Cloud-based Motion Plan Computation for Power-Constrained Robots

Jeffrey Ichnowski, Jan Prins, and Ron Alterovitz

University of North Carolina at Chapel Hill, USA
{jeffi,prins,ron}@cs.unc.edu

Abstract. We introduce a method for splitting the computation of a robot's motion plan between the robot's low-power embedded computer, and a high-performance cloud-based compute service. To meet the requirements of an interactive and dynamic scenario, robot motion planning may need more computing power than is available on robots designed for reduced weight and power consumption (e.g., battery powered mobile robots). In our method, the robot communicates its configuration, its goals, and the obstacles to the cloud-based service. The cloud-based service takes into account the latency and bandwidth of the connection between it and the robot and computes and returns a motion plan within the time frame necessary for the robot to meet requirements of a dynamic and interactive scenario. The cloud-based service parallelizes construction of a roadmap, and returns a sparse subset of the roadmap giving the robot the ability to adapt to changes between updates from the server. In our results, we show that with typical latency and bandwidth limitations, our method gains significant improvement in the responsiveness and quality of motion plans in interactive scenarios.

1 Introduction

Cloud-based computing offers a vast amount of low-cost computation power on-demand. It offers the ability to quickly scale up and down compute resources so that you can have more computing when you need it, and not pay for it when you do not. To place in context the price of cloud computation power, the July 2016 prices for 1 second of 360-cores of computation can be less than \$0.0047 [1]. This implies that with an embarrassingly parallel algorithm [2], a 5-minute computation can be cut to less than 1 second. And because you pay for the resources that you use, the same computation would require \$0.0047 whether using 1-core for 360 seconds, or 360-cores for 1 second. To access these immense computing resources, the only thing that is required is a connection to the internet.

Mobile robots are often designed and built to keep weight and power consumption as low as possible to achieve an acceptable duration of autonomy before requiring recharging. This design concern naturally dictates that the computation power on such a robot is limited—for example, to a low-power single-core processor. Motion planning is a computationally intensive process [3], and as

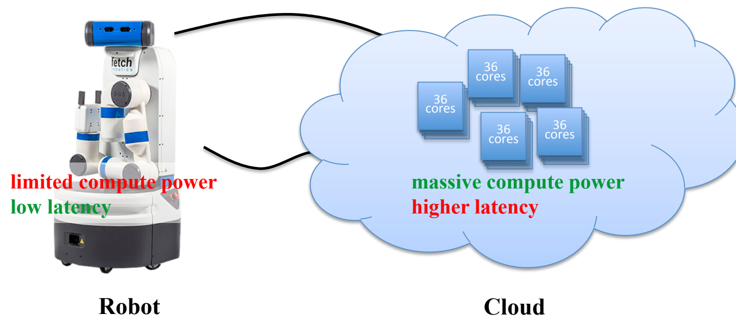


Fig. 1. Comparison of robot only and cloud computing for robot motion planning. The robot has limited computing power in order to reduce weight and increase battery life, however it has low latency access to its sensors and actuators. The cloud-computing has vast amounts of on-demand computing power available, but has a higher latency access to the robot and the information it sends.

such, if the mobile robot has more than a few degrees of freedom, its computational demands for motion planning can quickly exceed its available computational power.

In a static environment, the robot can compute its motion plan a priori and execute it. If the robot has no demands on when it needs to compute the motion plan, it can sit motionless while it computes the motion plan locally. On the other hand, if it needs a motion plan quickly, it can use cloud computing resources to greatly decrease the time to compute a motion plan, and start executing sooner.

In a dynamic environment, however, the robot must not only compute a complete motion plan, but it must also sense changes in the task's goal and the robot's environment and update its motion plan accordingly. As in a static environment, the robot can use a cloud-based computation to rapidly produce an initial motion plan. However, the network complicates matters when it comes to updating the plan due to changes in the environment since the network has limited bandwidth and introduces a network latency-based delay. The delay due to network latency and bandwidth may introduce enough of a lag that the mobile robot relying solely on cloud-based motion planning would not be able to respond to changes in its environment quickly enough to avoid a collision.

In this paper we propose a method for a mobile robot to compute and execute a motion plan by offloading much of the computational cost of motion planning to the cloud, while remaining reactive enough to respond to a dynamic environment and avoid obstacles.

2 Related Work

The NIST definition of cloud computing [4], provides a good high-level overview of the capabilities of the cloud. Broadly, cloud computing encompasses a "ubiquitous, convenient, on-demand network access to a shared pool of computing

resources that can be rapidly provisioned and released...". *Cloud-robotics and automation* are a subset of cloud-based computing related to robotics—it encompasses a broad range of topics, including access to big-data libraries, high-performance computing, collective robot learning, and remote human interaction. Kehoe, et al. provides an excellent survey of cloud-robotics in [5].

In this paper we focus on cloud-computing as an on-demand high-performance computing platform to accelerate motion planning. Bekris et al. [6] use the cloud to precompute manipulation roadmaps. The robot uses the roadmap to compute the shortest collision-free path, lazily determining if edges on the roadmap are blocked as determined by the latest sensor data. They observe that a dense precomputed roadmap, while covering more space and capable of producing shorter paths between configurations, has the negative effect of increasing bandwidth requirements to transfer the roadmap and taking more time to perform a search. They thus use techniques such as SPARS and IRS (described below) to reduce the roadmap size and evaluate the tradeoffs. Our approach follows from that observations, but instead computes and updates the roadmap at an interactive rate.

In [7], Kehoe et al. use a cloud-based data service to facilitate recognition of objects for grasping. The approach uses a custom Google image recognition service that is trained to recognize objects and estimate grasp points. In a subsequent related paper [8], Kehoe et al. use cloud-based computation to massively accelerate through parallel computation, a Monte Carlo sampling-based grasp analysis and planning. The paper demonstrates the cloud's ability to scale to 500 compute nodes and achieve a $445\times$ speedup.

Parallel processing has been successfully used to accelerate motion planning computations. In [2], Amato et al. demonstrate that probabilistic roadmap generation is *embarrassingly parallel*—meaning that little effort is needed to separate the sample generation into multiple parallel processes. The method described in [9] uses lock-free synchronization to parallelize multi-core shared-memory sampling-based motion planning algorithms with minimal overhead and observe linear and super-linear speedup. Carpin et al. describes an OR-parallel RRT method [10] that allows for distributed generation of sampling-based motion plans among independent servers—the algorithm chooses the best plan generated from the servers participating, and the result is a probabilistically better plan. Otte et al.'s C-FOREST [11] algorithm improves upon OR-parallel RRT by exchanging information between computers about the best path found, resulting in speedup in the motion planning on all parallel threads.

Robots are increasingly integrated into networks of computers. With the advent of ROS [12] and similar systems, network connected robots are becoming the norm. ROS's network stack is designed for a high-bandwidth, low-latency, local private/protected network to facilitate unified access to the robot's sensor, actuators, and embedded systems. Cloud-based computing, on the other hand, has lower bandwidth, higher latency, and is generally publicly accessible (except, for example, when using a VPN), and thus requires additional consideration above the network stack.

The probabilistic roadmap method (PRM) [13] generates a connected graph of robotic configurations in a precomputing offline phase. The robot later uses the roadmap to find a path from an initial configuration to a goal configuration by following along the edges of the graph. The k-PRM* [14] method improves upon PRM by defining a connectivity level (k) needed to guarantee asymptotic optimality.

Sparse roadmaps and roadmap spanners such as SPARS [15] are an effective technique in reducing the complexity of motion planning roadmaps. They can produce asymptotically near-optimal roadmaps, which maintain reachability of the non-sparse graph, while limiting the size of the graph to thresholds needed for lower-end computing platforms. In our method we adopt and parallelize the incremental roadmap spanner (IRS) of [16] to reduce the roadmap size for transmission over the internet.

Once the robot has a roadmap, whether sparse or not, it needs a path finding algorithm to navigate its structure. Shortest-path finding algorithms such as Dijkstra's algorithm and A* search find optimal paths, but can suffer from a slow compute time that makes them inappropriate for reactive path planning. D* and D* Lite algorithms perform a search from goal to start and track information in the graph that allows them to be incrementally updated when changes to the roadmap (e.g., from moving obstacles) occur—this provides a performance benefit in that only a partial graph search is needed anytime there is a change in the roadmap. The Anytime Repairing A* [17] and Anytime D* Lite [18] algorithms use an inadmissible heuristic in A* to find a path quickly, then incrementally improve the plan in subsequent iterations.

3 Problem Definition

Let $\mathcal{C}$ be the *configuration space* for the robot—the k-dimensional space of all possible configurations the robot take. Let $\mathcal{C}_{\text{free}} \subseteq \mathcal{C}$ be the subset of configurations that are collision free. Let $\mathbf{q} \in \mathcal{C}$ be the k-dimensional complete specification of a single robotic configuration (e.g., the joint angles of an articulated robot). Let $\mathbf{Q}_{\text{goal}} \subseteq \mathcal{C}_{\text{free}}$ be the set of goal configurations. Given a starting configuration $\mathbf{q}_0$, the objective of motion planning in a static environment is to compute a path $\boldsymbol{\tau} = (\mathbf{q}_0, \mathbf{q}_1, \ldots, \mathbf{q}_n)$, such that the path between $\mathbf{q}_i$ and $\mathbf{q}_{i+1}$ is in $\mathcal{C}_{\text{free}}$ as traversed by a local planner, and $\mathbf{q}_n \in \mathbf{Q}_{\text{goal}}$

When the robot operates in a dynamic environment, $\mathcal{C}_{\text{free}}$ changes over time. Let $\mathcal{C}_{\text{free}}(t) \subseteq \mathcal{C}$ be the obstacle-free configuration space at time t, and let $\mathbf{Q}_{\text{goal}}(t) \subseteq \mathcal{C}_{\text{free}}(t)$ be the goal at time t. Given the robot starting configuration $\mathbf{q}_0$ at time t_0, the objective of motion planning in a dynamic environment is to compute a path $\boldsymbol{\tau} = \left(\left[\mathbf{q}_0^\intercal, t_0\right]^\intercal, \left[\mathbf{q}_1^\intercal, t_1\right]^\intercal, \ldots, \left[\mathbf{q}_n^\intercal, t_n\right]^\intercal\right)$, such that the path between $\mathbf{q}_i$ and $\mathbf{q}_{i+1}$ is in $\mathcal{C}_{\text{free}}(\cdot)$ as traversed by a local planner from time t_i to t_{i+1}, and $\mathbf{q}_n \in \mathbf{Q}_{\text{goal}}(t_n)$.

In a dynamic environment $\mathcal{C}_{\text{free}}(t)$ may only be known at time t, and within the sensing capabilities of the robot. We consider obstacles in the environment that fall into the following categories: (1) *known static* obstacles that do not

change over the course of the task (e.g., a wall), (2) *unknown static* obstacles that are static, but are not known until sensed by the robot, and (3) *dynamic* obstacles that are moving through the environment and whose motion is unknown in advance.

The robot, being in its environment, has fast access to the input from its sensors, and is able to incorporate them into its planning to avoid moving obstacles. The cloud computing service does not have sensors relevant to the robot's scenario and thus only has access to the sensed environment via what the robot communicates to it.

Motion planning computation is split between two computing resources: (1) the robot's embedded *local computer*, and (2) the remote *cloud computer(s)*. Without loss of generality, we assume the robot's computer is a low-power single-core processor with some percentage of compute time dedicated to motion computations. The cloud-computing servers are fast multi-core computers.

The two computing resources communicate via a network with quantifiable bandwidth and latency. Bandwidth (R) is measured in bits per second, and is much lower than the bandwidth achievable between CPU and RAM. Latency (t_L) is measured as the time between when a bit is sent and when it is received. The bandwidth is low enough that sending a complete roadmap from client to server would hamper the robot's ability to adapt quickly to changing environment. The latency is high enough that the planning process must compensate for it in it requests updates to the motion plan.

4 Method

We introduce a new set of algorithms to effectively split motion plan computation between a robot and a cloud-based compute service based upon the strengths of each system. The robot is in the environment and has fast access to sensors, but it has a low-power processor—it is thus responsible for sensing the environment (i.e., detecting obstacles and estimating current state), reacting to dynamic obstacles, and executing collision-free motions. The cloud-based compute service is connected to the robot by a possibly high-latency low-bandwidth network, but has fast on-demand computing power—it is thus responsible for rapidly computing and sending to the robot a motion planning roadmap that encodes feasible collision-free motions.

When the robot starts a new task, it initiates a cloud-planning session by sending a request with the task and environment description to the cloud-based computing service. The cloud computer receives the request, starts a new cloud-based motion planning session, and computes a motion plan. Once the motion plan is of sufficient quality (as determined by the task), the cloud-based service sends the motion plan to the robot so that the robot can begin execution of the task.

The cloud-based service operates as a request-response service; each request the client makes results in a single response from the service. In the algorithms presented, the request-response communication is *asynchronous* unless otherwise

Algorithm 1 Robot Computation

Input: the initial configuration $\mathbf{q}_{\text{robot}}$, goal region $\mathbf{Q}_{\text{goal}}$, known static obstacles $\mathcal{W}$
1: $\mathbf{G} = (\mathbf{V}, \mathbf{E}) \leftarrow (\varnothing, \varnothing)$ {roadmap is initially empty}
2: $\boldsymbol{\tau} \leftarrow \varnothing$ {path is initially empty}
3: **send** `plan_req`$(t_0, \mathbf{q}_{\text{robot}}, \mathcal{W}, \mathbf{Q}_{\text{goal}}) \Rightarrow$ cloud
4: **while** $\mathbf{q}_{\text{robot}} \notin \mathbf{Q}_{\text{goal}}$ **do**
5: $(\mathcal{W}, \mathcal{D}) \leftarrow$ (sensed static obstacles, tracked dynamic obstacles) {sense}
6: **if recv** `roadmap_update` $\Leftarrow$ cloud **then**
7: Incorporate update into robot's roadmap $\mathbf{G}$
8: $t_{\text{req}} \leftarrow$ (current time) $+ t_{\text{step}}$
9: $\mathbf{q}_{\text{req}} \leftarrow$ compute where robot will be at t_{req}
10: **send** `plan_req`$(t_{\text{req}}, \mathbf{q}_{\text{req}}, \mathcal{W}, \mathbf{Q}_{\text{goal}}) \Rightarrow$ cloud
11: **if** changes in ($\mathbf{G}$, $\mathcal{W}$, $\mathcal{D}$) **or** (Anytime D*'s ϵ) > 1 **then**
12: $\boldsymbol{\tau} \leftarrow$ compute/improve path using Anytime D*
13: $\mathbf{q}_{\text{robot}} \leftarrow$ follow edges of shortest path $\boldsymbol{\tau}$ {move}

stated. Within a planning session, the service retains state from one request-response cycle to the next so that it does not start from scratch at each point in the process.

4.1 Roadmap-Based Robot Computation

The robot's algorithm is shown in Alg. 1. It initializes the process and starts the cloud-planning session in lines 1 to 3. As part of initialization it creates an empty graph for the roadmap and sends an initial planning request. It then starts a sense-plan-move loop (line 4) in which it will remain until it reaches a goal.

The sensing process at the start of each loop iteration is responsible for processing sensor input to construct a model of the static obstacles in the environment ($\mathcal{W}$), and to track the movement of dynamic obstacles ($\mathcal{D}$). Since the static environment changes infrequently (e.g., as the robot rounds a corner to discover construction blocking its path), an implementation can save bandwidth by only sending changes to the static environment as it discovers them.

In the planning part of the loop, the robot incorporates new data from the cloud service, computes a local path around dynamic obstacles, and requests plan updates as it needs them. The robot internally represents its estimate of $\mathcal{C}_{\text{free}}$ using a roadmap encoded as a graph $\mathbf{G} = (\mathbf{V}, \mathbf{E})$, where $\mathbf{V}$ are configurations (the vertices) of the graph, and $\mathbf{E}$ are the collision-free motions (edges) between configurations in $\mathbf{V}$. On line 6, the robot checks if the cloud service has responded to the robot's most recent request with an update to the roadmap. When the robot receives the cloud's roadmap, it incorporates the new data into the robot's roadmap, and initiates a new cloud planning request with the latest information from the environment.

Alg. 1 requests updates as frequently as possible, however if excessive network utilization shortens battery life in an implementation, requests can be made less frequently, for example, only when the robot has moved sufficiently out of its available roadmap. To send a request, the robot computes where it will be at time

t_{step} in the future following its current plan. The value of t_{step} is a parameter of the system, and accounts for the network round-trip and cloud processing time to compute the update.

If the robot has encountered a change to the graph, or any of the static or dynamic obstacles, or its current path (τ) can be refined further, it computes or improves the path using an Anytime D* planner [18], with a time component as described in [19]. Anytime D* defines and uses a runtime value in ϵ (line 11) to incrementally refine the robot's path. It starts by setting the value of $\epsilon > 1$ which it uses to modify the A* heuristic to find a sub-optimal solution quickly. As the algorithm iterates, it decreases ϵ and correspondingly refines the path with the new heuristic, resulting in an improved plan. When $\epsilon = 1$, the solution is optimal. As the last part of the loop, Alg. 1 moves the robot along the shortest path it computed.

When the robot computes its local path it saves computation time by only considering collisions between paths on the roadmap and the dynamic obstacles. The robot does not need to recompute self-collision avoidance, collisions with static obstacles, or other motion constraints, as this information is incorporated into the roadmap that the cloud service computes.

4.2 Roadmap-Based Cloud Computation

Cloud-based computation in our algorithm computes a roadmap for a robot to use when navigating through an environment and around obstacles. Because this algorithm runs on the cloud-based compute service, it has access to immense computational resources, enabling computation of a large, detailed roadmap. When building a roadmap, the cloud-based computation only considers the obstacles in the environment that are sent to the cloud from the robot—since the robot only sends static obstacles, the roadmap does not include avoidance of dynamic obstacles.

The robot starts a cloud planning *session* with an initial request for a roadmap. A session corresponds to a single robotic task and cloud-computing process that spans multiple requests from the robot. At the start, both the cloud and the robot have an empty graph as a roadmap. The cloud computes an initial roadmap and sends the relevant portion of the roadmap to the robot to begin execution of the task. As the robot needs additional areas of the roadmap, it sends additional requests to the server, and the server responds with updates to the roadmap. Optionally, in parallel, cloud process optimizes and extends the roadmap between request/response cycles.

Alg. 2 shows the cloud computing process for a single cloud-based motion planning session. The session starts with an empty graph $\mathbf{G} = (\mathbf{V}, \mathbf{E}) = (\varnothing, \varnothing)$. The algorithm builds the graph (Sec. 4.3) by generating vertices ($\mathbf{V}$) and *dense* edges ($\mathbf{E}$); and selects and maintains a *sparse* subset of edges $\mathbf{E}_{\text{s}} \in \mathbf{E}$. The sparse edges retain graph connectivity and are used to reduce the transfer size, while the dense edges give the robot more options to react to dynamic obstacles. Alg. 2 also maintains a subgraph $\mathbf{G}_{\text{robot}} = (\mathbf{V}_{\text{robot}} \subseteq \mathbf{V}, \mathbf{E}_{\text{robot}} \subseteq \mathbf{E})$ that tracks the portion of the $\mathbf{G}$ sent to the robot.

Algorithm 2 Cloud Planning Session

```
1: G = (V, E_s ⊆ E) ← (∅, ∅)
2: G_robot = (V_robot, E_robot) ← (∅, ∅)
3: W ← ∅ {static obstacles}
4: loop
5:    recv plan_req(t_req, q_req, W, Q_goal) ⇐ robot {blocking wait for next request}
6:    V ← {q ∈ V ∪ {q_req} | ∀w ∈ W : clear(q | w)}
7:    E ← {(q_a, q_b) ∈ E | ∀w ∈ W : link(q_a, q_b | w)}
8:    while t_now < t_req − t_res and not satisfactory solution do
9:       update G and q_goal using k-PRM*+IRS on W and Q_goal
10:   (V'_robot, E'_robot) ← serialize_graph(G, q_req, Q_goal, G_robot)
11:   send plan_res(V'_robot \ V_robot, E'_robot \ E_robot) ⇒ robot
12:   (V_robot, E_robot) ← (V'_robot, E'_robot)
```

The cloud planning session starts when it receives a plan_req (plan request) from the robot (line 5). This request corresponds to the plan_req sent by the robot in Alg. 1 line 3. The cloud computer adds the requested configuration $\mathbf{q}_{\text{req}}$ to the graph and updates the existing graph for any new static obstacles that are added $\mathcal{W}$ (lines 6 and 7). This step makes use of two application-specific functions to produce a valid roadmap: clear($\mathbf{q}$) computes whether or not $\mathbf{q} \in \mathcal{C}_{\text{free}}$ (e.g., via collision detection algorithms); and link($\mathbf{q}_a, \mathbf{q}_b$) checks if the path between $\mathbf{q}_a$ and $\mathbf{q}_b$ is in $\mathcal{C}_{\text{free}}$ as traversed by the robot's local planner. It then builds the roadmap until it runs out of time or it has a solution of satisfactory quality (lines 8 and 9). The compute time limit is the target completion time t_{req} minus the amount of time for the robot to receive the response t_{res}. Thus t_{res} is computed as the sum of graph serialization time and total network transfer time. The graph is then serialized using the method described in section 4.4, and the new vertices and edges selected for serialization are sent back to the robot as a plan_res (plan response) in line 11. Optionally, at the end of the loop the cloud computer may continue to update the roadmap in the background until it receives another plan_req from the robot.

4.3 Lock-free Parallel *k*-PRM* with a Roadmap Spanner

The cloud-based service computes a roadmap using k-PRM* [14] with the Incremental Roadmap Spanner (IRS) [16], accelerated by a lock-free parallelization construction we introduce in this section. k-PRM* is an asymptotically optimal sampling-based method that generates a roadmap. IRS selects an asymptotically near-optimal sparse subset of the edges generated by k-PRM* and results in a graph with significantly fewer edges as compared to k-PRM*. The edges from k-PRM* are the *dense* graph edges ($\mathbf{E}$). The edges selected by IRS are the *sparse* graph edges ($\mathbf{E}_\text{s} \subseteq \mathbf{E}$).

The server computes k-PRM*+IRS using a parallel lock-free algorithm in which all provisioned cores run Alg. 3 simultaneously to generate and add random samples to a graph in shared memory. The main portion of the algorithm proceeds similarly to the non-parallel version, with the key differences being that:

Algorithm 3 Lock-free Parallel k-PRM* IRS Thread

Input: $G = (V, E)$ is an initialized graph shared between threads, $\exists v \in V$: `is_goal`(v)
1: **while not** done **do**
2: $v_{\text{rand}} \leftarrow$ new vertex with random sample and connected component C_{rand}
3: C_{rand}.goal $\leftarrow$ `is_goal`(v_{rand})
4: **if** `clear`(v_{rand}) **then**
5: **for all** $v_{\text{near}} \in$ `k_nearest`$(V, v_{\text{rand}}, \{k =\}\lceil \log(|V|+1) * k_{\text{RRG}} \rceil)$ **do**
6: **if** `link`$(v_{\text{rand}}, v_{\text{near}})$ **then**
7: sparse $\leftarrow$ `shortest_path_dist`$(v_{\text{rand}}, v_{\text{near}}) < w_{\text{stretch}} *$ `dist`$(v_{\text{rand}}, v_{\text{near}})$
8: `add_edge`$(v_{\text{rand}}, v_{\text{near}}, \text{sparse})$
9: `add_edge`$(v_{\text{near}}, v_{\text{rand}}, \text{sparse})$
10: solved $\leftarrow$ solved **or** merge_components$(v_{\text{rand}}.\text{cc}, v_{\text{near}}.\text{cc})$
11: $V \leftarrow V \cup v_{\text{rand}}$

(1) nearest neighbor searching is fast and non-blocking due to the use the lock-free kd-tree described in [9], (2) graph edges are stored in lock-free linked lists (Alg. 4), and (3) progress towards a solution is tracked via connected components that are stored in lock-free linked trees (Alg. 5). As with k-PRM*, in each iteration this algorithm generates a random robot configuration and searches for its k-nearest neighbors using k from [14]. The algorithm checks if the path to each neighbor is obstacle-free (line 6), and if so, adds edges to the PRM graph (lines 8 and 9). As the algorithm builds the graph, it adds *dense* edges consistent with k-PRM*. When the shortest path distance between two vertices in the graph is shorter than a stretch weighted (w_{stretch}) straight-line distance, it adds *sparse* edges consistent with IRS.

Algorithm 4 `add_edge`$(v_{\text{from}}, v_{\text{to}}, \text{sparse})$

1: $e_{\text{dense}} \leftarrow$ new edge to v_{to} with e_{dense}.next $= v_{\text{from}}$.dense_list_head
2: **while not** `CAS`$(v_{\text{from}}.\text{dense_list_head}, e_{\text{dense}}.\text{next}, e_{\text{dense}})$ **do**
3: e_{dense}.next $\leftarrow v_{\text{from}}$.dense_list_head
4: **if** sparse **then**
5: add edge to v_{to} to sparse list of edges with CAS loop similar to one for dense list
6: **while** v_{from}.cc.parent $\neq$ `nil` **do**
7: v_{from}.cc $\leftarrow v_{\text{from}}$.cc.parent {Lazy update of vertex's connected component}

The algorithm adds edges to the graph using Alg. 4. Each vertex in the graph has a reference to the head of two linked lists: one for $\mathbf{E}$, and one for $\mathbf{E}_{\text{s}}$. Updating the list makes use of a"compare-and-swap" (CAS) operation available on modern multi-core CPU architectures. `CAS`(mem, old, new), in one atomic action, compares the value in *mem* to an expected *old* value, and if they match, updates *mem* to the *new* value. CAS, combined with the loop in line 2, updates the lists correctly even in the presence of competing concurrent updates.

The algorithm tracks progress towards a solution by maintaining information on each connected component ("cc" in Alg. 4) in the roadmap. When it adds an

Algorithm 5 `merge_components`(C_a, C_b)

```
1: repeat
2:     while C_a.parent ≠ nil do C_a ← C_a.parent
3:     while C_b.parent ≠ nil do C_b ← C_b.parent
4: until CAS(C_a.parent, nil, C_b)
5: repeat
6:     while C_b.parent ≠ nil do C_b ← C_b.parent
7:     C_merged ← new component
8:     (C_merged.start, C_merged.goal) ← (C_a.start or C_b.start, C_a.goal or C_b.goal)
9: until CAS(C_b.parent, nil, C_merged)
10: return C_merged.start and C_merged.goal
```

Algorithm 6 `serialize_graph`$(\mathbf{G}, \mathbf{q}_{\text{req}}, \mathbf{Q}_{\text{goal}}, \mathbf{G}_{\text{robot}})$

Input: $\mathbf{G} = (\mathbf{V}, \mathbf{E}_{\text{s}} \subseteq \mathbf{E})$, $\mathbf{G}_{\text{robot}} = (\mathbf{V}_{\text{robot}}, \mathbf{E}_{\text{robot}})$, s.t. $\mathbf{V}_{\text{robot}} \subseteq \mathbf{V}$, $\mathbf{E}_{\text{robot}} \subseteq \mathbf{E}$

```
1: (V'_robot, E'_robot) ← (V_robot, E_robot)
2: V_frontier = forward_frontier(q_req, E)
3: p(·) ← path_to_frontier(Q_goal, V_frontier, E)
4: V'_robot ← V'_robot ∪ V_frontier
5: Q ← {q ∈ V_frontier} {populate FIFO queue}
6: while |Q| > 0 do
7:     q_i ← remove head from Q
8:     for all (q_i, q_s) ∈ E_s : q_s ∉ V'_robot do
9:         (V'_robot, E'_robot) ← (V'_robot ∪ {q_s}, E'_robot ∪ {(q_i, q_s)})
10:        append q_s to Q
11:    if p(q_i) ≠ nil and (q_i, p(q_i)) ∉ E'_robot then
12:        if p(q_i) ∉ V_robot then
13:            append p(q) to Q
14:            V'_robot ← V'_robot ∪ p(q_i)
15:        E'_robot ← E'_robot ∪ (q_i, p(q_i))
16: return (V'_robot, E'_robot)
```

edge between two vertices, it also merges the connected components associated with the vertices (Alg. 5). This is done by maintaining a "parent" link from the pre-merged component to the post-merged component. The most recently merged component is thus found by repeatedly following parent links to the root of the connected components. Each connected component also maintains booleans tracking whether or not the component contains a vertex at the goal and/or start. Once a connected component is found that includes both a start and goal vertex, the graph contains a path between the two.

4.4 Roadmap Subset for Serialization

The roadmap serialization process selects a compact, relevant subset of a roadmap and converts it into a serial (linear) structure suitable for transmission over a network. Alg. 6 selects which vertices and edges of the graph to serialize. The process of converting the selected vertices and edges to sequence of bytes is left an implementation detail. Since bandwidth is limited, the process selects a

Algorithm 7 `path_to_frontier`$(\mathbf{q}_{\text{goal}}, \mathbf{V}_{\text{frontier}}, \mathbf{E})$

1: $g(\mathbf{q}_{\text{goal}}) \leftarrow 0$ {cost to goal}
2: $p(\mathbf{q}_{\text{goal}}) \leftarrow$ `nil` {forward pointers}
3: $\mathbf{U} \leftarrow \{\mathbf{q}_{\text{goal}}\}$ {priority queued ordered by $g(\cdot)$}
4: **while** $|\mathbf{V}_{\text{frontier}}| > 0$ **do**
5: $\quad\mathbf{q}_{\text{min}} \leftarrow$ remove $(\min \mathbf{U})$ from $\mathbf{U}$
6: $\quad\mathbf{V}_{\text{frontier}} \leftarrow \mathbf{V}_{\text{frontier}} \setminus \{\mathbf{q}_{\text{min}}\}$
7: $\quad$**for all** $(\mathbf{q}_{\text{from}}, \mathbf{q}_{\text{min}}) \in \mathbf{E}$ **do**
8: $\quad\quad d \leftarrow g(\mathbf{q}_{\text{min}}) + \texttt{cost}(\mathbf{q}_{\text{from}}, \mathbf{q}_{\text{min}})$
9: $\quad\quad$**if** $\mathbf{q}_{\text{from}} \notin \mathbf{U}$ **or** $d < g(\mathbf{q}_{\text{from}})$ **then**
10: $\quad\quad\quad g(\mathbf{q}_{\text{from}}) \leftarrow d$
11: $\quad\quad\quad$insert/update $\mathbf{q}_{\text{from}}$ in $\mathbf{U}$
12: $\quad\quad\quad p(\mathbf{q}_{\text{from}}) \leftarrow \mathbf{q}_{\text{min}}$
13: **return** $p(\cdot)$

small subset of the configurations in the roadmap to send to the robot. To allow the robot to navigate around dynamic obstacles in its immediate vicinity, as well as find the best route to goal, the cloud selects a subset of configurations that includes ones reachable from $\mathbf{q}_{\text{req}}$ within a time bound t_{max}, as well as the path to goal for each such vertex.

Serialization selection begins by finding the frontier between the vertices reachable from $\mathbf{q}_{\text{req}}$ within the time bound t_{max}, and vertices not reachable (line 2). The forward_frontier algorithm is a modified Dijkstra's algorithm that terminates once it finds paths longer than t_{max}. Since Dijkstra's expands paths in increasing path length, this will terminate once it has found all paths reachable within t_{max}. It returns all vertices $\mathbf{V}_{\text{frontier}}$ reachable within the frontier. The selection process then computes the shortest path from all goals to the vertices in $\mathbf{V}_{\text{frontier}}$ (line 3). This process, shown in Alg. 7, is a modified Dijkstra's algorithm that terminates once it has found a path to all vertices in $\mathbf{V}_{\text{frontier}}$.

In the last step in Alg. 6, the vertices from the frontier set are appended to $\mathbf{V}'_{\text{robot}}$ along with all configurations along their shortest paths to goal and reachable by the sparse edges. Line 5 populates the queue from $\mathbf{V}_{\text{frontier}}$. The loop starting on line 6 iterates through each configuration in the queue, adding sparse neighbors and steps along the shortest path to goal as it encounters them. By checking the graph before appending to the queue, the algorithm ensures that vertices are queued at most once. When the loop completes, the new graph subset is ready for sending to the robot. Then the cloud service sends only the changes in the graph from one response to the next (Alg. 6 line 11).

5 Results

We evaluate our algorithm on a Fetch robot [20] by giving it an 8 degree-of-freedom task in an environment with a dynamic moving obstacle. Our cloud-compute server runs on a system with four Intel x7550 2.0-GHz 8-core Nehalem-EX processors for a total 32-cores. The cloud-computing process makes use of all 32-cores. The cloud-compute server is physically located approximately 6 km

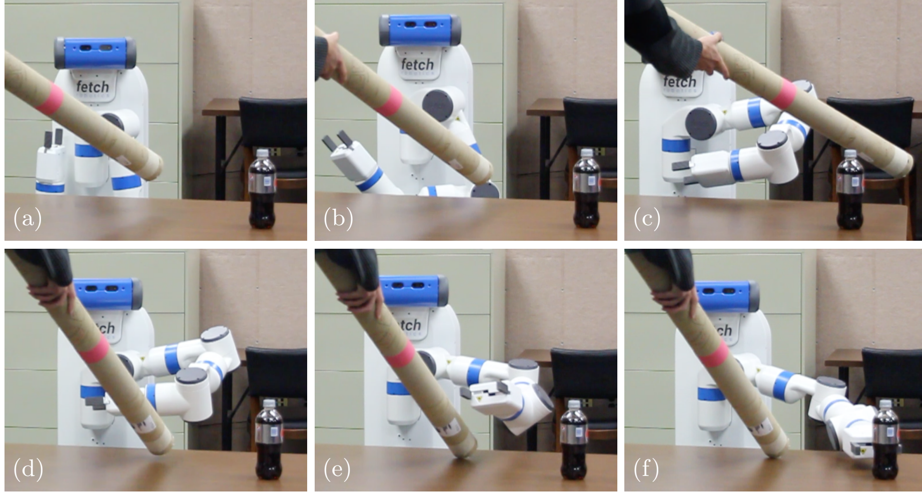
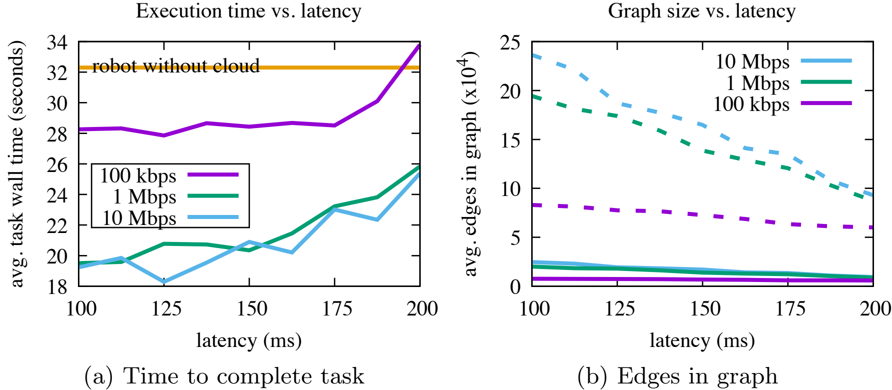


Fig. 2. The Fetch robot using our cloud-based motion planning for the task of grasping the bottle resting on the table while avoiding both the static obstacles (e.g., table) and the dynamic obstacle (a tube sensed via an RGBD camera). In frame (a) after the Fetch approaches the table with its arm in its standard rest configuration and it initiates the cloud-computation process. The Fetch's embedded CPU is tasked with sensing and avoiding dynamic obstacles, while a cloud-computer simultaneously generates and refines its roadmap. In frame (b), the Fetch begins its motion, only to be blocked in frame (c) by a new placement of the obstacle. The Fetch is again blocked in frame (d), moves again around the obstacle in frame (e), and reaches the goal in frame (f).

away from the robot, and the network connection between the server and robot supported a bandwidth in excess of 100 Mbps with a latency less than 20 ms. To model the impact of slower network connections, in our experiments we deliberately slowed packet transmission to model a fixed maximum bandwidth of R_{sim} and a fixed minimum round-trip latency of $t_{L_{\mathrm{sim}}}$ subject to noise sampled from a Gaussian distribution with standard deviation of $0.16\,t_{L_{\mathrm{sim}}}$.

We implemented our algorithm as a web-service accessible via HTTP [21]. The robot initiates a request by sending an HTTP POST to the server, and the server responds with an HTTP response code appropriate to the situation (e.g., "200 OK" for a successful plan, "503 Service Unavailable" when the server cannot acquire sufficient computing resources). Requests and responses are sent in a serialized binary form. To minimize overhead associated with establishing connections, both the cloud server and the robot use HTTP keep-alive to reuse TCP/IP connections between updates, and are configured to have a connection timeout that far exceeds expected plan computation time.

The Fetch robot has a 7 degree of freedom arm, a prismatic torso lift joint, and a mobile base. In our scenarios, prior to the cloud-based computation task, the Fetch robot navigates to the workspace using its mobile base without using the cloud service. This process introduces noise to the robot's base position and orientation. Once at the workspace, we give the Fetch robot the task of moving from a standard rest configuration (Fig. 2(a)) to a pre-grasp configuration over



(a) Time to complete task (b) Edges in graph

Fig. 3. Effect of different values for R_{sim} and $t_{L_{\text{sim}}}$. Graph (a) shows the wall-clock time for the Fetch robot to complete its pre-grasp motion task, where the orange line is the time for the robot to complete the task without the cloud service. Graph (b) compares the number of edges generated by the cloud computer (dashed lines) and the number of edges sent to the robot (solid lines) for the varying network conditions. The simulated network latency affects the amount of compute time that the cloud has for each update. Longer latencies lead to less time for available for computation, and thus leads to slower task completion time and fewer edges on the roadmap.

a table (Fig. 2(f)), requiring it to plan a motion using 8 degrees of freedom (i.e., the arm and prismatic torso lift joint). In this setting, the static obstacles are the table, floor, and surrounding office space. We also include a dynamic obstacle: a cylindrical tube that moves through the environment.

The sequence in Fig. 2 shows the full integrated system running, with the Fetch robot successfully moving its arm around the obstacles. At the beginning of a task, the Fetch communicates its position and orientation in the workspace to the cloud service and requests a roadmap for its task. The software uses custom tracking software and the Fetch's built in RGBD camera to determine the location of dynamic obstacles. When it computes a change in trajectory (e.g., to avoid a dynamic obstacle, or in response to a refined roadmap from the cloud), it sends the trajectory to the controller via a ROS/moveit interface.

We also ran our method in simulation to evaluate performance under varying networking conditions. We simulated the tube dynamic obstacle sweeping periodically over the table at a rate of 0.25 Hz (approximately 1 m/s). While the dynamic obstacle has a predictable motion consistent through all runs, the simulated sensors only sense the tube's position and orientation and do not predict its motion. As the tube obstacle is considered dynamic, the robot does not send information about it to the cloud computer, and it must avoid the tube by computing a path along the roadmap using its local graph. The robot and cloud are not given any pre-computation time; once given the task, the robot must begin and complete its motion as soon as it is able. We measure this as the "wall clock time to complete task."

The Fetch robot has a 2.9 GHz Intel i5-4570S processor with 4 cores. For our scenario, we limit our client-side planner to fully utilize a single core, under the assumption that in a typical scenario the remaining cores would leave sufficient compute power to run other necessary tasks, such as sensor processing.

As a baseline for comparison, we have the robot's computer generate a k-PRM* using a separate thread. This thread updates the graph used by the reactive planner at a period of 250 ms. The k-PRM* planner considers only the static environment and self-collision avoidance as the constraints on the roadmap generation, and generates a fully dense roadmap (no sparse edges). The reactive planner uses the roadmap to search for a path to the goal. While searching the roadmap, the robot lazily checks for collisions with the dynamic obstacle. In 50 runs, the robot completes the task with an average of 32.3 seconds.

We run the scenario using our method and simulate and vary the latency and bandwidth of the network between the robot and the 32-core cloud-computer. To maintain reactivity, the robot requests an update as soon as it receives the response to the previous request. Since the requested solve time (t_{req}) is set to 250 ms, an update is requested and received every 250 ms. The latency means that only a portion of the 250 ms can be used to compute a roadmap. The results in Fig. 3(a), averaging over 100 runs, show that the robot assisted by the cloud computation outperforms robot-only computation in almost all simulated cases. As we might expect, the slowest bandwidth and highest latency cause the performance benefit of using the cloud-based service to disappear. At the lowest latencies, the cloud-based solution outperforms the robot-only computation by 1.7×, reducing the task completion time to 19.0 seconds.

In Fig. 3(b), we show the savings that result from using the roadmap spanner and our serialization method. When latency is low, the cloud computer can spend more time computing, producing a roadmap that has on average 232649 edges. IRS and serialization reduce it to an average of 24236, a savings of close to 90%.

Fig. 4 shows the effect of roadmap serialization parameter $t_{\max}$ on our cloud-based motion planning. A smaller $t_{\max}$ implies less of the roadmap is sent to the robot, which results in reduced bandwidth usage but at a cost to the quality of the roadmap. As the robot executes its task, a proportionately higher portion of the server's dense roadmap is sent to the robot (see Fig. 4 (a)). From Fig. 4 (b), we see that if $t_{\max}$ is too small, the robot is slower to find a collision-free path past the dynamic obstacle. Conversely, there is little gain for increasing $t_{\max}$ beyond a certain threshold since unnecessary portions of the graph are sent to the robot, essentially wasting network bandwidth, leading to diminished performance.

6 Conclusion

Cloud computing offers access to vast amounts of computing power on demand. We introduce a method for power-constrained robots to accelerate their motion planning by splitting the motion planning computation between the robot and a high-performance cloud computing service. Our method rapidly computes an initial roadmap and then sends a mixed sparse/dense subgraph to the robot.

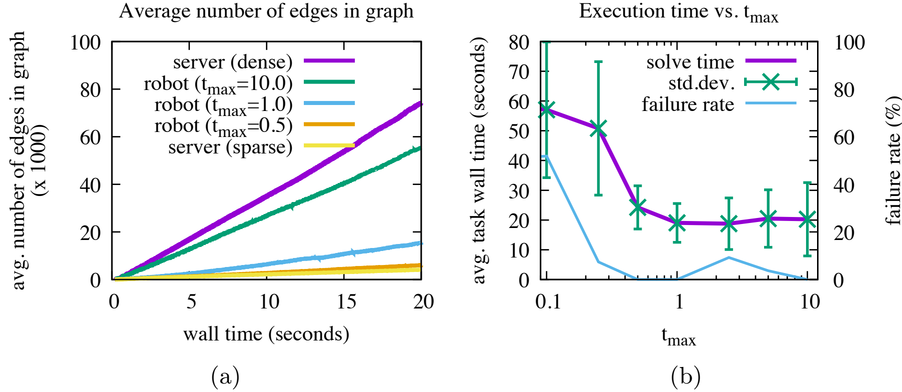


Fig. 4. The serialization parameter $t_{\max}$ affects the size of the graph on the robot and the robot's task completion wall time. In these graphs the simulated network is fixed at $R_{\text{sim}} = 1$ Mbps and $T_{L_{\text{sim}}} = 200$ ms and the server solve time is 250 ms. Graph (a) shows that larger values of t_{max} result in more of the dense edges of the graph being serialized and sent to the robot. In (b), we see that having t_{max} be too small results in a high failure rate (where failure means not reaching goal after 2 minutes), while having it too large increases the variance of the execution time.

The sparse portions of the graph retain connectivity and reduced transfer size, while the dense portions give the robot the ability to react to obstacles in its immediate vicinity. As the robot executes the plan, it periodically gets updates from the cloud to retain its reactive ability.

In our experiments, we applied our method to a Fetch robot, giving it an 8 degree of freedom task with a simulated dynamic obstacle. With our method, the split cloud/robot computation allows the robot to react to dynamic obstacles in the environment while attaining a more dense roadmap than possible with computation on the robot's embedded processor alone. The scenario requires a minimal amount of pre-computation time (less than a second) before the robot starts to execute its task. As a result, the task time-to-completion is significantly improved over the alternative without cloud computing.

In future work we will incorporate mobility into the planning for the Fetch robot to make use of its wheeled base. We will also implement the proposed framework using a commercial cloud-based service and investigate approaches that efficiently allocate cloud-computing resources, including for the short intervals of computation needed for single tasks. We will also evaluate this method on different scenarios and different robot types.

Acknowledgement

This research was supported in part by the U.S. National Science Foundation (NSF) under Award CCF-1533844 and Award IIS-1149965.

References

1. Amazon: EC2 instance pricing. https://aws.amazon.com/ec2/pricing/ Accessed: 2016-07.
2. Amato, N.M., Dale, L.K.: Probabilistic roadmap methods are embarrassingly parallel. In: Proc. IEEE Int. Conf. Robotics and Automation (ICRA). (May 1999) 688–694
3. Reif, J.H.: Complexity of the Mover's Problem and Generalizations. In: 20th Annual IEEE Symp. on Foundations of Computer Science. (1979) 421–427
4. Mell, P., Grance, T.: The NIST definition of cloud computing. http://dx.doi.org/10.6028/NIST.SP.800-145 (2011)
5. Kehoe, B., Patil, S., Abbeel, P., Goldberg, K.: A survey of research on cloud robotics and automation. IEEE Transactions on Automation Science and Engineering **12**(2) (2015) 398–409
6. Bekris, K., Shome, R., Krontiris, A., Dobson, A.: Cloud automation: Precomputing roadmaps for flexible manipulation. IEEE Robotics & Automation Magazine **22**(2) (2015) 41–50
7. Kehoe, B., Matsukawa, A., Candido, S., Kuffner, J., Goldberg, K.: Cloud-based robot grasping with the google object recognition engine. In: Robotics and Automation (ICRA), 2013 IEEE International Conference on, IEEE (2013) 4263–4270
8. Kehoe, B., Warrier, D., Patil, S., Goldberg, K.: Cloud-based grasp analysis and planning for toleranced parts using parallelized monte carlo sampling. IEEE Transactions on Automation Science and Engineering **12**(2) (2015) 455–470
9. Ichnowski, J., Alterovitz, R.: Scalable multicore motion planning using lock-free concurrency. IEEE Transactions on Robotics **30**(5) (2014) 1123–1136
10. Carpin, S., Pagello, E.: On parallel RRTs for multi-robot systems. In: Proc. 8th Conf. Italian Association for Artificial Intelligence. (2002) 834–841
11. Otte, M., Correll, N.: C-FOREST: Parallel shortest path planning with superlinear speedup. IEEE Trans. Robotics **29**(3) (2013) 798–806
12. ROS.org: Robot Operating System (ROS). http://ros.org (2012)
13. Kavraki, L.E., Svestka, P., Latombe, J.C., Overmars, M.: Probabilistic roadmaps for path planning in high dimensional configuration spaces. IEEE Trans. Robotics and Automation **12**(4) (1996) 566–580
14. Karaman, S., Frazzoli, E.: Sampling-based algorithms for optimal motion planning. Int. J. Robotics Research **30**(7) (June 2011) 846–894
15. Dobson, A., Bekris, K.E.: Sparse roadmap spanners for asymptotically near-optimal motion planning. The International Journal of Robotics Research **33**(1) (2014) 18–47
16. Marble, J.D., Bekris, K.E.: Asymptotically near-optimal planning with probabilistic roadmap spanners. IEEE Transactions on Robotics **29**(2) (2013) 432–444
17. Likhachev, M., Gordon, G.J., Thrun, S.: ARA*: Anytime A* with provable bounds on sub-optimality. In: Advances in Neural Information Processing Systems. (2003)
18. Likhachev, M., Ferguson, D.I., Gordon, G.J., Stentz, A., Thrun, S.: Anytime dynamic A*: An anytime, replanning algorithm. In: ICAPS. (2005) 262–271
19. Van Den Berg, J., Ferguson, D., Kuffner, J.: Anytime path planning and replanning in dynamic environments. In: Proc. IEEE Int. Conf. Robotics and Automation (ICRA). (2006) 2366–2371
20. Fetch Robotics: Fetch research robot. http://fetchrobotics.com/research/
21. Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., Berners-Lee, T.: Hypertext transfer protocol–http/1.1. Technical report (1999)