is valid according to a local planner, and 0 otherwise. Let $d : \mathcal{C} \times \mathcal{C} \to \mathbb{R}_{\geq 0}$ be a function that reflects the relative difficulty of traversing from one configuration to another, e.g., in terms of distance travelled, time taken, or energy expended. Given a starting configuration $\mathbf{q}_{\text{start}}$ and a set of goal configurations $\mathcal{C}_{\text{goal}}$, the objective of motion planning is to compute a sequence of configurations $\tau = (\mathbf{q}_0, \mathbf{q}_1, \ldots, \mathbf{q}_n)$ such that $\mathbf{q}_0 = \mathbf{q}_{\text{start}}$, $\mathbf{q}_n \in \mathcal{C}_{\text{goal}}$, $\mathbf{q}_i \in \mathcal{C}_{\text{free}}$ for all $i \in [0, n]$, and $L(\mathbf{q}_i, \mathbf{q}_{i+1}) = 1$ for all $i \in [0, n)$. Let $c(\tau) = \sum_{i=1}^{n} d(\mathbf{q}_{i-1}, \mathbf{q}_i)$. The objective of optimal motion planning is to find a τ that minimizes $c(\tau)$.

B. Parallel-computation budget problem

Let $p \in \mathbb{Z}^+$ be the number of computing processes concurrently solving a motion planning problem. Let $v > 0$

be the cost of a serverless computing resource (e.g., in dollars per second). The objective of a parallel-computation budget problem is to find a p that minimizes a real-world objective, such as cost to find a solution with high probability in a given amount of time:

$$\underset{p \in \mathbb{Z}^+}{\operatorname{argmin}} \quad p \cdot t \cdot v$$
$$\text{s.t.} \quad t < t_{\max} \quad \hat{\mathbb{E}}_{\text{solved}}(p,t) > x,$$

where $\hat{\mathbb{E}}_{\text{solved}}(p,t)$ is an *approximation* of the probability that the motion planner find a solution at time t, and x is the desired probability. To solve this and similar minimization problems using standard optimization techniques, one must be able to estimate $\hat{\mathbb{E}}_{\text{solved}}(p,t)$—computing this function is thus part of the parallel-computation budget problem.

IV. Method

This section proposes two approaches for multi-lambda motion planning. The first is probabilistically complete, and the second is asymptotically optimal.

A. Probabilistically-Complete Multi-Lambda Planning

Robots using probabilistically-complete motion planning algorithms can compute motion plans more rapidly with the simultaneous invocation of multiple lambdas. This is a result of the probabilistically-complete property that means that for a given motion planning problem S running on 1 CPU and using 1 core, the probability of finding a solution $P_{\text{solve}}(t; S, 1, 1)$ increases with increasing time t, converging to 1 as $t \to \infty$. By simultaneously computing with p functions on the same problem, the probability of finding a solution $P_{\text{solve}}(t; S, p, 1)$ is:

$$P_{\text{solve}}(t; S, p, 1) = 1 - (1 - P_{\text{solve}}(t, S, 1, 1))^p .$$

The RRT-OR [15] algorithm exploits this property by computing p independent RRTs and stopping as soon as any process finds a motion plan. The multi-lambda PRRT-OR similarly uses the property to scale to run concurrently on multiple lambdas (see Fig. 2), with the difference that it also exploits shared-memory multi-core parallelism in each lambda using PRRT [11] to increase the sampling rate within each lambda. Thus,

$$P_{\text{solve}}(t; S, p, c) = P_{\text{solve}}(t/c; S, p, 1),$$

where c is the number of cores on which each function is run, assuming linear speedup[1]. This is the property that that allows PRRT to compute solutions more rapidly.

Alg. 1 combines the favorable aspects of RRT-OR and PRRT into a single motion-planning whole. In this process, the robot acts as an initiator of, and coordinator for, all of the computing functions[2]. This algorithm initiates the process

Algorithm 1 MLPRRT (Multi-Lambda Parallel RRT)

```
 1: ip ← get robot's network address
 2: $s$ ← socket to listen for network connections on ip
 3: for $i \leftarrow 1$ to $p$ do
 4:     $\Lambda \leftarrow \Lambda \cup \{$call_lambda(
            $k$, PRRT-OR_Lambda, $\mathbf{q}_{\text{start}}, \mathcal{C}_{\text{goal}}, \text{C}_{\text{free}}, L,$ ip)$\}$
 5: $S \leftarrow \varnothing$
 6: while $\tau = \varnothing$ do
 7:     wait for connection on $s$ or new result in $\Lambda$
 8:     if incoming connection on $s$ then
 9:         $S \leftarrow S \cup \{$accept incoming connection from s$\}$
10:     for all $\lambda_i \in \Lambda$ do
11:         $\tau_i \leftarrow$ poll_lambda_result($\lambda_i$)
12:         if $\tau_i \neq \varnothing$ then
13:             $\tau \leftarrow \tau_i$
14: broadcast "solved" to all $S$ (and handle any stragglers)
```

Algorithm 2 PRRT-OR_Lambda($\mathbf{q}_{\text{start}}, \mathcal{C}_{\text{goal}}, \mathcal{C}_{\text{free}}, L,$ ip)

```
 1: $G = (V, E) \leftarrow (\mathbf{q}_{\text{start}}, \varnothing)$                 // initialize tree
 2: $s$ ← open connection to ip
 3: $\tau \leftarrow \varnothing$
 4: for each available hardware thread, in parallel do
 5:     RNG ← uniquely-seeded random number generator
 6:     while $\tau = \varnothing$ and not poll_solved(s) do
 7:         $\mathbf{q}_{\text{rand}} \leftarrow$ RNG()
 8:         $\mathbf{q}_{\text{near}} \leftarrow$ neighbor($V, \mathbf{q}_{\text{rand}}$)        // concurrent
 9:         if $\mathbf{q}_{\text{near}} \in \mathcal{C}_{\text{free}}$ then
10:             $\mathbf{q}_{\text{new}} \leftarrow$ steer($\mathbf{q}_{\text{near}}, \mathbf{q}_{\text{rand}}$)
11:             if $L(\mathbf{q}_{\text{near}}, \mathbf{q}_{\text{new}})$ then
12:                 $(V, E) \leftarrow (V \cup \{\mathbf{q}_{\text{new}}\}, E \cup \{(\mathbf{q}_{\text{near}}, \mathbf{q}_{\text{new}})\})$
13:                 if $\mathbf{q}_{\text{new}} \in \mathcal{C}_{\text{goal}}$ then
14:                     $\tau \leftarrow$ path from $\mathbf{q}_{\text{start}}$ to $\mathbf{q}_{\text{new}}$ // concurrent
15: return $\tau$
```

by establishing a broadcast mechanism (lines 1–2) and then spawing the desired number of lambdas (lines 3–4). The algorithm then waits for the first completed motion plan computation, and as with RRT-OR, stops all other running lambdas, in this case, with a broadcast message (line 14). This last step saves on computation budget.

The algorithm that computes the motion plan and runs in the FaaS is the multi-lambda PRRT-OR lambda shown in Alg. 2. This algorithm is PRRT modified to run as a lambda. PRRT is a highly-scalable multi-core version of the RRT algorithm [11], we summarize here: PRRT uses a concurrent nearest-neighbor searching data structure and lock-free updates to a graph to allow multiple threads to simultaneously add samples to the same tree without slowing down due to contention over locks. The lines that require special attention, and for which we refer the reader to PRRT, are listed with the "*// concurrent*" comment.

B. Asymptotically-Optimal Multi-Function Planning

While the focus of the probabilistically-complete motion planning was on the tradeoff between computing costs asso-

[1]In practice, FaaS providers expose hardware threads as a portion of a shared computing core, and thus more modest speedups are likely.

[2]FaaS providers often restrict functions to not allow incoming connections. While other broadcast formulations are possible through additional services, we present a solution here that presumes the robot can allow incoming connections and thus act as the broadcast coordinator.

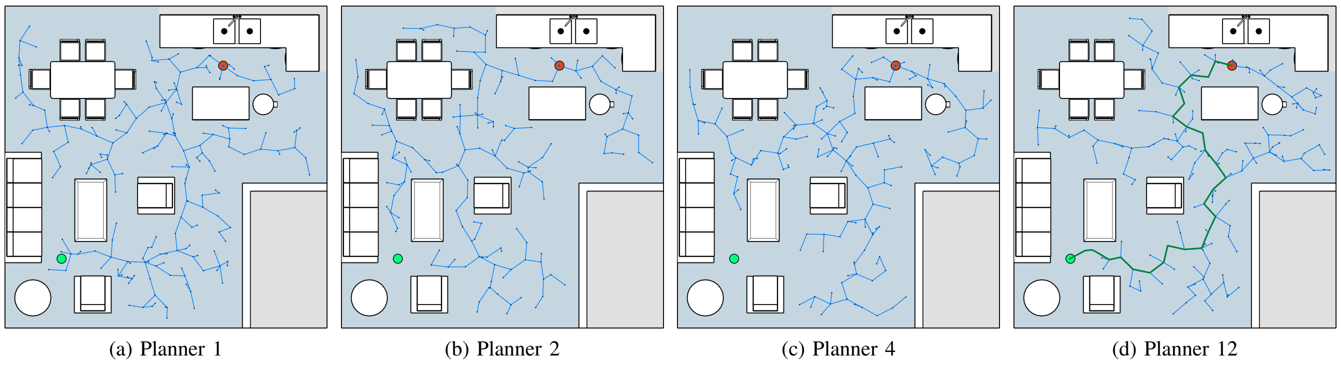

Fig. 2. **PRRT-OR operation.** With PRRT-OR, uniquely seeded PRRT planners run in parallel, stopping as soon as any planner finds a solution. The multi-core parallelism in PRRT allows each planner to shorten the expected solution time proportional to the number of cores it uses. The parallel execution exploits the probabilistic nature of the algorithm to find a solution faster. This figure shows the state of 4 of the 16 planners in a single PRRT-OR plan computation. Each is in varying proximity to finding the goal, and only the right-most one has found the goal.

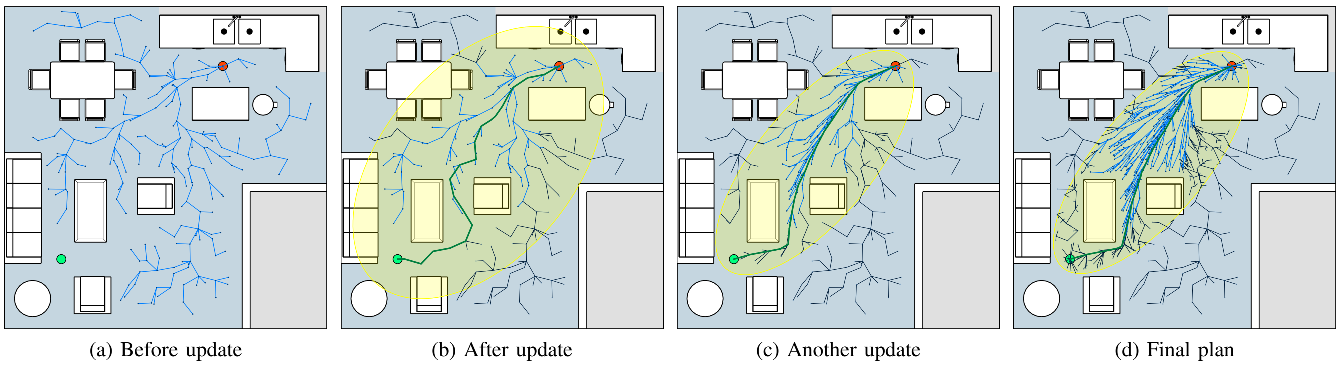

Fig. 3. **PC-FOREST operation.** With PC-FOREST, uniquely seeded PRRT* planners run in parallel, updating (via the robot) the other planners with the best paths as they find them. This figure shows the state of a single PRRT* planner of 16 planners before receiving an update (a), after an update (b), and after interleaved sampling and a successive update (c). The final state after 1000 samples computed by all 16 planners is in (d). When the planner continues sampling, it only needs to sample within the green ellipse, focusing exploration and speeding up the convergence rate.

ciated with motion planning and the probability of finding a motion plan, the focus of asymptotically-optimal motion planning is to find a plan that minimizes the sum of a distance function. One could thus place a multi-lambda motion planner in the context of a larger problem, such as minimizing the overall costs or energy required to operate a robot, or maximizing the profitability of robot operation. With an asymptotically-optimal motion planner, the sum of distances decreases with additional compute time, either by running longer or through the use of parallel computation.

The multi-lambda asymptotically-optimal motion planning algorithm we propose is a combination of the PRRT* shared-memory motion planner, and C-FOREST distributed-memory motion planner. For brevity, we do not include the robot's coordinating algorithm as it is similar to Alg. 1, with the differences being that: (1) motion planning does not stop on the first solution, but instead continues until a user-specified termination condition (e.g. time limit); and (2) as the robot receives motion plans, it broadcasts them out to all running functions to improve their efficiency.

Combined C-FOREST and PRRT (Alg. 3):* The combined algorithm, outlined in Alg. 3, offers advantages from both PRRT* and C-FOREST. PRRT* can scale to available multi-

Algorithm 3 `PC-FOREST_Lambda()`

```
1:  $s \leftarrow$ open connection to ip
2:  $G = (V, E) \leftarrow (\{\mathbf{q}_{\text{start}}\}, \varnothing)$
3:  $c(\mathbf{q}_{\text{start}}), c(\mathcal{C} \setminus \{\mathbf{q}_{\text{start}}\}) \leftarrow 0, \infty$
4:  $\tau_{\text{best}}, c_{\text{best}} \leftarrow \varnothing, \infty$
5:  for each available hardware thread, in parallel do
6:      while stopping criteria not met do
7:          recv_broadcast()                          // ← only 1 thread
8:          $\mathbf{q}_{\text{rand}} \leftarrow$ rejection_sample(RNG)
9:          $\mathbf{q}_{\text{near}} \leftarrow$ neighbor$(V, \mathbf{q}_{\text{rand}})$        // concurrent
10:         $\mathbf{q}_{\text{new}} \leftarrow$ steer$(\mathbf{q}_{\text{near}}, \mathbf{q}_{\text{rand}})$
11:         if $L(\mathbf{q}_{\text{near}}, \mathbf{q}_{\text{new}})$ then
12:             add_vertex$(\mathbf{q}_{\text{near}}, \mathbf{q}_{\text{new}})$
```

core shared-memory parallelism, generating samples faster, and thus converging to optimal faster. C-FOREST can scale out to distributed-memory parallelism, effectively sharing information about progress to convergence between independently running motion planners. Since this algorithm is run in the context of a FaaS function, it starts by both initializing the motion planning structures (lines 2–4) and opening a connection for sending and receiving updates. PRRT* stores

Algorithm 4 `add_vertex`

```
1: $r \leftarrow$ RRT* radius of search based on $\|V\|$
2: $N \leftarrow \texttt{neighbors}(V, \mathbf{q}_{\text{new}}, r) \cup \{\mathbf{q}_{\text{near}}\}$   // concurrent
3: $\mathbf{q}_{\text{parent}} \leftarrow \text{argmin}_{\mathbf{q}_i \in N : L(\mathbf{q}_i, \mathbf{q}_{\text{new}})}\, c(\mathbf{q}_{\text{i}}) + d(\mathbf{q}_i, \mathbf{q}_{\text{new}})$
4: $\texttt{update\_parent}(\mathbf{q}_{\text{parent}}, \mathbf{q}_{\text{new}})$
5: for all $\mathbf{q}_i \in N$ do
6:     $\texttt{update\_parent}(\mathbf{q}_{\text{new}}, \mathbf{q}_i)$
7: $V \leftarrow V \cup \{\mathbf{q}_{\text{new}}\}$   // $\mathbf{q}_{\text{new}}$ may already be in $V$
```

Algorithm 5 `update_parent`$(\mathbf{q}_{\text{parent}}, \mathbf{q}_{\text{child}})$

```
1: repeat
2:     $c' \leftarrow c(\mathbf{q}_{\text{parent}}) + d(\mathbf{q}_{\text{parent}}, \mathbf{q}_{\text{child}})$
3:     if $c(\mathbf{q}_{\text{child}}) < c'$ then
4:         return   // existing route to child is shorter
5: until successful atomic update of $E$ and $c(\mathbf{q}_{\text{child}})$
6: if $\mathbf{q}_{\text{child}} \in \mathcal{C}_{\text{goal}}$ and $c' < c_{\text{best}}$ then
7:     $c_{\text{best}}, \tau_{\text{best}} \leftarrow c'$, (path following edges to root)
8:     broadcast $c_{\text{best}}, \tau_{\text{best}}$
9: for all $(\mathbf{q}_{\text{child}}, \mathbf{q}_i) \in E$ do
10:    $\texttt{update\_parent}(\mathbf{q}_{\text{child}}, \mathbf{q}_i)$
```

the shortest sum of distances to a configuration using $c(\cdot)$. PRRT* builds a shared graph using all available hardware parallelism (line 5), using underlying data structures that allow for lock-free and concurrent updates with minimal overhead. Each thread then continually generates random samples (line 8) and adds them to a shared graph if the path is feasible (lines 11). The algorithm delegates one thread to have the additional responsibility of integrating updates from other running functions (line 7).

Adding Vertices to the Graph (Alg. 4): The `add_vertex` operation (Alg. 4) adds a vertex to the graph while locally rewiring the graph towards optimality. The local rewiring follows from RRT* [5] and makes the planner asymptotically-optimal. Rewiring is a two-step process. The first step selects the shortest path to the added vertex (lines 3). The second step checks all vertices in the neighborhood, and rewires them through the added vertex if the path is shorter (line 6).

Updating Edges in the Graph (Alg. 5): As the motion planner adds vertices to the graph, the effect of local rewiring operations needs to be pushed out to the rest of the tree [36]. Alg. 5 both updates a vertex's parent node (and thus path), and pushes updates to child nodes effected by a lowered $c(\cdot)$ to the node. The update, from PRRT*, updates the parent, path, and associated $c(\cdot)$ of a vertex in a single atomic operation. This atomic operation checks that the update will lower $c(\cdot)$ as it expects to do, replaces if it does, or fails if it would not. Since threads may be competing to update the same part of the graph, one thread's successful update may cause another thread's update to fail. When this happens, the thread whose update failed, compares to the new $c(\cdot)$ of the node, and if the update would still produce a better path, it tries the atomic operation again (line 5).

Incorporating Updates from Other Running Functions: Alg. 3 runs simultaneously in multiple lambdas. Each independent lambda shares its progress towards an optimal motion plan, and thus benefits from the progress of other lambdas. This sharing of progress towards optimality, as originally introduced by C-FOREST, includes two operations that speed up motion planning: (1) broadcasting and incorporating the best path into each planner's tree, and (2) sampling within a bounded ellipse to insure that the planner only considers configurations that can improve the best path.

Algorithm 6 `recv_broadcast`

```
1: if broadcast available then
2:     $c'_{\text{best}}, \tau'_{\text{best}} \leftarrow$ receive broadcast
3:     if $c'_{\text{best}} < c_{\text{best}}$ then
4:         $c_{\text{best}}, \tau_{\text{best}} \leftarrow c_{\text{best}}, \tau'_{\text{best}}$
5:         for all $(\mathbf{q}_i, \mathbf{q}_j) \in \tau_{\text{best}}$ do
6:             $\texttt{add\_vertex}(\mathbf{q}_i, \mathbf{q}_j)$
```

Algorithm 7 `rejection_sample`

```
1: repeat
2:     $\mathbf{q}_{\text{rand}} \leftarrow \texttt{RNG}()$
3:     $\mathbf{q}'_{\text{goal}}, c'_{\text{best}} \leftarrow$ atomically load from $\tau_{\text{best}}, c_{\text{best}}$
4: until $h(\mathbf{q}_{\text{start}}, \mathbf{q}_{\text{rand}}) + h(\mathbf{q}_{\text{rand}}, \mathbf{q}'_{\text{goal}}) < c'_{\text{best}}$
5: return $\mathbf{q}_{\text{rand}}$
```

The first operation, broadcasting and incorporating best paths, starts with the the broadcast of any improvement to the best solution (Alg. 5, line 8), and ends with incorporating the best path into the tree in `recv_broadcast` (Alg. 6). The `recv_broadcast` algorithm checks for broadcasts, and when it finds one, it incorporates the path into the lambda's local graph. Since each lambda may be running on multiple threads, the function only runs `recv_broadcast` from a single thread in order to avoid duplication of effort. Similarly, it makes use of the `add_vertex` function to insure that the path is properly incorporated into the graph in the presence of concurrent updates.

The second operation, sampling within a bounded ellipse, improves the convergence rate by only considering samples that can improve the best path to goal. This operation, shown in Alg. 7, uses information about the current best path to reject any samples whose straight-line distance from start and goal would be longer than the current best route.

In implementation, the updates to the best path and its associated cost make use of PRRT*'s data structures to allow for atomic updates. The data structure represents a path and its cost in a single object with an immutable linked-list path to the starting configuration. By representing the path and cost in a single immutable data structure, updating the best path (in `recv_broadcast`) is a matter of a single atomic update to a pointer. Similarly, accessing the best path (in `rejection_sample`) is a single atomic read (i.e., the goal configuration $\mathbf{q}_{\text{goal}}$ and the cost of the path to it c_{best} are considered together).

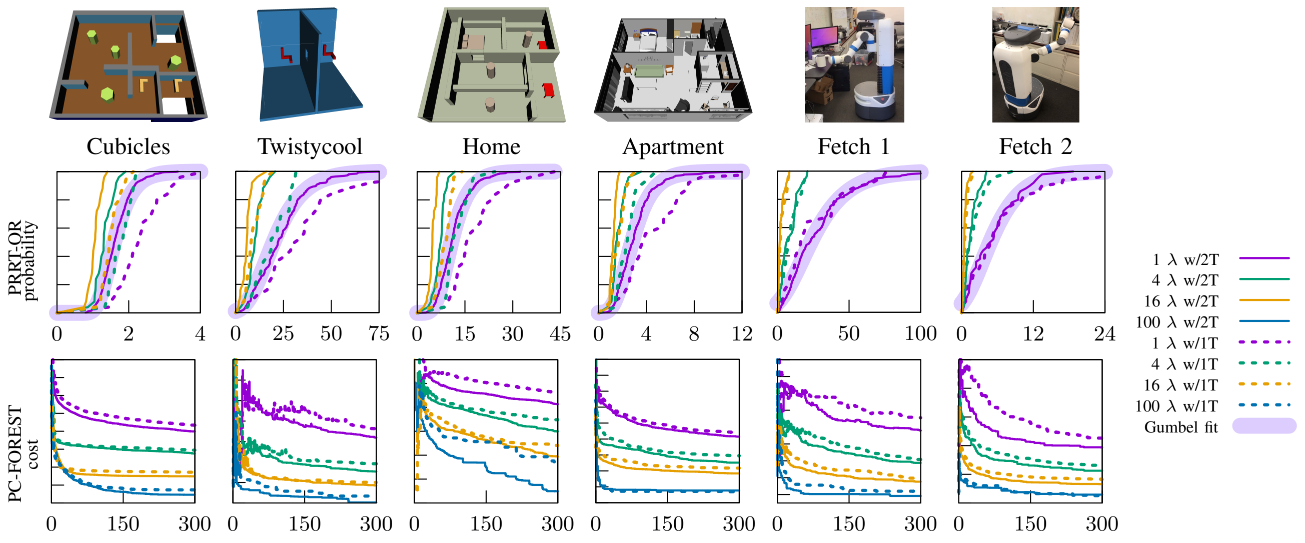

Fig. 4. **Multi-lambda planners computing solutions to various motion planning problems.** Each column shows a different motion planning problem. In all graphs, the x-axis is wall-clock time in seconds, the solid lines show the performance of lambdas making use of the 2 hardware threads (2T), and the dotted lines show the performance using one thread (1T). The color codes (see key) correspond to the number of lambdas run in parallel. We run up to 16 lambdas (16 λ) for the PRRT-OR, and 100 lambdas (100 λ) for the PC-FOREST. The **top row** shows the motion planning scenario. The **middle row** shows the multi-lambda PRRT-OR empirical cumulative distribution function. The lighter wide line shows the Gumbel distribution fit on 1 two-threaded (2T) lambda. The **bottom row** shows the multi-lambda PC-FOREST convergence rate.

C. *Budget-Constrained Motion Planning*

Given that lambda computing provides an unbounded source of parallelism that is pay-per-use, it becomes necessary to compute an estimate of how many lambdas to allocate to solving a motion planning problem. Since motion planning is PSPACE-complete [1], and due to the probabilistic nature of sampling-based motion planning, determining the precise computational resources required is essentially impossible. Instead, we focus on learning an estimate based on observations of previous motion plan computations. We fit a standard probability distribution to the time it takes to compute a solution. In the results section, we found the Gumbel distribution [37], as it considers the maximum of multiple samples, fit the probability distribution exhibited in our experiments. Its cumulative distribution function is

$$e^{-e^{-(x-\mu)/\beta}},$$

and has the benefit that it can be linearized and quickly least-squares fit and updated.

With this probability estimated, we propose that the amount and type of lambda computing to allocate to a motion planning problem can be readily computed based on the desired minimization objective.

V. Results

We experiment using Amazon's AWS Lambdas to compute motion plans for 6 degree-of-freedom (DOF) rigid-body motion planning problems from OMPL [38] and for a sequence of motion planning problems for the Fetch [39] robot. The Fetch robot is tasked with decluttering an office space. Moving between decluttering tasks is a 3-DOF problem that does not require lambda computing as it can be quickly solved by the Fetch robot's onboard CPU. Each decluttering task is an 8-DOF task (7 revolute joints and 1 prismatic lift joint) that sporadically uses lambda computing.

Fig. 4 shows the scenarios (first row), cumulative distribution function for the probabilistically-complete planner (second row), and convergence rate over time of the asymptotically-optimal planner (third row). The second row also shows a parameterization of the Gumbel distribution fit to the data, allowing the motion planner to estimate the amount of computing to allocate for a particular problem and given a time or budget constraint.

The graphs suggest that the planners are both able to scale to make use of multiple lambdas—allocating more lambdas allows the system to solve the problem sooner, and converge to a better solution faster. The graphs also suggest that the lambdas are also able to speed up motion plan computation using the multi-core computing available within each lambda.

VI. Conclusion

In this paper, we presented motion planning algorithms that make use of multi-core serverless lambda computing and present a method for estimating the amount of computing to allocate to a motion planning problem. The multi-core lambda algorithms make use of both multi-core parallelism within the lambda and scale with the computing power of additional lambdas running in parallel. Experimental results running these planners suggest that a system making use of this lambda can effectively scale to solve complex motion planning problems quickly.

In future work, we will explore addressing the variable latency associated with lambda invocation, scaling to more lambdas, and dynamically changing the parallelism allocated to a running problem (e.g., [35]).

References

[1] J. Canny, *The complexity of robot motion planning*. MIT press, 1988.

[2] Amazon Web Services, Inc. AWS Lambda – pricing. [Online]. Available: https://web.archive.org/web/20190909111142/https://aws.amazon.com/lambda/pricing/

[3] L. E. Kavraki, P. Svestka, J.-C. Latombe, and M. Overmars, "Probabilistic roadmaps for path planning in high dimensional configuration spaces," *IEEE Trans. Robotics and Automation*, vol. 12, no. 4, pp. 566–580, 1996.

[4] S. M. LaValle and J. J. Kuffner, "Randomized kinodynamic planning," *The International Journal of Robotics Research*, vol. 20, no. 5, pp. 378–400, May 2001.

[5] S. Karaman and E. Frazzoli, "Sampling-based algorithms for optimal motion planning," *The International Journal of Robotics Research*, vol. 30, no. 7, pp. 846–894, June 2011.

[6] J. D. Gammell, S. S. Srinivasa, and T. D. Barfoot, "Batch informed trees (bit*): Sampling-based optimal planning via the heuristically guided search of implicit random geometric graphs," in *2015 IEEE international conference on robotics and automation (ICRA)*. IEEE, 2015, pp. 3067–3074.

[7] Y. Li, Z. Littlefield, and K. E. Bekris, "Asymptotically optimal sampling-based kinodynamic planning," *The International Journal of Robotics Research*, vol. 35, no. 5, pp. 528–564, 2016.

[8] N. M. Amato and L. K. Dale, "Probabilistic roadmap methods are embarrassingly parallel," in *Proceedings IEEE Int. Conf. Robotics and Automation (ICRA)*, May 1999, pp. 688–694.

[9] W. Sun, S. Patil, and R. Alterovitz, "High-frequency replanning under uncertainty using parallel sampling-based motion planning," *IEEE Transactions on Robotics*, vol. 31, no. 1, pp. 104–116, 2015.

[10] I. A. Şucan and L. E. Kavraki, "Kinodynamic motion planning by interior-exterior cell exploration," in *Algorithmic Foundation of Robotics VIII*. Springer, 2009, pp. 449–464.

[11] J. Ichnowski and R. Alterovitz, "Scalable multicore motion planning using lock-free concurrency," *IEEE Transactions on Robotics*, vol. 30, no. 5, pp. 1123–1136, 2014.

[12] E. Plaku, K. E. Bekris, B. Y. Chen, A. M. Ladd, and L. E. Kavraki, "Sampling-based roadmap of trees for parallel motion planning," *IEEE Transactions on Robotics*, vol. 21, no. 4, pp. 597–608, 2005.

[13] S. A. Jacobs, N. Stradford, C. Rodriguez, S. Thomas, and N. M. Amato, "A scalable distributed rrt for motion planning," in *2013 IEEE International Conference on Robotics and Automation*. IEEE, 2013, pp. 5088–5095.

[14] C. Rodriguez, J. Denny, S. A. Jacobs, S. Thomas, and N. M. Amato, "Blind rrt: A probabilistically complete distributed rrt," in *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE, 2013, pp. 1758–1765.

[15] S. Carpin and E. Pagello, "On parallel RRTs for multi-robot systems," *Proceedings 8th Conference Italian Association for Artificial Intelligence*, 2002.

[16] D. Devaurs, T. Siméon, and J. Cortés, "Parallelizing rrt on distributed-memory architectures," in *2011 IEEE International Conference on Robotics and Automation*. IEEE, 2011, pp. 2261–2266.

[17] ——, "Parallelizing rrt on large-scale distributed-memory architectures," *IEEE Transactions on Robotics*, vol. 29, no. 2, pp. 571–579, 2013.

[18] M. Otte and N. Correll, "C-FOREST: Parallel shortest path planning with superlinear speedup," *IEEE Transactions on Robotics*, vol. 29, no. 3, pp. 798–806, 2013.

[19] J. Pan, C. Lauterbach, and D. Manocha, "g-Planner: Real-time motion planning and global navigation using GPUs," in *AAAI Conference on Artificial Intelligence (AAAI-10)*, July 2010, pp. 1245–1251.

[20] J. J. Bialkowski, S. Karaman, and E. Frazzoli, "Massively parallelizing the RRT and the RRT*," in *Proceedings IEEE/RSJ Int. Conf. Intelligent Robots and Systems (IROS)*, San Francisco, CA, Sept. 2011, pp. 3513–3518.

[21] B. Kehoe, A. Matsukawa, S. Candido, J. Kuffner, and K. Goldberg, "Cloud-based robot grasping with the google object recognition engine," in *Proceedings IEEE Int. Conf. Robotics and Automation (ICRA)*. IEEE, 2013, pp. 4263–4270.

[22] K. Bekris, R. Shome, A. Krontiris, and A. Dobson, "Cloud automation: Precomputing roadmaps for flexible manipulation," *IEEE Robotics & Automation Magazine*, vol. 22, no. 2, pp. 41–50, 2015.

[23] N. Tian, M. Matl, J. Mahler, Y. X. Zhou, S. Staszak, C. Correa, S. Zheng, Q. Li, R. Zhang, and K. Goldberg, "A cloud robot system using the dexterity network and berkeley robotics and automation as a service (brass)," in *2017 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2017, pp. 1615–1622.

[24] A. K. Tanwani, N. Mor, J. Kubiatowicz, J. E. Gonzalez, and K. Goldberg, "A fog robotics approach to deep robot learning: Application to object recognition and grasp planning in surface decluttering," *CoRR*, vol. abs/1903.09589, 2019. [Online]. Available: http://arxiv.org/abs/1903.09589

[25] J. Ichnowski, J. Prins, and R. Alterovitz, "Cloud-based motion plan computation for power-constrained robots," in *Algorithmic Foundations of Robotics (Proceeding of WAFR)*. Springer, 2016.

[26] ——, "The economic case for cloud-based computation for robot motion planning," in *Proceedings International Symposium on Robotics Research (ISRR)*, 2017, pp. 1–7.

[27] J. M. Hellerstein, J. M. Faleiro, J. E. Gonzalez, J. Schleier-Smith, V. Sreekanti, A. Tumanov, and C. Wu, "Serverless computing: One step forward, two steps back," in *Conference on Innovative Data Systems Research (CIDR '19)*, 1 2019. [Online]. Available: https://arxiv.org/abs/1812.03651

[28] E. Jonas, J. Schleier-Smith, V. Sreekanti, C.-C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. M. Carreira, K. Krauth, N. Yadwadkar, J. E. Gonzalez, R. A. Popa, I. Stoica, and D. A. Patterson, "Cloud programming simplified: A berkeley view on serverless computing," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2019-3, 2 2019. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2019/EECS-2019-3.html

[29] S. Hendrickson, S. Sturdevant, T. Harter, V. Venkataramani, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Serverless computation with openlambda," in *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*, 2016.

[30] G. McGrath and P. R. Brenner, "Serverless computing: Design, implementation, and performance," in *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*. IEEE, 2017, pp. 405–410.

[31] I. E. Akkus, R. Chen, I. Rimac, M. Stein, K. Satzke, A. Beck, P. Aditya, and V. Hilt, "SAND: Towards high-performance serverless computing," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, 2018, pp. 923–935.

[32] V. Sreekanti, C. Wu, X. C. Lin, J. Schleier-Smith, J. M. Faleiro, J. E. Gonzalez, J. M. Hellerstein, and A. Tumanov, "Cloudburst: Stateful functions-as-a-service," 2020.

[33] N. J. Yadwadkar, B. Hariharan, J. E. Gonzalez, B. Smith, and R. H. Katz, "Selecting the best VM across multiple public clouds: A data-driven performance modeling approach," in *Proceedings of the 2017 Symposium on Cloud Computing*, ser. SoCC '17. New York, NY, USA: ACM, 9 2017, pp. 452–465. [Online]. Available: http://doi.acm.org/10.1145/3127479.3131614

[34] A. Chung, J. W. Park, and G. R. Ganger, "Stratus: cost-aware container scheduling in the public cloud," in *Proceedings of the ACM Symposium on Cloud Computing*, 2018, pp. 121–134.

[35] M. Kröhnert, R. Grimm, N. Vahrenkamp, and T. Asfour, "Resource-aware motion planning," in *2016 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2016, pp. 32–39.

[36] O. Arslan and P. Tsiotras, "Use of relaxation methods in sampling-based algorithms for optimal motion planning," in *2013 IEEE International Conference on Robotics and Automation*. IEEE, 2013, pp. 2421–2428.

[37] E. J. Gumbel, "The return period of flood flows," *The annals of mathematical statistics*, vol. 12, no. 2, pp. 163–190, 1941.

[38] I. A. Şucan, M. Moll, and L. E. Kavraki, "The Open Motion Planning Library," *IEEE Robotics and Automation Magazine*, vol. 19, no. 4, pp. 72–82, Dec. 2012. [Online]. Available: http://ompl.kavrakilab.org

[39] Fetch Robotics, "Fetch research robot," http://fetchrobotics.com/research/.