Efficient Distribution of Robotics Workloads using Fog Computing

Raghav Anand

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2020-47
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2020/EECS-2020-47.html

May 15, 2020

Efficient Distribution of Robotics Workloads using Fog Computing

by

Raghav Anand

A project report submitted in partial satisfaction of the

requirements for the degree of

Master of Science

in

Electrical Engineering and Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Ken Goldberg,
Professor Joseph Gonzalez

Spring 2020

The project report of Raghav Anand, titled Efficient Distribution of Robotics Workloads using Fog Computing, is approved:

Digitally signed by Ken Goldberg
DN: cn=Ken Goldberg, o, ou, email=goldberg@berkeley.edu, c=US
Date: 2020.05.15 09:37:33 -08'00'

Date 15 May 2020

Date 15/May/2020

University of California, Berkeley

Efficient Distribution of Robotics Workloads using Fog Computing

Abstract

Efficient Distribution of Robotics Workloads using Fog Computing

by

Raghav Anand

Master of Science in Electrical Engineering and Computer Science

University of California, Berkeley

Professor Ken Goldberg,

As more and more robots are used to perform tasks in homes, offices and warehouses, there will be a need to scale algorithms to large fleets of robots to allow for fast, reliable and safe operation. These algorithms will often have to leverage the power of the Edge and the Cloud (together called the Fog) in unison to deliver the most efficient compute that any robot requires at any point in time.

In the first part of the thesis, we introduce Robot Inference and Learning as a Service for low-latency and secure inference serving of deep models that can be deployed on robots. Unique features of RILaaS include: 1) low-latency and reliable serving with gRPC under dynamic loads by distributing queries over multiple servers on Edge and Cloud, 2) SSH based authentication coupled with SSL/TLS based encryption for security and privacy of the data, and 3) front-end REST API for sharing, monitoring and visualizing performance metrics of the available models. We report experiments to evaluate the RILaaS platform under varying loads of batch size, number of robots, and various model placement hosts on Cloud, Edge, and Fog for providing benchmark applications of object recognition and grasp planning as a service. We address the complexity of load balancing with a reinforcement learning algorithm that optimizes simulated profiles of networked robots.

In the second part of the thesis we propose a sampling-based multi-query graph-based motion planner for robots that parallelizes the search process using cloud-based serverless computing (AWS Lambda). Using graph-based motion planning instead of tree-based alternatives allows for efficient reuse of a precomputed road map between tasks in the same workspace. By parallelizing the precomputation and reusing exploration, a robot executing multiple actions in the same workspace can leverage an already dense graph to create more efficient motion plans in a short amount of time. We introduce an algorithm to parallelize Probablistic Roadmaps (PRM) over serverless nodes, provide proofs of asymptotic optimality and probabilistic completeness and run a suite of experiments on the Fetch robot for a pick-and-place task to measure the provided speedup.

To Amma, Appa and Thuths

Contents

List of Figures

List of Tables

Acknowledgments

I would like to thank Prof. Ken Goldberg for providing me with the opportunity to do research in AUTOLab for the last two years, for helping me find projects that I enjoyed working on and supporting me during difficult times. I would also like to thank Prof. Joey Gonzalez, Jeff Ichnowski, Chenggang Wu and all the other amazing people in AUTOLab and RISELab without whom my research experience would have been that much more hollow.

My family provided constant support since I came to Cal: doing this would have been impossible without them. I want to thank Vibha Seshadri for celebrating with me during my best moments and being there for me during my worst ones. My roommates Howard Ki, Manny Lujan and Sushanth Varma (and my pseudo-roommate Emma Jaeger) made Berkeley an unforgettable experience, and I will be forever grateful. Finally, I want to thank everyone else I met here for all I have learnt during the last 4 years, and leave with the hope that this premature end to school in 2020 is not the last I see of you.

Chapter 1

Introduction

As more and more robots are used to perform tasks in homes, offices and warehouses, there will be a need to scale algorithms to large fleets of robots to allow for fast, reliable and safe operation. These algorithms will often have to leverage compute on both the Edge and the Cloud (together called the Fog) to allow cost-efficient and speedy operation.

A large class of newly developed algorithms rely on computationally expensive deep neural networks for perception, planning and control. There is often a difficulty in deploying these algorithms to large robot fleets because of the additional infrastructure development that needs to accompany the algorithm development. Chapter 3 presents Robot Inference and Learning as a Service (RILaaS): a framework that allows easy deployment and distribution of deep models over a heterogenous Fog infrastructure. RILaaS was built in collaboration with Ajay Tanwani and has been accepted for publication to IEEE-RAL 2020 [44] *. My contribution to RILaaS was to architect and build out the backend for the platform (including containerization), setup the load balancing framework (including the Q-Learning algorithm) and run all the experiments to validate the reinforcement learning based load-balancer.

On the other hand, some robotics algorithms not involving deep neural networks require more specific algorithms to efficiently parallelize them in a cost-effective manner. Motion planning for robots is a common subprocedure that is required to complete more complex robotics tasks. For robotic arms, motion planning using the onboard CPU is often slow due to the high dimensional configuration space in which these plans have to be generated. Chapter 4 presents a novel way of parallelizing graph based motion planning using serverless computing to allow the generation of fast and cost-efficient motion plans. This work was done in collaboration with Jeffrey Ichnowski and Chenggang Wu [3] †. My contribution here was to start from previous work on tree based motion planning [14], formulate extensions to graph based planners, implement these extensions in a C++ codebase and run experiments to measure speedups for the end-to-end motion planning pipeline.

*Ajay Tanwani, Raghav Anand, Joseph Gonzalez, and Ken Goldberg. "RILaaS: Robot Inference and Learning as a Service". In:IEEE Robotics and Automation Letters(2020).

†Raghav Anand, Jeff Ichnowski, Chenggang Wu, Joseph Hellerstein, Joseph Gonzalez, Ken Goldberg "Distributed Multi-Query Serverless Motion Planning".

Chapter 2

Related Work

2.1 Robot Inference and Learning as a Service

Cloud and Fog Robotics

Cloud Robotics provides on-demand availability of configurable resources to support robots' operations [22]. The centralized Cloud approach alone often limits the latency and throughput of data than deemed feasible for many robotics applications. Fog Robotics distributes the resource usage between the Cloud and the Edge in a federated manner to mitigate the latency, security/privacy, and network connectivity issues with the remote Cloud data centers [11, 43]. Popular cloud robotics platforms include RoboEarth [48] – a world-wide web style database to store knowledge generated by humans and robots accessed via Rapyuta platform; KnowRob [45] – a knowledge processing system for grounding the knowledge on a robot; RoboBrain [37] – a large scale computational system that learns from publicly available resources over internet; rosbridge [34] – a communication package between the robot and the Robot Operating System (ROS) over Cloud; while Berkeley robotics and automation as a service (BraaS) [46] and Dex-Net as a Service (DNaaS) [33] are recent efforts to provide Cloud-based services for analytical grasp planning.

To the best of our knowledge, RILaaS is the first user-based data-driven general purpose inference serving platform for programming robots. We provide grasp planning and single-shot object recognition services as an example where the robots send RGB and/or depth images of the environment and retrieve the recognized objects and the grasp locations for robotic manipulation.

Inference Serving

Inference serving is emerging as an important part of a machine learning pipeline for deploying deep models. The growing demand of machine learning based services such as image recognition, speech synthesis, recommendation systems etc. is resulting in tighter latency requirements and more congested networks. Large tech companies have built their private model serving infrastructure to handle scaling, performance, and life cycle management in production, however, their adoption in a wider machine learning and robotics community is rather limited.

A simple way to deploy a trained model is to make a REST API using Flask. Although simple and quick, it often causes scale, performance, and model life cycle management issues in production. **Tensorflow-serving** uses SavedModels to package the trained models for scaling and sharing of the deployed models [31]. The serving, however, does not support arbitrary pre-processing and post-processing of the data which limits a range of applications. **Clipper** supports a wide variety of frameworks including Caffe, Tensorflow and Scikit-learn for inference serving in the Cloud. Additionally, it uses caching and adaptive batching to improve the inference latency and throughput [10]. **InferLine** combines a planner and a reactive controller to continuously monitor and optimize the latency objectives of the application [9]. **Rafiki** optimizes for model accuracy with a reinforcement learning algorithm subject to service level latency constraints [49]. **INFaaS** automatically navigates the decision space on behalf of users to meet user-specified objectives [36]. Recently, a number of companies have entered the model serving space with Amazon Apache MXNet, Nvidia TensorRT, Microsoft ONNX and Intel OpenVino to satisfy the growing application demands. All these services are typically optimized to serve specific kinds of models in the Cloud only. Moreover, creation or updating of the models at the back end is manual and cumbersome. In comparison to these services, RILaaS allows users to upload trained deep models, share with other users and/or make them publicly available for others to test models with custom data and easily deploy on new robots for querying the trained models. It distributes the queries over Cloud and Edge to satisfy more stringent service level objectives than possible with inference serving in the Cloud only.

Inference Optimization

Deploying deep learning models is not just about setting up the web server API, but ensuring that the service is scalable and the requests are optimized for service level objectives. The Cloud provides seemingly infinite resources for compute and storage, whereas resources at the Edge of the network are limited. Edge and Fog Computing brings Cloud-inspired computing, storage, and networking closer to the robot where the data is produced [39]. Quality of service provisioning depends upon a number of factors such as communication latency, energy constraints, durability, size of the data, model placement over Cloud and/or Edge, computation times for learning and inference of the deep models, etc. This has motivated several models for appropriate resource allocation and service provisioning [29]. Schaerf et al. investigate adaptive load balancing in a decentralized and distributed system [38]. Tian et al. present a fog robotic system for dynamic visual servoing with an ayschronous heartbeat signal [47]. Chinchali et al. use a deep reinforcement learning strategy to offload robot sensing tasks over the network [7]. Nassar and Yilmaz [30] and Baek et al. [4] allocate resources in the Fog network with a reinforcement learning based load balancing algorithm.

RILaaS takes a distributed approach to inference serving where a load-balancer receives inference requests from nearby robots/clients at the Edge and learns to decide whether to process the requests on Cloud or Edge servers based on their resource consumption. We show its application to vision-based grasping and object recognition and investigate the inference scaling problem by simulating increasing number of requests in the network.

2.2 Parallelizing Graph based Motion Planning

Sampling based Motion Planning

Sampling-based motion planners solve motion-planning [8] problems by generating random robot configurations and connecting them into a graph of feasible motions. Planners such as PRM [21] and RRT [24] are probabilistically complete, meaning that with enough time, they will find a solution with probability 1. With attention to sampling and connection strategy, these planners can be asymptotically-optimal (e.g., PRM* and RRT* [19] and SST [26]), meaning that with enough time, they will find an optimal solution with probability 1. In some scenarios, finding a single solution to a motion planning problem or *single-query* is sufficient, while in other scenarios it can be beneficial to precompute a *multi-query* graph or road map of motions that can later be quickly search with different start and goal configurations. In this paper, we build on prior work in graph-based planners to build a multi-query motion planner by parallelizing the sampling loop using cloud-based serverless computing to construct a graph.

Parallel Algorithms for Motion Planning

Amato et al. [1] showed that sampling-based motion planners are well-suited for parallel computation. Prior work exploring parallelizing sampling-based motion planning takes mulitple forms, including building a single graph in shared memory with locks [41] and without locks [13], or in distributed memory [32, 6, 18]. In this work, we parallelize the construction of a single graph by distributing graph generation to multiple concurrent serverless computing processes.

Spliting a motion planning problem into smaller regions is a tactic prior work explores either to tackle simpler problems or to distribute work in a parallel computing environment. KPIECE [41] prioritizes sampling cells to break up motion planning problems based on complexity of each cell. Jacobs et al. [17] divide space for sampling using radial splits. Ichnowski et al. [15] recursively split sampling regions to always keep the working data set small enough to fit in the CPU cache.

Serverless Computing

Serverless computing has gained wide attention in recent years. Compared to traditional serverful computing, where users provision virtual machines (VM) and perform computation on these VMs, serverless computing has two key advantages. First, it abstracts away the notion of servers; users only need to register functions with the system and define when to trigger function execution. This drastically simplified the deployment process as users no longer need to manually provision VMs and worry about finding the VM that has the optimal combination of CPU, memory, and network resources. Second, serverless platforms automatically adapt to workload changes; they dynamically scale up as the workload spikes and scale down as the workload troughs, and users only pay for the compute allocated during the function execution. In our setting, as the robot moves across different environment and performs the search, the amount of compute required across environment dynamically changes, making serverless computing an attractive option.

Using the Cloud for Motion Planning

Cloud-based computation for robotics shows promise in offloading compute-intensive processes from a robot's onboard computer [16], allowing robots to have low-power CPUs and light-weight batteries to power them. Motion planning can be computationally challenging [5], and thus is a good candidate for cloud-based computation [23]. In prior work, Ichnowski et al. [14] showed that serverless computing of tree-based single-query motion planners has the potential to dramatically speed up motion plan computation. In this work, we propose speeding up graph-based multi-query motion planning using serverless computing. Unlike prior serverless motion planning work which could limit its network communication to a small portion of the graph (e.g., the solution), this work requires coordination in the construction of a shared graph on which it will later perform multiple queries.

Chapter 3

Robot Inference and Learning as a Service

3.1 Introduction

Robot programming has evolved from low level coding to more intuitive methods. Common ways of programming robots include use of a teaching pendant to record and playback a set of via-points, offline programming with the use of a simulator, programming by demonstration such as kinesthetic teaching, or programming by exploration for trial and error learning of the desired task. Despite the variety of interfaces, teaching a new task to a robot requires a skilled person which involves data collection, labeling and/or learning a policy from hundreds of hours of robot training [25]. In this work, we advocate the need of a **programming-by-abstraction** approach where high-level skills such as grasping and object recognition etc. can readily be acquired in a 'plug-and-play' manner to facilitate programming of complex skills.

Recent advancements in deep learning have led to a rise of robotic applications that rely on computationally expensive models such as deep neural networks for perception, planning and control. Typical usage of a deep learning model involves: **training**, **adaptation** and/or **inference**. The training stage involves estimation of model parameters on large scale data, adaptation is the process of transferring/fine-tuning the model to a new domain/environment, while inference requires predicting the model output for a given input. While training and adaptation of a deep model is computationally and resource intensive, inference is often lightweight and must be done in real-time to meet the performance requirements of the application. As an example, training a deep object recognition model on ImageNet-1k may last for days, adaptation may take hours, but the inference time is often less than 100 milliseconds.

Robots are increasingly linked to the network and thus not limited by the onboard resources for compute, storage and networking with Cloud and Fog Robotics [22, 43]. By offloading the computational and storage requirements over the network, the robots can share training, adaptation and inference of deep learning models and reduce the burden of collecting and labelling massive data for programming a separate model for each robot. Once trained, the models can be deployed to an inference serving system to meet the performance requirements of the application such as bandwidth, latency, accuracy and so on. To our surprise, there is very little research on how to

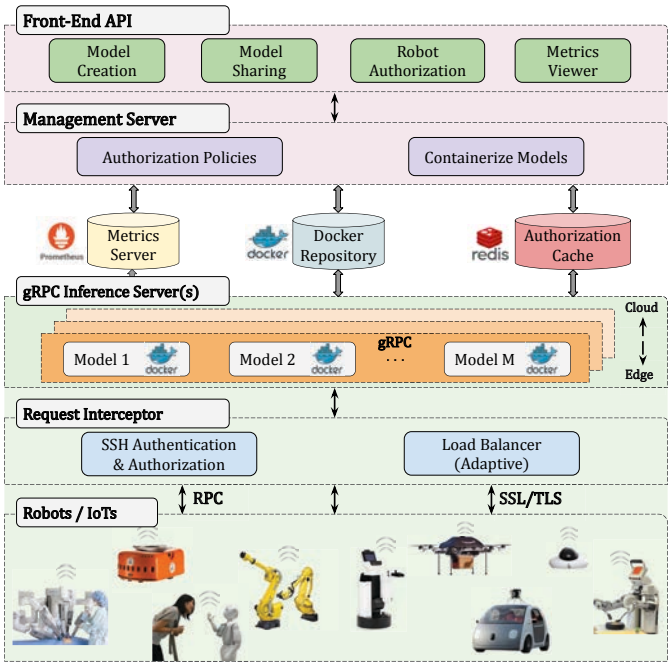

Figure 3.1: RILaaS uses a hierarchy of resources in the Cloud-Edge continuum to distribute inference/prediction serving of deep learning models such as grasp planning and object recognition on a fleet of robots. Users can manage robots and models with a front-end API that interacts with the inference loop through a metrics server, authorization cache, and a Docker model repository.

use/re-use and deploy such models once they are trained. The focus of this paper is on scalable inference serving of deep models on networked robots.

In this paper, we introduce a novel Robot-Inference-and-Learning-as-a-Service (RILaaS) platform to meet the service level objectives in inference serving of deep models on robots. RILaaS abstracts away applications from the training phase with virtualized computing and storage of models and datasets, thereby, removing hardware and software dependencies on custom middleware. It allows users to easily upload, test, share, monitor and deploy trained models on robots for querying the service ubiquitously. The service optimizes for lower latency and scalable inference across a fleet of robots by distributing queries over Cloud and Edge using a model-specific load balancing strategy (see Fig. 3.1). We also address the complexity of load balancing with a reinforcement learning algorithm that optimizes load profiles of networked robots by distributing queries over Cloud and Edge, and observe that it outperforms several baselines including round robin, least connections, and least model time. We show the application of RILaaS to deep object recognition and grasp planning, where a model takes as input images of the robot environment and returns objects and/or grasp configurations as output. We investigate the performance of RILaaS platform under varying batch sizes, number of robots, and simulated dynamic loads for vision-based decluttering, where a mobile robot grasps objects from a cluttered floor and sorts them into respective

bins.

Contributions

This chapter makes the following contributions:

1. We present RILaaS: a novel user-based low-latency inference serving platform to facilitate large-scale use/re-use of deep models for robot programming.

2. We provide examples of deep object recognition and grasp planning as a service with RILaaS and benchmark their performance with varying number of robots, batch sizes and dynamic loads.

3. We optimize the round-trip latency times for scalable inference serving by distributing queries over Cloud and Edge servers with a reinforcement learning algorithm that outperforms several baselines under simulated dynamic loads by at least 14.04% reduction in round-trip latency time compared to the next best least-connections strategy.

3.2 RILaaS Features and Challenges Addressed

Consider the multi-agent setting with a set of M robots $\langle r_1 \dots r_M \rangle$ each having access to a set of trained models or policies $\langle \pi_1 \dots \pi_D \rangle$ that are deployed on a set of N servers. Each model may be deployed on one or more servers, and the location of each server is fixed either on Cloud, Edge or anywhere along the continuum. The robot observes the state of the environment as $\{\boldsymbol{\xi}_t\}_{t=1}^{T_B}$ in a mini-batch of size T_B, sends the request to the inference service and receives the response $\{\boldsymbol{y}_t\}_{t=1}^{T_B}$. The job of the inference service is to compute the response $\{\boldsymbol{y}_t = \pi(\boldsymbol{\xi}_t)\}_{t=1}^{T_B}$ for the requested model such that the round-trip latency time is optimized in communication with the set of robots, while preserving the privacy and security of the data.

Next, we describe the specific challenges in developing the general purpose inference serving platform and discuss the RILaaS methodology to address the outlined issues.

Model Support: Prominent machine learning frameworks such as PyTorch, Tensorflow, Spark, Caffe are widely used for training and adaptation of deep models. Deploying these multiple frameworks on a robot or a set of inference servers is complex because of conflicting dependencies between each framework. RILaaS accepts any arbitrary model for deployment by using Docker containers to allow each framework to exist independently of the other. Each container can be customized to the requirements of a particular framework. The containers accept inputs of `Map<name, numeric array>` and return outputs of the same form, where the map function adapts the model inputs and outputs to the RILaaS format.

Rapidly Deployable: RILaaS abstracts away applications from models to facilitate ease of deployment on custom hardware with varying specifications. It only requires the public SSH key of the robot for authenticating and subscribing to the required models, after which the robot can readily access model outputs over a network call.

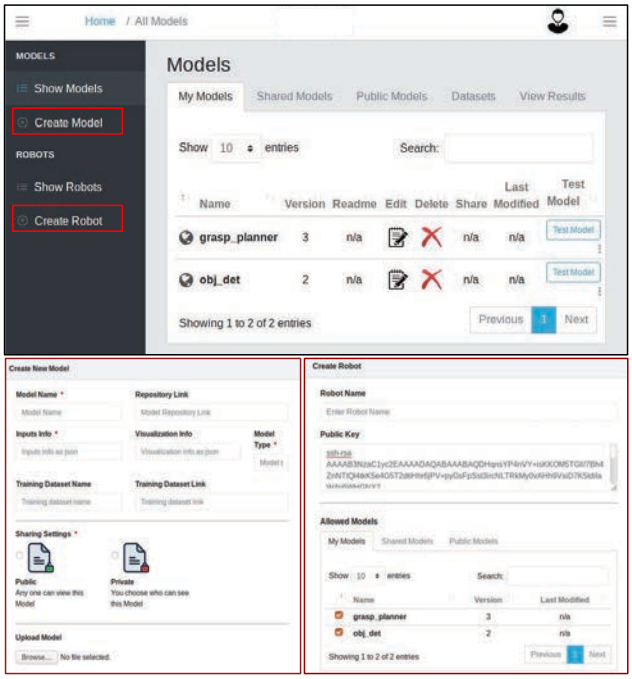

Figure 3.2: Front-end API snapshots (not shown to scale): *(top)* Users can upload, share and visualize models and datasets, *(bottom-left)* interface to upload new models and set access control policies, *(bottom-right)* interface to deploy available models on robots.

Security and Privacy: Inference serving often requires transmitting sensitive data over untrusted wireless networks (such as images of private locations) . Hosting models on the Edge of the network can keep data private and the network secure, but it comes at the cost of developing and maintaining a heterogeneous Edge infrastructure. RILaaS uses a Fog robotics approach to place models on the Cloud and the Edge servers depending upon the security requirements specified by the user. This allows access to the infinite compute capacity of the Cloud for low-sensitivity models while using secure but less powerful Edge infrastructure for private data. Moreover, RILaaS's front end allows easy management of access controls on a per-robot per-model basis.

Scalable Workloads: Robots with limited onboard compute may have to trade-off between doing fast inference on a remote server using hardware accelerators such as GPUs (while incurring additional network overhead) and doing slow inference locally on CPUs. Latency times need be optimized to deal with dynamic application dependent workloads. RILaaS optimizes the inference serving latency for each individual model by using reinforcement learning to distribute queries over the Cloud and the Edge servers according to their resource consumption.

Performance Monitoring: Monitoring the inference service is useful to evaluate desired accuracy and latency for real time applications. RILaaS allows users to specify and log metrics for each model and each robot over a front-end.

3.3 RILaaS Architecture

RILaaS is divided into four modules: **1) Front-end**, **2) Management Server**, **3) Inference Server** and **4) Request Interceptor**. The front-end provides a simple interface to upload trained models and deploy them on robots. The management server is responsible for storing the authorization policies and deploying the containerized models on requested servers. The inference server computes the response of the incoming queries using specified models. The request interceptor authorizes the use of specified models, while the load-balancer hosted on the request interceptor learns to distribute queries over multiple inference servers. Additionally, the monitoring server collects metrics about the model and the robot performance. The user first uploads or chooses a publicly shared model over the front-end where it is containerized and deployed on the inference server. Robots are added by specifying their public SSH key and subscribing to the desired models. Robots can then query the deployed models over the network using a minimalist client library. The monitoring server runs in the background to log the desired metrics for visualization via the front-end. The overall architecture is summarized in Fig. 3.1.

User End: Front-End and Management Server

RILaaS provides a user-facing REST API that interacts with the management server to create, view and update models, datasets, robots and metrics (see Fig. 3.2 for front-end snapshots). The front-end is a user-based platform that provisions for:

Model Creation: Users upload the model folder containing the pre-trained model weights and specify the input, output types and optional pre-processing and post-processing modules. The management server containerizes the model automatically and uploads the image in a docker repository hosted on AWS. We package each model in a separate Docker container to resolve system conflicts between models and prevent over-utilization of system resources.

Model Sharing: Users can make their models private, public or share with other users on the platform to facilitate re-usability of models across applications.

Robot Creation: Users deploy the uploaded models on robot(s) by adding their public SSH key for authentication. Note that all publicly available models are automatically made available to any robot registered with the service.

Dataset Creation: The front-end allows users to upload test datasets for querying the uploaded models and visualizing the model outputs. The test datasets can similarly be made public for other users to test the models. This allows users to ensure the functioning of their deployed models before querying them from the robot.

Metrics Viewer: A flexible query interface through Prometheus allows users to view metrics about their model/robot such as requests sent/received and the round-trip communication latency times. Additional end-points for metrics can be added via a dedicated endpoint that is asynchronously monitored by the management server.

Robot End: Request Interceptor and Inference Server

The **Request Interceptor** receives the incoming requests from the networked robots and distributes them to the inference servers. The request interceptor may be deployed on the robot itself or at the Edge of the network for a fleet of robots. Note that multiple request interceptors can also be deployed for the same application. The request interceptor is responsible for SSH based authentication of the robots and authorizing access control for the models. Authentication and authorization policies prevent misuse of compute resources by intruders. Authentication is done using JSON Web Tokens (JWT) signed with private SSH key of the robot, while authorization policies are stored in a database in the management server. Naively fetching model access policies from remote databases for every request can slow down inference, thereby, these access policies are stored on a local Redis cache to minimize network calls to a remote database for each robot query. The cache is updated using an event-triggered system that maintains the most recent version of access control policies from the management server. The request interceptor subsequently directs the authorized queries to the inference servers using a user-specified load balancing strategy to optimize the round-trip latency times.

Inference Servers deploy the containerized models on provisioned servers to process the incoming requests. The servers may be placed on Cloud, Edge and/or anywhere along the continuum depending upon the application requirements. Modular resource placement allows the robots to access resources from the Edge and seamlessly switch to the Cloud for scalability if Edge resources are lacking. Moreover, non-critical models can also be rate limited on a per-robot basis in order to prevent DoS attacks from occurring at the Edge and ensure high availability of important models.

Inference Query Life Cycle

RILaaS abstracts away the hardware and software dependencies required for inference of deep robot models. Once a model has been deployed on the RILaaS platform, a robot or a fleet of robots can readily access the deep models by a simple network call after installing the minimalist RILaaS client python package. As shown in the code snippet below, the `RobotClient` object contains the necessary parameters for authentication and authorization of the robot and the required deep model. The robot specifies the target address of the request interceptor, the model name and the model version for inference, the private SSH key of the robot for inference and the SSL certificate location. The SSL certificate encrypts the communication between the robot and the servers. The robot communicates with the servers using gRPC, an open source Remote Procedure Call library built on HTTP/2. Once it is created, the `RobotClient` object is used to make predictions with a simple function call.

```
from client import RobotClient
rc = RobotClient(
        TARGET_IP,
        MODEL_NAME,
        MODEL_VERSION,
        PRIVATE_SSH_KEY_PATH,
```

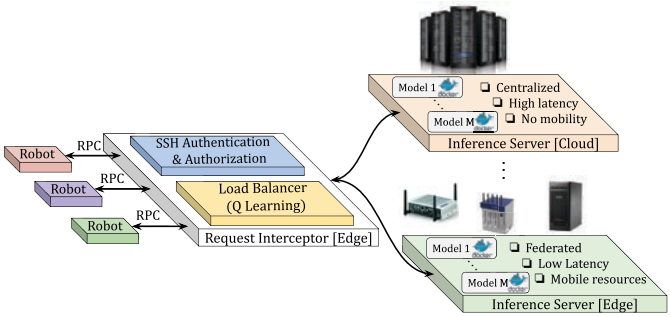

Figure 3.3: Inference optimization with adaptive load-balancing: A Q-Learning algorithm adapts the distribution of the incoming requests from the robots between the Cloud and the Edge resources to optimize the round-trip latency time.

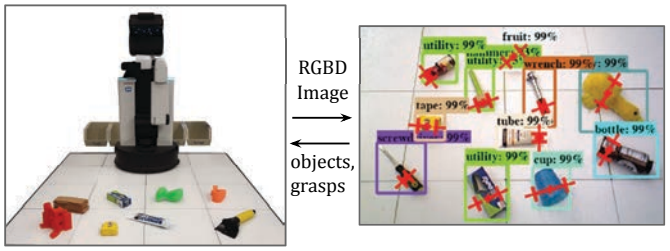

Figure 3.4: Vision-based decluttering application where the robots send the RGBD image of the environment to the inference service and retrieves the object categories and bounding boxes, along with their grasp locations to put the objects in their corresponding bins.

```
            SSL_CERTIFICATE_LOCATION
        )
    outputs = rc.predict(inputs)
```

3.4 Inference Optimization with Adaptive Load-Balancing

The inference requests from a robot or a fleet of robots can be optimized for large-scale serving of deep models. A-priori estimation of querying rate of the model and the round-trip inference time of the model provide a useful criteria for inference optimization. Ensemble modeling is also useful to deploy multiple models of the same task and optimize the inference times. Appropriate model selection can provide a trade-off between accuracy and latency to satisfy the service level objectives [10, 49]. Flexibility in placement and usage of resources can also increase the overall system efficiency. Consider, for example, a resource constrained network where GPUs are available on the Cloud and only CPUs are available at the Edge of the network. Even though a GPU provides superior computation capabilities compared to a CPU, the round-trip communication time

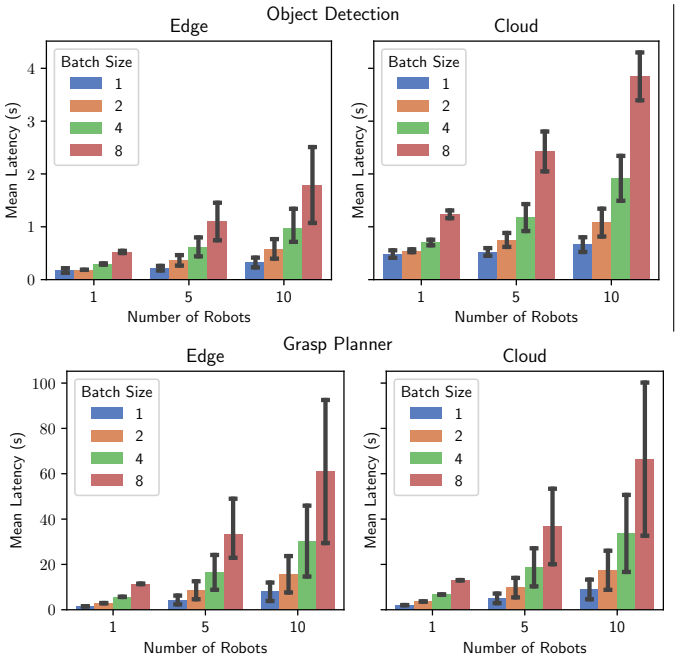

Figure 3.5: Comparison of the average round-trip latency times of the object recognition model on *(left)* and the grasp planning model on *(right)* with the use of Edge or Cloud resources. We make two observations: 1) the round-trip communication time scales sub-linearly with increasing batch size and number of robots across both models, 2) the difference between the Edge and the Cloud latency times is more dominant when the computation time is less than the communication time as for the object recognition model in comparison to the grasp planning model.

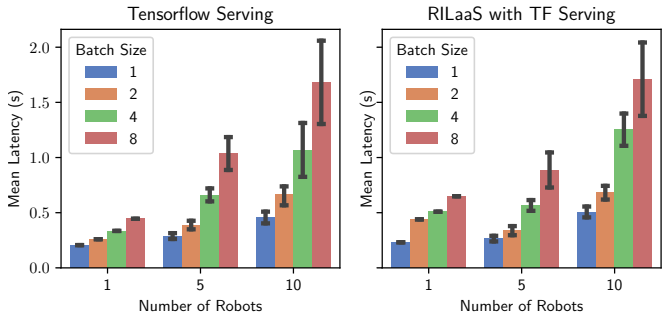

Figure 3.6: A comparison between RILaaS and tensorflow Serving deployed on the Edge for the object detection model. RILaaS performs on par with tensorflow serving and the gap closes further with more users.

of using a GPU in the Cloud vs a CPU locally is application and workload dependent. Note that the capital and operational expense for a CPU is far lower than that of a GPU. Simple application profiling may be used for resource placement in this context [20]. However, finding an appropriate balance for performance and cost is challenging when the application demands and the availability of resources keeps changing over time, making continuous re-evaluation necessary [50].

Load balancing across multiple servers is useful for optimizing resource utilization, reducing latency and ensuring fault-tolerant configurations [38]. Traditional load balancing strategies supported in RILaaS include,

Round Robin: Requests are distributed in a cyclic order regardless of the load placed on each server.

Least Connections: The next request is assigned to the server with the least number of active connections.

Least Model Time: Requests are assigned based on running estimate of average round-trip latency for each model. To prevent choosing a single server for extended periods of time, we randomize the server selection with a probability to explore all available resources.

We use *nginx* [35] for load-balancing with round robin or least connections. The nginx load balancing strategies naively assume homogeneity of servers, i.e., each request takes a similar amount of time to process on available resources. Moreover, the heuristics used in these strategies are not suitable for handling dynamic loads where the number of requests vary over time. In this work, we seek to optimize the inference times under dynamic loads by distributing queries over a set of non-homogeneous servers between the Edge and the Cloud (see Fig. 3.3 for an overview). Note that this work does not directly address the problem of autoscaling (creating new servers) to meet load spikes, instead the focus is on optimizing the load across a fixed set of servers.

We formulate the adaptive load-balancing as a reinforcement learning problem to minimize the expected round-trip latency for each request in a given time horizon on a per-model basis. We assign an 'agent' to each model to distribute the incoming queries, i.e., the number of agents scale linearly with the number of models used. Each agent keeps an estimate of each server in a Markov decision process tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{R} \rangle$ where $\boldsymbol{s}_t \in \mathcal{S}$ is the state representation of the server at time t, $\boldsymbol{a}_t \in \mathcal{A}$ is the action of sending request to one of the N servers which results in transition to a new state $\boldsymbol{s}'_t \in \mathcal{S}'$ along with the reward $r(\boldsymbol{s}_t, \boldsymbol{a}_t) \in \mathcal{R}$ as an estimate of the round-trip latency, i.e.,

$$\boldsymbol{s}_t = \begin{bmatrix} p_{t,1} \ , \ q_{t,1} \\ p_{t,2} \ , \ q_{t,2} \\ \vdots \\ p_{t,N} \ , \ q_{t,N} \end{bmatrix}, \quad \boldsymbol{a}_t = \begin{bmatrix} 1 \\ 2 \\ \vdots \\ N \end{bmatrix}, \quad r_t = -\left(1 + \mathcal{L}(\boldsymbol{s}_t, \boldsymbol{a}_t)\right)^2, \tag{3.1}$$

where $p_{t,i}$ is the number of requests of a model on server i at time t, $q_{t,i}$ represents the total number of active requests of all models on server i at time t, and $\mathcal{L}(\boldsymbol{s}_t, \boldsymbol{a}_t)$ is the round-trip latency of sending and receiving a inference request to a server. Note that the reward function penalizes the high latency times in a quadratic manner. The agent learns to choose the server by taking action $\boldsymbol{a}_t$ such that the expected latency in a given time horizon is minimized as estimated by the Q-function

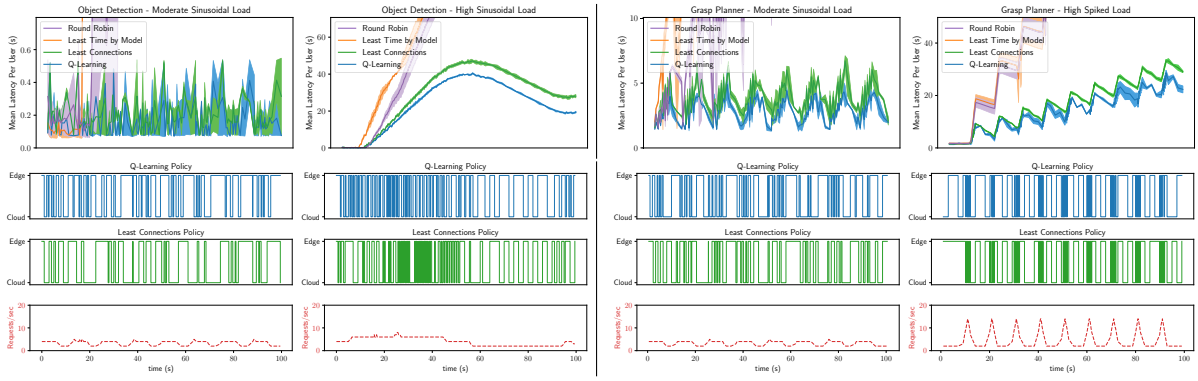

Figure 3.7: Inference optimization of varying test load profiles for object recognition on *(left)* and grasp planning on *(right)*. For each model, top row shows the round-trip latency of load-balancing strategies, second and third row shows the Q-learning and least connections policy output in allocating Edge or Cloud resources, fourth row shows the requests rate profile. Q-learning scales better with increasing loads than other load-balancing strategies by optimally figuring out how to use Edge resources more frequently to reduce the average round-trip latency times.

$Q(\boldsymbol{s}_t, \boldsymbol{a}_t)$,

$$Q(\boldsymbol{s}_t, \boldsymbol{a}_t) = \mathbb{E}\left[\sum_{t=0}^{T} \gamma^t r(\boldsymbol{s}_t, \boldsymbol{a}_t)\right], \quad \boldsymbol{a}_t = \underset{\boldsymbol{a}_t=1\ldots N}{\arg\max}\ Q(\boldsymbol{s}_t, \boldsymbol{a}_t), \tag{3.2}$$

where γ is the discount factor of future rewards. The Q-function is recursively updated using the Bellman equation [42]. With a small probability, a server is randomly chosen to encourage exploration of the state and action space. The agent continuously optimizes the action selection to drive down the latency times for each model based on the observed load profiles from the networked robots. Note that in a real world setting, this training can be done during periods of low usage by replaying past load profiles, while using a competitive baseline until the load balancing model is trained.

3.5 Experiments and Results

We now present experiments for evaluating the RILaaS platform to serve deep models of object recognition and grasp planning on a large scale. We empirically investigate the effect of varying batch size, number of users and resource placement, followed by the adaptive load-balancing experiments to optimize simulated dynamic load profiles with a fleet of robots. We use the Amazon EC2 (East) `p2.1xlarge` instance with 1 Tesla K80 GPU in Northern Virginia (us-east-1) for Cloud compute and use Amazon S3 buckets for Cloud storage. The Edge infrastructure comprises of a workstation with 1 NVidia V100 GPU located at a nearby data center.

Application Workloads

We consider real-world application scenarios where RILaaS is used to provide object recognition and grasp planning as a service for vision-based robot decluttering, building upon our previous work in [43, 40].

Object Recognition: We use the MobileNet-Single Shot MultiBox Detector (SSD) [27] model with focal loss and feature pyramids as the base model for object recognition (other well-known models including YOLO, Faster R-CNN etc. can similarly be used [12]). The input RGB image is fed to a pre-trained VGG16 network, followed by feature resolution maps and a feature pyramid network, before being fed to the output class prediction and box prediction networks. The model is trained on 12 commonly used household and machine shop object categories using a combination of synthetic and real images of the environment.

Grasp Planning: Robots in homes, factories or warehouses require robust grasp plans in order to interact with objects in their environment. We use the Dex-Net grasp planning model to plan grasps from the depth images of the environment [28]. The model samples antipodal pairs from a depth image and feeds them to a convolutional neural network to predict the probability of success. The sampled grasps are successively filtered with a cross-entropy method to return the most likely grasp. Note that the pre-processing step of sampling many different grasps requires CPU usage, whereas predicting the grasp success requires GPU resources for efficient grasp planning.

Vision-Based Decluttering: We sequentially pipeline the object recognition and grasp planning models together for vision-based surface decluttering [43]. The robot sends RGBD images of the environment, where the RGB image is used for object recognition and the cropped depth image from the output bounding box of the object recognition model is used by the grasp planning model to output the top ranked grasp for the robot to pick and place the object into its corresponding bin (see Fig. 3.4).

Scalability of RILaaS

We deployed the trained models on the RILaaS platform to receive images from the robot, perform inference, and send back the output results to the robot. We measure the round-trip time $t^{(\mathrm{rtt})}$, i.e., time required for communication to/from the server and the inference time $t^{(\mathrm{inf})}$. We experiment with two hosts for the inference service: EC2 Cloud (East), and Edge with GPU support.

Resource Placement with Cloud vs Edge: Results in Table 3.1 show that the communication time is a major component of the overall round-trip latency time. Deploying the inference service on the Edge significantly reduces the round-trip inference time and the timing variability in comparison to hosting the service on Cloud, with a communication overhead of around 100 milliseconds only. The difference in resource placement is less pronounced for grasp planning model where CPU computation time is a dominant factor. Moreover, the authentication time only takes 1 millisecond on average with Redis cache in comparison to 630 milliseconds with a relational database on AWS.

Effect of Batch Size and Number of Robots: We next vary the batch size and number of robots making concurrent requests to the service. Fig. 3.5 suggests that the average round-trip

Table 3.1: Computation time for inference $t^{(\text{inf})}$ vs round trip communication time $t^{(\text{rtt})}$ (in milliseconds) for inference over Edge and EC2-East Cloud. Results are averaged across 6 trials. Communication time dominates the computation time and increases as the distance to the server increases.

Location	$t^{(\text{inf})}$	$t^{(\text{rtt})}$
Object Detection		
EC2-East	42.79 ± 0.41	483.82 ± 70.87
Edge	$\mathbf{36.03 \pm 3.18}$	$\mathbf{172.77 \pm 43.55}$
Grasp Planner		
EC2-East	1501.61 ± 12.76	2051.48 ± 22.684
Edge	$\mathbf{1386.95 \pm 22.92}$	$\mathbf{1515.59 \pm 26.16}$

latency grows sub-linearly with the batch size and the number of robots querying the service. Moreover, deploying models on Edge yields lower round-trip latency times across both models, but the difference is more pronounced for the object recognition model with lower computation time than the grasp planning model.

Comparison with Tensorflow Serving: Fig. 3.6 suggests that RILaaS gives comparable results to tensorflow-serving for the object recognition model deployed at the Edge. Note that the tensorflow-serving does not provide out-of-the-box pre-processing/post-processing, authentication, authorization and metrics for models that it supports. Consequently, the grasp planning model cannot be hosted on tensorflow-serving as it iterates over preprocessing and inference. RILaaS supports a tensorflow-serving backend while providing the aforementioned features to make it feasible for a wide variety of models.

Inference Optimization under Dynamic Loads

We simulate time-varying requests of different profiles to evaluate the performance of inference optimization with adaptive load-balancing. We query the object recognition and grasp planning model alternatively at specified rates to simulate the decluttering setup, and compare the Q-learning based adaptive load-balancing with round robin, least connections and least model time strategies. The request profiles include: 1) **uniform loads** of $1, 2, 4, 8$ requests per second, 2) **step-wise increasing loads** of $1, 2, 3, 4$ requests per second, 3) **spiked loads** where nominal load of 2 requests per second is augmented with 13 requests per second for up to 2 seconds, 4) **Poisson distributed loads** where requests follow the Poisson process with arrival rate of $1, 2, 4, 8$ requests per second, 5) **sinusoidal loads** with varying amplitudes and frequencies of $0.05, 0.01, 0.08$ Hz. The first 4 types of load profiles are used for both training and testing, while the sinusoidal load profiles are only used for testing of the optimal inference serving policy.

Fig. 3.7 shows the plots of the object recognition and grasp planning model for various request profiles. It can be seen that the Q-learning strategy outperforms the commonly used load-balancing

strategies. Least-connections performance is better among the compared load-balancing strategies and its performance is similar to Q-learning for lighter workloads. The inference serving policy reveals that the Q-learning is able to decrease the average latency times by more frequently using the Edge resource as compared to the Cloud. Overall, the adaptive load-balancing strategy with Q-learning for object recognition gives 15.76% and 70.7% decrease in round-trip latency time compared to the next best least connections and worst performing round-robin baseline. Similarly, the grasp planning model shows 12.32% and 65.91% decrease in the round-trip latency time with Q-learning in comparison to least connections and round-robin strategies.

Chapter 4

Parallelizing Graph Based Motion Planning Using Lambda Serverless Computing

4.1 Introduction

Motion planning for robots is a common subprocedure that is required to complete more complex robotics tasks. For robots in cluttered environments that have many degrees of freedom, planning can be computationally challenging [5]. Additionally, motion planning problems can have highly varied computational costs. Investing in local computational resources to solve this problem will lead to low utilization of these resources on the one hand, and introduce problems with maintenance and scalability of resources on the other hand. Moreover general motion planning problems scale exponentially with the dimension of the space, which means that low dimensional problems might be tractable on robot hardware whereas high dimensional problems may not. Consider a robot tasked with cleaning an office space (see Fig. 4.1): computing motion plans to move between rooms is relatively inexpensive as the problem can be reduced to finding paths in a 2-dimensional space that the robot can track. Planning manipulator arm motions to grasp objects in each of these rooms requires solving higher-dimensional problems that requires vastly more computational power for a short amount of time. Similarly in a warehouse or a factory scenario, robots often have to plan manipulator arm trajectories for many motions within a single work cell. This paper proposes methods to leverage the elasticity of the cloud by using serverless computing to parallelize computations of complex motion plans for multi-query motion planning.

In our previous work [14], serverless computing provides on-demand parallelism for tree-based planning algorithms. One problem with tree-based planners is the need to replan from scratch with every new start and goal, even if the robot is operating in the same environment. For example, in the office decluttering scenario, separate motion plans need to be computed for each object on a desk. Reusing exploration from previous motion plans could greatly reduce the amount of time required to move each object. Similarly in a warehouse scenario, robots often have to execute multiple pick-and-place motions in the same environment to reach different grasping positions. In this paper, we propose a parallelized serverless graph-based motion planner as it allows for efficient

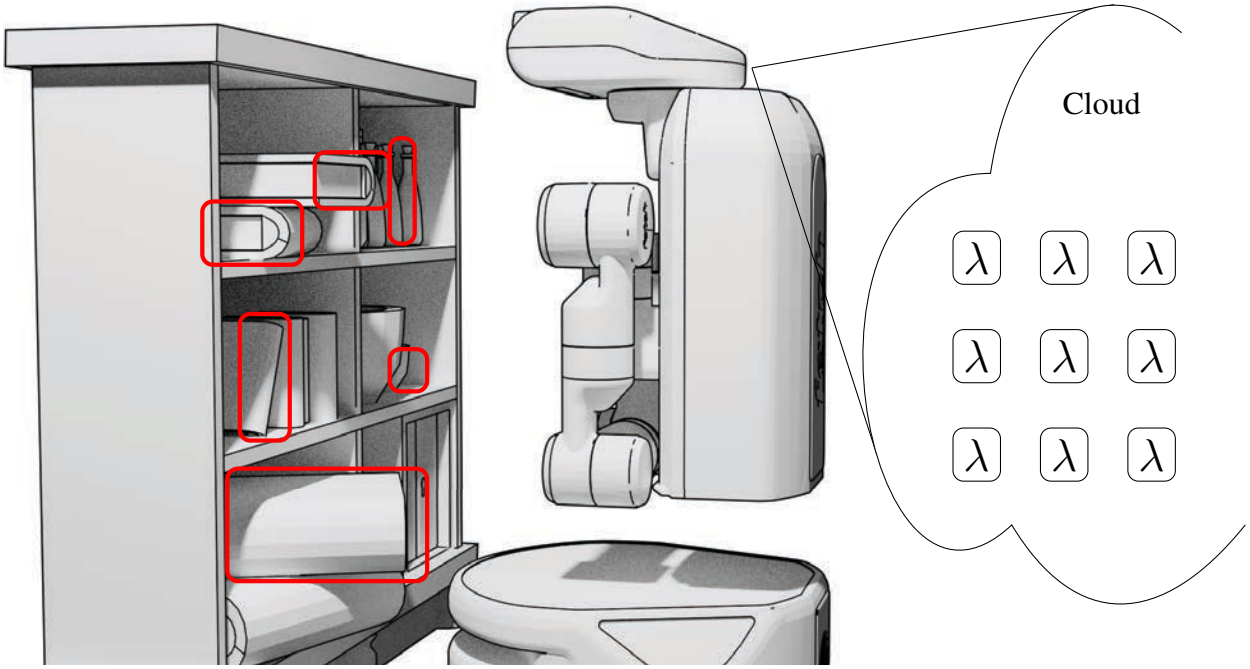

Figure 4.1: A robot organizes a shelf by computing a sequence of motions for its manipulator arm to grasp and place various objects (shown in red). As the obstacle environment does not change between tasks, the robot can compute a motion plan graph once for the sequence of tasks, and then use shortest searching on the graph to complete the sequence of tasks. We propose speeding up the precomputation of the graph using cloud-based parallel serverless "Lambda" computing, allowing the robot to more efficiently use computing while spending wall-clock time before it can start the tasks.

reuse of previous exploration in these multi-query paradigms.

Serverless computing, or Function-as-a-Service (FaaS), is a paradigm in which simple functions can be run on-demand on various cloud and edge systems, and pricing is granular at 100 ms resolution [2]. The intermittent workloads that home and office motion-planning scenarios present match serverless paradigm of elastically scaling compute resources to minimize cost and meed demand. Although each compute unit in a serverless scenario is limited in its computational capabilities, an efficient parallel algorithm can take advantage of the availability of a massive number of such units to achieve large speedups.

This chapter makes the following contributions:

1. a distributed parallel algorithm for computing probabilistically-complete and asymptotically-optimal motion plans using serverless computing

2. an implemented system of the above strategies on Amazon Web Services FaaS "Lambda"

environment

3. time-bounded allocation of resources for motion-planning for both single query and multi-query problems

4. experiments in simulation and on the Fetch mobile manipulator that suggesting that the proposed algorithms provide speedups against local baselines

4.2 Problem Statement

In this paper we propose speeding up multi-query motion planning through the use of cloud-based serverless computing. In this section we formalize the motion planning problem, and provide backgound on the serverless computing environment.

Multi-query Motion Planning Problem

Let $\vec{q} \in \mathcal{C}$ be the complete specification of a robot's degrees of freedom (e.g., joint angles, position and orientation in space), where $\mathcal{C}$ is the configuration space, the set of all possible configurations. Let $\mathcal{C}_{\text{obstacle}} \subset \mathcal{C}$ be the configurations that are in obstacle or otherwise violate task-specific constraints, and the remaining configurations $\mathcal{C}_{\text{free}} = \mathcal{C} \setminus \mathcal{C}_{\text{obstacle}}$ is the free space. Let $L : \mathcal{C} \times \mathcal{C} \to \{0, 1\}$ be an indicator function that is 1 if the path between two configurations is entirely in $\mathcal{C}_{\text{free}}$ according to a *local planner*, and 0 otherwise. Given a start configuration $\vec{q}_{\text{start}} \in \mathcal{C}_{\text{free}}$ and a goal configuration $\vec{q}_{\text{goal}}$, the objective of motion planning is to find a sequence $\tau = (\vec{q}_0, \vec{q}_1, \ldots, \vec{q}_n)$ such that $\vec{q}_0 = \vec{q}_{\text{start}}$, $\vec{q}_n = \vec{q}_{\text{goal}}$, and $L(\vec{q}_i, \vec{q}_{i+1}) = 1$ for all $i \in [0, n)$.

The objective of multi-query motion planning is to precompute a data structure that allows for the efficient computation of τ given changing $\vec{q}_{\text{start}}$ and $\vec{q}_{\text{goal}}$.

Given a cost function $d : \mathcal{C} \times \mathcal{C} \to \mathbb{R}^+$, let $c(\tau) = \sum_{i=0}^{n-1} d(\vec{q}_i, \vec{q}_{i+1})$. The objective of optimal motion planning is to compute a τ that minimizes $c(\tau)$. An asymptotically-optimal motion planner finds a τ such that $c(\tau) = c(\tau^*) + \epsilon$, where $c(\tau^*)$ is the optimal motion plan cost, and ϵ decreases towards 0 with additional computation.

Serverless Computing Environment

The robot has an onboard computer and networked access to a cloud-based computing service. There are two classes of cloud-based computing service: always-on computers (servers) of varying size and serverless computing. For server-based computing, users provision certain number of servers and are charged based on the computing capabilities and generally in 1 h increments. Serverless computing, on the other hand, allows for unbounded concurrent execution of single (possibly multi-threaded) functions, that do not store state between executions, cannot accept inbound network connections, and have bounded runtime. Serverless computing is charged in very short (e.g., 100 ms) increments. The goal of our serverless multi-query motion planning is to

perform parallel precomputation step of multi-query motion planning while avoiding the over-allocation (overcharging) of server-based computing.

4.3 Method

In this section we describe the parallelized serverless sampling-based motion planner. The motion planner is based on PRM*, for which we provide a brief background. We then present the algorithm that runs on the serverless cloud. Due to constraints on serverless computing, specifically statelessness and only allowing outbound network connections, we define a coordinator algorithm that runs on a separate computer (either in the cloud or not) that allows inbound connections and can keep state. Note that a robot with a public IP can be used as the coordinator and bypass a separate provisioned server. Moreover if a coordinator server is provisioned, it can be a lightweight instance with a far lower cost than the more compute intensive instances required for motion planning.

Probabilistic Road Maps (PRM) and PRM* Background

The Probabilistic Road Maps (PRM) [21] motion planner randomly samples configurations to build a graph of the connectivity of the environment. This graph is then subsequently be searched to find paths between any two points. The original version of the algorithm samples n configurations in $\mathcal{C}_{\text{free}}$ and connect pairs of configurations that are at most a distance of r_{prm} away provided there is a collision-free path between them according to a local planner. The next phase is the query phase in which a shortest-path searches (e.g., Djikstra's) compute a path connection arbitrary start and goal configurations. PRM is probablistically complete, but not necessarily asymptotically optimal, depending on choice of r_{prm}. Karaman et al. [19] presented PRM* which provides a lower bound r^*_{prm} such that PRM with $r_{\text{prm}} > r^*_{\text{prm}}$ is asymptotically optimal. This value of r^*_{prm} is:

$$r^*_{\text{prm}} = \gamma^*_{\text{prm}} \cdot \left(\frac{\log(n)}{n}\right)^{\frac{1}{d}},$$

where γ^*_{prm} is determined by the volume of obstacle-free space and the dimension of the planning environment.

Serverless Algorithm

To parallelize the computation of the PRM, we propose exploiting the determinism of sampling methods used to construct these graphs. In particular, random number generators create deterministic sequences of points when provided with a particular seed. Thus as long as all lambdas are initialized with the same random seed, they will sample the same set of points. Additionally, the sampling stage of the PRM algorithm is much cheaper than the nearest neighbor queries and the connection of edges. For instance, sampling and validating 1000 points for an 8 dimensional

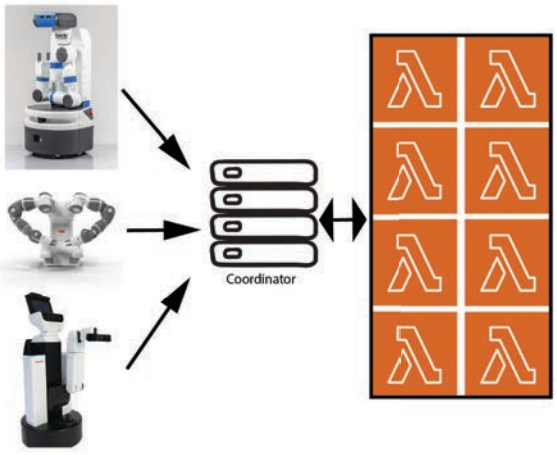

Figure 4.2: A central coordinator handles initializing lambdas and maintains open connections to them to allow communication between lambdas. Note that the coordinator need not be a very large instance as it performs mostly a network bound task. Moreover, multiple robots can reuse a single coordinator for maximum efficiency.

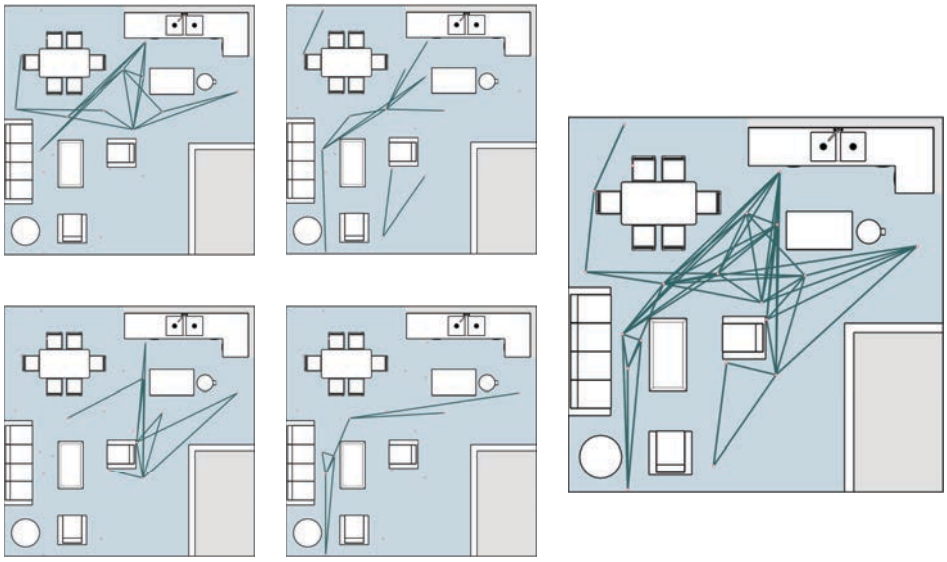

Figure 4.3: Each of the 4 lambdas on the left connect a subset of the edges in the sampled vertices. These edges are sent to the coordinator which combines them into the complete graph on the right.

Algorithm 1 Lambda Algorithm

Require: L is the planner over subspace S
$G = (V = \varnothing, E = \varnothing)$
while not done **do**
  $\vec{q}_{\text{rand}} \leftarrow$ random sample
  if $\vec{q}_{\text{rand}} \in \mathcal{C}_{\text{free}}$ **then**
    if $i \bmod \text{num\_lambdas} = lambda\_id$ **then**
      for $\vec{q}_{\text{near}}$ in `near` ($\vec{q}_{\text{rand}}$, r_{prm}) **do**
        if ID($\vec{q}_{\text{near}}$) > ID($\vec{q}_{\text{rand}}$) **then**
          `steer`($\vec{q}_{\text{rand}}$, $\vec{q}_{\text{near}}$)
        end if
      end for
    end if
    $i = i + 1$
  end if
  if n_{samples} = sample threshold **then**
    $n_{\text{samples}} = 0$
    Send new vertices/ edges to coordinator
  end if
  Update r_{prm}
end while

Algorithm 2 Coordinator Algorithm

Receive problem specification from robot
$G = (V = \varnothing, E = \varnothing)$
for lambda in num_lambdas **do**
  `initializeLambda`(lambda)
end for
while not done **do**
  for lambda in lambdas **do**
    Receive new vertices and edges G_l from lambda
    $G = G \bigcup G_l$
  end for
  if time limit exceeded **then**
    exit and return graph to robot
  end if
end while

space took 0.063 seconds, whereas connecting the edges for the above samples took 6.194 seconds. These two properties allow us to tradeoff some redundant sampling work for communication of vertices between lambdas.

Figure 4.4: A Fetch robot performing two decluttering tasks in the experiments. As the robot approaches a desk (left) and a bookshelf (right) it computes a road map of motions for its 8-DOF manipulator arm using serverless computing, it then uses to perform a sequence of decluttering tasks.

In particular, all the lambdas perform the sampling step. However, we cyclically choose which vertices each lambda is responsible for connecting to the existing graph. Additionally, since the constructed graph is undirected, each edge is added to the coordinator twice. To avoid this redundant computation an ordering is enforced where an edge is only attempted to be connected if the source vertex has an id that is less than the target vertex. While this ordering strategy can create load imbalances for a small number of vertices, in the limit of many vertices which are required for sufficient exploration of the space, each vertex should have approximately the same number of target edges to connect.

We maintain the probabilistic completeness and asymptotic optimality of the PRM algorithm as the edges and vertices are exactly the same ones that would be created in the serial version of the algorithm.

4.4 Experiments and Results

To test the proposed motion plannner's ability to speed up motion planning with serverless computation, we experiment on a Fetch robot tasked with decluttering an office space. We use Amazon's Lambda as the serverless computing environment, selecting the maximum CPU and memory allocation setting for each lambda. The coordinating server runs on an c5.xlarge instance (two 64-bit Arm Neoverse cores on a Graviton Processor) in the same region as the lambda processes. Since the algorithm is based on random sampling, all experiments are run for 10 trials and the median values are plotted (since the median is less sensitive to outliers).

Bandwidth Optimization

To provide a framework for the remainder of the results presented, we will first describe bandwidth bottlenecks that we faced when scaling to larger numbers of lambdas due to the centralized

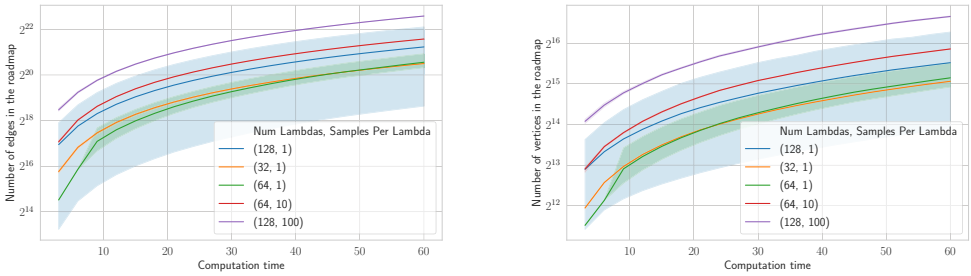

Figure 4.5: Tuning the communication to optimize for bandwidth. The second parameter in the legend indicates how many points were sampled by each lambda before communicating with the coordinator. A higher number of samples before communication increases the packet size sent. In the graph we see that a small packet size of 1 does not scale to large numbers of lambdas, but increasing the packet size results in better scaling of the number of edges (left) and number of vertices (right).

coordinator. The parameter that decides the bandwidth is s: the number of points each lambda samples before communicating any newly found edges to the coordinator. A higher s will increase the packet size, reduce the frequency of communication and improve the bandwidth of communication. However, if s is too large, lambdas might exit before sending any points to the coordinator due to the time-bound nature of the computation. Thus it is ideal to set s to be as small as possible until the coordinator is unable to keep up with the frequency of packets.

As seen in Fig. 4.5 for more than 64 lambdas the coordinator is unable to receive all the vertices and edges in time. Note that for 64 and 128 lambdas, higher values of s at around $10 - 100$ are sufficient to unblock the coordinator and continue to allow the algorithm to scale. For the remainder of the experiments, different numbers of lambdas will be compared only with the optimized value of s.

Graph Size Scaling

In order to measure the parallel efficiency of the algorithm, we look at how three different quantities scale across both Fetch scenarios: the number of samples generated, the number of vertices generated and the number of edges generated. We expect sublinear scaling for the number of vertices and samples as each new vertex added to the graph has a higher cost for nearest neighbor queries and more edges to add. On the other hand, we expect atleast linear scaling for the number of edges generated as each edge should be generated exactly once globally. Parallel efficiency for these three quantities (denoted by n) are computed at $t = 60s$ for k lambdas using $e = \frac{n_k}{k \cdot n_1}$. From Fig. 4.6 and Table. 4.1 we observe the expected superlinear scaling for the number of edges with a mean parallel efficiency of 1.16, sublinear scaling for the number of samples generated with a mean parallel efficiency of 0.472 and sublinear scaling for the number of vertices generated with a

Num Lambdas	Sample Scaling	Vertex Scaling	Edge Scaling
1	1.00	1.00	1.00
4	0.71	0.73	1.10
8	0.60	0.62	1.15
32	0.40	0.45	1.19
64	0.35	0.39	1.26
128	0.30	0.34	1.26

Table 4.1: Parallel efficiency results for the common seed algorithm. Edge scaling is linear or superlinear as expected, while sample scaling and vertex scaling are sublinear due to the additional cost incurred for every vertex added.

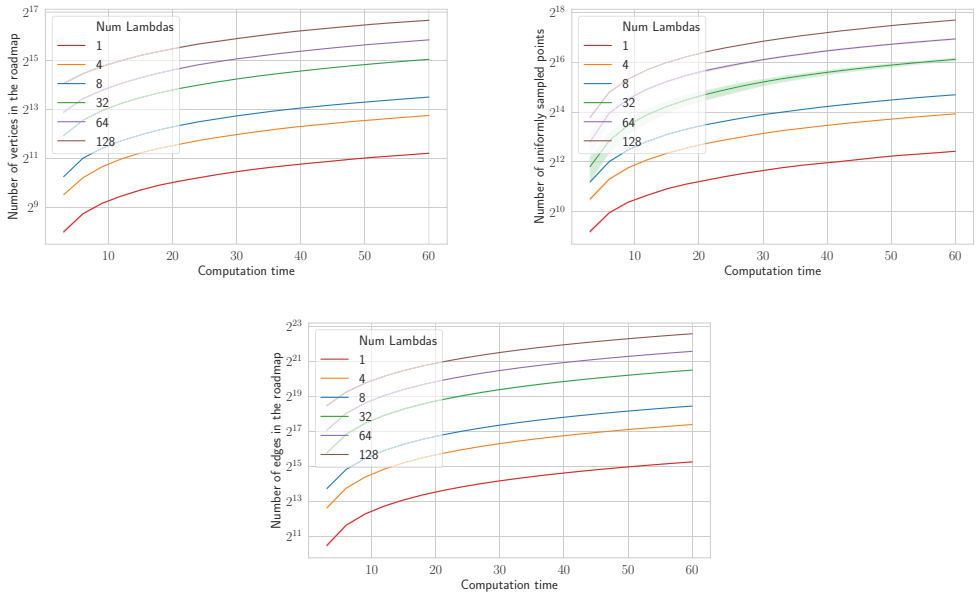

Figure 4.6: Scaling results for the common seed algorithm. Edge scaling (bottom) is linear or superlinear as expected, while sample scaling (top right) and vertex scaling (top left) are sublinear due to the additional cost incurred for every vertex added.

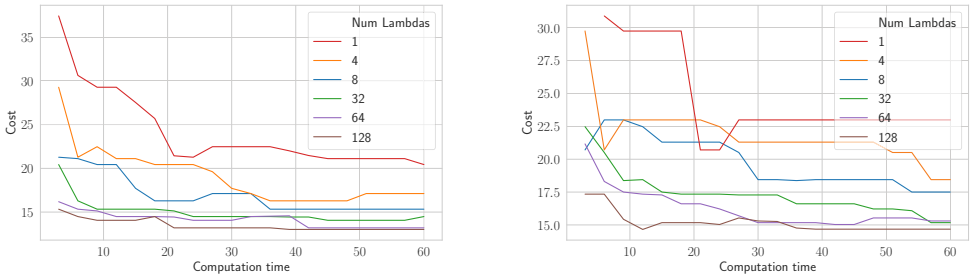

Figure 4.7: Path costs for two randomly sampled goal positions with increasing time and varying the number of lambdas. The marginal performance improvement reduces with more lambdas, however there is usually some benefit to increased scaling.

Num Lambdas	Path Cost at 3s	Path Cost at 60s	$t_{\text{to\_best\_at\_3}}$	$t_{\text{to\_best\_at\_9}}$
1	inf	20.23	11.68	9.68
4	27.72	17.16	10.41	9.17
8	22.70	15.99	7.56	8.41
32	19.52	15.07	2.68	4.32
64	16.90	14.44	1.74	2.34
128	15.63	14.30	1.00	1.00

Table 4.2: Path scaling results for the common seed algorithm. The first two columns indicate the mean path cost at 3s and the mean path cost at 60s. The next two columns represent a time multiplier of how much additional time fewer lambdas take to find equivalent solutions to the solution 128 lambdas finds in 3s and 9s.

mean parallel efficiency of 0.506.

Path Cost Scaling

In order to measure the degree of scaling in terms of path costs, 20 randomly sampled goals in front of the robot were chosen and path costs from the start position to these goals were computed. The following 4 metrics are computed across both Fetch scenarios:

- The mean path cost across all goals at 3 seconds
- The mean path cost across all goals at 60 seconds
- The mean increase in time (as a multiple) taken for smaller number of lambdas to get an equivalent path found by 128 lambdas in 3 seconds

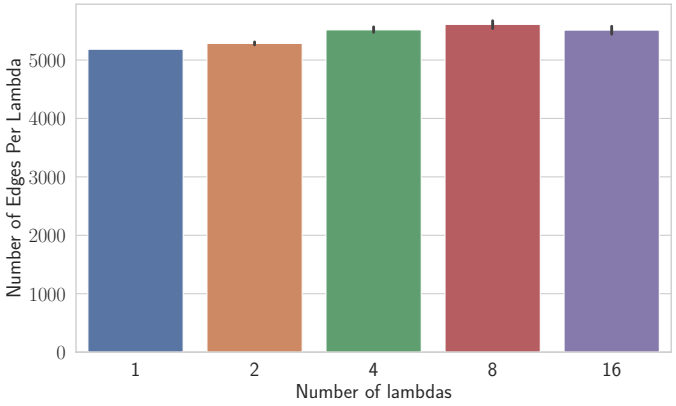

Figure 4.8: Edges sampled per lambda over a 20 second period are nearly equal indicating good load balancing of work even though the edge connections are random.

- The mean increase in time (as a multiple) taken for smaller number of lambdas to get an equivalent path found by 128 lambdas in 9 seconds

The last 2 metrics are used to demonstrate the time speedup provided by using the maximum number of lambdas. Note that for these two metrics, if no equivalent solution was found by 60 seconds then the time to find was set to 60. From Table. 4.2 we see that there is a large improvement in the path cost with increasing numbers of lambdas. Additionally we see that the time benefit provided by using more lambdas is often sufficient to justify the use of increased numbers of lambdas: 128 lambdas finds solutions atleast $9-11$ times quicker than a single lambda. In practice this speedup will be higher because a single lambda often cannot find the same solution as 128 lambdas even after 60 seconds as is observed in Fig. 4.2.

Distribution of Work

Due to the random nature of this algorithm, the distribution of work among lambdas is also measured. Ideally we would want the number of edges sampled by each lambda to be nearly the same to have an equal distribution of work. The number of edges each lambda generates for varying numbers of lambdas is plotted in Figure. 4.8. From the figure it is clear that each lambda does roughly the same amount of work over a 20s period, indicating good load balancing for this problem.

Chapter 5

Conclusion and Future Work

Virtualizing robot storage, compute and programming is a key enabler for large-scale learning and inference of deep models for robotic applications. In chapter 3 we introduced RILaaS as a novel user-based inference serving platform for deploying deep learning models on robots that satisfies heterogeneous model support, rapid deployment, security and privacy, and low latency requirements of the applications. We used reinforcement learning for scalable inference serving that adapts better with dynamic loads than commonly used load balancing strategies. We provide deep object recognition and grasp planning as a service and showed its application to vision-based decluttering of objects from the floor and depositing them in target bins. To the best of our knowledge, RILaaS is the first of its kind user based inference serving platform of deep models for robotic applications. In future work, we plan to couple a digital twin/simulator with the uploaded models for efficient sim-to-real transfer and federated learning with a fleet of robots. Moreover, we will test various models for segmentation, hierarchical task planning etc. in a multi-agent distributed environment with a set of robots.

In chapter 4 we proposed using cloud-based serverless computing to rapidly compute a probabilistically complete and asymptotically-optimal road map for multi-query motion planning. The serverless computing environment provides a nearly unbounded source of parallelism that we exploit by determining how much parallelism we wish to use depending on the speed and quality of solution required. Each lambda then samples and builds a graph on a subset of vertices, periodically sharing information with the coordinator about parts of the graph. In experiments for a Fetch robot, the proposed serverless computing sped up motion planning computation by atleast 9 times while providing linear scaling in the number of edges in the graph with a parallel efficiency of 1.16, suggesting this approach can be used in practice to speed up sproadically computationally-intensive motion-planning problem while being more cost effective than an always-on high-end computer. In future work, we plan to explore parallelizing the sampling stage of the process, as well as dynamically allocating more lambdas to work on specific regions of the state space that are difficult to explore.

Overall, we explored two different ways to distribute computation of robot algorithms over a heterogenous Fog infrastructure: the first a general purpose platform to deploy deep models and the second a problem specific parallelization of motion planning using serverless computing. As

robots become more prevalent in our lives, it will be necessary to expand upon these paradigms and ensure that roboticists have access to simple ways to deploy their algorithms at scale and take advantage of the Fog.

Bibliography

[1] Nancy M. Amato and Lucia K. Dale. “Probabilistic roadmap methods are embarrassingly parallel”. In: *Proceedings IEEE Int. Conf. Robotics and Automation (ICRA)*. May 1999, pp. 688–694.

[2] Amazon Web Services, Inc. *AWS Lambda – Pricing*. URL: https://web.archive.org/web/20190909111142/https://aws.amazon.com/lambda/pricing/ (visited on 09/09/2019).

[3] Raghav Anand, Jeff Ichnowski, et al. “Distributed Multi-Query Serverless Motion Planning”. In: 2020.

[4] Jung-Yeon Baek, Georges Kaddoum, et al. “Managing Fog Networks using Reinforcement Learning Based Load Balancing Algorithm”. In: *CoRR* abs/1901.10023 (2019).

[5] John Canny. *The complexity of robot motion planning*. MIT press, 1988.

[6] Stefano Carpin and Enrico Pagello. “On parallel RRTs for multi-robot systems”. In: *Proceedings 8th Conference Italian Association for Artificial Intelligence* (2002).

[7] S. Chanchali, A. Sharma, et al. “Network Offloading Policies for Cloud Robotics: a Learning-based Approach”. In: *CoRR* abs/1902.05703 (2019).

[8] Howie Choset, Kevin M. Lynch, et al. *Principles of Robot Motion: Theory, Algorithms, and Implementations*. MIT Press, 2005.

[9] Daniel Crankshaw, Gur-Eyal Sela, et al. “InferLine: ML Inference Pipeline Composition Framework”. In: *CoRR* abs/1812.01776 (2018). arXiv: 1812.01776.

[10] Daniel Crankshaw, Xin Wang, et al. “Clipper: A Low-Latency Online Prediction Serving System”. In: *CoRR* abs/1612.03079 (2016). arXiv: 1612.03079.

[11] Siva Leela Krishna Chand Gudi, Suman Ojha, et al. *Fog Robotics for Efficient, Fluent and Robust Human-Robot Interaction*. 2018. arXiv: 1811.05578 [cs.RO].

[12] Jonathan Huang, Vivek Rathod, et al. “Speed/accuracy trade-offs for modern convolutional object detectors”. In: *CoRR* abs/1611.10012 (2016). arXiv: 1611.10012.

[13] Jeffrey Ichnowski and Ron Alterovitz. “Scalable multicore motion planning using lock-free concurrency”. In: *IEEE Transactions on Robotics* 30.5 (2014), pp. 1123–1136.

[14] Jeffrey Ichnowski, William Lee, et al. “Fog Robotics Algorithms for Distributed Motion Planning Using Lambda Serverless Computing”. In: *Proceedings IEEE Int. Conf. Robotics and Automation (ICRA)*. June 2020.

[15] Jeffrey Ichnowski, Jan F. Prins, and Ron Alterovitz. "Cache-aware asymptotically-optimal sampling-based motion planning". In: *Proceedings IEEE Int. Conf. Robotics and Automation (ICRA)*. June 2014.

[16] Jeffrey Ichnowski, Jan Prins, and Ron Alterovitz. "The Economic Case for Cloud-based Computation for Robot Motion Planning". In: *Proceedings International Symposium on Robotics Research (ISRR)*. 2017, pp. 1–7.

[17] Sam Ade Jacobs, Kasra Manavi, et al. "A scalable method for parallelizing sampling-based motion planning algorithms". In: *Proceedings IEEE Int. Conf. Robotics and Automation (ICRA)*. 2012, pp. 2529–2536.

[18] Sam Ade Jacobs, Nicholas Stradford, et al. "A scalable distributed RRT for motion planning". In: *Proceedings IEEE Int. Conf. Robotics and Automation (ICRA)*. May 2013, pp. 5073–5080. ISBN: 9781467356428.

[19] Sertac Karaman and Emilio Frazzoli. "Sampling-based algorithms for optimal motion planning". In: *The International Journal of Robotics Research* 30.7 (June 2011), pp. 846–894. ISSN: 0278-3649.

[20] A. Kattepur, H. K. Rath, and A. Simha. "A-Priori Estimation of Computation Times in Fog Networked Robotics". In: *2017 IEEE International Conference on Edge Computing (EDGE)*. 2017, pp. 9–16. DOI: 10.1109/IEEE.EDGE.2017.11.

[21] L E Kavraki, P Svestka, J.-C. Latombe, and M Overmars. "Probabilistic roadmaps for path planning in high dimensional configuration spaces". In: *IEEE Trans. Robotics and Automation* 12.4 (1996), pp. 566–580.

[22] B. Kehoe, S. Patil, P. Abbeel, and K. Goldberg. "A Survey of Research on Cloud Robotics and Automation". In: *Automation Science and Engineering, IEEE Transactions on* 12.2 (2015), pp. 398–409.

[23] Ben Kehoe, Sachin Patil, Pieter Abbeel, and Ken Goldberg. "A survey of research on cloud robotics and automation". In: *IEEE Transactions on Automation Science and Engineering* 12.2 (2015), pp. 398–409.

[24] Steven M LaValle and James J. Kuffner. "Randomized kinodynamic planning". In: *The International Journal of Robotics Research* 20.5 (May 2001), pp. 378–400.

[25] Sergey Levine, Peter Pastor, et al. "Learning hand-eye coordination for robotic grasping with deep learning and large-scale data collection". In: *IJRR* 37.4-5 (2018), pp. 421–436. DOI: 10.1177/0278364917710318.

[26] Yanbo Li, Zakary Littlefield, and Kostas E Bekris. "Asymptotically optimal sampling-based kinodynamic planning". In: *The International Journal of Robotics Research* 35.5 (2016), pp. 528–564.

[27] Tsung-Yi Lin, Priya Goyal, et al. "Focal Loss for Dense Object Detection". In: *2017 IEEE International Conference on Computer Vision (ICCV)* (Oct. 2017). DOI: 10.1109/iccv.2017.324.

[28] Jeffrey Mahler, Jacky Liang, et al. "Dex-Net 2.0: Deep Learning to Plan Robust Grasps with Synthetic Point Clouds and Analytic Grasp Metrics". In: *CoRR* abs/1703.09312 (2017). arXiv: 1703.09312.

[29] M. Mukherjee, L. Shu, and D. Wang. "Survey of Fog Computing: Fundamental, Network Applications, and Research Challenges". In: *IEEE Communications Surveys Tutorials* (2018), pp. 1–1. DOI: 10.1109/COMST.2018.2814571.

[30] Almuthanna T. Nassar and Yasin Yilmaz. "Reinforcement-Learning-Based Resource Allocation in Fog Radio Access Networks for Various IoT Environments". In: *CoRR* abs/1806.04582 (2018). arXiv: 1806.04582.

[31] Christopher Olston, Fangwei Li, et al. "TensorFlow-Serving: Flexible, High-Performance ML Serving". In: *Workshop on ML Systems at NIPS*. 2017.

[32] Michael Otte and Nikolaus Correll. "C-FOREST: Parallel Shortest Path Planning With Superlinear Speedup". In: *IEEE Transactions on Robotics* 29.3 (2013), pp. 798–806. ISSN: 1552-3098. DOI: 10.1109/TRO.2013.2240176.

[33] Li Pusong, B. DeRose, et al. "Dex-Net as a Service (DNaaS): A Cloud-Based Robust Robot Grasp Planning System". In: *14th International Conference on Automation Science and Engineering (CASE)*. 2018, pp. 1–8.

[34] Morgan Quigley, Ken Conley, et al. "ROS: an open-source Robot Operating System". In: *ICRA Workshop on Open Source Software*. Vol. 3. Jan. 2009.

[35] Will Reese. "Nginx: The High-Performance Web Server and Reverse Proxy". In: *Linux J.* 2008.173 (2008). ISSN: 1075-3583.

[36] Francisco Romero, Qian Li, Neeraja J. Yadwadkar, and Christos Kozyrakis. "INFaaS: Managed & Model-less Inference Serving". In: *CoRR* abs/1905.13348 (2019). arXiv: 1905.13348.

[37] Ashutosh Saxena, Ashesh Jain, et al. "RoboBrain: Large-Scale Knowledge Engine for Robots". In: *CoRR* abs/1412.0691 (2014). arXiv: 1412.0691.

[38] Andrea Schaerf, Yoav Shoham, and Moshe Tennenholtz. "Adaptive Load Balancing: A Study in Multi-Agent Learning". In: *CoRR* cs.AI/9505102 (1995).

[39] W. Shi, J. Cao, et al. "Edge Computing: Vision and Challenges". In: *IEEE Internet of Things Journal* 3.5 (2016), pp. 637–646. DOI: 10.1109/JIOT.2016.2579198.

[40] Benno Staub, Ajay Kumar Tanwani, et al. "Dex-Net MM: Deep Grasping for Surface Decluttering with a Low-Precision Mobile Manipulator". In: *IEEE International Conference on Automation Science and Engineering (CASE)*. 2019, pp. 1–7.

[41] Ioan A Şucan and Lydia E Kavraki. "Kinodynamic motion planning by interior-exterior cell exploration". In: *Algorithmic Foundation of Robotics VIII*. Springer, 2009, pp. 449–464.

[42] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. Cambridge, MA, USA: A Bradford Book, 2018. ISBN: 0262039249.

[43] Ajay Kumar Tanwani, Nitesh Mor, et al. "A Fog Robotics Approach to Deep Robot Learning: Application to Object Recognition and Grasp Planning in Surface Decluttering". In: *IEEE Intl Conf. on Robotics and Automation (ICRA)*. 2019, pp. 4559–4566. DOI: 10.1109/ICRA.2019.8793690.

[44] Ajay Tanwani, Raghav Anand, Joseph Gonzalez, and Ken Goldberg. "RILaaS: Robot Inference and Learning as a Service". In: *IEEE Robotics and Automation Letters* (2020).

[45] Moritz Tenorth and Michael Beetz. "KnowRob – A Knowledge Processing Infrastructure for Cognition-enabled Robots". In: *Int. Journal of Robotics Research* 32.5 (Apr. 2013), pp. 566–590.

[46] N. Tian, M. Matl, et al. "A cloud robot system using the dexterity network and berkeley robotics and automation as a service (Brass)". In: *IEEE International Conference on Robotics and Automation (ICRA)*. 2017, pp. 1615–1622. DOI: 10.1109/ICRA.2017.7989192.

[47] Nan Tian, Ajay Kumar Tanwani, et al. "A Fog Robotic System for Dynamic Visual Servoing". In: *CoRR* abs/1809.06716 (2018).

[48] Markus Waibel, Michael Beetz, et al. "RoboEarth". In: *IEEE Robotics & Automation Magazine* 18.2 (2011), pp. 69–82. DOI: 10.1109/MRA.2011.941632.

[49] Wei Wang, Sheng Wang, et al. "Rafiki: Machine Learning as an Analytics Service System". In: *CoRR* abs/1804.06087 (2018). arXiv: 1804.06087.

[50] Xiaolong Xu, Shucun Fu, et al. "Dynamic Resource Allocation for Load Balancing in Fog Environment". In: *Wireless Communications and Mobile Computing* (2018), pp. 1–15.