SCALING ROBOT MOTION PLANNING TO MULTI-CORE PROCESSORS AND THE CLOUD

Jeffrey Ichnowski

A dissertation submitted to the faculty at the University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science.

Chapel Hill
2020

Approved by:

Ron Alterovitz

Jan F. Prins

James Anderson

Jack Snoeyink

Kostas Bekris

ABSTRACT

Jeffrey Ichnowski: Scaling Robot Motion Planning to Multi-core Processors and the Cloud
(Under the direction of Ron Alterovitz)

Imagine a world in which robots safely interoperate with humans, gracefully and efficiently accomplishing everyday tasks. The robot's motions for these tasks, constrained by the design of the robot and task at hand, must avoid collisions with obstacles. Unfortunately, planning a constrained obstacle-free motion for a robot is computationally complex—often resulting in slow computation of inefficient motions. The methods in this dissertation speed up this motion plan computation with new algorithms and data structures that leverage readily available parallel processing, whether that processing power is on the robot or in the cloud, enabling robots to operate safer, more gracefully, and with improved efficiency.

The contributions of this dissertation that enable faster motion planning are novel parallel lock-free algorithms, fast and concurrent nearest neighbor searching data structures, cache-aware operation, and split robot-cloud computation. Parallel lock-free algorithms avoid contention over shared data structures, resulting in empirical speedup proportional to the number of CPU cores working on the problem. Fast nearest neighbor data structures speed up searching in SO(3) and SE(3) metric spaces, which are needed for rigid body motion planning. Concurrent nearest neighbor data structures improve searching performance on metric spaces common to robot motion planning problems, while providing asymptotic wait-free concurrent operation. Cache-aware operation avoids long memory access times, allowing the algorithm to exhibit superlinear speedup. Split robot-cloud computation enables robots with low-power CPUs to react to changing environments by having the robot compute reactive paths in real-time from a set of motion plan options generated in a computationally intensive cloud-based algorithm.

We demonstrate the scalability and effectiveness of our contributions in solving motion planning problems both in simulation and on physical robots of varying design and complexity. Problems

include finding a solution to a complex motion planning problem, pre-computing motion plans that converge towards the optimal, and reactive interaction with dynamic environments. Robots include 2D holonomic robots, 3D rigid-body robots, a self-driving 1/10 scale car, articulated robot arms with and without mobile bases, and a small humanoid robot.

To my spouse, Chrissy Kistler, whose support and encouragement made this possible. To my little dudes, Roy and Carl, who made this worthwhile. To my mom, Jeanne. To my sister, Amy and her little dude, Alexander (Sasha). To Peggy and John Seymore. To Charlie and Nancy Kistler. In loving memory of my father, Arthur Ichnowski, who saw me start this journey, and is with me in spirit at its end.

ACKNOWLEDGEMENTS

I would like to thank Ron Alterovitz for his invaluable guidance over the course of pursuing my PhD and for creating and supporting opportunities beyond just the research and writing. I would also like to thank my committee for their time, feedback, and the unique perspectives that they each provided that improved my dissertation: Jan Prins, for his guidance and discussions that help inspire and put the finishing touches on many aspects of the research; James Anderson, for discussions on the importance of *real* real-time computing in robots and for creating the opportunity to apply my research to a self-driving car; Jack Snoeyink, for his valuable discussions and advice on both research and writing; and Kostas Bekris, for his advice, discussions, and prior contributions that helped extend this work to the cloud.

The research in this dissertation was supported in part by U.S. National Science Foundation (NSF) under awards IIS-1117127, IIS-1149965 and CCF-1533844, and by the National Institutes of Health (NIH) under award R21EB017952.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

Introduction

Consider a robot that needs to plan its motions to autonomously complete a task. The planned motion needs to avoid obstacles, obey task-specific constraints, and reach a goal within a timely manner. Computing motion plans quickly can be computationally demanding; the general motion planning problem is PSPACE-hard [98], and the time required grows exponentially in the robot's configurable degree's of freedom. Thus, while rapid online motion planning around moving obstacles for robots with few degrees of freedom (e.g., a disc robot vacuuming a floor, or a self-driving car) may be tractable with modern motion planning algorithms, adding just a few degrees of freedom to the problem (e.g., an articulated robot arm) requires new tools and more computational processing power. Fortunately we are living in an era in which computational processing power is growing exponentially—but tapping into that power requires novel work in parallel and cache-aware algorithms and concurrent data structures. Moreover, due to physical limits and power constraints, the computing power required to plan motions may not be housed within the robot's physical body—and thus we also need novel approaches to utilizing computing power outside the robot's body. This dissertation presents and demonstrates the effectiveness of multi-core parallel computation, made scalable through concurrent data structures and sped up by making the computation cache-aware, to solve challenging motion planning problems quickly, both within the robot and via cloud-based computers.

Taking advantage of multi-core parallelism is increasingly important due to the growth trends in CPU computational power. Gordon Moore famously predicted that computational power would grow exponentially [86]. This trend, dubbed Moore's law, continued for decades and was readily measured by the number of operations a single thread of execution on a CPU could compute per second (see Fig. 1.1). Around 2005, single-threaded execution speed reached physical limits, and no longer continued on the same trend—spelling an end of an age. At about the same time, CPU manufacturers started introducing multiple computing cores to their CPUs—each of these cores

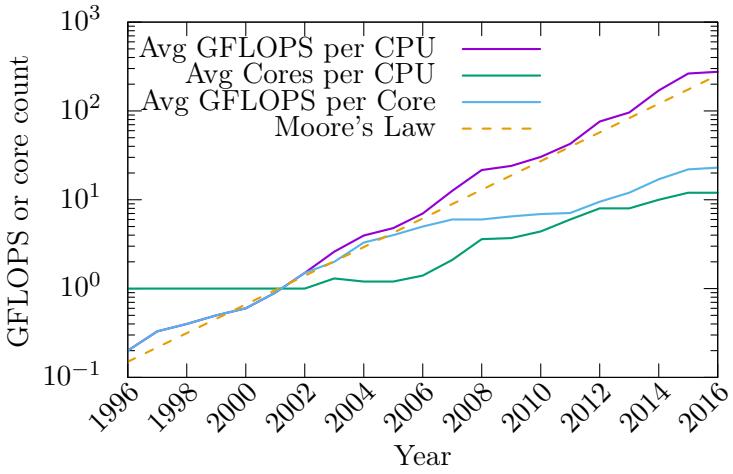


Figure 1.1: **CPU trends vs. Moore's law** [115]. Moore's law predicts an exponential trend in processing performance. Around 2005, due to physical limitations, the trend in single-core processing power began to taper. Around the same time CPU manufacturers began introducing additional processing cores to each CPU—allowing them to perform mutiple concurrent threads of operation. When the single-core performance trend is multipled by the number of cores, the resulting trend continues on along the Moore's law prediction. The implication is that in order to continue to gain exponential growth in performance, computer algorithms must make use of computational parallelism.

being capable of running one or more independent threads of execution. When we multiply these trends out, we can observe that computation power has indeed continued on the exponential growth that Moore's law predicted. But the implication is that we need to make use of computational parallelism in order to get this benefit.

Gaining the benefit of computational parallelism in motion planning is made problematic by many robot motion planning algorithms being inherently sequential in design. Parallelizing these algorithms requires attention to details such as gaining speedup on all parts of the algorithm (not just a small portion of it), as well as how to concurrently update shared data structures without causing data corruption or a program crash. To prevent data corruption and crashes, algorithm threads can ensure correct operation by locking shared data structures for exclusive access. But when one thread locks a data structure, other threads end up waiting for exclusive access. When waiting, these threads are not computing, and thus the algorithm does not speed up proportionally to the computational parallelism. This problem is made worse by increasing parallelism, as more threads compete for the same shared data structures and thus spend time waiting instead of computing. Since increasing parallelism is the trend in modern CPUs, approaches to correct concurrent access

to shared data structures must look beyond locking for exclusive access. This dissertation presents and evaluates lock-free data structures, as well as using characteristics inherent to motion planners that allow for asymptotically diminishing wait times. The result is motion planners that can speed up in proportion to the amount of computational parallelism available.

One of the side effects of accelerating a motion planner with computational parallelism is that they can generate a lot of data quickly—quickly enough that the slow access time of RAM becomes a performance issue. RAM's slow access time (relative to a CPU's computing speed) is often effectively hidden by a CPU's high-performance memory cache. The cache allows a program's frequently accessed data to be serviced quickly without being delayed by RAM; however, the performance benefit of these caches disappears when a program's working data set exceeds the size of the memory cache. The data set grows faster, thus exceeds the cache size sooner, when generated by a highly parallelized motion planner. In order to avoid having the working data set exceeding the cache size, this dissertation explores the use of cache-aware algorithms and data structures in motion planning. Keeping the motion planner's working data set in cache avoids the bottleneck of slow RAM access times, and keeps the motion planner running faster for longer.

In order for a robot to plan motions in a *dynamic environment*, that is, an environment with moving obstacles, the robot may need more computing power than it can carry and power onboard. Moving obstacles may block a robot's previously computed motion plan, or unblock a better path that the robot should follow—necessitating rapid updates to the existing motion plan, or generation of a new motion plan. The CPU of a sufficiently high degree-of-freedom robot may not be able to keep pace with environment changes (e.g., the Nao small humanoid robot [108] in Fig. 1.2 (c)). This problem is worsened when the physical design of a robot mandates using smaller, lighter, and lower energy-consuming mobile CPUs. While mobile CPUs benefit from a Moore's law-like trend in increasing parallelism, their small size and low energy usage means that they may not be able to compute motion plans fast enough to interact with dynamically changing environments. Thus, to get sufficient computing power for motion planning, some robots will have to leverage computing power from somewhere else.

The cloud is a promising source of networked computing power that is external to robots, scalable, and cost-effective. Cloud-based computers are unconstrained in size, weight, and power-consumption in relation to the robot's design. Cloud-based computers are a scalable solution, since the robot can

request as little or as much computational parallelism as it needs depending on the complexity of its motion planning tasks. Cloud-based computers can be cost-effective, since they allow the robot to potentially have a cheaper onboard CPU, and pay only for the computation that they use, as they use it. But for all the computational benefits of the cloud, there are challenges in how to coordinate the split in computation between the robot and the cloud and overcome bottlenecks introduced by the network connection between the robot and cloud-based computers. In order to overcome these bottlenecks, this dissertation introduces algorithms in which the robot navigates a dynamically changing environment using a small, but relevant, portion of the result of a computationally intensive cloud-based motion planner.

This dissertation advances the ability of robots to solve complex motion planning problems quickly. It does this through contributions of novel approaches to scalable parallel computation and concurrent data structures, and by embedded cache awareness. It further enables these advances to be effectively utilized whether the motion planner runs purely on the robot or in tandem with the cloud. The effectiveness of all the advances are demonstrated on physical robots in real-world scenarios. In concert with the research for this dissertation, we also released open-source projects that will allow these advances to be utilized in practice everywhere.

1.1 Problem and Motivation

In this section we define our motion planning problem, and we motivate extending it to a cloud-based solution.

1.1.1 Robot Motion Planning and Parallel Computation

Motion planning solves the problem of how to move a robot from a start configuration to a goal configuration while avoiding obstacles and remaining within task-specific constraints. As the definition of what is a robot is broad (e.g., Figs. 1.2 and 1.3), so too is the definition of motion planning for robots. Solving the general case of motion planning typically requires identifying and planning the motion for the robot's degrees of freedom—e.g., joint angles for an articulated robot, or position and orientation on a floor for a vacuuming robot. The degrees of freedom form the configuration space ($\mathcal{C}$-Space) of the robot's motion planning problem. Motion planning in the $\mathcal{C}$-Space allows the motion plan to be naturally converted to a sequence of actions to execute (e.g., a sequence of angles a joint should take). Unfortunately, while planning in the $\mathcal{C}$-Space ties naturally

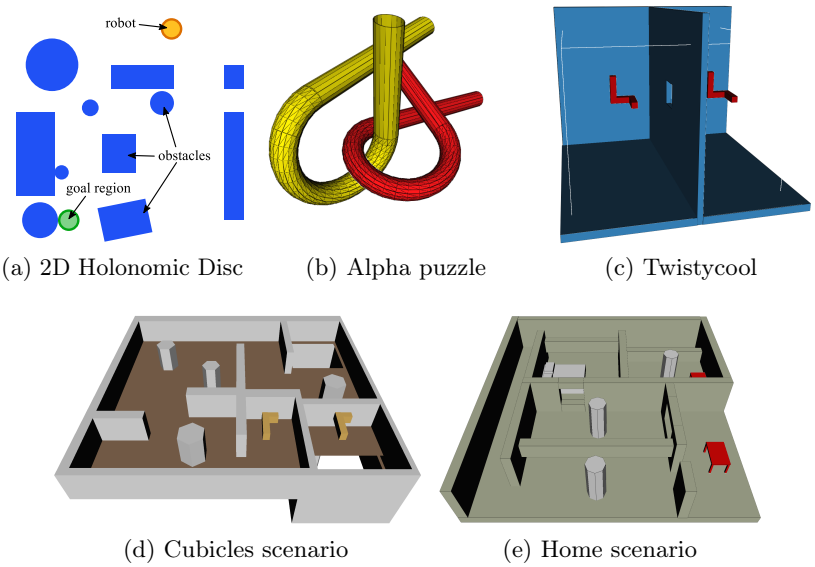
(a) Fetch robot (b) Baxter robot (c) Nao humanoid (d) F1/10 self-driving

Figure 1.2: **Physical robots used in experiments.** The motion planning algorithm research in this dissertation computes motion plans that we use to move physical and simulated robots around obstacles to a goal. The robots have varying characteristics including: (a) a mobile base and single arm, (b) two articulated arms and a stationary base, (c) arms, legs, and feet, and (d) a wheeled vehicle.

to the robot's motions, it causes the dimensionality of the planning problem to grow with the degrees of freedom and induces complex obstacle geometries in the $\mathcal{C}$-Space that are impractical to compute (see example in Fig. 1.4). This is one the main reasons that motion planning is computationally difficult.

In a *dynamic motion planning problem*, the goal and/or obstacles in the environment change over the course of the robot's motion. The robot must still avoid obstacles in such a *dynamic environment*, potentially by rapidly recomputing a new motion plan or updating its existing motion plan. If the robot fails to compute fast enough, the robot may not reach its goal in a timely manner, or worse, may collide with an obstacle.

A highly successful set of approaches to solving the motion planning problems in the general case are sampling-based motion planners [20]. Sampling-based motion planners operate by sampling random robot configurations, testing them against collisions and task-specific constraints, and connecting them to form a graph of valid motions. By testing random configurations, sampling-based planners do not need to directly compute obstacle geometries in $\mathcal{C}$-Space and thus avoid one of the computational complexities of motion planning. When the graph of motions connects the robot's starting configuration to a goal configuration through an unbroken path through the graph of valid motions, the motion planning problem is solved. Solutions for sampling-based motion planners are probabilistic—if a solution path exists, a *probabilistically complete* motion planner will find a solution with probability 1.0 given infinite time. The implication is that probabilistically complete motion planners will produce a feasible plan with increasing probability as they spend more time computing. In a similar vein, an *asymptotically optimal* motion planner will find the optimal solution (according



(a) 2D Holonomic Disc (b) Alpha puzzle (c) Twistycool

(d) Cubicles scenario (e) Home scenario

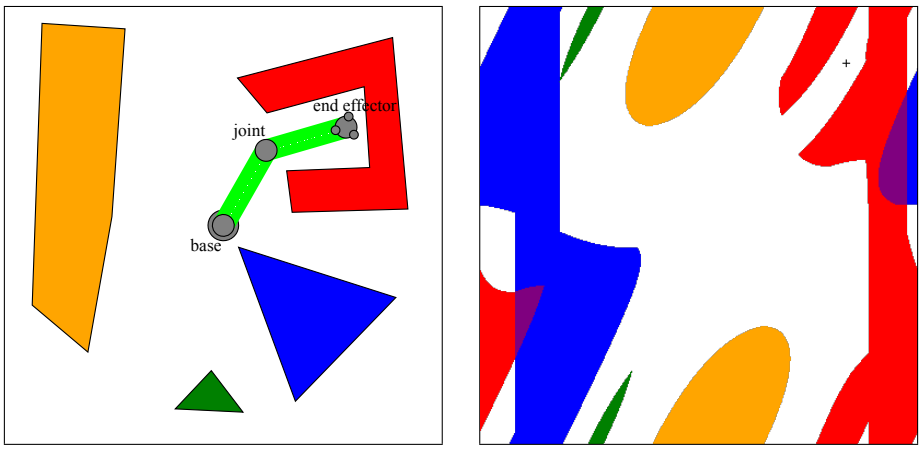
Figure 1.3: **Simulated robots used in experiments.** We experiment with simulations of various robot types and scenarios. In (a) is an example of a 2D holonomic disc motion planning problem, and in (b) – (e) are 3D rigid-body problems from OMPL [112] for planning in SE(3).

to a cost function) with probability 1.0, given infinite time. The implication is that asymptotically optimal motion planners converge towards an optimal solution as they spend more time computing.

In order to find motion plans sooner, or converge towards optimal motion plans faster, a sampling-based motion planner must thus be able to generate and evaluate more random samples at a faster rate. Evaluating samples at a faster rate, regardless of algorithmic advances, is inherently tied to the diminishing single-core performance trend of CPUs, unless the motion planner can make use computational parallelism. Exploiting computation parallelism in sampling-based motion planners that are sequential in design (e.g., RRT [76], RRT* [60]), requires novel approaches in order to gain parallelism that is *scalable*. With parallelism that is scalable, increasing parallelism by a factor of p leads to decreasing the solution time by a factor of $1/p$. With CPUs having as many as 32 cores (64 threads) becoming readily available, a parallelized motion planning algorithm has the potential to enable difficult motion planning problems that take minutes on a single-core, to take seconds when computed in parallel. But scaling to 64 threads requires novel approaches to coordinating the work between parallel threads in order to avoid slowdown.

1.1.2 Approaches to Parallelizing Motion Planning

One approach to speed up motion planning is to identify and parallelize the chief time-consuming component of the algorithm's implementation. For example, if the algorithm spends most of its time



(a) 2D 2-link planar robot (b) Obstacles in $\mathcal{C}$-space

Figure 1.4: **$\mathcal{C}$-space obstacles for a 2D 2-link planar robot** [44]. In (a) a 2-link robot operates in a workspace with four polygonal robots of varying colors. The robot's links can rotate around a fixed base and at a single joint, giving the robot two degrees of freedom. These obstacles are shown with the same colors in the visualization of the $\mathcal{C}$-Space in (b). A rotation around the base corresponds to moving the + horizontally, and a rotation around the joint corresponds to moving the + vertically. The shape of the $\mathcal{C}$-Space obstacles are irregular and difficult to define in closed form, especially in higher dimensions.

checking for collisions, then this approach leads to a solution that parallelizes the collision-detection subroutines [16]. Or, as analysis shows that the big-O runtime of a motion planner will eventually be dominated by nearest neighbor searching [68, 45], one might focus attention on speeding up nearest neighbor searching. However, focusing attention on speeding up one portion of the motion planning is inherently limited by Amdahl's law [6]—which argues that the maximum speedup one can observe is limited by the portion of the code that *cannot* be parallelized. Thus parallelizing the "outer loop" of the motion planner avoids this limitation, and is the focus of this dissertation.

Parallelized motion planning algorithms require access to shared data structures to be correct under concurrent operation. If two or more threads concurrently modify the same portion of a data structure, the result can be data corruption, data loss, or program failure. To avoid these issues, threads can employ locks to gain mutually exclusive access to the data structure. Exclusive access however, leads to slowdown, as threads spend time waiting on a lock instead of computing. This problem becomes worse with increasing parallelism, as more threads means increased likelihood of threads contending for the same lock. To enable scaling to increased parallelism, simple lock-based approaches are often not sufficient, and thus instead, this dissertation contributes approaches that

use lock-free atomic updates to shared data structures, data structures designed for concurrency, as well as exploiting properties of the motion planning algorithms to reduce likelihood of contention.

The longer a sampling-based motion planner runs, the larger its underlying data structures grow. Eventually the data structures will grow to exceed the size of the CPU's cache. When that happens, threads will increasingly access the data that they need from much slower RAM. The result can be a dramatic slowdown in the algorithm's performance. With a parallelized motion planner, this cache-based effect happens sooner, and thus this dissertation contributes novel approaches that make motion planning algorithms *cache-aware*.

1.1.3 The Economic Case for Robot Cloud Computing

Computing motion plans for high-degree-of-freedom robots typically requires a capital expense of thousands of dollars to purchase a high-end computer capable of computing timely solutions. As an alternative, would you prefer gaining access to the latest computational hardware on demand, and for cents per task? That is the promise of cloud computing for robots—a potential to lower costs and improve efficiency for a variety of robotics applications.

Cloud computing has the potential to change the way we design, use, and pay for robotic systems. Unlike traditional robots, which are purchased upfront, cloud computers are billed in units of usage time. Thus, when using cloud computing one can and should approach solving problems in the most cost-effective way possible. To illustrate, for \$10 000 one could purchase a high-end computer, or one could get 117 647 hours on a compute-optimized single-core cloud computer, 3 267 hours on a 36-core cloud computer, or one hour of 117 647 cores[1]. With a parallel algorithm for motion planning [4] and externalizing the robot's computation (e.g., [52]) to a cloud-based computer, one could dramatically reduce the time to solve a complex task. New robotics algorithms that leverage this computing power may extend a robot's service life and battery-based operation time, and reduce its initial and operating costs.

The cloud is already changing the way we think about computing for robots, but its full potential has not been tapped. To date, many data-centric, and pre-computation approaches leverage the cloud [62]. What about solving complex tasks with *near-term* deadlines by using the cloud to add

[1] As of March 2018, Amazon offers single-core servers at \$0.085/hr, and 36-core at \$3.06/hr.

computing power in response to the changing demands of a problem? This will be particularly valuable for network-connected robots that face challenging motion planning problems that involve high-degree-of-freedom systems, dense cluttered environments, learning complex task models, or managing high levels of uncertainty. In this section, we present an economic motivation for, and the research challenges posed by, leveraging cloud-based computation in online and interactive robot motion planning algorithms. Bringing the benefits of cloud-based computing to robots poses multiple open research challenges, such as: how to cost-effectively allocate computing, how to design algorithms around network bottlenecks, and how to split computation between a robot and the cloud.

The cloud changes the cost model of computing by shifting it from a capital expense (CapEx) to an operational expense (OpEx). Typically, robots require a large upfront CapEx, driven in part by the cost of the robot's computer. Using the cloud makes computing become an OpEx over the service life of the robot. With the right algorithms and utilization, an increase in a robot's OpEx will be offset by, not only a reduced CapEx, but also an increased service life, increased battery-based operation time, and a net improvement in operational efficiency.

Lower CapEx by extending a robot's service. A robot's service life may be extended through the use of cloud computing. The service life starts at purchase and ends when the robot's utility decreases to the point it is removed from service. Increasing the service life reduces the number of robots purchased over time, leading to a reduced CapEx. Consider a home assistance robot that aids someone with a variety of daily tasks of living. Such a robot could gain additional functionality by following a process similar to that of installing applications and updates to a smartphone or tablet. In this scenario, the robot becomes obsolete and needs replacement due to either physical component wear or due to advances in software exceeding the capabilities of the robot's computing hardware.

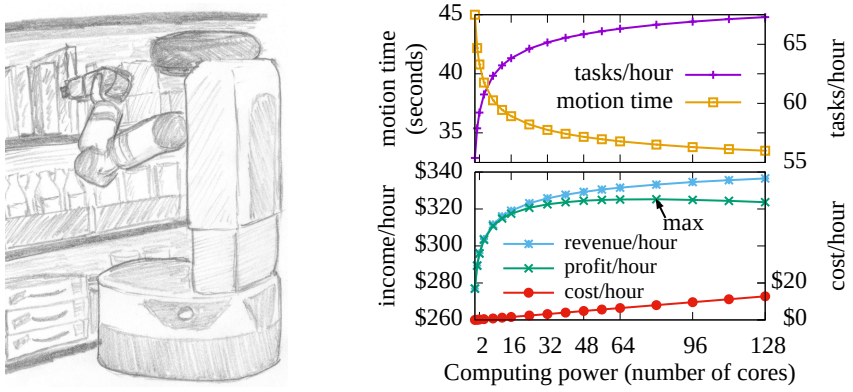
Historically, computing hardware has become obsolete much more quickly than non-computing hardware (e.g., motors, sensors). Smartphones, as a proxy for a robot's computing platform, have a life expectancy in the range of 3 to 4.7 years [8, 28]. Cars, as a proxy for a robot's non-computing hardware, have an average age in the US of 11.1 years [34]. The short service life of mobile computing devices is unsurprising when considering Moore's law, which observes an 18-month doubling in computation power as measured by transistor count. At the end of a 4.7 year service life, a robot

will have almost 9 times less computing power than its replacement. At the end of a car's 11.1 years, a robot will have almost 170 times less. The computing platform on a robot is fixed, but cloud services offer computers that are routinely upgraded. A robot that effectively utilizes the cloud for computation could thus potentially extend the time before it becomes computationally obsolete, and correspondingly extend its service life.

Cheaper robots with longer battery life. Incorporating a reliance on cloud computing into the physical design of a mobile robot will allow for cheaper robots with longer battery-based operation time. The computing platform in a robot is necessarily limited by economic factors, including price and, for mobile robots, physical size and battery capacity. Embedding high-end CPUs and GPUs enables higher performance computing, but comes at a cost of dramatically increased price and energy drain for the robot. Higher energy drain either requires increased battery size and weight, or results in reduced battery-based operation time. If instead, a robot's designers look to lower-power computing platforms sufficient to running baseline algorithms, while offloading intensive computation tasks to the cloud, their robot design can offer decreased battery size or allow for an increased battery-based operation time, all for a reduced upfront cost.

Robots that learn from their environment and from humans are examples that could naturally benefit from such a cloud-enabled robot design [120]. Cloud-based computation accelerates the learning of a model, while the robot need only use the learned model with low-powered computation. Such a system could be used in robots that are deployed to unfamiliar environments and expected to adapt rapidly to them as they operate. Novel cloud-based learning solutions [59], and low-powered fast convolutional network processors [30] and FPGAs [90], are making this closer to reality.

Improved operational efficiency. Cloud computation can not only reduce the initial purchase costs of a robot, but it can also increase a robot's operational efficiency, potentially increasing associated revenue and reducing the need to purchase more robots. Motion planning can be computationally intensive, whether attempting to find a feasible solution in a complex space, maximizing a task's success rate in the presence of uncertainty, or minimizing a motion's cost (e.g., path length, time to completion, energy required). Asymptotically optimal [60] and near-optimal [82] motion planners work to minimize a motion's cost by converging towards optimality. They converge



(a) Robot picking from shelf

(b) Efficiency gained with cloud computers

Figure 1.5: **Example of maximizing profit with cloud computing.** A warehouse packs packages using robots (a). The robots avoid collision with an ever-changing inventory by using motion planning algorithms. In (b), the warehouse wishes to maximize *profit/hour*, which here is computed as (*revenue/hour*) − (*cost/hour*). Each task the robot completes results in revenue for the company, thus more *tasks/hour* means more *revenue/hour*. The robot uses cloud computation of an asymptotically optimal motion planner to reduce *motion time* and thus pack more boxes per hour. The warehouse can adjust the amount of cloud-based computing resources it uses so that it maximizes profits.

faster when given more computing power or computational parallelism [49]. By leveraging parallelism of cloud computers for motion planning [12, 51], robots can complete tasks faster. When accelerating robot motions results in more revenue, the OpEx associated with cloud computing could be justified by net improved profits (see Fig. 1.5). When a fixed number of tasks are required per unit time, faster completion times means fewer robots are required, thus lowering CapEx.

1.2 Research Challenges

In this section we present the research challenges of accelerating motion planning on multi-core computers and extending it to a cloud-based solution.

1.2.1 Research Challenges in Parallel Computation of Robot Motion Plans

The problem of accelerating motion planning is broken down here into the design of parallel algorithms, novel concurrent data structures, and cache-aware planning.

Accelerating motion planning with parallel processing. Parallel processing in the form of multi-core CPUs is readily available in both cloud-based and robot-embedded computers, though the scale of parallelism is different between the two. For example, the robots in our lab have 4-core CPUs typically found in modern desktop computers, while cloud services readily offer computers

with up to 72-cores CPUs. The trend in both robot and cloud-based CPUs is towards increasing core counts. This problem is thus creating sampling-based motion planning algorithms that make use of this parallelism, while overcoming scaling limitations, ideally gaining linear (or better) speedup.

Nearest neighbor searching in SO(3) and SE(3). Nearest neighbor searching is an important performance bottleneck in sampling-based motion planners. Its computational complexity grows as the motion planner runs longer, and eventually dominates the computation time. Two important topological spaces in robot motion planning are SO(3) and SE(3), which are used for planning rigid body motions. This problem is thus accelerating nearest neighbor searching within the context of our concurrent nearest neighbor data structure using a novel space-partitioning approach.

Concurrent nearest-neighbor searching. In parallelized sampling-based motion planners, nearest-neighbor searching data structures need to be updated and queried concurrently. In traditional approaches to nearest neighbor searching data structures, exclusive access is required during inserts to avoid data structure corruption. Exclusive access, by definition, does not allow concurrent operation, and thus causes parallel sampling-based motion planners to slow down. This problem is thus creating a *concurrent* nearest-neighbor searching data structure that allows concurrent queries and updates and minimizes the possibility of waits.

Cache-aware motion planning. Gaining the benefits of a parallelized sampling-based motion planner means that motion plans will have significantly more samples in the same amount of wall-clock time. This presents a problem not typically seen with non-parallelized motion-planners—the motion planning graph and nearest-neighbor data-structures exceed the size of the CPU's high-speed memory caches. As a result, the traditional approach to sampling-based motion planners begins to dramatically slow down as memory access time can be orders of magnitude slower than cache access times. This problem is thus making motion planners cache-aware, and thus able to maintain the speed benefits of using the CPU's cache.

Efficient reusable motion planner implementation. Implementing a highly efficient motion planning algorithm often requires writing robot-specific code. The alternative of leveraging a reusable software library often trades off efficiency in favor of generality. We wish to put algorithms and

data structures from this dissertation into a reusable, highly efficient, motion planning software library. The challenge is thus creating a reusable motion planning software library that has an expressive language capable of supporting a wide variety of robots, and that produces highly efficient robot-specific code.

1.2.2 Research Challenges in Cloud-based Computation of Robot Motion Plans

Cloud computing offers many potential benefits, but realizing them presents several open research problems. Cloud computing services offer scaling computing power in a wide variety of options, from a single core virtualized on a server, to all cores on a high-end multi-core computer, to arrays of GPUs, to networked combinations of these. Motion planning algorithms that benefit from parallelism typically run with fixed parallelism configured a priori. With cloud computing, the amount of parallelism to allocate to a problem becomes a question of balancing benefit to the cost (instead of availability) of computing. Robots also must interact with a changing world, and in order to respond to changes (e.g., to sense and avoid collisions with obstacles) they must take into account the network latency (i.e., round-trip time) and bandwidth limits. One option is mixing or splitting computing between multiple sites: the robot's onboard computer and the cloud-based computers, ideally gaining the benefits of each site's strengths while avoiding the weaknesses. This research focus on using the cloud to speed up the motion planning computation—when the cloud is unavailable, the robot will have to fallback to slower planning using on-board computing or simply refuse to operate. The research challenges are thus: how to utilize the parallelism afforded by cloud computing, and how to adapt algorithms to work around limitations of the network.

Splitting computation between multiple sites. Algorithms can potentially address the resource allocation and network concerns by splitting computation between the robot's on-board computer, a co-located computer, and cloud-based computers. The robot's computer has the lowest latency and highest bandwidth access to its environment via its sensors and actuators. A cloud-based computer has relatively high latency and low bandwidth. Depending on the scenario in which the robot operates, some portion of the robot's computation can be split between the different sites. As an example, vision processing and motion tracking require a large amount of bandwidth and low latency in order to react to changes in the environment—matching the characteristics and (hopefully) capabilities of the robot's onboard or co-located computer. On the other hand, an

intensive pre-computation of a robot's path through its environment (barring dynamic obstacles) can be rapidly computed by high-performance parallel computing in the cloud. As a general research challenge, can we design robot algorithms that split portions of computation between multiple computing sites and thus gain the benefits of the cloud's massive computing power while meeting the demands of a problem that requires low-latency computation?

Network bottlenecks and deadlines. Robot algorithms that rely on cloud computing must consider and address the limitations imposed by the network. Advances in networking technology may improve the latency and bandwidth to an extent, but communication networks will always be slower than the interconnect between the robot and its onboard or co-located computer. This limit is fundamentally insurmountable, since it is a direct result of the speed of light. As such, network limitations vary by domain, and the challenges imposed by the network bottlenecks for robots in home and warehouse environments significantly differ from robots tasked with deep-sea and space exploration. For the class of algorithms and scenarios in which the results can be pre-computed, the network might not warrant concern. However, robots operate in the real world, and they must be able to sense and respond quickly to changes in the environment in order to avoid undesirable or harmful outcomes, especially in safety-critical scenarios, such as warehouse robots operating in close proximity to humans or with medical robots working with, or operating on, humans. To avoid undesirable outcomes, we pose the research challenge by borrowing language from the real-time computing community, and considering computing tasks with *hard deadlines* and *soft deadlines*. For robots computing tasks with hard deadlines (ones that cannot be missed), how can we ensure that a robot's motion planning algorithm will meet the deadline (or at worst, minimize the chance of missing the deadline)? For robotic tasks with soft deadlines (ones for which a miss results in reduced benefit or increased cost), how can a robotic algorithm maximize the benefit or minimize the cost of these tasks? More parallel processing can speed up computation to get ahead of the deadline, but the network remains a bottleneck of fundamental importance to these research challenges.

1.3 Contributions

This thesis makes a number of contributions to accelerating sampling-based motion planning for robots through the use of parallelization of motion planning algorithms, novel approaches to nearest neighbor searching, provably correct concurrent data structures, and cache-aware motion planning

algorithms. This thesis also extends these advancements to work on power-constrained robots using a novel algorithm and system architecture for partitioning the dynamic motion planning problem between a robot's CPU and a cloud-based high-power multi-core CPU. These advancements are made available as open-source software libraries that make use of a novel template-based software architecture. This section outlines these aforementioned contributions.

1.3.1 Motion Planning with Superlinear Speedup

To address the challenge of efficiently and scalably parallelizing a sampling-based motion planner, we take the approach parallelizing the "outer loop" of the RRT [76] and RRT* [60] sampling-based motion planners. By parallelizing at this level, we parallelize the entire algorithm, and thus avoid the limitations described by Amdahl's law. To get around slowdown and contention associated with locks, we present an algorithm that updates shared data structures through lock-free atomic operations. With these approaches, the algorithm demonstrates linear speedup with additional parallelism. With some inherent work-saving from this approach, and a simple partitioning of samples, the motion planner exhibits superlinear speedup.

This contribution appears in Chapter 2 and was originally presented at IROS [48] and later in journal form [49].

1.3.2 Fast Nearest Neighbor Searching in SO(3) and SE(3)

To address the challenge of having a fast nearest neighbor searching data structure for topological spaces common to robotic motion planning problems, we present a novel space-partitioning data structure for SO(3) and SE(3). For SO(3) searching it uses the distance metric defined by the shortest great-arc that subtends two rotations. For SE(3), it uses a metric that is the weighted sum of SO(3) and Euclidean. This data structure is based on the kd-tree [13], but uses partitioning hyperplanes that pass through the origin of a unit 4-sphere of the quaternion [70] representation of the rotation. The data structure's performance is demonstrated both with random samples, and embedded in a sampling-based motion planner.

This contribution appears in Chapter 3 and was originally presented at WAFR [46].

1.3.3 Concurrent Nearest Neighbor Searching

To address the challenge of having a fast nearest neighbor data structure that allows for *concurrent* operation, we present a novel concurrent data structure for nearest neighbor searching. This data structure is based on the kd-tree and thus supports our novel SO(3) and SE(3) partitioning approach.

This data structure defers partitioning decisions in order to generate splits that are better balanced than prior approaches, resulting in measurably faster performance in sampling-based motion planning problems. In order to support fast concurrent operation, the data structure makes use of lock-free atomic operations, memory-ordering directives, and fine-grain locks. We provide proofs of correct operation under concurrency through the use of linearization points, and we provide a proof of asymptotically wait-free operation in motion planning.

This contribution appears in Chapter 4 and was originally presented at WAFR [45].

1.3.4 Cache-Aware Sampling-based Motion Planning

To address the challenge induced by sampling-based motion planners generating working data sets that exceed the CPU's cache size, we present a novel approach to sampling-based motion planning that is *cache-aware*. This approach successively partitions the sampling space to keep the working data set to a size that fits in the cache. The proposed motion planner integrates its sampling strategy with a space-partitioning nearest neighbor structure thus constraining its queries to a small portion of the nearest neighbor data structure. In experiments, the cache-aware approach leads to measurable performance improvement—as much as halving the wall-clock time to compute a solution when compared to a non-cache aware approach.

This contribution appears in Chapter 5 and was originally presented at ICRA [50].

1.3.5 Cloud-based Motion Planning in Dynamic Environments

To address the challenge of gaining large-scale multi-core parallelism on power-constrained robots, we present a novel approach to partitioning the motion plan computation between a robot's CPU and a cloud-based computer. In this algorithm and system, the robot, with its fast access to its environment through its sensors and actuators, is responsible for reacting to and avoiding dynamic obstacles. The cloud-based computer, with its high degree of parallelism, is responsible for generating a large asymptotically optimal roadmap of options for the robot. The approach works around network bottlenecks by selecting and sending only the relevant portions of a roadmap given the robot's likely path. We experimentally validate the algorithm and system on a physical robot interacting with a obstacle sensed through an RGB+depth camera.

This contribution appears in Chapter 6 and was originally presented at WAFR [51].

1.3.6 Efficient Motion Planners via Templates

To make the contributions available in a reusable form, this thesis also presents and distributes an open-source software library for concurrent nearest neighbor searching and motion planning on multi-core system that is based upon C++ templates. The C++ template-based software architecture uses compile-time polymorphism to generate code that is custom to the robot's motion planning tasks. This type of customization produces motion planners run measurably more efficiently (both in memory and runtime) than equivalent planners based upon runtime polymorphism. While this library may have wide applicability, it is originally intended to run on power-constrained robots running multi-core CPUs one might find in mobile phones or similar mobile computing devices. We demonstrate the performance impact of this software architecture on a suite of benchmarks running on low-power CPUs.

This contribution is described in Chapter 7, will be presented at ICRA [47], and is available in source code with a free-to-use license.

1.4 Thesis Statement

This dissertation proposes and demonstrates that motion planning for robots can be accelerated through the use of algorithmic and data structure advances that leverage multi-core CPU architecture—whether the CPU is inside the robot or accessed through the cloud. With computational speed that effectively scales with increasing core count, robots are able to accomplish their tasks faster, thus enabling new tasks. This dissertation aims to prove the following thesis statement:

Robot motion planning algorithms using multi-core parallelism, concurrent data structures, and cache-awareness can demonstrate superlinear speedup. With this speedup, robots can solve complex motion planning problems sooner and converge towards optimal motion plans faster. The resulting faster motion planning can enable robots to effectively operate in dynamically evolving scenarios, including cases in which a robot with a low-power CPU gains access to faster motion planning through computers deployed in the cloud.

The chapters in this dissertation support this thesis statement as outlined below.

1.5 Organization

In Chapter 2 we present a parallelized algorithm for sampling-based motion planning that speeds up linearly, and in some cases superlinearly, with additional cores. In Chapter 3, we present an

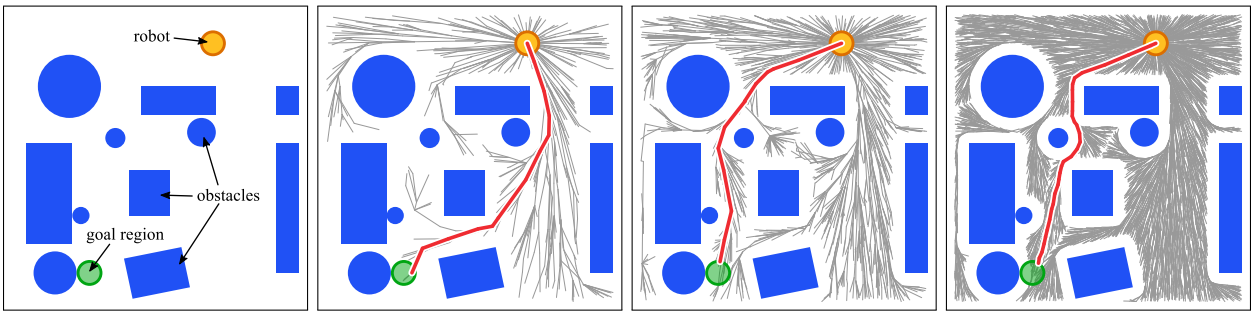
approach for fast nearest neighbor searching in SO(3) and SE(3), resulting in faster motion planning of 3D rigid body problems. In Chapter 4, we further accelerate nearest neighbor searching with a novel provably correct concurrent data structure that defers partitioning decisions to produce a better balanced tree. In Chapter 5, we introduce a *cache-aware* sampling-based motion planning, enabling faster motion planner of problems requiring many samples. In Chapter 6, we split motion planning between the robot's CPU and a cloud-based computer while overcoming network bottlenecks in order to make robots with low-power CPUs interact better with dynamic environments. In Chapter 7, we present the architecture of an open-source library that allows the aforementioned advancements to be used in real-world situations. Finally we conclude in Chapter 8 with a discussion of the promising implications and potential future directions of this work.

CHAPTER 2

Motion Planning with Superlinear Speedup

Incremental sampling-based motion planners, such as the Rapidly-exploring Random Tree (RRT) and RRT*, are used in a variety of robotics applications including autonomous navigation, manipulation, and computational biology [75, 60]. The objective of these planners is to find a feasible or optimal path through the robot's free configuration space from a start configuration to a goal configuration. In this chapter, we introduce PRRT (Parallel RRT) and PRRT* (Parallel RRT*), parallelized versions of the single-tree RRT and RRT* motion planners that are tailored to execute on modern multi-core CPUs.

Most modern PCs and mobile devices have between 2 and 32 processing cores with shared memory, and the number of cores is increasing. PRRT and PRRT* are designed to scale and efficiently utilize all available cores concurrently, enabling motion planning with substantial *speedup* with respect to the number of cores processing in parallel (see Fig. 2.1). Speedup, defined as the factor by which compute time is reduced with additional processing cores, is ideally proportional to the number of processing cores. In practice though, speedup is typically hindered by the overhead of coordinating updates between multiple cores. The methods proposed in this chapter reduce this overhead to the point at which speedup is near *linear* with the number of cores—thus, PRRT and PRRT* computing with p cores can reduce compute time to $1/p$ over an equivalent single-core motion plan computation. We have also observed that PRRT and PRRT* in some cases achieve a speedup that exceeds the number of processing cores, and thus appears to be *superlinear*. This superlinear speedup effect is based upon a comparison between the multi-core parallel motion planners presented here and the standard single-threaded algorithms on which they are based (or, equivalently, the parallel motion planner running on a single core). While the measured speedup can be superlinear, it should be understood that in theoretic terms speedup can never be superlinear [38], as one could devise a (potentially complicated) single-threaded algorithm that mimics the operations of a parallel process, e.g., through time-slicing. The superlinear effect measured in PRRT and PRRT* is a result of PRRT



(a) The scenario (b) 1 core (c) 4 cores (d) 32 cores

Figure 2.1: **PRRT* on 2D holonomic scenario**. We ran PRRT* for a 2D holonomic motion planning problem for a disc-shaped robot for 10 ms on 1, 4, and 32 processor cores. The red line shows the optimal path found. With the same wall clock time, adding more processor cores increases the size of the tree, enabling fast computation of higher quality motion plans on modern multi-core computers.

and PRRT* algorithmic enhancements that allow parallel operation to effectively reduce both the amount of work required and the time it takes to do that work.

Our focus is on challenging motion planning scenarios for which a large number (tens or hundreds of thousands) of configuration samples is typically necessary to find a feasible path or to compute a plan with the desired closeness to optimality. In RRT and RRT*, the time spent computing nearest neighbors grows logarithmically with each iteration as the number of samples rises, whereas the time spent per iteration on collision detection decreases as the expected distance between samples shrinks. Collision detection typically dominates computation time in the early iterations. But as the number of iterations rises and the number of samples increases, nearest neighbor search will dominate the overall computation.

To enable speedup regardless of the computational bottleneck (e.g. collision detection or nearest neighbor searching), we parallelize the outer loop of RRT and RRT*: we create multiple threads that each generate samples and incrementally extend the motion planning tree based on those samples. To parallelize at this level, independently working threads must share access to a nearest neighbor searching data structure and to the motion planning tree.

Shared access is often controlled using locks; when a thread must access a shared data structure, it first locks the data structure, then accesses it, and finally unlocks it. When another thread attempts to access a locked data structure it waits (i.e., is blocked) until the data structure is unlocked. When locking a data structure, there is often a trade-off with granularity—resulting in either blocked

threads or high overhead; either approach typically results in sublinear speedup. Blocked threads result from locking too large of a data structure, leading to threads spending time waiting instead of computing. High overhead results from repeatedly locking small portions of a data structure. With increasing processor counts, the sublinear effect of locking is only compounded as more threads must contend for the same resources.

To reduce causes of sublinear speedup and create opportunities, but not a guarantee, for superlinear speedup, PRRT and PRRT* introduce three key components relevant to multi-core concurrency. The first is *lock-free concurrency using atomic operations*. To eliminate slowdowns caused by lock overhead and contention, PRRT and PRRT* use lock-free shared data structures that are updated using an atomic compare-and-swap (CAS) operation, a universal primitive [116]. A CAS operation has three arguments: a location in shared memory, the expected value stored therein, and a new value to replace the previous. In a single atomic step, CAS loads the value stored in memory, compares it to the expected value and, only if they are the same, stores the new value in memory. Without the atomic guarantee, another concurrent thread would be able to store a different value between the CAS's load and store. The atomic operation removes the need for locks when updates to shared data structures can be formulated into a single update. When a comparison fails due to a change made by another thread, the update is reformulated with the new information and tried again until it succeeds or is no longer necessary. In PRRT and PRRT* we observe that as the number of nodes n in a motion planning tree increases, the probability that any of the p threads are updating the same part of the motion planning tree decreases ($\lim_{n\to\infty} O(p/n) = 0$). As a consequence, CAS operations rarely fail, and we avoid the unnecessary blocking and overhead associated with locks. Lock-free operations eliminate the need for locks and hence reduce the overhead that might otherwise be associated with concurrent access to a shared-memory data structure. Lock-free operations by themselves at best enable linear speedup, but can be used in conjunction with other components to create opportunities for superlinear speedup.

The second component introduced in PRRT and PRRT* that sets up conditions in which superlinear speedup might occur is *cache-friendly partition-based sampling*. To reduce the size of each thread's working data set, we partition the configuration space into non-overlapping regions and assign a partition to each thread. Partitioning has two benefits. First, it reduces the likelihood that two threads will simultaneously attempt to modify the same part of the shared data structures,

reducing CAS failures. Second, as each processor core is expected to work in a smaller subset of the nearest neighbor data structure, more of the relevant structure can reside in each core's cache [1], thus creating an opportunity for superlinear speedup. Cache-efficiency, while not affecting the algorithmic complexity, can lead to significant real-world performance gains on modern CPU architectures [73].

The third component introduced to create opportunities for superlinear speedup in PRRT* is *parallel work-saving*. During the rewiring phase of RRT*, the algorithm evaluates the costs of paths to nearby nodes, rewires them through the new node if such routing would produce a shorter path, and percolates updates up the tree. To reduce the number of rewiring operations in RRT*, we ensure that when multiple threads attempt to rewire the same portion of the tree, only the one with the better update continues. This frees the other threads to continue expanding the RRT*, effectively reducing computation effort relative to single-threaded RRT* for percolating rewiring up the tree. Parallel work-saving can enhance an algorithm's performance and can in some cases enable superlinear speedup.

PRRT and PRRT* are designed to run on standard shared-memory, multi-core, CPU-based computing platforms (rather than, for example, a cluster or a GPU). This facilitates easy direct integration with existing libraries for collision detection, robot kinematics, and physics-based simulation [102, 112]. The contributions of this chapter were originally introduced in a conference paper [48] and a journal paper [49]. We provide pseudocode sufficiently detailed to show where CAS operations are used, how they impact the surrounding instructions, and how we ensure correctness under concurrency. We demonstrate the fast performance and scalability of PRRT for feasible motion planning using the Alpha Puzzle scenario and a random spheres scenario, and we demonstrate PRRT* for optimal motion planning using the Cubicles scenario, a holonomic disc-shaped robot, and a SoftBank Nao [108] small humanoid robot performing a 10 degree of freedom 2-handed task.

2.1 Related Work

Sampling-based motion planners include several components that can naturally be parallelized, and prior work has taken multiple avenues to exploit this parallelism using multi-core and multi-processor CPUs, clusters, and GPUs. Early work by Amato et al. [4] showed that the batched operations of sampling-based probabilistic roadmaps (PRMs) can be parallelized. In this chapter, our focus is on parallelizing the *anytime* motion planners RRT and RRT*.

Parallelizing RRT introduces new challenges since the validity of the tree must be maintained as it is updated by multiple concurrent threads of execution. A direct approach on a shared-memory system is to use locks on shared data structures, which is one of the methods proposed by Carpin et al. [19] and implemented as pRRT in OMPL [112]. Parallelizing RRT has also been investigated for distributed-memory systems common in clusters. Devaurs et al. [23] propose collaborative building of an RRT across multiple processes using message passing. This approach achieves a sublinear speedup as the number of available processors increases. Jacobs et al. [57] introduce speedups by adjusting the amount of local computation before making an update to a global data structures and by radially subdividing the configuration space into regions. Approaches targeting distributed-memory systems (e.g., [23, 57]) can also be run on shared-memory systems, but they do not take advantage of shared-memory primitives that can offer additional opportunities for speedup. KPIECE [114] prioritizes cells in a discretized grid for sampling based upon a notion of each cell's importance to solving a difficult portion of the motion plan and has been demonstrated to parallelize on shared-memory systems using locking primitives. Our focus is on shared-memory systems (common in PCs and mobile devices), which enables us to utilize atomic CPU operations and cache-friendly algorithms to set up conditions under which superlinear speedup might occur for a single RRT.

Several approaches to parallelizing motion planning across multiple cores/processors have utilized multiple tree-based data structures. Carpin et al. [19] propose an "OR" parallel algorithm in which several RRT processes run in parallel and the algorithm stops when the first RRT process finds a solution. Plaku et al. [96] introduced the Sampling-based Roadmap of Trees (SRT) algorithm, which subdivides the motion planning problem into subproblems that are distributed, solved by another planner (e.g., RRT), and then connected together. SRT achieves near-linear speedup that slightly tapers at higher processor counts. Otte et al. [94] also distribute the generation of independent path planning trees among several processes and achieve significant speedups by sharing information between processes about the best known path. Unlike the above methods that rely on multiple trees, we focus on building a single motion planning tree as in RRT and RRT*. Hence, our approach is complementary to the above multi-tree methods, which utilize multiple single-tree data structures. Our lock-free methods for shared-memory, multi-core concurrency result in an empirical superlinear speedup for some scenarios for both feasible and optimal single-tree motion planning.

Bialkowski et al. [16] parallelize RRT* and related methods by improving the rate of collision detection. This approach results in substantial speedups for environments where collision detection dominates processing time. But due to Amdahl's law [6], parallel performance will taper as the number of samples increases and nearest neighbor checks begin to dominate computation time.

Partitioning of configuration space has been used to various effect in motion planning. For example, Rosell et al. [103] hierarchically decomposes C-space to perform a deterministic sampling sequence that allows uniform and incremental exploration. Morales et al. [87] automatically decompose a motion planning problem into (possibly overlapping) partitions well-suited for one of many (sampling-based) planners in a planning library. Yoon et al. [125] show how cache-efficient layouts of bounding volume hierarchies provide performance benefits in the context of collision detection.

GPU-based parallel computation has also been used to accelerate motion planning, including GPU-based methods for the PRM [95], rasterization-based planning [77], Voronoi diagram-based sampling [65, 35], and R* [66]. Implementing GPU-based algorithms is challenging in part because the single-instruction-multiple-data (SIMD) execution model of GPU's constrains algorithm design. When each thread needs to do something different (inherently divergent), such as traversing a space partitioning tree, the SIMD model loses nearly all ability to parallelize [43]. Another challenge with GPU approaches is that, while they can gain the benefit of the high computational throughput associated with GPUs, they sacrifice some interoperability with standard systems and libraries based upon CPUs.

2.2 Problem Formulation

In this section, we formally define the computing environment and motion planning problem.

2.2.1 Parallel Computing Environment

Our target computing environment is the one available in almost every modern computer: a multi-core/multi-processor concurrent-read-exclusive-write (CREW) shared-memory system with atomic operations that synchronize views of memory between threads running on different cores [40]. This is the model in the current generation of x86-64 and ARM multi-core processors as well as many other CPU architectures.

In this environment, a computer contains one or more *processors*. Each processor may contain one or more *cores*. Each core acts as an independent CPU capable of having a single *thread* running

simultaneous to the threads running on the other cores. The total number of cores in the system is:

$$p = (\#\text{ of cores per processor}) \times (\#\text{ of processors}).$$

For example, a system with four processors, where each processor has 8 cores, has $p = 32$.

Speedup refers to how much a parallel algorithm is faster than a corresponding sequential algorithm. Let T_p be the execution time of a program that is executed using p cores. Formally, speedup S_p is the ratio of the sequential (single-threaded) execution time T_1 to parallel execution time T_p with p cores:

$$S_p = \frac{T_1}{T_p}.$$

Linear speedup means $S_p = p$, and superlinear speedup means $S_p > p$.

To avoid sublinear speedup, we use the atomic compare-and-swap (CAS) operation for fast lock-free updates to data structures. To help enable superlinear speedup, we exploit the fast, but limited in size, CPU memory cache. Modern processors typically have a cache hierarchy between the core and RAM that includes one or more small but fast caches local to each core (L1 and L2) and a larger and slower cache shared among cores (L3). When the data set in use by a core is smaller, the core uses the faster local caches more often and gains a proportional speed benefit. CPU caches can be leveraged to gain superlinear speedups by distributing the working dataset into smaller chunks across multiple cores.

2.2.2 Problem Definition

Let $\mathbf{q} \in \mathcal{C}$ be a d-dimensional vector representing the configuration of a robot, d is the number of degrees of freedom, and $\mathcal{C}$ is the set of all possible configurations the robot may take (the *configuration space*). Let $\mathcal{C}_{\text{free}} \subseteq \mathcal{C}$ denote the subset of the configuration space for which the robot is not in collision with an obstacle.

The objective of PRRT (feasible motion planning) is to find a path in the robot's configuration space that is feasible (e.g., avoids obstacles) and reaches the goal region. Formally, the objective of PRRT is to compute a path $\Pi : (\mathbf{q}_0, \mathbf{q}_1, \mathbf{q}_2, \ldots, \mathbf{q}_{\text{end}})$ such that $\mathbf{q}_0 = \mathbf{q}_{\text{init}}$, $\mathbf{q}_{\text{end}} \in Q_{\text{goal}}$, Π lies in $\mathcal{C}_{\text{free}}$, $\mathbf{q}_{\text{init}}$ is the starting configuration of the robot, $Q_{\text{goal}} \subseteq \mathcal{C}_{\text{free}}$ is the set of goal configurations. The objective of PRRT* (optimal motion planning) is to compute a feasible path that reaches the

goal region and minimizes a user-defined cost function. An example cost function is the minimum total Euclidean length of the segments in the planned path.

2.2.3 Problem-specific Functions

Similar to their sequential motion planning counterparts RRT and RRT*, PRRT and PRRT* require as input the definition of problem-specific functions. For two configurations $\mathbf{q}_1, \mathbf{q}_2 \in \mathcal{C}$, the function $\texttt{STEER}(\mathbf{q}_1, \mathbf{q}_2)$ returns a new configuration that would be reached if taking a trajectory from $\mathbf{q}_1$ heading toward $\mathbf{q}_2$ up to some maximum user-specified distance. The function $\texttt{FEASIBLE}(\mathbf{q}_1, \mathbf{q}_2)$ returns `false` if the local path from $\mathbf{q}_1$ to $\mathbf{q}_2$ collides with an obstacle or violates some motion constraint, and `true` otherwise. For PRRT*, the function $\texttt{COST}(\mathbf{q}_1, \mathbf{q}_2)$ specifies the cost associated with moving between two configurations $\mathbf{q}_1$ and $\mathbf{q}_2$, which can equal control effort, Euclidean distance, or any problem-specific user-specified metric that can be used with RRT* [60]. We also require a function $\texttt{GOAL}(\mathbf{q})$ that returns `true` if $\mathbf{q} \in Q_{\text{goal}}$ and `false` otherwise.

The above problem-specific functions are standard in RRT and RRT*, which enables current implementations of these problem-specific functions to be used in PRRT and PRRT* unchanged, provided the functions allow for correct concurrent evaluation.

2.3 PRRT

In this section, we present Parallel RRT (PRRT), a lock-free parallel extension of the RRT algorithm. We describe the algorithm in sufficient detail to show where atomic operations are used, how they impact the algorithm design, and how we ensure correctness under concurrency.

The PRRT algorithm maintains data structures that are shared across all threads, including the data structure for nearest neighbor searching, the RRT tree τ, the approximate iteration number, and whether or not a path to the goal has been found. As shown in Algorithm 1, PRRT begins by partitioning the configuration space into non-overlapping regions and launching an independent thread for each partition. For peak performance, each thread runs on a dedicated core. The impact of partitioning is that it localizes each thread's operations (e.g. random sampling, nearest neighbor searching, and collision detection) to a smaller portion of the configuration space. This allows for more effective use of each core's caches and contributes in some cases to our method's empirical superlinear performance.

Algorithm 1 `PRRT`

1: initialize τ
2: **for** $i = 1 \ldots$ thread_count **do**
3: $\quad s \leftarrow$ partition(i, thread_count)
4: $\quad w_i \leftarrow$ start new thread `PRRT_Thread`(τ, s)

2.3.1 PRRT Threads

The algorithm for each thread of PRRT is shown in Algorithm 2. PRRT is nearly identical to the standard RRT algorithm except that (1) each thread only samples in its partition, (2) PRRT uses a lock-free nearest-neighbor data structure (introduced in Sec. 2.3.2), and (3) all graph updates are lock-free. We note that although sampling is local to a partition, the nearest-neighbor data structure and graph of motions spans the entire configuration space and is shared by all threads.

As in the standard RRT algorithm, the function PRRT creates a new node for $\mathbf{q}_{\text{new}}$ and sets its parent pointer to the node of $\mathbf{q}_{\text{near}}$ (line 6) and then inserts the node into the lock-free kd-tree (line 7). The ordering is important since PRRT must ensure that other threads only see fully initialized nodes, and the new node will become visible as soon as it is inserted into the kd-tree.

Complicating matters, modern CPUs and compilers may speculatively execute memory reads and writes out-of-order as a performance optimization. These optimizations are done in a manner that guarantees correctness from the view of a single thread, but out-of-order writes may cause a thread executing concurrently on another core to see uninitialized or partially initialized values, resulting in an incorrect operation. The solution to this problem is to issue a memory barrier (also known as a memory fence) [83]. A memory barrier tells the compiler and CPU that all preceding memory operations must complete before the barrier, and similarly no memory operations may speculate ahead of the barrier until after the barrier completes. In many CPU architectures, atomic operations such as compare-and-swap imply a memory barrier. For `PRRT_Thread` to operate correctly, it must ensure that a memory barrier is issued before a new node becomes visible to another thread, which is done in the lock-free kd-tree insertion algorithm described next.

2.3.2 Building a Lock-Free kd-Tree

The RRT algorithm requires an algorithm `Nearest`$(\tau, \mathbf{q})$ for computing the nearest neighbor in τ to a configuration $\mathbf{q}$ in configuration space. Using a logarithmic nearest neighbor search rather than a brute-force linear algorithm often results in a substantial performance gain [123]. In PRRT,

Algorithm 2 `PRRT_Thread`(τ, s)

while not done **do**
  $\mathbf{q}_{\text{rand}} \leftarrow$ random sample from s
  $\mathbf{q}_{\text{near}} \leftarrow$ `Nearest`$(\tau, \mathbf{q}_{\text{rand}})$
  $\mathbf{q}_{\text{new}} \leftarrow$ `STEER`$(\mathbf{q}_{\text{near}}, \mathbf{q}_{\text{rand}})$
  if `FEASIBLE`$(\mathbf{q}_{\text{near}}, \mathbf{q}_{\text{new}})$ **then**
    $\tau \leftarrow \tau \cup \text{edge}(\mathbf{q}_{\text{near}}, \mathbf{q}_{\text{new}})$
    `LockFreeKDInsert`$(\mathbf{q}_{\text{new}})$
    if `GOAL`$(\mathbf{q}_{\text{new}})$ **then**
      done $\leftarrow$ **true**

Algorithm 3 `LockFreeKDInsert`$(\mathbf{q}_{\text{new}})$

$n_{\text{new}} \leftarrow \{\text{value:}\mathbf{q}_{\text{new}}, \text{split:}\varnothing, \text{a:}\varnothing, \text{b:}\varnothing\}$
$\mathbf{q}_{\text{min}} \leftarrow$ minimum bounds of sample space
$\mathbf{q}_{\text{max}} \leftarrow$ maximum bounds of sample space
$n_{\text{ptr}} \leftarrow$ pointer to kd_root
for $d = 0 \rightarrow \infty$ **do**
  $a \leftarrow d \bmod \kappa$
  if node in n_{ptr} is `null` **then**
    n_{new}.split $\leftarrow$ `Split`$(\mathbf{q}_{\text{min}}, \mathbf{q}_{\text{max}}, \mathbf{q}_{\text{new}}, a)$
    — memory barrier —
    if `CAS`$(n_{\text{ptr}}, \texttt{null}, n_{\text{new}})$ **then**
      return
  if $\mathbf{q}[a] < n_{\text{ptr}}$.split **then**
    $\mathbf{q}_{\text{max}}[a] \leftarrow n_{\text{ptr}}$.split
    $n_{\text{ptr}} \leftarrow$ pointer to n_{ptr}.a
  else
    $\mathbf{q}_{\text{min}}[a] \leftarrow n_{\text{ptr}}$.split
    $n_{\text{ptr}} \leftarrow$ pointer to n_{ptr}.b

for nearest neighbor searches we use a variant of a kd-tree data structure [13] that we adapt to allow for concurrent lock-free inserts using CAS.

Each node of the kd-tree is a k-dimensional point (i.e., a configuration in PRRT), where $k = d$ is the dimension of the configuration space. The kd-tree is a binary tree in which each non-leaf node represents an axis-aligned splitting hyperplane that divides the space in two—points on one side of this hyperplane are in the left subtree of that node and the other points are in the right subtree. The axis associated with a node is based on its depth (i.e., level) in the tree. For example, in 3D Euclidean space the hyperplane for a node in the first level of the kd-tree is perpendicular to the x-axis based on that node's x dimension value. For successive layers, the splitting is perpendicular to the y-axis, then the z-axis, and then repeating $x, y, z, x, y, z, \ldots$ down the tree.

To insert a node in the kd-tree for fast nearest neighbor searching, `PRRT_Thread` calls the lock-free kd-tree insert function `LockFreeKDInsert` shown in Algorithm 3. It starts with a pointer to the root (line 4), then traverses down the kd-tree by different dimensions (lines 5, 6) until it finds an empty branch (line 7). Once found, it generates and records the split (line 8), performs a memory barrier, and then a CAS (lines 9, 10) to change the pointer from `null` to the new node that was allocated and initialized in line 1. If the CAS succeeds, the node is inserted and the algorithm returns. If another thread already updated the pointer, the CAS will fail, and the algorithm will continue to walk down the tree until it can attempt another insert. The memory barrier before the CAS ensures that the node is fully initialized before it is visible to other threads when the CAS succeeds.

In line 8, `Split` denotes a function that generates the hyperplane. The split is generated based upon the bounds of the region of the node's parent. The bounds are initialized in lines 2 and 3 and updated in lines 15 and 18. If the bounds are known and finite, `Split` forces a mid-point split [81] by returning $(\mathbf{q}_{\min} + \mathbf{q}_{\max})/2$. If the bounds are not known, as might happen with the initial values at the root of the tree, `Split` returns $\mathbf{q}_{\text{new}}[a]$, causing the inserted value to define the split.

The kd-tree handles most spaces relevant to motion planning in configuration spaces, including $\mathbb{R}^n$, $\mathcal{T}^n$, and combinations thereof with an appropriate distance metric [123]. For $\mathbb{R}^n$ spaces, we consider Euclidean distance metrics. For $\mathcal{T}^n$ spaces (with unbounded revolute joints where $\theta = \theta + 2n\pi$ for any integer n) we consider distance metrics based on a circular distance in the form $\text{dist}_{S^1}(\theta_1, \theta_2) = \min(|\theta_1 - \theta_2|, 2\pi - |\theta_1 - \theta_2|)$. For a combination of these spaces, we consider the root sum of squares.

We augment the lock-free kd-tree to support SE(3) and SO(3) by defining splits based on the approach of vantage-point trees (vp-trees) [124]. The kd-tree defines a split on an SO(3) component using an orientation $\mathbf{a}_{\text{split}}$ in space and a pre-defined distance ϕ from the orientation. The distance function is the shortest arc-length between two orientations and thus ranges from 0 to π. Representing orientations using quaternions [39], $\text{dist}_{\text{SO3}}(\mathbf{a}_1, \mathbf{a}_2) = \arccos |\mathbf{a}_1 \cdot \mathbf{a}_2|$. Orientations that are less than ϕ away from $\mathbf{a}_{\text{split}}$ are on one side of the split, and orientations greater than ϕ away are on the other side. We preselect ϕ as $\sec 30°$, as that produces an even split on the orientations in SO(3). The `Split` function on the SO(3) component generates a split orientation by rotating the orientation component $\mathbf{a}_{\text{new}}$ of the inserted point by ϕ about an arbitrary axis. This causes $\mathbf{a}_{\text{new}}$ to lie exactly

on the split. This vp-tree-based approach enables the lock-free kd-tree to efficiently support the SE(3) and SO(3) configuration spaces.

PRRT and PRRT* builds up the lock-free kd-tree on the fly by inserting randomly generated configuration samples. The resulting tree remains relatively balanced. It can be shown that the expected number of comparisons required to insert a random sample into a binary tree generated with uniform random insertions is about $2 \ln n$ [69, p. 430–431].

The kd-tree can be used for any number of dimensions, but may become inefficient in very high dimensional spaces [123]. Even in such cases, kd-trees distribute random updates throughout the tree, leading to low contention over insertion points. In brute-force approaches based upon arrays or lists, inserts at a single insertion point (e.g. the tail of the list/array) may result in contention.

2.3.3 Querying a Lock-Free kd-Tree

For a given query sample, `Nearest` and `Near` search the lock-free kd-tree for the sample closest to it, or all samples within a radius of it, respectively. They successively compare the query to each traversed node's splitting hyperplane, and recurse down the side on which the query sample lies (the "near" side). Recursion ends when encountering empty branches. Upon return from the near side, the methods traverse the "far" side of the hyperplane only if it is possible that points in that part of the tree would be closer than the closest found so far (`Nearest`) or within the search radius (`Near`).

In practice PRRT can be used with other nearest-neighbor search approaches that allow for non-blocking searches and low-contention updates, and provide partitioned locality properties. The alternative of using a nearest-neighbor data structure with locks is also possible, but as shown in the results in Sec. 2.5, unlike the lock-free kd-tree, a lock-based kd-tree will result in sublinear speedup as different threads contend for access to the structure.

In our implementation, we consider two schemes for configuration space partitioning that naturally align with the nearest neighbor search kd-tree: (1) an even subdivision created by "slicing" along the first dimension of configuration space, and (2) a multi-dimensional grid. In both cases, each thread samples within their assigned (and unchanging) partition. While more sophisticated partitioning approaches (e.g. [114, 103, 87]) might look for ways to focus sampling on regions of difficulty (such as regions containing narrow passages), our motivation in partitioning is to create locality with sampling and nearest neighbor searches, and thus improve CPU cache utilization. As seen in the

Algorithm 4 PRRT*

1: initialize τ
2: **for** $i = 1 \ldots$ thread_count **do**
3: $s \leftarrow$ partition(i, thread_count)
4: $w_i \leftarrow$ start new thread PRRT*_Thread(τ, s)

results, the choice of partitioning scheme has an impact on the overall performance of the motion planner depending on the scenario.

2.4 PRRT*

In this section, we present Parallel RRT* (PRRT*), a lock-free parallel extension of the RRT* algorithm. The PRRT* algorithm shares across all threads the data structure for nearest neighbor searching, the RRT* tree τ, the approximate iteration number, and the best path to the goal found by any of the threads. PRRT*, shown in Algorithm 4, begins just like PRRT except it launches threads of PRRT*_Thread(τ, s).

2.4.1 PRRT* Threads

PRRT* expands the motion planning tree much like PRRT except that it includes the additional step of "rewiring" a small neighborhood of the tree to enable finding optimal paths. PRRT*_Thread, shown in Algorithm 5, is the main loop of a thread of PRRT*.

At a high level, PRRT* works much like standard RRT*. In the outer loop, it randomly samples a configuration, finds the sample's nearest neighbor in the motion planning tree, and computes a new configuration by steering from the nearest neighbor toward the sampled configuration (lines 2–5). PRRT* then searches for all the configurations in a ball around the new configuration (line 6) using the ball radius from RRT*[60]. PRRT* then connects the new configuration to the configuration in the ball that produces the shortest path (lines 8–17), and then inserts the newly connected configuration into the nearest neighbor structure (line 21). Finally, it rewires any configuration in the ball radius that produces a shorter path to goal through the newly added configuration.

The notable differences from standard RRT* are: (1) each thread samples within a partition of the configuration space (line 2), (2) nearest neighbors are found using a lock-free kd-tree (lines 3 and 6), (3) new configurations are added to the RRT* tree in a manner that accounts for parallelism by fully initializing them before adding them to the nearest-neighbor structure (lines 18–20), and (4) rewiring is accomplished entirely via lock-free operations.

Algorithm 5 PRRT*_Thread(τ, s)

1: **while** not done **do**
2: $\mathbf{q}_{\text{rand}} \leftarrow$ random sample from s
3: $n_{\text{nearest}} \leftarrow$ Nearest($\tau, \mathbf{q}_{\text{rand}}$)
4: $\mathbf{q}_{\text{new}} \leftarrow$ STEER(n_{nearest}.config, $\mathbf{q}_{\text{rand}}$)
5: **if** FEASIBLE(n_{nearest}.config, $\mathbf{q}_{\text{new}}$) **then**
6: $N_{\text{near}} \leftarrow$ Near($\tau, \mathbf{q}_{\text{new}}, \min\{\gamma \left(\frac{\log|\tau|}{|\tau|}\right)^{1/d}, \eta\}$)
7: $c_{\min} \leftarrow \infty$
8: **for all** $n_{\text{near}} \in N_{\text{near}}$ **do**
9: **if** FEASIBLE(n_{near}.config, $\mathbf{q}_{\text{new}}$) **then**
10: $c_{\text{link}} \leftarrow$ COST(n_{near}.config, $\mathbf{q}_{\text{new}}$)
11: $c_{\text{path}} \leftarrow n_{\text{near}}$.edge.cost $+ c_{\text{link}}$
12: **if** $c_{\text{path}} < c_{\min}$ **then**
13: $n_{\min} \leftarrow n_{\text{near}}$
14: $c_{\min} \leftarrow c_{\text{path}}$
15: n_{new}.config $\leftarrow \mathbf{q}_{\text{new}}$
16: $e_{\text{new}} \leftarrow (n_{\text{new}}, c_{\min}, n_{\min})$
17: n_{new}.edge $\leftarrow e_{\text{new}}$
18: LockFreeKDInsert(n_{new})
19: **if** e_{new} is expired **then**
20: PRRT*_Update(n_{new}.edge, e_{new})
21: **if** GOAL(e_{new}) **then**
22: record goal
23: **for all** $n_{\text{near}} \in N_{\text{near}} \setminus \{n_{\min}\}$ **do**
24: PRRT*_Rewire($\tau, n_{\text{near}}, n_{\text{new}}$)

2.4.2 PRRT* Rewiring

During the rewiring phase of RRT*, the algorithm considers paths to configurations nearby the newly added configuration, and it rewires the RRT* tree if re-routing those paths through the newly added configuration is both FEASIBLE and results in a shorter path. Following the approach of prior implementations of RRT* [60, 112], we store the path cost to that node's configuration within each RRT* node and push updates down the tree when a node is rewired.

PRRT* formulates rewiring (Algorithm 6) into a CAS operation that guarantees rewiring is completed correctly, even if another thread is concurrently accessing or rewiring the same node. If the CAS update fails, the assertion about the new trajectory being shorter may now be incorrect. In that case, the update is re-evaluated and tried again if the rewiring would still result in a shorter path.

CAS operations only work on single memory operands. The rewiring assertion however is made about two pieces of information: the trajectory and the cost of that trajectory. We thus modify

Algorithm 6 `PRRT*_Rewire`$(\tau, n_{\text{near}}, n_{\text{new}})$: conditionally rewires a near node through a newly created node, if doing so creates a short path

```
1: e_new ← n_new.edge
2: e_near ← n_near.edge
3: c_link ← COST(n_new.config, n_near.config)
4: c'_near ← e_new.cost + c_link
5: if c'_near ≥ e_near.cost or
       not FEASIBLE(n_new.config, n_near.config) then
6:     return
7: repeat
8:     e'_near ← (n_near, c'_near, n_new)
9:     — memory barrier —
10:    if CAS(n_near.edge, e_near, e'_near) then
11:        add e'_near to e_new.children
12:        PRRT*_Update(e'_near, e_near)
13:        if e_new is expired then
14:            PRRT*_Update(n_new.edge, e_new)
15:        remove e_near from e_near.parent.children
16:        return
17:    e_near ← n_near.edge
18: until c'_near ≥ e_near.cost
```

the data structures to encapsulate both trajectory and cost into a single unit making it suitable for a CAS. The data structures we define are *nodes*, representing reachable valid configurations, and *edges*, representing trajectories from one node to another. The edges form a linked tree structure that represents known trajectories to any nodes. To get from the initial configuration to any node's configuration, the edge structure is followed (in reverse) from the node back to the root of the tree where the initial configuration is stored. An edge's path to root never changes, and thus its computed trajectory cost never changes. When PRRT* finds a shorter path to a node, the node's *edge* is CAS with the better edge. Here again, we issue a memory barrier and ensure that the new edge is fully initialized before the CAS. The old edge will still essentially be present in the edge tree, but is no longer referenced from the node. We call an edge in this state "expired", and detect it when *edge.node.edge* $\neq$ *edge*. Expired edges can be garbage collected and their associated memory reused, but care must be taken to avoid the "ABA" problem [116]. (The ABA problem occurs when a thread reads 'A' from a shared memory location and, before it performs the CAS, another thread modifies the shared location to 'B' and back to 'A', which causes the first thread to treat the shared memory location as unmodified.)

Algorithm 7 PRRT*_Update($e_{\text{new}}, e_{\text{old}}$): Moves all the active children from a now expired parent edge to the new parent edge.

```
 1: n_parent ← e_new.parent
 2: done ← false
 3: repeat
 4:    e_child ← remove_first e_old.children
 5:    if e_child = ∅ then
 6:       if e_new is expired then
 7:          PRRT*_Update(e_new.node.edge, e_new)
 8:       done ← true
 9:    else if e_child is not expired then
10:       n_child ← e_child.node
11:       c'_child ← e_new.cost +COST(n_child, n_parent)
12:       if c'_child < e_child.cost then
13:          e'_child ← (n_child, c'_child, e_new)
14:          — memory barrier —
15:          if CAS(n_child.link, e_child, e'_child) then
16:             add e'_child to e_new.children
17:             PRRT*_Update(e'_child, e_child)
18: until done
```

By computing CAS operations around an edge, PRRT* guarantees that any update it makes results in an equal or better path, a requirement for the solution to converge towards optimality. After rewiring a node through a better path, the new shorter path is recursively percolated to the nodes that link in to the rewired node. This update process (Algorithm 7) atomically replaces edges to the expired parent with shorter ones. It repeatedly removes the old children one at a time (line 4) from a lock-free list structure [85, 116] until no more children remain (line 5). It then creates the new child edge with the updated cost, and CAS it into place (line 15). A memory barrier before the CAS ensures that the edge is fully initialized before another thread can access it. Note that by using the lock-free list removal, the algorithm ensures that only one thread is updating a particular child at any time. In the case in which two threads are competing to update the same portion of the tree, the thread(s) producing the longer update terminate early (lines 10, 13), and only the thread producing the shorter update proceeds, thus providing work savings and improving speedup.

2.4.3 Asymptotic Optimality of PRRT*

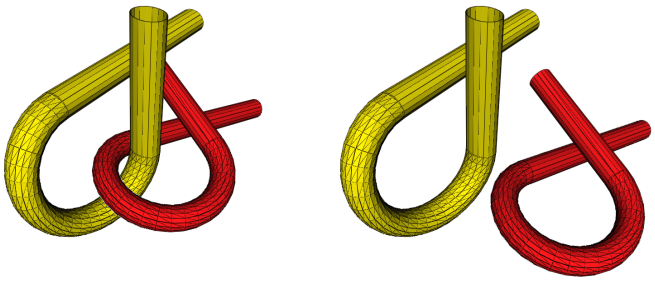
In the case of single-threaded execution, PRRT* runs exactly like sequential RRT* and hence is asymptotically optimal.

Next, let us consider PRRT* running with multiple threads and without partitioning. Each of the p threads is operating independently on a shared RRT* graph. Each thread begins its computation by observing the size n_t of the current graph and ends an iteration adding a configuration to the graph that is of size n'_t. When a single thread is running, $n'_t = n_t$. When multiple threads are running concurrently, $n'_t \geq n_t$ due to updates from other threads. Since the ball radius used in iteration t is based on n_t, as t increases and the ball radius shrinks, each thread is operating with a ball radius greater than or equal to what is necessary for asymptotic optimality according to the proofs from RRT* [60]. Thus it follows from the proof of asymptotic optimality of RRT* [60] that PRRT* when running without partitioning is asymptotically optimal.

Finally, let us consider PRRT* running with multiple threads and with partitioning. The impact of partitioning on the sampling distribution is that (1) PRRT* samples uniformly in independent static partitions rather than globally, and (2) each partition (due to the nature of the underlying planning problem) may sample at a different rate. If all threads sample their partition at the same rate, the sampling distribution of the entire space, in the limit, is uniform. We will denote this RRT* graph resulting from these samples at iteration t as G_t. If the sampling rate differs between threads, then we can consider G_t as the graph that results from running all the threads at the sampling rate of the slowest thread. Samples added by the threads with a faster sampling rate result in a graph G'_t that is a superset of G_t. The rewiring step of PRRT* guarantees that the quality of plans found on G'_t are at least as good as the plans found on G_t. If the ball radius of PRRT* is thus defined to guarantee asymptotic optimality of the slowest thread's partition, we guarantee asymptotic optimality of G_t as t increases. The graph G'_t, as a superset, is thus also guaranteed to be asymptotically optimal as t increases. Hence, PRRT* carries the same asymptotic optimality guarantee as RRT*.

2.5 Results

We evaluate our method with five scenarios: (1) PRRT on the Alpha Puzzle [121] scenario, (2) PRRT on a 10,000 random spheres scenario, (3) PRRT* on the OMPL [112] Cubicles scenario, (4) PRRT* on a holonomic disc-shaped robot moving in a planar environment, and (5) PRRT* on an SoftBank Nao [108] small humanoid robot performing a 2-handed task using 10 DOF. Results are computed on a system with four Intel x7550 2.0GHz 8-core Nehalem-EX processors for a total of 32



(a) starting configuration (b) goal configuration

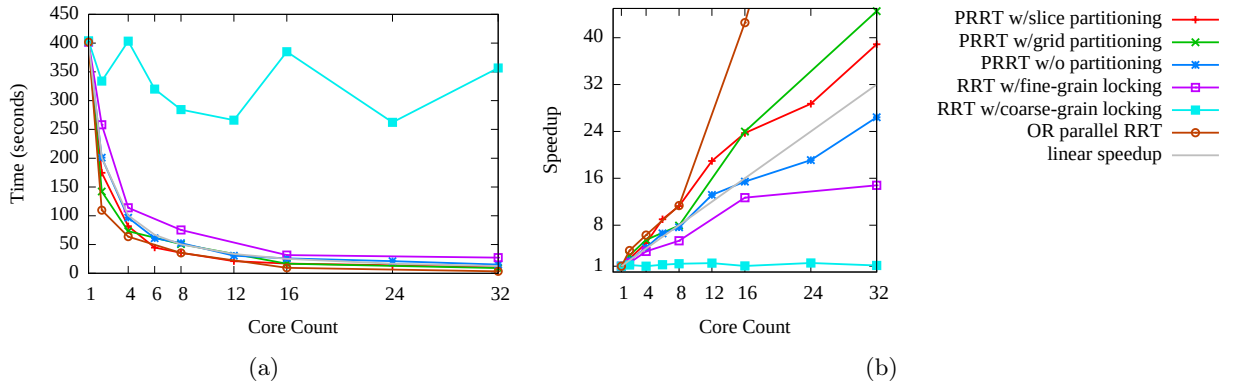
Figure 2.2: **The Alpha 1.2 scenario**. The yellow alpha is the obstacle, and the red alpha is the robot in SE(3). The robot must move from inside the obstacle (a) to outside the obstacle (b) by sliding through the narrow passage at an appropriate orientation.

cores. Each processor has an 18MB shared L3 cache and each core has a private 256KB L2 cache as well as 32KB L1 data and instruction caches.

2.5.1 PRRT on the Alpha Puzzle Scenario

The Alpha Puzzle scenario [121] is a motion planning problem containing a narrow passage in the configuration space. The puzzle consists of two tubes, each twisted into an alpha shape. The objective is to separate the intertwined tubes, where one tube is considered a stationary obstacle and the other tube is the moving object (robot), as shown in Fig. 2.2. We specifically use the Alpha 1.2 variant included in OMPL [112], where different variants scale the size of the narrow passage (with smaller numbers being more difficult to solve).

Using the Alpha 1.2 scenario, we evaluate PRRT's ability to speed up computation as the number of available CPU cores rises. We note that there has been much work on developing sampling strategies that improve RRT's ability to solve the Alpha Puzzle scenario quickly—we however use the standard uniform sampling (with and without partitioning) to demonstrate the multi-core performance of PRRT. As with other RRT variants, customized sampling strategies could be used with PRRT (with and without partitioning) to obtain results even more quickly. We evaluated PRRT for both slice and grid-based partitioning on different numbers of processor cores up to 32. For each core count, we ran 500 trials. We also consider PRRT with lock-free data structures but without partition-based sampling. We plot the median computation times and speedups in Fig. 2.3 (a) and (b), respectively. For comparison, we include results from multi-threaded locked variants of RRT in which each thread independently samples and computes feasibility, but the shared kd-tree is locked



(a) (b)

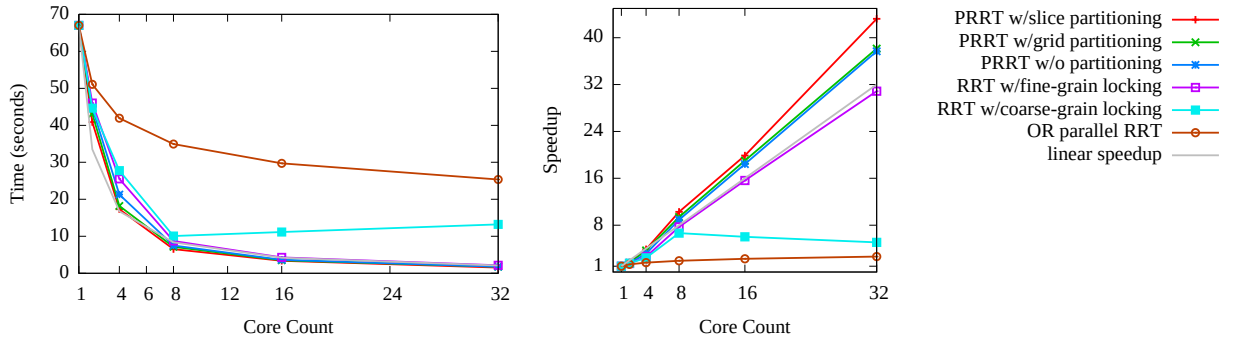
Figure 2.3: **Performance of PRRT and related methods run on the Alpha Puzzle scenario**. PRRT finds a solution with superlinear speedup with respect to the number of processor cores. PRRT without partition-based sampling finds solutions with a slightly sublinear speedup but good scalability. In contrast, RRT using a locked kd-tree does not scale as well. Coarse-grain locking causes too much lock-contention, and fine-grain avoids some lock-contention but adds the overhead of repeated locking. For this scenario, the multi-tree OR parallel RRT acheives greater speedups than accelerating the construction of a single tree.

either at the tree level ("coarse-grain locking") or at the node level ("fine-grain locking"). We also compare to the multi-tree OR-parallel RRT [19] in which each thread creates its own tree and all threads stop as soon any find a solution.

As shown in Fig. 2.3, PRRT achieves a superlinear speedup for the Alpha 1.2 scenario for all processor counts. PRRT's speedup for 32 cores was 39.4x. PRRT without partitioning achieves sublinear speedup, but due to the lock-free data structures still scales well as the number of cores rises. In contrast, RRT with a locked nearest neighbor data structure scales poorly; lock contention is very high due to the large number of configuration samples necessary to solve this motion planning problem. PRRT's use of lock-free data structures and partitioning enable a superlinear speedup for the Alpha 1.2 scenario on the multi-core computer. OR-parallel RRT performs best on this scenario, which requires creating samples inside a short, narrow passage. We hypothesize that the independence of the RRT's in OR parallel RRT facilitates landing the critical samples inside the short, narrow passage, and hence is better for this scenario than an approach that accelerates construction of a single RRT.

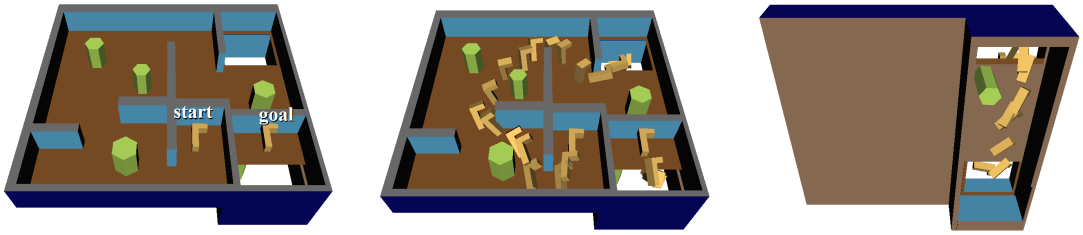
2.5.2 PRRT on 6-DOF, 10,000 Random Spheres

We apply PRRT and related methods to a random spheres scenario in which a holonomic spherical robot must navigate through an obstacle course of 10,000 randomly placed spheres in 6-dimensional



(a) (b)

Figure 2.4: **PRRT and related methods run on the 6-DOF random spheres scenario**. PRRT scales well with additional cores, which allow it to rapidly generate configuration samples and make progress towards the goal.

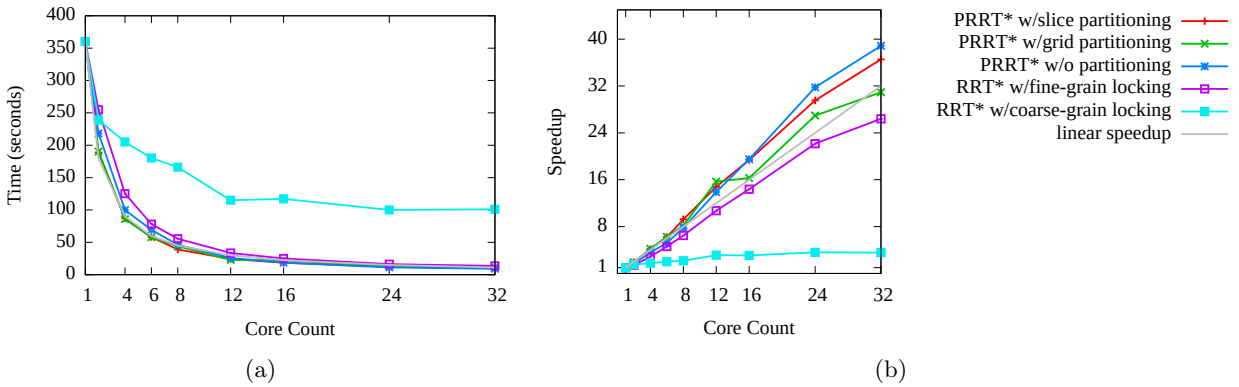


(a) (b) (c)

Figure 2.5: **PRRT* solves on the Cubicles scenario**. The "L"-shaped robot must move from its start pose on one side of a wall to the goal pose on the other side of the wall by moving through a lower floor (a). We illustrate an example path produced with 50,000 configurations (b, c).

C-space. The objective for the robot is to navigate from the center of the C-space to a corner while avoiding collision with the obstacles. The problem does not have a single difficult narrow passage like the Alpha problem, but the problem is still difficult because solutions necessarily have many segments.

In the random spheres scenario, OR parallel RRT does not perform as well as in the Alpha Puzzle scenario, likely because this scenario does not include a short, narrow passage requiring a "lucky" few samples to solve. In contrast, PRRT scales well with additional cores, which allow it to rapidly generate configuration samples and make progress towards the goal. The results are plotted in Fig. 2.4.



(a) (b)

Figure 2.6: **Performance of PRRT* and related methods run to 50,000 configurations on the Cubicles scenario**. PRRT* without partitioning and with slice partitioning both achieve superlinear speedups with respect to the number of processor cores. PRRT* with grid partitioning suffers in performance as some cores are confined to sampling inside partitions that are disconnected by obstacles from the start and goal. RRT* with a locked kd-tree nearest neighbor data structure scales poorly due to lock contention.

2.5.3 PRRT* on the Cubicles Scenario

The Cubicles scenario, included in OMPL [112], is a motion planning problem in which an "L"-shaped robot must move in SE(3) through a 2-story office-like environment. As shown in Fig. 2.5, to move from the start pose to the goal pose, the robot must find a path through SE(3) that includes traveling through a different floor. For computing path cost, we use OMPL's configuration space distance metric that sums the weighted spatial and orientation components. The objective is to compute a feasible path from the start pose to the goal pose that minimizes path cost.

Using the Cubicles scenario, we evaluate PRRT*'s ability to speed up computation as the number of available CPU cores rises. We evaluate PRRT* with and without partition-based sampling on different numbers of processor cores up to 32. For each core count, we ran 100 trials of each method, generating trees with 50,000 configurations in each trial. We plot the median computation times and speedups in Fig. 2.6(a) and (b), respectively. As with RRT, we compare against multi-threaded locked variants of RRT*. In the locked-RRT* fine-grain variant, access to the kd-tree and the rewiring updates of the tree are locked at the node (i.e., configuration) level—at most times multiple locks must be acquired to guarantee only one thread is updating a portion of the graph at any given moment, and locks are always acquired in the same order to avoid deadlock. We also compare against a multi-threaded OR-parallel RRT*, in which each thread computes an independent RRT* graph, and the final computed path is the one with the minimum cost selected from all graphs.

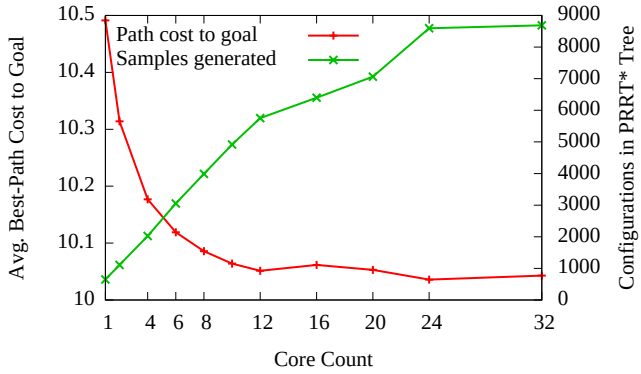


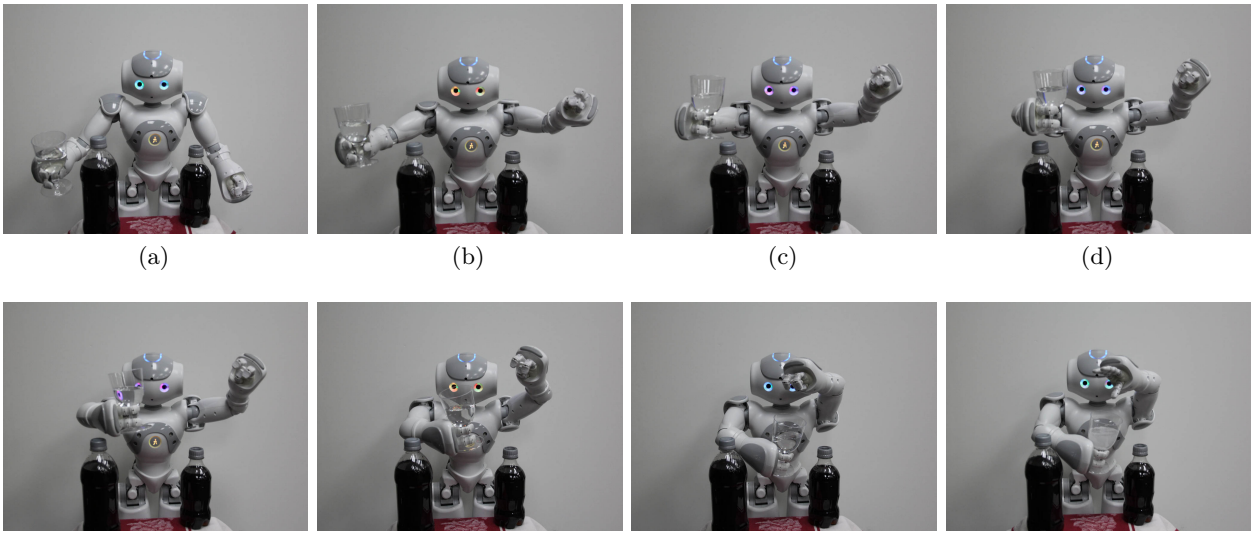
Figure 2.7: **PRRT* run for 10 ms on the 2D holonomic disc-shape robot scenario**. PRRT* generates more samples, and produces a better quality solution with more cores, even in this short time interval.

PRRT* with slice partitioning and PRRT* without partitioning achieve superlinear speedup on the Cubicles scenario. On 32 cores, PRRT* with slice partitioning achieves a speedup of 36.6x and PRRT* without partitioning achieves a speedup of 38.9x. All methods achieved median solution path costs that are within 1% of one another, indicating that parallelization and partitioning do not significantly affect path quality when the size of the tree (50,000 configurations in this case) is held constant. In this scenario, PRRT* with grid partitioning does not perform as well as other PRRT* variants because some of the threads sample in partitions that are unreachable (i.e., the space on the left of Fig. 2.5(c)) from the start and goal configurations. At 32 cores, grid partitioning allocates 8 cores to partitions entirely in the unreachable space. PRRT* performs substantially better than RRT* with a locked kd-tree for nearest neighbor searching, which achieved sublinear speedup for both fine and coarse grain locking due to lock overhead and contention.

2.5.4 PRRT* for a 2D Holonomic Disc-shaped Robot

We executed PRRT* for a 2D holonomic disc-shaped robot that must move to the goal in the environment shown in Fig. 2.1(a). We executed RRT* on one core and PRRT* on 4 and 32 cores for 10 ms of wall clock time. The quality of paths is shown visually in Fig. 2.1 and quantitatively in Fig. 2.7. With more cores, the size of the constructed tree in the 10 ms increases substantially, visibly improving the quality of the computed motion plan. More space is explored and more narrow passages are discovered.

As stated in section 2, the focus of PRRT and PRRT* is on challenging scenarios requiring tens or hundreds of thousands of samples, and this 10 ms scenario does not fall into that category. In



(a) (b) (c) (d)

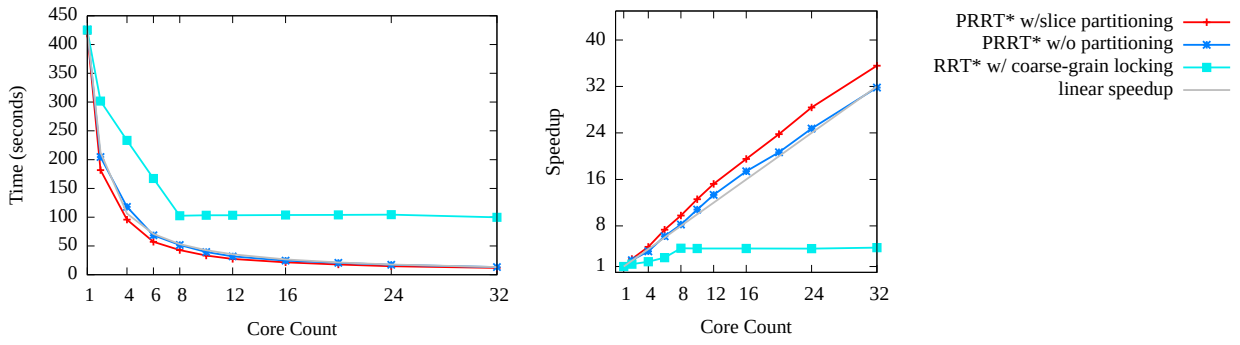
(e) (f) (g) (h)

Figure 2.8: **An example PRRT* motion plan created for the Aldebaran Nao robot**. The robot carries an effervescent antacid in one hand and places it over a glass of water held in the other hand, all while avoiding the bottles on the table and not spilling the water (i.e. `FEASIBLE` is constrained to keep the glass mostly level). In the last frame, after the robot reaches the goal configuration, it drops the antacid into the water.

Fig. 2.7, we see that as we add more cores above 12, PRRT* begins to show a diminishing return on samples generated and quality of solution due to several factors: (1) the PRRT* tree grows faster thus causing the per-query time for nearest neighbor to also increase, (2) PRRT* is rapidly converging towards the optimal solution, and (3) 10 ms is a short enough interval that we observe the overhead of startup. In the early growth of the roadmap, where the number of samples n is small, as we add more cores p, the expected contention rises ($\lim_{p\to\infty} O(p/n) = \infty$). As we show in Sec. 2.5.5, the PRRT* startup overhead quickly disappears with additional computation time. We also note that this 10 ms scenario performs well for current readily available multi-core systems (typically in the range of 2–12 cores), producing the significant and visible improvements shown in Fig. 2.1.

2.5.5 PRRT* for a 2-handed SoftBank Nao 10 DOF Task

We evaluated PRRT* on an SoftBank Nao small humanoid robot [108] with the task of dropping an object held in one hand into a cup held in the other hand while avoiding obstacles. Each arm of the Nao robot has 5 degrees of freedom (shoulder pitch/roll, elbow yaw/roll and wrist yaw), resulting in a 10 dimensional configuration space for this problem. All joints are bounded revolute joints, and we define `COST` as a Euclidean distance in configuration space. The robot must avoid obstacles on the table in front of it while keeping the cup upright throughout its motion—i.e. the function



(a) (b)

Figure 2.9: **Performance of PRRT* and related methods run on the Nao 10 DOF task for 100,000 configurations**. PRRT* achieves superlinear speedups with respect to the number of processor cores. In contrast, RRT* with a course-grain locked kd-tree nearest neighbor data structure cannot exceed 4x speedup due to lock contention.

`FEASIBLE` tests if the robot will collide with objects in the environment and also tests if the robot's joint angles will result in the cup being upright subject to a tolerance. We define `GOAL` to return true for configurations that satisfy the following constraints within a tolerance: (1) the (x, y) coordinates for the left hand and the right hand are the same, (2) the left hand's z coordinate is higher than the right hand, (3) the object in the left hand is pointing down, and (4) the cup in the right hand is held upright. We show the Nao robot using PRRT* successfully performing the task in Fig. 2.8.

To demonstrate PRRT*'s ability to compute high quality solutions faster on multiple cores, we executed the Nao 10 DOF task for $n = 100,000$ configurations with varying core counts and averaging over 10 runs. As shown in Fig. 2.9, we observe superlinear speedup with PRRT*. Executing PRRT* on one core (thus making it equivalent to standard RRT*) requires 420 seconds. On 32 cores, PRRT* required only 11.6 seconds for the same number of samples. PRRT* was 36x faster with no significant difference in the quality of the computed paths.

The use of lock-free data structures and partitioning in PRRT* both have an impact on performance. PRRT* without partition-based sampling performed slightly worse than PRRT*, achieving approximately a linear speedup as shown in Fig. 2.9. We also executed RRT* parallelized by locking the kd-tree. At 100,000 configurations, nearest neighbor searches dominate the computation time, so threads spend most of their time waiting for access to the kd-tree when using locks. Consequently, the lock-based approach cannot exceed 4x speedup.

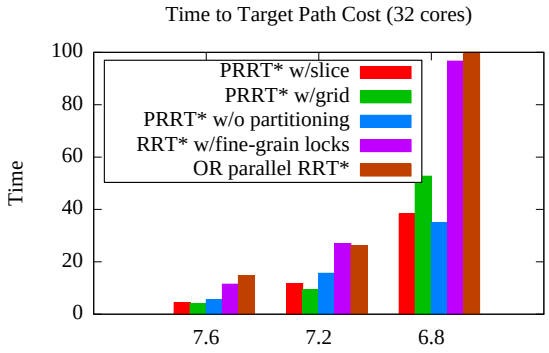


Figure 2.10: We give PRRT* and RRT* variants a specified target path cost and show the time it takes to reach the target in the Nao scenario. In this graph we also include OR parallel RRT*, a multi-tree RRT* in which 32 RRT* trees are built in parallel and the best result is chosen from among them. For target path cost 6.8, OR parallel RRT* exceeded the allotted time and is plotted only to 100 seconds. We do not include the coarse-grained locking in this graph—in all cases it exceeded the allotted time.

We note that the relative performance of motion planning using lock-free and lock-based nearest neighbor searching varies with the size of the motion planning tree τ. When the size of the tree τ is smaller, collision-detection dominates computation time and the lock-based approach achieves a more reasonable speedup. At 2,000 samples on 32 cores, we observe a 16.4x speedup with locked kd-trees, although PRRT* still outperforms with a 28.9x speedup. The locked version's speedup diminishes as more samples are added, as shown in Fig. 2.11. In contrast, the lock-free PRRT* overcomes thread startup overhead and reaches 32x speedup by the 20,000th configuration before increasing to 36x speedup by 100,000 configurations.

To demonstrate how PRRT* can be used to produce better results per unit time, we also ran the Nao 10 DOF task 50 times for 3 seconds at various processor core counts. As shown in Fig. 2.12, increasing the number of processor cores enables us to build trees with more samples per second and find better solutions. The path cost from the initial configuration to the goal shows convergence to an optimal solution as the number of samples increases, as expected with RRT*. In contrast to the 10 ms runs for the holonomic disc-shaped robot, in these 3-second runs for the Nao robot the impact of startup overhead is no longer significant and we see the number of samples generated scale well with the number of cores. We also observed that RRT* would find paths to the goal in only 80% of the 3-second runs on one core. With two cores, PRRT* found solutions in 98% of the runs. At higher core counts, PRRT* found solutions in all runs.

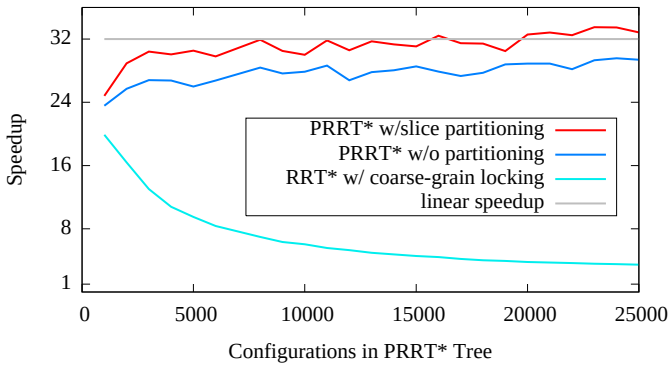


Figure 2.11: PRRT* running on 32 cores overcomes startup overhead and speedup increases as the number of configurations increases. In contrast, using a locked nearest neighbor structure shows good speedup initially, but as the number of configurations increases, contention over locked data structures slows the algorithm down.

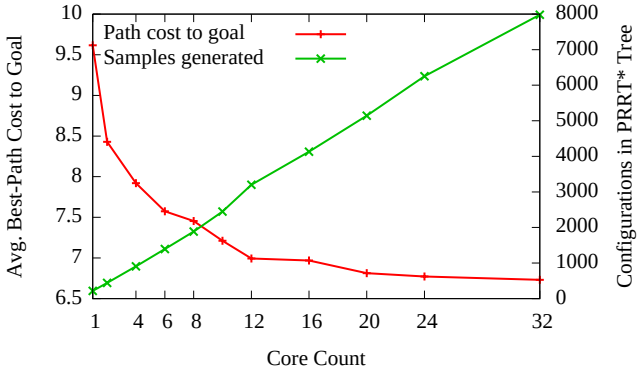
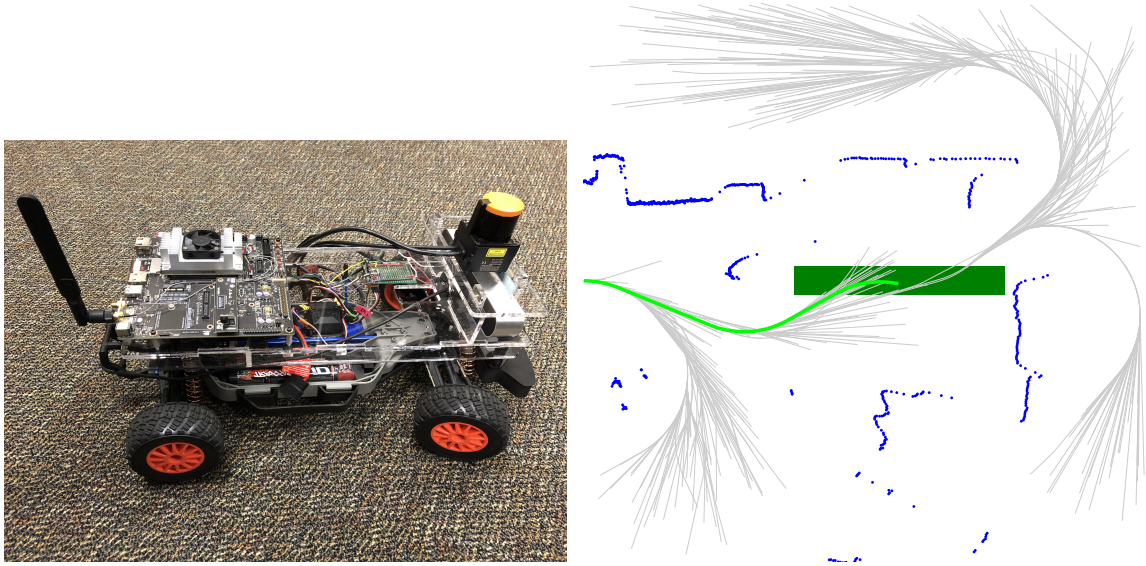


Figure 2.12: **PRRT* run for 3 seconds on the Nao 10 DOF task**. Increasing the number of processor cores results in samples being generated at a higher rate and better quality solutions.

2.5.6 PRRT* for 1/10 Scale Self-Driving Car

To demonstrate the ability of PRRT* to compute motion plans for additional robot types, we have the 4-core processor on a 1/10 scale self-driving car (Fig. 2.13 (a)) repeatedly compute a path around obstacles sensed using its LIDAR. Motion planning in this demonstration connects configurations in the tree using a Dubins path [26]. Dubins paths are forward-only paths with constraints on curvature (i.e., turning radius) which can be visualized in the tree and the path that the planner generates as shown in Fig. 2.13 (b). Due to the constraints on the path, it is non-trivial to compute a space-partitioning nearest neighbor data structure, so we modify nearest neighbor searching to use a lock-free linked list [116]. This results in an $O(n)$ nearest neighbor search, which



(a) 1/10 scale car (b) PRRT* path

Figure 2.13: **PRRT* planning for 1/10 scale self-driving car**. A 1/10 scale self-driving car (a) uses PRRT* to plan a path around an obstacle (in blue) detected by LIDAR. The car starts on the left side and has a goal of entering the green rectangle to its right. The tree it computes (b) is shown in grey, while the best path it found (and thus will follow), is shown in light green. The onboard computer, an NVIDIA Jetson TX1, has a 4-core ARM-based processor.

we found acceptable in this case due to the short time we spent planning, and the ability to scale with the parallelism of the onboard multi-core processor.

2.6 Conclusion

In this chapter, we presented PRRT (Parallel RRT) and PRRT* (Parallel RRT*), single-tree sampling-based methods for feasible and optimal motion planning that are tailored to execute on modern multi-core CPUs. Using atomic updates and lock-free data structures, PRRT and PRRT* remove barriers to scaling to higher processor core counts. We further observe that using a non-overlapping partition-based sampling strategy increases cache efficiency by localizing each thread's computation to a region of memory. While not guaranteed, we empirically observed that these contributions enable PRRT and PRRT* in some scenarios to achieve superlinear speedup.

Our method is best suited for challenging motion planning problems in which a large number of samples is required to find a feasible or near optimal solution. As the number of samples increases, computation time gradually changes from being dominated by collision detection to being dominated by nearest neighbor search. PRRT and PRRT* parallelize the entire computation of the motion planning tree and thus maintain speedup ratios regardless of which portion of the computation is

dominating. We demonstrated fast performance and significant speedups in 5 scenarios including the Alpha Puzzle and Cubicles scenarios and an SoftBank Nao small humanoid robot performing a two-handed, 10 DOF task.

The methods in this chapter demonstrate scalable parallelism that should be applicable to a variety of sampling-based motion planners. However it is possible to do better with methods outlined in subsequent chapters. Planning for rigid-body motions will benefit from a faster nearest neighbor searching strategy that is presented in chapter 3. Additionally, while the kd-tree presented here ensures lock-free operation, it is optimized for fast inserts that come at the expense of reduced query performance—a problem that the concurrent nearest neighbor data-structure presented in chapter 4 addresses. Finally, the static partitioning in this chapter, while having an impact on many real-world level problems, does not produces a sustainable cache-locality in the limit. Eventually, the cache-benefit of the static partitioned locality will run out. Other work in the field of cache-aware and cache-oblivious algorithms (e.g., [37, 27]) has shown how to create a sustained cache-based performance improvement, regardless of problem size, and this concern is addressed in chapter 5.

CHAPTER 3

Fast Nearest Neighbor Searching in SO(3) and SE(3)

Nearest neighbor searching is a critical component of commonly used motion planners. Sampling-based methods, such as probabilistic roadmaps (PRM), rapidly exploring random trees (RRT), and RRT* [20, 60], create a motion plan by building a graph in which vertices represent collision-free robot configurations and edges represent motions between configurations. To build the graph, these motion planners repeatedly sample robot configurations and search for nearest neighbor configurations already in the graph to identify promising collision-free motions.

Because nearest neighbor search is a fundamental building block of most sampling-based motion planners, speeding up nearest neighbor searching will accelerate many commonly used planners. This is especially true for asymptotically optimal motion planners, which typically require a large number of samples to compute high-quality plans. As the number of samples in the motion planning graph rises, nearest neighbor search time grows logarithmically (or at worst linearly). As the samples fill the space, the expected distance between samples shrinks, and correspondingly reduces the time required for collision detection. Collision detection typically dominates computation time in early iterations, but as the number of iterations rises, nearest neighbor search will dominate the overall computation—increasing the importance of fast nearest neighbor searches.

In this chapter, we introduce a fast, scalable exact nearest neighbor search method for robots modeled as rigid bodies. Many motion planning problems involve rigid bodies, from the classic piano mover problem to planning for aerial vehicles. A planner can represent the configuration of a rigid body in 3D by its 6 degrees of freedom: three translational (e.g., x, y, z) and three rotational (e.g., yaw, pitch, roll). The group of all rotations in 3D Euclidean space is the special orthogonal group SO(3). The combination of SO(3) with Euclidean translation in space is the special Euclidean group SE(3).

Our approach uses a set of kd-trees specialized for nearest neighbor searches in SO(3) and SE(3) for dynamic data sets. A kd-tree is a binary space partitioning tree data structure that successively

splits space by axis-aligned planes. It is particularly well suited for nearest neighbor searches in Minkowski distance (e.g., Euclidean) real-vector spaces. However, standard axis-aligned partitioning approaches that apply to real-vector spaces do not directly apply to rotational spaces due to their curved and wrap-around nature.

The primary contribution of this chapter is the novel way of partitioning SO(3) space to create a kd-tree search structure for SO(3) and by extension SE(3). This chapter's contribution, and its evaluation, are single-threaded—chapter 4 makes use of this chapter's contribution to allow for concurrent and faster nearest neighbor searching operations with novel modifications to the kd-tree data structure. Our SO(3) partitioning approach can be viewed as projecting the surface of a 4-dimensional cube onto a 3-sphere (the surface of a 4-dimensional sphere), and subsequently partitioning the projected faces of the cube. The 3-sphere arises from representing rotations as unit quaternions which in turn can be represented as 4-dimensional vectors. The projection and partitioning we describe has two important benefits: (1) the dimensionality of the rotation space is reduced from its 4-dimensional quaternion representation to 3 (its actual degrees of freedom), and (2) the splitting hyperplanes efficiently partition space allowing the kd-tree search to check fewer kd-tree nodes. We propose efficient methods to handle the recursion pruning checks that arise with this kd-tree splitting approach, and also discuss splitting strategies that support dynamic data sets. Our approach for creating rotational splits enables our kd-tree implementation to achieve fast nearest neighbor search times for dynamic data sets.

We demonstrate the speed of our nearest neighbor search approach on scenarios in OMPL [112] and demonstrate a significant speedup compared to state-of-the-art nearest neighbor search methods for SO(3) and SE(3).

3.1 Related Work

Nearest neighbor searching is a critical component in sampling-based motion planners [20]. These planners use nearest neighbor search data structures to find and connect configurations in order to compute a motion plan.

Spatial partitioning trees such as the kd-tree [13, 36, 109], quadtrees and higher dimensional variants [33], and vp-trees [124] can efficiently handle exact nearest neighbor searching in lower dimensions. These structures generally perform well on data in a Euclidean metric space, but because

of their partitioning mechanism (e.g., axis-aligned splits), they do not readily adapt to the rotational group SO(3). Kd-trees have a static construction that can guarantee a perfectly balanced tree for a fixed (non-dynamic) data set. Bentley showed how to do a static-to-dynamic conversion [14] that maintains the benefits of the balanced structure produced by static construction, while adding the ability to dynamically update the structure without significant loss of asymptotic performance.

Yershova and LaValle [123] showed how to extend kd-trees to handle $\mathbb{R}^n$, S^1, SO(3), and the Cartesian product of any number of these spaces. Similar to kd-trees built for $\mathbb{R}^n$, they split SO(3) using rectilinear axis-aligned planes created by a quaternion representation of the rotations [105]. Although performing well in many cases, rectilinear splits produce inefficient partitions of SO(3) near the corners of the partitions. Our method eschews rectilinear splits in favor of splits along rotational axes, resulting in splits that more uniformly partition SO(3).

Non-Euclidean spaces, including SO(3), can be searched by general metric space nearest neighbor search data structures such as GNAT [17], cover-trees [15], and M-trees [21]. These data structures generally perform better than linear searching. However, except for rare pathological cases, these methods are usually outperformed by kd-trees in practice [123].

Nearest neighbor searching is often a performance bottleneck of sampling-based motion planning, particularly when the dimensionality of the space increases [53, 97]. It is sometimes desirable in such cases to sacrifice accuracy for speed by using approximate methods [53, 97, 10, 72, 89]. These methods can dramatically reduce computation time for nearest neighbor searches, but it is unclear if the proofs of optimality for asymptotically optimal motion planners hold when using approximate searches. Our focus is on exact searches, though we believe that some approximate kd-tree speedups can be applied to our method.

3.2 Problem Definition

Let $\mathcal{C}$ be the configuration space of the robot. For a rigid-body robot, the configuration space is $\mathcal{C} = \mathbb{R}^m$ if the robot can translate in m dimensions, $\mathcal{C} = \mathrm{SO}(3) = P^3$ if the robot can freely rotate in 3 dimensions, and $\mathcal{C} = \mathrm{SE}(3) = \mathbb{R}^3 P^3$ if the robot can freely translate and rotate in 3 dimensions. Let $\mathbf{q} \in \mathcal{C}$ denote a configuration of the robot. When $\mathcal{C} = \mathbb{R}^m$, $\mathbf{q}$ is an m-dimensional real vector. When $\mathcal{C} = P^3$, we define $\mathbf{q}$ as a 4-dimensional real vector in the form $[a\ b\ c\ d]^{\mathrm{T}}$ representing the components of a unit quaternion $\mathbf{q} = a + b\mathbf{i} + c\mathbf{j} + d\mathbf{k}$, where $\mathbf{i}$, $\mathbf{j}$, and $\mathbf{k}$ are the fundamental quaternion units, and $\|\mathbf{q}\| = 1$. The unit quaternion representation of rotations have a *double-coverage* property [105] in

which $\mathbf{q}$ and $-\mathbf{q}$ represent the same rotation. We use the notation $\mathbf{q}[x]$ to represent the x component of a configuration $\mathbf{q}$.

Computation of nearest neighbors depends on the chosen distance metric. Let $\texttt{DIST}(\mathbf{q}_1, \mathbf{q}_2)$ be the distance between two configurations. We will focus on distance functions commonly used in robot motion planning software (e.g., OMPL [112]). In $\mathbb{R}^m$ we use the Minkowski distance of order $p = 2$:

$$\texttt{DIST}_{\mathbb{R}^m}(\mathbf{q}_1, \mathbf{q}_2) = \left(\sum_{i=1}^{m} |\mathbf{q}_1[i] - \mathbf{q}_2[i]|^p \right)^{1/p},$$

which is the L^2 or Euclidean distance. In P^3 we use a distance of the shorter of the two angles subtended along the great arc between the rotations [105, 112, 123]. This metric is akin to a straight-line distance in Euclidean space mapped on a 3-sphere:

$$\texttt{DIST}_{P^3}(\mathbf{q}_1, \mathbf{q}_2) = \arccos |\mathbf{q}_1 \cdot \mathbf{q}_2| = \arccos \left| \sum_{i \in \{a,b,c,d\}} \mathbf{q}_1[i]\mathbf{q}_2[i] \right|.$$

In $\mathbb{R}^3P^3$, we use the weighted sum of the $\mathbb{R}^3$ and P^3 distances [112]:

$$\texttt{DIST}_{\mathbb{R}^m P^3}(\mathbf{q}_1, \mathbf{q}_2) = \alpha \, \texttt{DIST}_{\mathbb{R}^m}(\mathbf{q}_1, \mathbf{q}_2) + \texttt{DIST}_{P^3}(\mathbf{q}_1, \mathbf{q}_2).$$

where $\alpha > 0$ is a user-specified weighting factor. As defined, the distance function is symmetric, i.e., $\texttt{DIST}(\mathbf{q}_1, \mathbf{q}_2) = \texttt{DIST}(\mathbf{q}_2, \mathbf{q}_1)$. We also define $\texttt{DIST}(\mathbf{q}, \varnothing) = \infty$.

We apply our approach to solve three variants of the nearest neighbor search problem commonly used in sampling-based motion planning. Let $\mathbf{Q}$ denote a set of n configurations $\{\mathbf{q}_1 \ldots \mathbf{q}_n\} \subset \mathcal{C}$. Given a configuration $\mathbf{q}_{\text{search}}$, the *nearest neighbor search* problem is to find the $\mathbf{q}_i \in \mathbf{Q}$ with the minimum $\texttt{DIST}(\mathbf{q}_{\text{search}}, \mathbf{q}_i)$. In the *k-nearest neighbors* variant, where k is a positive integer, the objective is to find a set of k configurations in $\mathbf{Q}$ nearest to $\mathbf{q}_{\text{search}}$. In the *nearest neighbors in radius r* search, where r is a positive real number, the objective is to find all configurations in $\mathbf{Q}$ with $\texttt{DIST}(\mathbf{q}_{\text{search}}, \mathbf{q}_i) \leq r$.

Sampling-based motion planners make many calls to the above functions when computing a motion plan. Depending on the planner, the set of nodes $\mathbf{Q}$ is either a static data set that is constant for each query or $\mathbf{Q}$ is a dynamic data set that changes between queries. Our objective is to achieve

efficiency and scalability for all the above variants of the nearest neighbor search problem for static and dynamic data sets in SO(3) and SE(3).

3.3 Method

Our method enables fast nearest neighbor searching by partitioning samples into a kd-tree-like data structure. A kd-tree is a binary tree in which each branch node recursively subdivides space by an axis-aligned hyperplane, and each child's subtree contains only configurations from one side of the hyperplane. The recursive subdivision speeds up nearest neighbor searching enabling the search algorithm to test a small portion of the entire set of samples. In a real vector metric space, such as Euclidean space, it is common for each split to be defined by an axis-aligned hyperplane, though other formulations are possible [109]. For performance reasons it is often desirable for the splits to evenly partition the space, making median or mean splits good choices [88]. We will describe these methods and how to apply our SO(3) partition scheme to them.

In our method, we eschew rectilinear axis-aligned splits in favor of partitions that curve with the manifold of SO(3) space. The set of all unit quaternion representations of rotations in SO(3) forms the surface of a 4-dimensional sphere (a 3-sphere). We create top-level partitions of this 3-sphere by projecting the 8 faces of a 4-dimensional cube onto the 3-sphere. Because of the double-coverage property, half of the top-level projected surface partitions are redundant, and thus we only need 4 top-level partitions. After creating the top-level partitions, we build four kd-trees (one for each projected face) by recursively subdividing the top-level projected surface partitions. Similar projections are used in [122] to generate deterministic samples in SO(3), and in [91] to create a minimum spanning tree on a recursive octree subdivision of SO(3). When subdividing a top-level surface partition into a kd-tree, we apply a novel approach in which the partitioning hyperplanes pass through the center of the 3-sphere, and thus radially divide space. These partitions are curved, and thus standard kd-tree approaches that apply to real-vector spaces must be adapted to maintain consistency with the great arc distance metric we use for SO(3). In Fig. 3.1, we depict a lower dimensional analog consisting of the faces of a 3-dimensional cube projected onto a 2-sphere, with only one of the projected cube faces subdivided into a kd-tree.

3.3.1 Projected Partitioning of SO(3)

The top-level partitioning on the 3-sphere, requires four top-level partitions, and provides two advantages: (1) we reduce the dimensionality of the rotation representation from a 4-dimensional

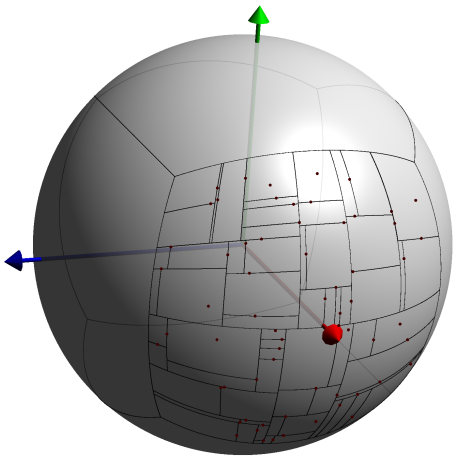


Figure 3.1: **Kd-tree projected onto the surface of a 2-sphere**. An axis-orthogonal cube is projected into a sphere. Each face of the cube is a separately computed kd-tree; however, for illustrative purposes, we show the kd-tree of only one of the faces. In our method we extend the analogy to 4-dimensional ambient space for use with quaternions.

quaternion to a 3-dimensional position on the top-level projected partition, and (2) it allows radially aligned splitting hyperplanes that follow the curve with the manifold. There is, however, a small cost for these benefits. The projection leads to building four kd-trees, although asymptotically the cost is at worst a constant factor.

In the projection of the surface of a 4D cube onto the surface of a 3-sphere we label each of the top-level projected 3D surface partitions by their projected axis of greatest magnitude, thus $+a$, $+b$, $+c$, and $+d$. In the lower dimensional analog in Fig. 3.1, these volumes coorespond to the red, green, and blue axes. The double-coverage property of quaternions means that the negative top-level partitions (i.e., $-a$, $-b$, $-c$, $-d$) are redundant. We negate any configuration whose quaternion is in a negative top-level partition, and thus place the quaternion in the positive top-level partition.

To determine in which top-level projected partition a quaternion $\mathbf{q}$ lies, we find its component of greatest absolute magnitude. Thus:

$$\texttt{top\_level\_partition}\,(\mathbf{q}) = \underset{i \in a,b,c,d}{\operatorname{argmax}} \left|\mathbf{q}[i]\right| .$$

In order to build a kd-tree within each top-level partition, the space needs to be recursively subdivided. As with rectilinear kd-trees on Minkowski spaces, we use a partitioning hyperplane to define the recursive split. Unlike rectilinear kd-trees, the hyperplane we use for SO(3), passes through

the origin, and can thus be defined by the hyperplane's normal (instead of offset). To understand how to create and use these splitting hyperplanes, we first look at how to compute the angle between a unit quaternion and a hyperplane normal. If θ is the angle between the unit quaternions $\mathbf{q}$ and $\mathbf{n}$, then $\mathbf{q} \cdot \mathbf{n} = \cos\theta$. This property allows us to represent bounding and splitting hyperplanes by their normals $\mathbf{n}$. Determining on which side a quaternion $\mathbf{q}$ lies is a matter of evaluating the sign of the dot product—positive values are on one side, negative values are on the other, and a dot product of 0 lies on the hyperplane.

We will focus our discussion on the top-level projected $+a$-partition, with the other top-level partitions ($+b$, $+c$, and $+d$) being permutations on it. The normals bounding the 6 sides of the projected a-partition are the unit quaternions:

$$\begin{array}{lll} \sqrt{0.5}\,[1\ 1\ 0\ 0]^{\mathrm{T}} & \sqrt{0.5}\,[-1\ 1\ 0\ 0]^{\mathrm{T}} & +b\text{-axis bounds} \\ \sqrt{0.5}\,[1\ 0\ 1\ 0]^{\mathrm{T}} & \sqrt{0.5}\,[-1\ 0\ 1\ 0]^{\mathrm{T}} & +c\text{-axis bounds} \\ \sqrt{0.5}\,[1\ 0\ 0\ 1]^{\mathrm{T}} & \sqrt{0.5}\,[-1\ 0\ 0\ 1]^{\mathrm{T}} & +d\text{-axis bounds} \end{array}$$

We observe that within the projected $+a$-partition, the a component of the hyperplane normals varies between $\sqrt{0.5}$ and $-\sqrt{0.5}$, the axis component varies between $\sqrt{0.5}$ at the boundaries to 1 at $a = 0$, and the other components are always zero. The bounds for the b, c, and d projected partition follow similarly.

In order to define a splitting hyperplane, it can be useful to determine the normal of the hyperplane that passes through a quaternion in the set we are subdividing. To determine the normal, we solve for $\mathbf{n}$ in $\mathbf{q} \cdot \mathbf{n} = 0$. We define $\texttt{axisnorm}_{\text{vol,axis}}(\mathbf{q})$ as the axis-aligned normal within a top-level projected partition for quaternion $\mathbf{q}$. The $+a$-partition definitions for `axisnorm` are:

$$\begin{aligned} \texttt{axisnorm}_{a\text{-vol},b\text{-axis}}(\mathbf{q}) &= \texttt{normalize}(-\mathbf{q}[b], \mathbf{q}[a], 0, 0) \\ \texttt{axisnorm}_{a\text{-vol},c\text{-axis}}(\mathbf{q}) &= \texttt{normalize}(-\mathbf{q}[c], 0, \mathbf{q}[a], 0) \\ \texttt{axisnorm}_{a\text{-vol},d\text{-axis}}(\mathbf{q}) &= \texttt{normalize}(-\mathbf{q}[d], 0, 0, \mathbf{q}[a]), \end{aligned}$$

where $\texttt{normalize}(\mathbf{q})$ normalizes its input vector to a unit quaternion. From the `axisnorm` we also gain the useful property of being able to define an angle of rotation about the axis. This angle forms the basis for a relative ordering around the axis, which we will use later to select median partitioning

Algorithm 8 `BuildKDTreeQ`

Require: $\mathbf{Q}$ is a set of configurations of size $\mathbf{n} > 0$

1: **if** $\mathbf{Q}$ has 1 configuration **then**
2: **return** leaf node with $\mathbf{Q}_1$
3: **else**
4: $\texttt{axis} \leftarrow \texttt{CHOOSE\_PARTITION\_AXIS}(\mathbf{Q})$
5: $(\mathbf{Q}', \texttt{split}, \texttt{m}) \leftarrow \texttt{PARTITION}(\mathbf{Q}, \texttt{axis})$
6: $\texttt{left} \leftarrow \texttt{BuildKDTree}\left(\mathbf{Q}'_{1..m-1}\right)$
7: $\texttt{right} \leftarrow \texttt{BuildKDTree}(\mathbf{Q}'_{m..n})$
8: **return** branch node with split on $(\texttt{axis}, \texttt{split})$ and children $(\texttt{left}, \texttt{right})$

hyperplanes. The angle about the axis is the arctangent of the normal's partition component over the normal's axis component, thus for example, $\mathbf{q}$'s angle about the b-axis in the $+a$-partition is $\tan^{-1}(-\mathbf{q}[a]/\mathbf{q}[b])$. When computing relative ordering of rotation about and axis, the exact angle is unimportant, and we can shortcut the trigonometric computation by comparing the partition component alone, as follows:

$$\mathbf{q}_1[a] < \mathbf{q}_2[a] \iff \tan^{-1}(-\mathbf{q}_1[a]/\mathbf{q}_1[b]) > \tan^{-1}(-\mathbf{q}_2[a]/\mathbf{q}_2[b]).$$

3.3.2 Static KD-Tree

In a static nearest neighbor problem, in which $\mathbf{Q}$ does not change, we can use an efficient one-time kd-tree construction that allows for well-balanced trees. Alg. 8 outlines a static construction method for kd-trees on real-vector spaces. The algorithm works as follows. First it checks if there is only one configuration, and if so it returns a leaf node with the single configuration (lines 1–2). Otherwise the set of configurations is partitioned into two subsets to create a branch. $\texttt{CHOOSE\_PARTITION\_AXIS}(\mathbf{Q})$ in line 4 chooses the axis of the partition. A number of policies for choosing the axis are possible, e.g., splitting along the axis of greatest extent. Then, $\texttt{PARTITION}(\mathbf{Q}, \texttt{axis})$ (line 5) splits $\mathbf{Q}$ along the $\texttt{axis}$ into the partially ordered set $\mathbf{Q}'$ such that $\forall \mathbf{q}_i \in \mathbf{Q}'_{1..m-1} : \mathbf{q}_i[\texttt{axis}] \leq \texttt{split}$ and $\forall \mathbf{q}_j \in \mathbf{Q}'_{m..n} : \mathbf{q}_j[\texttt{axis}] \geq \texttt{split}$. Thus a median split chooses $m = n/2$ and creates a balanced tree.

The `PARTITION` function is implemented efficiently either by using a partial-sort algorithm, or sorting along each axis before building the tree. Assuming median splits, `BuildKDTree` builds a kd-tree in $O(n \log n)$ time using a partial-sort algorithm.

In our SO(3) projection, we define an axis comparison that allows us to find the minimum and maximum along each projected axis, and to perform the partial sort required for a median partition.

The axis comparison is the relative ordering of each quaternion's `axisnorm` angle for that partition and projection.

The minimum and maximum extent along each axis is the quaternion for which all others are not-less-than or not-greater-than, respectively, any other quaternion in the set, according to the ordering of `axisnorm`. The angle of the arc subtending the minimum and maximum `axisnorm` values is the axis's extent. Thus, if we define $\mathbf{N}$ as the set of all `axisnorm` values for $\mathbf{Q}$ in the $+a$-partition and along the b-axis therein: $\mathbf{N}_{a,b} = \{\texttt{axisnorm}_{a\text{-vol},b\text{-axis}}(\mathbf{q}) : \mathbf{q} \in \mathbf{Q}\}$, then the minimum and maximum `axisnorm` along the b-axis is:

$$\mathbf{n}_{\min} = \operatorname*{argmin}_{\mathbf{n}_i \in \mathbf{N}_{a,b}} \mathbf{n}_i[a] \qquad \mathbf{n}_{\max} = \operatorname*{argmax}_{\mathbf{n}_j \in \mathbf{N}_{a,b}} \mathbf{n}_j[a]$$

and the angle of extent is $\arccos |\mathbf{n}_{\min} \cdot \mathbf{n}_{\max}|$. After computing the angle of extent for all axes in the partition, we select the greatest of them and that becomes our axis of greatest extent.

3.3.3 Dynamic KD-Tree

Sampling-based motion planners, such as RRT and RRT*, generate and potentially add a random configuration to the dataset at every iteration. For these algorithms, the nearest neighbor searching structure must be dynamic—that is, it must support fast insertions interleaved with searches. In [14], Bentley and Saxe show that one approach is to perform a "static-to-dynamic conversion". Their method builds multiple static median-split kd-trees of varying sizes in a manner in which the amortized insertion time is $O(\log^2 n)$ and the expected query time is $O(\log^2 n)$. In the text that follows, we describe our implementation for modifying the kd-tree to a dynamic structure, and we compare the approaches in Sec. 3.4.

The kd-tree may also be easily modified into a dynamic structure by allowing children to be added to the leaves of the structure, and embedding a configuration in each tree node. When building such a dynamic kd-tree, the algorithm does not have the complete dataset, and thus cannot perform a balanced construction like the median partitioning in Sec. 3.3.2. Instead, it chooses splits based upon an estimate of what is likely to be the nature of the dataset. When values are inserted in random order into a binary tree, Knuth [69, p. 430–431] shows that well-balanced trees are common, with insertions requiring about $2 \ln n$ comparisons, and the worst-case $O(n)$ is rare. In our experiments, we observe results suggesting that the generated trees are indeed well-balanced across a variety

Algorithm 9 `DynamicKDInsert` $(\mathbf{q})$

1: $\mathtt{n} \leftarrow \&\mathtt{kdroot}$
2: $(\mathbf{C}_{\min}, \mathbf{C}_{\max}) \leftarrow$ partition bounds
3: **for** $\mathtt{depth} = 0 \rightarrow \infty$ **do**
4: $\quad(\mathtt{axis}, \mathtt{split}) \leftarrow \mathtt{KD\_SPLIT}(\mathbf{C}_{\min}, \mathbf{C}_{\max}, \mathtt{depth})$
5: $\quad$**if** $\mathtt{n} = \varnothing$ **then**
6: $\quad\quad*\mathtt{n} \leftarrow$ new node with $(\mathtt{axis}, \mathtt{split}, \mathbf{q})$
7: $\quad\quad$return
8: $\quad$**if** $\mathbf{q}[\mathtt{axis}] < \mathtt{split}$ **then**
9: $\quad\quad\mathtt{n} \leftarrow \&\,(*\mathtt{n}_{\text{left}})$
10: $\quad\quad\mathbf{C}_{\max}[\mathtt{axis}] \leftarrow \mathtt{split}$
11: $\quad$**else**
12: $\quad\quad\mathtt{n} \leftarrow \&\,(*\mathtt{n}_{\text{right}})$
13: $\quad\quad\mathbf{C}_{\min}[\mathtt{axis}] \leftarrow \mathtt{split}$

of sampling-based motion planning scenarios. In the results section, we split at the midpoint of the bounding box. A few possible choices that empirically work well with sampling-based motion planners are: (1) split at the midpoint of the bounding box implied by the configuration space and the prior splits, (2) split at the hyperplane defined by the point being added, or (3) an interpolated combination of the two.

`DynamicKDInsert` (Alg. 9) adds a configuration into a dynamic kd-tree. In this formulation, each node in the kd-tree contains a configuration, an axis and split value, and two (possibly empty) child nodes. Given the bounding box of the partition and a depth in the tree, the `KD_SPLIT` function (line 4) generates a splitting axis and value. In Euclidean space, `KD_SPLIT` can generate a midpoint split along the axis of greatest extent by choosing the `axis` that maximizes $\mathbf{C}_{\max}[\mathtt{axis}] - \mathbf{C}_{\min}[\mathtt{axis}]$, and the split value of $(\mathbf{C}_{\min}[\mathtt{axis}] + \mathbf{C}_{\max}[\mathtt{axis}])/2$.

In our SO(3) projection, the axis of greatest extent is computed from the angle between $\mathbf{c}_{\min}$ and $\mathbf{c}_{\max}$, where $\mathbf{c}_{\min}$ and $\mathbf{c}_{\max}$ are an axis's bounding hyperplane normals from $\mathbf{C}_{\min}$ and $\mathbf{C}_{\max}$. An interpolated split is computed using a spherical linear interpolation [105] between the bounds:

$$\mathbf{c}_{\text{split}} = \mathbf{c}_{\min}\frac{\sin t\theta}{\sin\theta} + \mathbf{c}_{\max}\frac{\sin(1-t)\theta}{\sin\theta} \qquad \text{where} \qquad \theta = \arccos\left|\mathbf{c}_{\min}\cdot\mathbf{c}_{\max}\right|.$$

A split at the midpoint ($t = 0.5$) simplifies to $\mathbf{c}_{\text{mid}} = (\mathbf{c}_{\min} + \mathbf{c}_{\max})/(2\cos\frac{\theta}{2})$.

Algorithm 10 `DynamicKDSearch` $(\mathbf{q}_{\text{search}}, \mathtt{n}, \mathtt{depth}, \mathbf{C}_{\min}, \mathbf{C}_{\max}, \mathbf{q}_{\text{nearest}}, \mathbf{s}, \mathbf{a})$

1: **if** $\mathtt{n} = \varnothing$ **then**
2: **return** $\mathbf{q}_{\text{nearest}}$
3: **if** `DIST` $(\mathbf{q}_{\text{search}}, \mathbf{q}_n) <$ `DIST` $(\mathbf{q}_{\text{search}}, \mathbf{q}_{\text{nearest}})$ **then**
4: $\mathbf{q}_{\text{nearest}} \leftarrow \mathbf{q}_n$ // $\mathbf{q}_n$ is the configuration associated with $\mathtt{n}$
5: $(\mathtt{axis}, \mathtt{split}) \leftarrow$ `KD_SPLIT` $(\mathbf{C}_{\min}, \mathbf{C}_{\max}, \mathtt{depth})$
6: $(\mathbf{C'}_{\min}, \mathbf{C'}_{\max}) \leftarrow (\mathbf{C}_{\min}, \mathbf{C}_{\max})$
7: $\mathbf{C'}_{\min}[\mathtt{axis}] \leftarrow \mathbf{C'}_{\max}[\mathtt{axis}] \leftarrow \mathtt{split}$
8: **if** $\mathbf{q}_{\text{search}}[\mathtt{axis}] < \mathtt{split}$ **then**
9: $\mathbf{q}_{\text{nearest}} \leftarrow$ `DynamicKDSearch` $(\mathbf{q}_{\text{search}}, \mathtt{n}_{\text{left}}, \mathtt{depth}+1, \mathbf{C}_{\min}, \mathbf{C'}_{\max}, \mathbf{q}_{\text{nearest}}, \mathbf{s}, \mathbf{a})$
10: **else**
11: $\mathbf{q}_{\text{nearest}} \leftarrow$ `DynamicKDSearch` $(\mathbf{q}_{\text{search}}, \mathtt{n}_{\text{right}}, \mathtt{depth}+1, \mathbf{C'}_{\min}, \mathbf{C}_{\max}, \mathbf{q}_{\text{nearest}}, \mathbf{s}, \mathbf{a})$
12: $\mathbf{s}[\mathtt{axis}] \leftarrow \mathtt{split}$
13: $\mathbf{a}[\mathtt{axis}] \leftarrow 1$
14: **if** `BOX_DIST` $(\mathbf{q}_{\text{search}}, \mathbf{s}, \mathbf{a}) \leq$ `DIST` $(\mathbf{q}_{\text{search}}, \mathbf{q}_{\text{nearest}})$ **then**
15: **if** $\mathbf{q}[\mathtt{axis}] < \mathtt{split}$ **then**
16: $\mathbf{q}_{\text{nearest}} \leftarrow$ `DynamicKDSearch` $(\mathbf{q}_{\text{search}}, \mathtt{n}_{\text{right}}, \mathtt{depth}+1, \mathbf{C'}_{\min}, \mathbf{C}_{\max}, \mathbf{q}_{\text{nearest}}, \mathbf{s}, \mathbf{a})$
17: **else**
18: $\mathbf{q}_{\text{nearest}} \leftarrow$ `DynamicKDSearch` $(\mathbf{q}_{\text{search}}, \mathtt{n}_{\text{left}}, \mathtt{depth}+1, \mathbf{C}_{\min}, \mathbf{C'}_{\max}, \mathbf{q}_{\text{nearest}}, \mathbf{s}, \mathbf{a})$
19: **return** $\mathbf{q}_{\text{nearest}}$

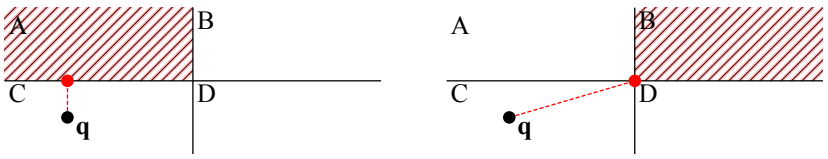
If instead we wish to split at the hyperplane that intersects the point being inserted, we use the `axisnorm` to define the hyperplane's normal. Furthermore, we may combine variations by interpolating between several options.

3.3.4 Kd-Tree Search

In Alg. 10, we present an algorithm of searching for a nearest neighbor configuration $\mathbf{q}_{\text{search}}$ in the dynamic kd-tree defined in Sec. 3.3.3. This algorithm queries a minimal portion of the kd-tree required to ensure a correct result.

The search algorithm begins with $\mathtt{n}$ as the root of the kd-tree, a $\mathtt{depth}$ of 0, $\mathbf{C}_{\min}$ and $\mathbf{C}_{\max}$ as the root partition bounds, an empty $\mathbf{q}_{\text{nearest}}$, and the split vectors $\mathbf{s} = \mathbf{a} = \mathbf{0}$. The search proceeds recursively, following the child node on the side of the splitting hyperplane on which $\mathbf{q}_{\text{search}}$ resides (lines 8–11).

Upon return from recursion, it is possible that the tree will contain a point in the other child that is closer than any point found so far. Thus the search algorithm must check if it is possible that the other child could contain a configuration closer to $\mathbf{q}$ than the nearest one. This check is performed by testing computing the distance from $\mathbf{q}_{\text{search}}$ to the closest point within bounding box



(a) single axis (b) double axis

Figure 3.2: **A kd-tree search for q determining if it should traverse the second node**. The search checks if it is possible for any configuration in the region contained within the node to have a point closer than the one already found. In (a), the search computes the distance between $\mathbf{q}$ and region A—this is a 1-dimensional L^2 distance between $\mathbf{q}$ and the hyperplane that splits regions A and C. In (b), the search computes the distance between $\mathbf{q}$ and region B—and it computes a 2-dimensional L^2 distance. Our method extends this computation to the curved projection on a 3-sphere.

of the other child. If this distance is smaller than the distance to $\mathbf{q}_{\text{nearest}}$, then it is possible for the other child to contain a closer node, and the algorithm recurses into the other child to search it.

Checking the distance to the bounding box does not require a full bounding box check—instead the search algorithm can be sped up by only computing the distance from the $\mathbf{q}_{\text{search}}$ to the nearest point within the bounding box. In the algorithm, the bounding box of a child is defined by $\mathbf{C'}_{\text{min}}$ and $\mathbf{C'}_{\text{max}}$, and the `BOX_DIST` function computes this bounding-box distance. The computation required for `BOX_DIST` is depicted in Fig. 3.2.

In the search algorithm, $\texttt{BOX\_DIST}(\mathbf{q}, \mathbf{s}, \mathbf{a})$ on line 14, computes the distance between a configuration $\mathbf{q}$ and the corner of a partition defined by $\mathbf{s}$ and $\mathbf{a}$, and thus tests to see if it possible for a point in the other child to be closer than the nearest one found so far. The components of $\mathbf{s}$ are the split axis values between the current region and the region in which $\mathbf{q}$ resides. The components of $\mathbf{a}$ are 1.0 for each axis which is defined in $\mathbf{s}$ and 0.0 otherwise. With these values for $\mathbf{a}$ and $\mathbf{s}$, the `BOX_DIST` definition for the L^2 distance metric: $\texttt{BOX\_DIST}_{\texttt{L}^2}(\mathbf{q}, \mathbf{s}, \mathbf{a}) = \left(\sum_{i=1}^{d} (\mathbf{q}_i - \mathbf{s}_i)^2 \mathbf{a}_i\right)^{1/2}$.

For the search algorithm to produce a correct result, the box distance function can be relaxed—it is sufficient that it returns a distance less than or equal to the closest possible configuration in the node's region. This for example, $\texttt{BOX\_DIST}(\cdots) = 0$, is sufficient to produce a correct result but inefficient in that it would result in a searching the entire kd-tree. In general, a poorly bounded box distance is valid, but results in reduced search performance. Thus a tightly bounded `BOX_DIST` is critical to performance.

In order to extend the `BOX_DIST` function to our projected partition mapping of SO(3), we must compute the distance between a configuration $\mathbf{q}$ and a partition defined by hyperplanes partitioning

a unit 3-sphere. For this function to be tightly bounded, it must take into account that the partition defined by the bounds on our projected manifold are curved (see Fig. 3.1). When only 1 hyperplane is defined (i.e. the first split in SO(3)), the distance is the angle between a configuration and a great circle defined by a splitting hyperplane's normal $\mathbf{n}_{\text{split}}$ and its intersection with the unit 3-sphere. This distance is:

$$\texttt{BOX\_DIST}_{P^3|\mathbf{n}_{\text{split}}} = \arcsin(\mathbf{q} \cdot \mathbf{n}_{\text{split}})$$

When 2 of the 3 axes are split, the distance is the angle between the configuration and an ellipse. The ellipse results from projecting the line defined by the two splitting hyperplanes onto a unit 3-sphere. If the split axis values are the normals $\mathbf{n}_b$ and $\mathbf{n}_c$ in the projected a partition, and thus the d-axis is not yet split, the partial distance is:

$$\texttt{BOX\_DIST}_{P^3|\mathbf{n}_b,\mathbf{n}_c} = \min_{\omega} \arccos |\mathbf{q} \cdot \texttt{ell}(\mathbf{n}_b, \mathbf{n}_c, \omega)|$$

where `ell` is an ellipsoid parameterized by the normals $\mathbf{n}_b$ and $\mathbf{n}_c$, and varied over ω:

$$\texttt{ell}(\mathbf{n}_b, \mathbf{n}_c, \omega) = \left(\omega, -\omega \frac{\mathbf{n}_b[a]}{\mathbf{n}_b[b]}, -\omega \frac{\mathbf{n}_c[a]}{\mathbf{n}_c[c]}, \pm \sqrt{1 - \omega^2 - \left(\omega \frac{\mathbf{n}_b[a]}{\mathbf{n}_b[b]}\right)^2 - \left(\omega \frac{\mathbf{n}_c[a]}{\mathbf{n}_c[c]}\right)^2} \right)$$

The distance is minimized at $\omega = \gamma / \sqrt{\eta(\gamma^2 - \eta \mathbf{q}[a])}$ where

$$\gamma = \mathbf{q}[a] - \mathbf{q}[b] \frac{\mathbf{n}_b[a]}{\mathbf{b}[b]} - \mathbf{q}[c] \frac{\mathbf{n}_c[a]}{\mathbf{n}_c[c]}, \qquad \eta = 1 + \left(\frac{\mathbf{n}_b[a]}{\mathbf{n}_b[b]}\right)^2 + \left(\frac{\mathbf{n}_b[a]}{\mathbf{n}_c[c]}\right)^2.$$

When all three axes are split (e.g., the b, c, and d axes in the a projected partition), the distance is the angle between the configuration and the corner of the hyperplane bounded partition defined by the three axes. If the split axis values are the normals $\mathbf{n}_b$, $\mathbf{n}_c$, and $\mathbf{n}_d$ (in the projected a partition), the partial distance is:

$$\texttt{BOX\_DIST}_{P^3|\mathbf{n}_b,\mathbf{n}_c,\mathbf{n}_d} = \arccos |\mathbf{q} \cdot \mathbf{q}_{\text{corner}}|$$

$$\text{where:} \qquad \mathbf{q}_{\text{corner}} = \texttt{normalize}\left(1, -\frac{\mathbf{n}_b[a]}{\mathbf{n}_b[b]}, -\frac{\mathbf{n}_c[a]}{\mathbf{n}_c[c]}, -\frac{\mathbf{n}_d[a]}{\mathbf{n}_d[d]}\right)$$

Each of these `BOX_DIST` functions for P^3 successively provide a tighter bound, and thus prunes recursion better.

Each query in the SO(3) subspace must search up to four kd-trees created from the top-level projected partitions on the 3-sphere. The projected partition in which the query configuration lies we call the *primary* partition, and the remaining three partitions are the *secondary* partitions. The search begins by finding the nearest configuration in the kd-tree in the primary partition. The search continues in each of the remaining secondary partitions only if it is possible for a point within its associated partition to be closer than the nearest point found so far. For this check, the box distance is computed between the query configuration and the two hyperplanes that separate the primary and each of the secondary partitions. There are two hyperplanes due to the curved nature of the manifold and the double-coverage property of quaternions. Since a closer point could lie near either boundary between the partitions, we must compare to the minimum of the two partial distances, thus:

$$\min\left(\texttt{BOX\_DIST}_{P^3|\mathbf{n}_{ab}}(\mathbf{q}), \quad \texttt{BOX\_DIST}_{P^3|\mathbf{n}_{ba}}(\mathbf{q})\right)$$

where $\mathbf{n}_{ab}$ and $\mathbf{n}_{ba}$ are the normals of the two hyperplanes separating the top-level partitions a and b.

3.3.5 Nearest, k-Nearest, and Nearest in Radius r Searches

With minor modification, the nearest-neighbor searching algorithm in Alg. 10 extends to support k-nearest and radius-based nearest neighbor searches.

We extend it to k-nearest neighbor search by replacing $\mathbf{q}_{\text{nearest}}$ with a priority queue. The priority queue contains up to k configurations and is ordered based upon distance from $\mathbf{q}_{\text{search}}$, with the top being the farthest of the contained configurations from $\mathbf{q}_{\text{search}}$. The queue starts empty, and until the queue contains k configurations, the algorithm adds all visited configurations to the queue. From then on, $\texttt{DIST}(\mathbf{q}_{\text{search}}, \mathbf{q}_{\text{nearest}})$ (lines 3 and 14) is the distance between $\mathbf{q}_{\text{search}}$ and the top of the priority queue. When the search finds a configuration closer than the top of the queue, it removes the top and adds the closer configuration to the queue (line 4). Thus the priority queue always contains the k nearest configuration visited.

To search for all nearest neighbors within radius r of $\mathbf{q}_{\text{search}}$, we modify $\mathbf{q}_{\text{nearest}}$ in Alg. 10 to be a set. Distance comparisons on lines 3 and 14 treat $\texttt{DIST}(\mathbf{q}_{\text{search}}, \mathbf{q}_{\text{nearest}}) = r$. When the algorithm finds a configuration closer than r, it adds it to the result set in line 4.

3.4 Results

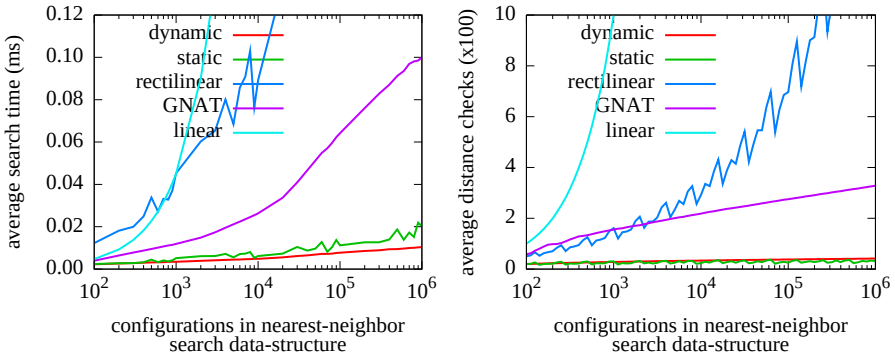
We evaluate our method for nearest neighbor searches in four scenarios: (1) uniform random rotations in SO(3), (2) uniform random rotations and translations in SE(3), (3) configurations generated by RRT [74] solving the "Twistycool" motion planning scenario in OMPL [112], and (4) configurations generated by RRT* [60] solving the "Home" motion planning scenario in OMPL [112]. We compare four methods for nearest neighbor searching: (1) "dynamic" is a dynamic kd-tree using our method and midpoint splits, (2) "static" is a static-to-dynamic conversion [14] of a median-split kd-tree using our method, (3) "rectilinear" is a static-to-dynamic conversion of a median-split kd-tree using rectangular splits [123] on SO(3), and (4) "GNAT" is a Geometric Near-neighbor Access Tree [17]. All runs are computed on a computer with two Intel X5670 2.93 GHz 6-core Westmere processors, though multi-core capabilities are not used.

3.4.1 Random SO(3) Scenario

To show our method's ability to speed up searching on 3D rotations, we generated uniformly distributed random configurations in SO(3) and compute nearest neighbors for random configurations. We compute the average search time and the average number of distance computations performed to search a nearest neighbor data structure of size n. We vary n from 100 to 1 000 000 configurations, and plot the result in Fig. 3.3. The average nearest neighbor search time in Fig. 3.3(a) shows an order of magnitude performance benefit when using our method. The number of distance computations in Fig. 3.3(b) is a rough metric for how much of the data structure each method is able to prune from the search. The performance gain in Fig. 3.3(b) gives insight into the reasons for the performance gains shown in Fig. 3.3(a).

3.4.2 Random SE(3) Scenario

To show our method's ability to speed up searching of 3D rigid body transforms, we build nearest neighbor search structures with random configurations generated in SE(3). Using $\texttt{DIST}_{\mathbb{R}^m P^3}$, we evaluate performance for $\alpha = 1$ and 10 in Fig. 3.4. For small α, the SO(3) component of a configuration is given more weight, and thus provides for greater differentiation of our method. In Fig. 3.4 (a), we observe a 2 to 5× improvement in performance between our method and the rectilinear method, and an order of magnitude performance improvement over GNAT. As α increases, more weight is given to the translation component, so our SO(3) splits have less impact on performance. Hence, our improvement drops, but is still 2 to 3× faster than rectilinear, and 8× faster than GNAT.



(a) nearest neighbor search time

(b) distance checks

Figure 3.3: **Comparison of nearest neighbor search time and distance checks plotted with increasing configuration count in the searched dataset**. In (a) we plot the average time to compute a single nearest neighbor for a random point. In (b) we track the average number of distance computations performed by a search.

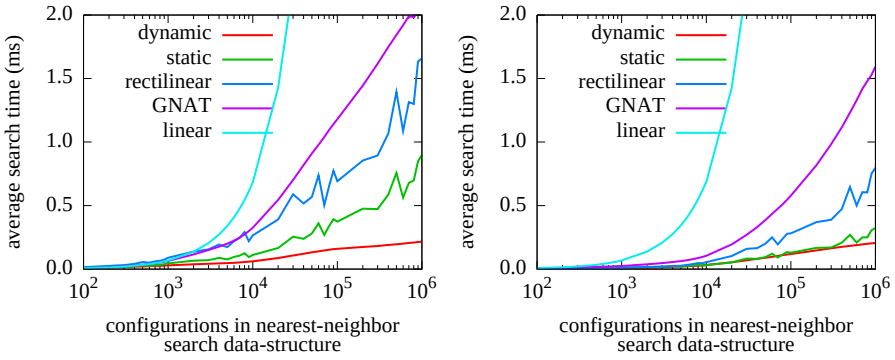
3.4.3 RRT on the Twistycool Scenario

To show the impact of our method on the performance of sampling-based motion planning, we evaluate embed our method into an RRT planner and have it solve the "Twistycool" motion planning scenario from using OMPL . The Twistycool puzzle, shown in Fig. 3.5(a), is a motion planning problem in which a rigid-body object (the robot) must move through a narrow passage in a wall that separates the start and goal configurations. At each iteration, the RRT motion planner computes a nearest neighbor for a random sample against all samples it has already added to its motion planning tree. We have adjusted the relative weighting α for translation and rotation from its default, such that each component has approximately the same impact on the weighted distance metric.

As we see in Fig. 3.5(b), the performance of our method with the dynamic kd-tree is more than $5\times$ faster than GNAT and rectilinear split kd-trees. This matches our expectations formed by the uniform random scenario results, and shows little degradation with the non-uniform dataset created by this motion planning problem.

3.4.4 RRT* on the Home Scenario

To show our methods impact on an asymptotically optimal sampling-based motion planner, we embed our nearest neighbor method into RRT* and have it solve the "Home" scenario included in OMPL. As shown in Fig. 3.6(a), the motion planner computes a plan that moves a table from one room to another while avoiding obstacles. The RRT* planner incrementally expands a motion



(a) search times ($\alpha = 1$)

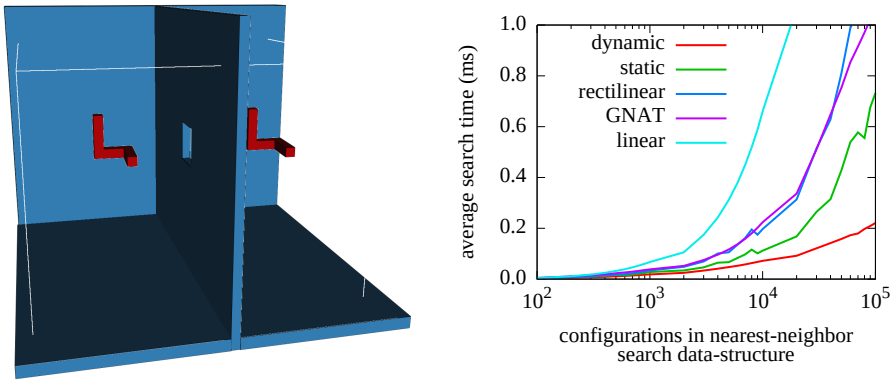
(b) search times ($\alpha = 10$)

Figure 3.4: **Comparison of nearest neighbor search time for random configurations in SE(3)**. In (a) and (b) the translation space is bounded to a unit cube, and the translation distance is weighted 1.0 and 10.0 respectively. In (a) the SO(3) component of a configuration is given more weigh, and thus has more impact on each search.

planning tree, while "rewiring" it towards optimality as it goes. In each iteration RRT* finds an extension point using a nearest neighbor search, and then rewires a small neighborhood after a k-nearest neighbor search. Unlike RRT, we can allow RRT* to continue for as many iterations as desired, and get incrementally better results. As with the RRT scenario, we proportionally scale α so that the SO(3) and translation components have approximately equivalent impact on the distance metric. As shown in Fig. 3.6 (b), our method in both variants outperforms GNAT and rectilinear splits by roughly a factor of 3. In these results we observe also that the median split of "static" and the midpoint split of "dynamic" perform equally well, and the main differentiating factor between the kd-tree methods is thus the SO(3) partitioning.

3.5 Conclusion

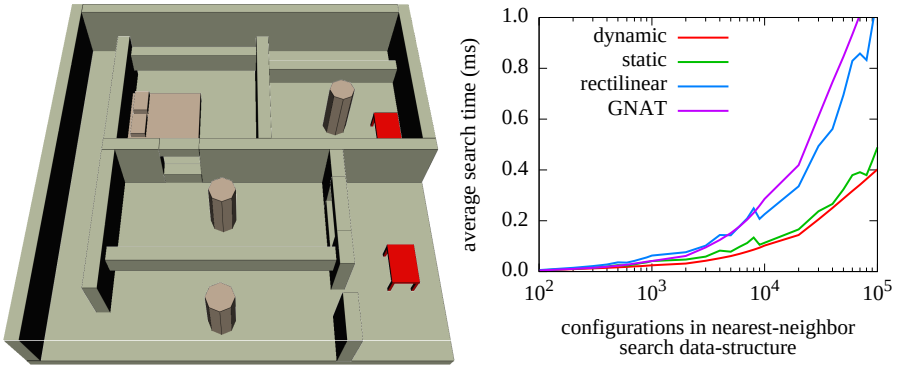
In this chapter we presented a novel approach to partitioning SO(3) and by extension, SE(3), and used that approach to create a fast nearest-neighbor searching data structure. This data structure, based upon a kd-tree, offers two key benefits: (1) it reduces the dimensionality of the rotation representation from 4-dimensional quaternion vector to match its 3 degrees of freedom, and (2) creates an efficient partitioning of the curved manifold of the rotational group. We integrated our approach into RRT and RRT* and demonstrated that the fast nearest-neighbor searching performance improved the solution time and convergence rate in rigid-body motion planning problems when compared to prior work.



(a) Twistycool scenario (b) RRT nearest neighbor search times

Figure 3.5: **Twistycool scenario and RRT nearest neighbor search times**. The scenario in (a) requires the red robot to move from its starting configuration on the left, through a narrow passage in the wall, to its goal configuration on the right. The average time per nearest neighbor search is plotted in (b).

In the chapter 4, we further speed up nearest neighbor searching and enable concurrent nearest neighbor data structure searching and inserting. The methods from this chapter enable the concurrency advances to apply to SO(3) and SE(3) metric spaces, and thus enable highly-scalable and fast motion planning on rigid-body and related motion planning problems.



(a) Home scenario (b) RRT* nearest neighbor search times

Figure 3.6: **Home scenario and RRT* nearest neighbor search times**. In the scenario in (a), the motion planner must find a path that moves the red table "robot" from its starting configuration in the lower right room to the goal configuration in the upper right. The average time for nearest neighbor search is plotted in (b).

CHAPTER 4

Concurrent Nearest Neighbor Searching

Nearest neighbor searching data structures are a fundamental building block for many algorithms in robotics. Algorithms such as sampling-based robot motion planners [20], typically need to repeatedly search and insert data into a nearest neighbor data structure, and thus their performance benefits from nearest neighbor operations that are fast. However, with the trend of modern CPUs towards increasing computational parallelism in the form of multiple processor cores, it is no longer sufficient for a data structure to just enable operations to be fast. To harness the full computational power of a multi-core processor, algorithms must also allow for concurrent operations across multiple cores without slowdown. Slowdown is unfortunately worsened by increasing parallelism when the data structure requires concurrent operations to *wait* for mutually exclusive access to data to ensure correct operation. A *concurrent* data structure, on the other hand, avoids this source of slowdown, by minimizing or eliminating the requirement for mutual exclusion and the associated wait time. In this chapter we present a concurrent data structure, and associated algorithms, for fast exact nearest neighbor searching that is geared towards robotics applications.

In this chapter we improve upon the performance of the concurrent nearest neighbor data structure that we introduced in chapter 2. In that chapter, a simple binary lock-free kd-tree provided wait-free queries and lock-free inserts. The fast lock-free inserts of that approach reduced the likelihood of insert waiting, but come at the expense of increased search times due to imbalances in the resulting tree. In this chapter, insert operations produce a more balanced tree resulting in faster queries, and we provide proofs of correct operation and the low probability of waits.

The data structure in this chapter is based on a kd-tree [13], and thus as presented in chapter 3, provides for fast insert and search operations on metric spaces important to many robotics applications—including Minkowski spaces (a common example being Euclidean), SO(3) [46], and Cartesian products thereof [123]. This data structure, like kd-trees, partitions space into spatially separated sub-trees using branching nodes. Fast insertion of a new point (e.g., a robot configuration

for a sampling-based motion planner) into the data structure comes from the ability to quickly traverse the partitions to an insertion point. Fast searches for a set of nearest neighbors to a query point come from the ability to use the partitions to confine traversal to a spatially relevant portion of the tree. With minor modifications to the searching algorithm, searches can also produce nearest neighbor sets that are bounded in cardinality or bounded to be within a radius from the query point.

The data structure presented in this chapter supports provably correct concurrent operations. This is in contrast to the traditional approach to kd-trees, in which concurrent operation without mutual exclusion leads to data structure corruption. Corruption occurs when concurrent operations interleave mutations that invalidate the computations of each other. For example, two (or more) threads inserting similar values into a kd-tree may decide to split the same node, causing one overwriting the result of the other. The problem is only exacerbated by modern compilers and CPUs as they often automatically and unpredictably change the order of memory accesses to improve the performance of non-concurrent operations. For example, if one operation writes to memory location 'A' and then to 'B', a concurrent operation may see the change to 'B' before it sees the change to 'A'. While the reordered memory writes do not affect the correctness of the operation in which they occur, they may become problematic for the correctness of an operation running concurrently. An effective way to prevent corruption caused by interleaved mutations and reordering of memory writes is to only allow one insert operation to happen at any moment in time by using a mutual exclusion locking mechanism. But, by definition, locking prevents concurrent operations, and thus all but one attempted concurrent insert operation will spend time waiting. When an thread spends time waiting instead of computing, the algorithm effectively slows down. To avoid this slowdown, the data structure supports concurrent wait-free queries, and it also supports inserts that wait with asymptotic probability of zero.

We embed the proposed method in the parallelized sampling-based motion planning algorithm from chapter 2 to demonstrate its performance and ability to operate under concurrency on a 32-core computer. The improvements proposed in this chapter double nearest-neighbor search performance when compared to our prior lock-free nearest neighbor search data structure, and lead to up to 30% faster convergence rates of the motion planner. Sampling-based motion planners parallelize well [4], but as the results show, contention over exclusive access to a non-concurrent nearest neighbor data structures can slow them down significantly. The concurrent data structure we propose allows the

parallelized motion planner to find solutions and converge faster by generating orders of magnitude more samples than a parallelized motion planner that must lock its data structures.

4.1 Related Work

Our proposed nearest neighbor searching approach loosely follows that of a kd-tree [13, 36, 109]. A kd-tree is a space-partitioning binary tree that splits branching nodes along axis-aligned hyperplanes in $\mathbb{R}^n$. When splitting hyperplanes occur at the median of values in the subtrees, it creates perfectly balanced trees. However, as originally proposed, kd-trees are limited to $\mathbb{R}^n$ with a Minkowski metric.

Yershova et.al. [123] extended the metric spaces supported by kd-trees to include SO(2), SO(3), and cartesian products thereof and with $\mathbb{R}^n$. The SO(3) partitions of this approach are along axis-aligned hyperplanes in $\mathbb{R}^n$. In chapter 3, we propose a method for partitioning SO(3) using hyperplanes that wrap around the 3-sphere manifold obtained from a quaternion representation of SO(3) rotations. While the data structure we propose in this chapter works with either SO(3) partitioning scheme, we expand upon the latter to address special handling required when inserting values under concurrency.

Generalized nearest-neighbor approaches, such as the Geometric Near-neighbor Access Tree (GNAT) [17] only require a well-behaved metric and thus support a broader set of topologies than kd-trees. The generalized nature of such structures does not take advantage of knowlege of the underlying topology as kd-trees do, and thus may not be as efficient as kd-trees. Additional work is also required to make such structures support concurrent and wait-free operations.

Approximate nearest neighbor searching approaches gain search efficiency by sacrificing accuracy. Methods include locality sensitive hashing (LSH) [7] and randomized kd-trees [106]. Our focus is on exact nearest neighbor searching as the proofs of many sampling-based motion planners' asymptotic feasibility (e.g., RRT [74]) and asymptotic optimality (e.g., RRT* [60]) implicitly rely on the nearest neighbor structure being exact. However, if the trade-off of accuracy for speed is appropriate, methods such as those proposed by Arya et al. [10] and Beis et al. [11] shorten the kd-tree search process producing approximate results faster. We believe similar methods could be readily applied to our proposed method to allow for approximate nearest neighbor searching under concurrency.

Concurrent data structures, such as the binary tree proposed by Kung and Lehman [71], allow correct operation while avoiding contention by having threads lock only the node that they are manipulating. In chapter 2, we proposed a kd-tree that allows concurrent modification and searching while avoiding contention through the use of a *lock-free* atomic update. When inserting into this kd-tree, the algorithm makes partitioning choices at the leaf of the kd-tree based upon the bounds of the region and/or the value in the leaf. Empirically this approach works well for the random insertion order of the associated sampling-based planner. However, better search performance is possible with a balanced kd-tree as would be created by median splits. To better approximate median splits in this work, we incorporate the approach described by Sherlock et al. [104] that accumulates a predetermined number of values into leaves before performing a median split on the values within the leaf.

4.2 Problem Definition

The problem definition is stated in two key parts: (1) correct concurrent operation, and (2) nearest neighbor searching.

Correct Concurrent Operation requires that memory *writes* of one operation must not adversely affect the memory *reads* or *writes* of a concurrent operation, while minimizing the time concurrent operations wait on each other. Once an operation running on a CPU core inserts a point into the data structure, the inserted point will eventually be reachable to all other cores. Once an operation running on a CPU core reaches a point in the data structure, all subsequent operations on that core must continue to reach the point.

Nearest Neighbor Searching finds all the nearest neighbors of a query point. Let $\mathcal{C}$ be a topological space which is composed of the Cartesian product of one or more sub-topologies in $\mathbb{R}^n$ and SO(3). Let $\mathbf{q} \in \mathcal{C}$ be a single configuration in the topological space with components from each sub-topology, e.g., $\mathbf{q} = \{\mathbf{p}_i, \ldots, \mathbf{r}_j, \ldots\}$, with $\mathbf{p}_i \in \mathbb{R}^{n_i}$ and $\mathbf{r}_j \in S^3$ for each i and j. Each SO(3) component is specified using the coefficients of a unit quaternion representing a rotation in 3D space [70].

Let $d(\mathbf{q}_1, \mathbf{q}_2)$ be the distance between two configurations, such that it is the weighted sum of the distances of each sub-topology's component:

$$d(\mathbf{q}_a, \mathbf{q}_b) = \sum_i \alpha_i d^p_{\mathbb{R}^n}(\mathbf{p}_{a_i}, \mathbf{p}_{b_i}) + \sum_j \alpha_j d_{\mathrm{SO(3)}}(\mathbf{r}_{a_j}, \mathbf{r}_{b_j}),$$

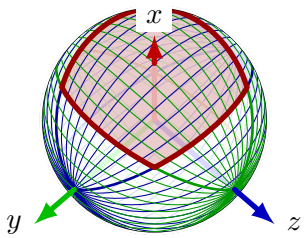


Figure 4.1: **Lower-dimensional analog of SO(3) partitioning scheme** [46]. In SO(3), quaternions are partitioned into four non-overlapping bounded regions of a 3-sphere, with the negative axis mapped onto the positive axis due to the double-coverage property. The 2-sphere analog shown here is partitioned into three bounded regions, with the x-centered bounded region highlighted in red. Within the bounded region, evenly separated partitioning hyperplanes are shown in green for one axis and in blue for the other.

where α_i and α_j are positive real weight values, $d^p_{\mathbb{R}^n}(\cdot,\cdot)$ is an L^p distance metric on $\mathbb{R}^n$, and $d_{\mathrm{SO}(3)}(\cdot,\cdot)$ is the length of the shorter of the two angles subtended along the great arc. Thus:

$$d^p_{\mathbb{R}^n}(\mathbf{p}_a, \mathbf{p}_b) = \Big(\sum_i^n |\mathbf{p}_{a,i} - \mathbf{p}_{b,i}|^p \Big)^{1/p}$$
$$d_{\mathrm{SO}(3)}(\mathbf{r}_a, \mathbf{r}_b) = \cos^{-1} |\mathbf{r}_a \cdot \mathbf{r}_b|.$$

If appropriate to the application, a similar effect to weighting the distance metric can also be obtained by scaling the $\mathbb{R}^n$ coefficients instead.

Given a set $\mathbf{Q} = \{\mathbf{q}_1, \mathbf{q}_2, \ldots, \mathbf{q}_n\}$ where $\mathbf{q}_i \in \mathcal{C}$, and a query point $\mathbf{q}_{\text{search}} \in \mathcal{C}$ for some topological space $\mathcal{C}$, the objective of *k-nearest neighbors search*, is to find the set $\mathbf{N} \subseteq \mathbf{Q}$, such that $|\mathbf{N}| = \min(k, |\mathbf{Q}|)$, and:

$$\max_{\mathbf{q}_i \in \mathbf{N}} d(\mathbf{q}_i, \mathbf{q}_{\text{search}}) \leq \min_{\mathbf{q}_j \in \mathbf{Q} \setminus \mathbf{N}} d(\mathbf{q}_j, \mathbf{q}_{\text{search}}),$$

where k is a positive integer. With $k = 1$ it thus finds the nearest neighbor.

The objective of *r-nearest neighbors search*, where r is a non-negative real value, is to find $\mathbf{N} \subseteq \mathbf{Q}$, such that:

$$\mathbf{N} = \{\mathbf{q}_i \mid d(\mathbf{q}_i \in \mathbf{Q}, \mathbf{q}_{\text{search}}) \leq r\}.$$

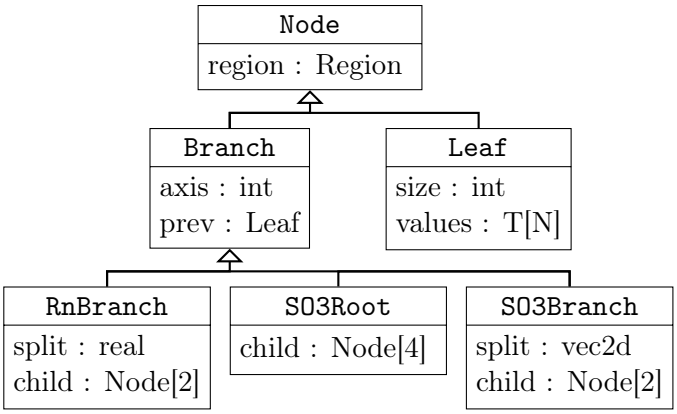


Figure 4.2: **Diagram of a possible node design needed to implement the proposed data structure**. Each box represents a *type* of node that can be in the tree, with its name (top) and its data members (below the separating line). Data members are listed as *name : type*. Array types have their capacity listed in square brackets. Nodes inherit all members from their ancestors (shown with open arrows), thus all node types have a *region* data member. The three node types that inherit from `Branch` include a split *axis* and *prev* pointer to the Leaf node that the branch replaced. The root of an SO(3) subtree has four children, while the other branch types have a split plane definition and two children. The `Leaf` node has a current *size*, and fixed capacity (N) array of *values* of the type (T) stored in the data structure.

4.3 Method

The proposed method is based upon a kd-tree. A kd-tree is a binary tree data structure in which each branch successively partitions space by an axis-aligned hyperplane, and the leaf nodes contain the points to search. Searching a kd-tree for a query point begins at the root of the tree. When the search encounters a branch, it recurses to the child on the same side of the branch's splitting hyperplane as the query point. When the search encounters a leaf, it checks the distance between the leaf's point and the query point, and adds the point to the result set if the distance is small enough. When returning from recursion, the search then checks the distance between the query point and the closest point on the splitting hyperplane. If the distance between the points is small enough to be added to the result set, then the algorithm recurses to search the other child of the branch.

The partitioning approach for SO(3) [46], requires special handling for the top-level SO(3) branch (see lower dimensional analog in Fig. 4.1). Unlike other branches, this branch partitions space into four top-level *volumes*, one for each of the four components of a quaternion. (See `SO3Root` in Fig. 4.2). Once the algorithm has partitioned a value to a top-level SO(3) volume, the branches in the subtree are binary splits—similar to branches in $\mathbb{R}^n$, but with a hyperplane through the origin and defined by a constrained normal (see [46])

Algorithm 11 `INSERT`$(T, \mathbf{u})$

Require: T is the kd-tree, $\mathbf{u}$ is the value to insert

```
1:  p ← root of T
2:  loop
3:    n ← load(p)
4:    if n is a branch then
5:      update n's region to contain u
6:      p ← FOLLOW(n, u)
7:    else if not try_lock(n) then {n is a leaf}
8:      yield/pause CPU
9:    else {acquired lock on n}
10:     m ← load(n.size)
11:     if m < leaf capacity then
12:       update n's region to contain u
13:       append u to leaf n
14:       store(n.size, m + 1)
15:       unlock(n)
16:       return
17:     c ← SPLIT(n, u)
18:     store(p, c)
```

4.3.1 Data Storage

In chapter 3, we proposed a lock-free kd-tree that created a new branch every time a leaf was inserted. That approach has the benefit of making insertions quick and lock-free, but introduces an expense to search performance from two factors: (1) there is little information from which to choose a splitting hyperplane, leading to suboptimal tree-balancing, and (2) traversing a branch is more time consuming than a simple point-to-point distance check of a leaf. This performance issue is further exacerbated in algorithms that search more frequently than they insert (e.g., sampling-based motion planning algorithms such as [74, 60] that reject samples after checking the validity of paths to nearest neighbors). In the approach proposed herein, we address these two factors to improve search performance, by batching many points into leaves before splitting them into branches [104]. In our implementation, the leaf node's batch size is a fixed tunable number of the data structure.

4.3.2 Inserting Data

Inserting a value into a concurrent batched kd-tree (Alg. 11) starts at the kd-tree's root node (line 1) and traverses down the tree until it finds a leaf into which it will insert the new point. At each level of the tree, the current node is checked to see if it is a branch or a leaf. Empty trees and children are stored as leaf nodes with 0 size, and thus do not require special handling. When the

Algorithm 12 `FOLLOW`$(n, \mathbf{u})$

Require: n is branch

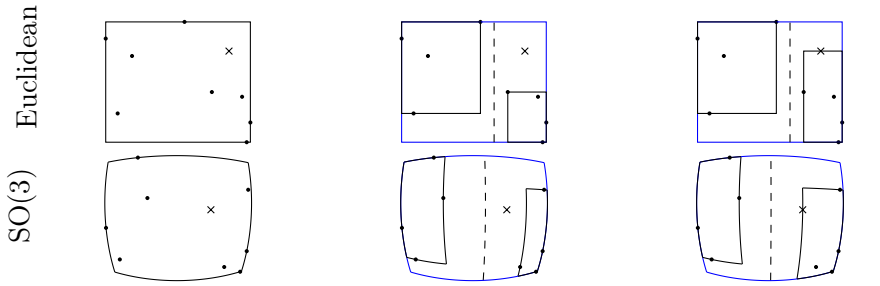
1: **if** n is SO(3) root **then**
2: $i \leftarrow$ `so3_volume_index`$(\mathbf{u})$
3: **return** n.child$[i]$
4: **else if** n is SO(3) branch **then**
5: **return** n.child$[H(\mathbf{u}[\text{axis}] \cdot n.\text{split})]$
6: **else if** n is $\mathbb{R}^n$ branch **then**
7: **return** n.child$[H(\mathbf{u}[\text{axis}] - n.\text{split})]$

algorithm encounters a branch node (line 4), it updates the branch node's region and traverses to the child under which the new value will be inserted. When the algorithm encounters a leaf it first attempts to lock the leaf (line 7) using a fast spin locking mechanism such as compare-and-swap (CAS) on a boolean flag. If the algorithm fails to lock the node, it issues an optional CPU-specific instruction (line 8) for efficient spin locking[1], and then it loops to try again. Once the algorithm successfully acquires the lock, it appends the value to the leaf if there is room (line 11), or splits the leaf (line 17) otherwise. When appending to a leaf, the algorithm ensures the new value is fully initialized before updating the leaf's size (line 14). The size update is a linearization point for making the inserted value reachable to other cores. When splitting the leaf, the algorithm replaces the leaf with the new branch (line 18), and then loops to insert the value into one of the branch's children.

Traversing to Insertion Point FOLLOW (Alg. 12) implements the branch traversal required by the INSERT algorithm. When it encounters an SO(3) root node, it traverses to the child whose partition contains the sample. When it encounters an SO(3) branch or an $\mathbb{R}^n$ branch, it computes the signed distance between the point to insert and the splitting hyperplane. The sign of the distance selects the child using a Heaviside step function $H(\cdot)$ defined as:

$$H(x) = \begin{cases} 0, & x < 0 \\ 1, & x \geq 0. \end{cases}$$

[1]Issuing the appropriate yield/pause instruction here can allow some CPU architectures to give more resources to concurrently running threads and/or reduce power consumption—for example, the `pause` instruction on Intel architectures [92] and the `yield` instruction on ARM-based CPUs.



(a) Leaf to split (b) Branch created (c) Region updated

Figure 4.3: **Steps of splitting a leaf while operating under concurrency**. In (a) the INSERT algorithm traversed to the leaf to add the 'x', finds the leaf is full, and thus calls SPLIT to create a branch. SPLIT partitions the branch along leaf's horizontal dimension resulting in the branch shown in (b). After SPLIT returns, INSERT then traverses to the right side, adds to the leaf, and updates the leafs region (c). During the SPLIT process, concurrent nearest neighbor searches traverse the old leaf. Once the INSERT replaces the leaf with the branch, searches will traverse the branch instead.

Algorithm 13 `SPLIT`$(n, \mathbf{u})$

Require: n is a Leaf, $\mathbf{u}$ is the value to insert

```
1:  axis ← best_axis(n's region)
2:  if axis is first SO(3) then
3:      c ← new SO3Root
4:      for all v ∈ n.values do
5:          i ← so3_volume_index(v)
6:          append v to c.child[i]
7:      return r
8:  else
9:      b_0, b_1 ← median_split(p, axis)
10:     split ← ½(max(b_0.values) + min(b_1.values))
11:     return new branch with axis, split, b_0, b_1
```

Splitting Leaf Nodes When inserting into a full leaf, the INSERT algorithm uses `SPLIT` (Alg. 13) to create a branch from the values in the full leaf. For an efficient kd-tree the splitting process will choose a partition that: (1) minimizes the maximum distance between points in the resulting subdivision and (2) divides the values into equal leaf nodes with the same number of elements (median split). To that end, the `SPLIT` algorithm first selects the best axis for partitioning as the one with the greatest extent between region bounds of the leaf. The region bounds are maintained by INSERT. For $\mathbb{R}^n$ axes, the extent is the difference between the minimum and maximum along each dimension of the bounds. For SO(3) root nodes, the extent is $\pi/2$. For SO(3) branch nodes, the extent is the arccosine of the dot product of the minimum and maximum normalized bounds for the axis [46].

Algorithm 14 `NEAREST`$(N, n, \mathbf{q}, k, r)$

Require: N is the set of nearest neighbor result so far, n is a pointer to the current node, $\mathbf{q}$ is the query, k is the maximum $|N|$ to return, r is the maximum radius

```
1: if |N| < k or dist(n.region, q) ≤ min(r, max N) then
2:   if n is leaf then
3:     for all i ∈ {0, ... load(n.size)} do
4:       if |N| < k or dist(n.values[i], q) < min(r, max N) then
5:         if |N| = k then
6:           N ← N \ (max N)
7:         N ← N ∪ n.values[i]
8:   else if n is SO(3) root then
9:     i ← so3_volume_index(q)
10:    N ← NEAREST(N, load(n.child[i]), q, k, r)
11:    for all v ∈ {0, 1, 2, 3} \ i do
12:      N ← NEAREST(N, load(n.child[v]), q, k, r)
13:  else
14:    if n is SO(3) branch then
15:      c ← H(q · n.split)
16:    else {n is ℝ^n branch}
17:      c ← H(q[axis] − n.split)
18:    N ← NEAREST(N, load(n.child[c]), q, k, r)
19:    N ← NEAREST(N, load(n.child[1 − c]), q, k, r)
20: return N
```

If the selected axis is the SO(3) root, the SPLIT algorithm creates a new SO3Root branch node, and copies the old leaf's values into the appropriate child of the SO3Root (lines 3 to 6). Otherwise, for the remaining axis types, `median_split` (line 9) partitions the values of the old leaf evenly into two new leaf nodes (see Fig. 4.3 (a) and (b)) using an efficient selection algorithm. The SPLIT algorithm returns with a new branch that is split halfway between the maximum of one child and the minimum of the other (line 11).

In the presence of concurrency, concurrent nearest neighbor searches will continue to traverse the old leaf until INSERT atomically replaces the old leaf with the new branch. This means that INSERT does not know if the old leaf is being concurrently accessed, and thus cannot release the memory associated with the leaf without risking a program error. The SPLIT algorithm presented here stores a reference to the old leaf to allow the memory associated with the leaf to be safely deallocated later.

4.3.3 Searching Operations

NEAREST (Alg. 14) implements k-nearest neighbor (with k as appropriate and $r = \infty$) and r-nearest neighbor (with $k = \infty$ and r as appropriate) searches. Traversal for searching for a nearest neighbor is similar to that of FOLLOW. The primary difference is that after searching one child of the branch, NEAREST may need to search the other children of a branch. The algorithms starts with a pointer n to the root node of the kd-tree, and an empty set N of nearest neighbors. It terminates recursion if the node's region (as maintained by INSERT) is too far away from the query point to be added to the nearest neighbor set. If the node is a leaf (lines 3 to 7), it iterates through each value in the leaf, updating the N as appropriate. Here it first loads the node's size, ensuring that it will only visit consistent values in the leaf based upon the linearization point in INSERT.

When traversing an SO3Root node, NEAREST navigates the search key's SO(3) axis-major volume first (lines 9 and 10). It then searches the remaining volumes in an arbitrary order (lines 11 to 12).

When traversing an SO3Branch node or RnBranch node (lines 15 to 19), the algorithm first traverses a child in the same order as FOLLOW does. After returning from recursion on that child, it then traverses the other child. By recursing on the closer child first, updates to N will cause the traversal on the farther child to terminate quickly on line 1.

4.4 Correctness and Analysis

In this section we prove that NEAREST is wait-free and correct with concurrent INSERTS (Lemma 2), and provide analysis on the probability that INSERT waits (Lemma 4). Correct operation relies upon *linearizable* operations which appear to occur instantaneously at a *linearization point* from the perspective of concurrent operations. Thus, before the linearization point, the linearizable operation has not occurred, and after the linearization point, the operation has occurred—there is no intermediate point in which the operation partially occurs. We prove that INSERT is *linearizable* (Lemma 1) and that once a value is reachable it remains reachable (Lemma 3). The following proofs depend upon *release* and *acquire* ordering semantics where noted in the algorithms. These semantics ensure that all memory writes that happen before the release-ordered store (via $\texttt{store}(a, \cdot)$) become visible side-effects of an acquire-ordered load (via $\texttt{load}(a)$). Implementations must explicitly ensure this ordering.

Lemma 1. *The INSERT operation is linearizable.*

Proof. INSERT can modify a leaf in one of two ways: (1) by appending a value to a leaf, or (2) splitting the leaf into a branch. As such, there are two linearization point cases to make INSERT linearizable.

Case (1): INSERTs do not store new values until they have exclusive write access to a leaf, and thus no two INSERT operations will concurrently store a value into the same leaf. INSERT stores the new value one past the leaf's size limit before incrementing the size with a release-order store. Concurrent operations do not read values in a leaf past the leaf's size limit, thus storing the incremented size is the linearization point for this case.

Case (2): INSERT splits a leaf by replacing it with a new branch node with children populated from the values from the leaf. As INSERT locks the leaf before populating the branch's children, the same values will be present in both leaf and branch. INSERT replaces the pointer to the leaf with the pointer to the branch using a release-order store. Since concurrent operations will either load a pointer to the leaf before the store, or to the branch after the store, the store is the linearization point for this case.

Both cases have linearization points, and thus INSERT is linearizable. □

In case (2), unlike case (1), the leaf is not (necessarily) unlocked, as concurrent INSERT operations waiting for the leaf will load the new branch after the linearization point, and recurse to operate on a child of the new branch.

Lemma 2. *The NEAREST operation is wait-free, and concurrent INSERT operations do not cause incorrect operation.*

Proof. The NEAREST operation contains no blocking waits or retry loops, and thus will not wait on other operations. Correct operation under concurrency results from the two linearization points of NEAREST.

In case (1), when NEAREST visits a leaf, it first performs an acquire-order load of the leaf's size before iterating through the values in the leaf. As incrementing the size is the linearization point, NEAREST will only iterate through values in the leaf stored before the linearization point, and thus it will only traverse consistent data.

In case (2), when NEAREST recurses to search a child of a branch, it performs an acquire-order load of a pointer to the child. NEAREST will either load the pointer before or after the corresponding

linearization point of INSERT. If NEAREST loads the child before the linearization point, it will recurse to visit the leaf. If NEAREST loads the child after the linearization point, it will recurse to visit the branch. All nodes remain valid once reached, including leaf nodes that have been replaced by branch nodes, thus NEAREST will operate correctly under concurrency. □

Lemma 3. *Once a value is reachable by NEAREST, the value will remain reachable to all subsequent NEAREST operations.*

Proof. A value is first reachable after linearization point case (1) of INSERT. The leaf in which the value resides remains reachable until linearization point case (2) of INSERT. After the linearization point case (2), all values from the original leaf reside in the child nodes of the branch that replaced the original leaf. The originally reachable value thus remains reachable before and after linearization point case (2), and the value will thus *always* remain reachable to subsequent NEAREST operations. □

Lemma 4. *With uniform random insertion, an INSERT operation waits, or causes a wait, with probability* $(1 - ((n-1)/n)^{p-1})$*, where* p *is the number of concurrent INSERT operations and* n *is the number of leaf nodes in the tree. INSERT asymptotically almost surely does not wait.*

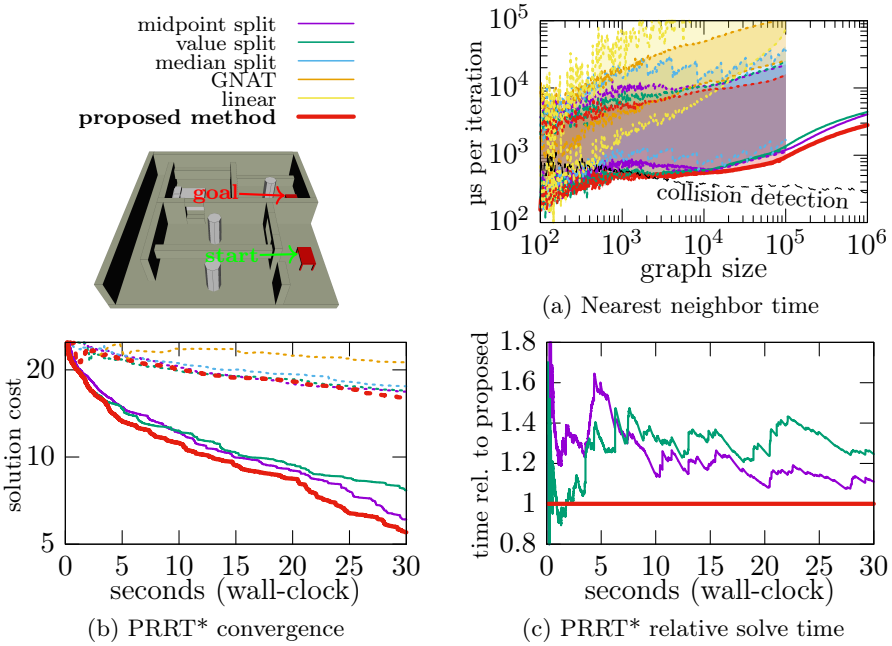
Proof. An INSERT will loop, and thus effectively wait on line 7, if a concurrent INSERT had a successful `try_lock` on the *same* leaf. Leaf nodes represent a bounded subregion of the space with uniform distribution. We cast this as the generalized birthday problem, and follow its derivation. Let $P(A_n)$ be the probability that an INSERT concurrently updates a leaf of the same bounded subregion as any of the other $(p-1)$ concurrent INSERTs. This is equivalent to $1 - P(A'_n)$, where $P(A'_n)$ is the probability that no other INSERT concurrently updates the same bounded region. We compute $P(A'_n)$ as the joint probability that $(p-1)$ INSERT operations are updating different regions. Thus,

$$P(A_n) = 1 - P(A'_n) = 1 - \left(\frac{n-1}{n}\right)^{p-1}.$$

It follows that $\lim_{n\to\infty} P(A'_n) = 1$, and thus INSERT asymptotically almost surely does *not* wait. □

4.5 Results

We evaluate the data structure by embedding it in PRRT* [49], the lock-free parallelized asymptotically optimal sampling-based motion planner introduced in chapter 2. The data structure

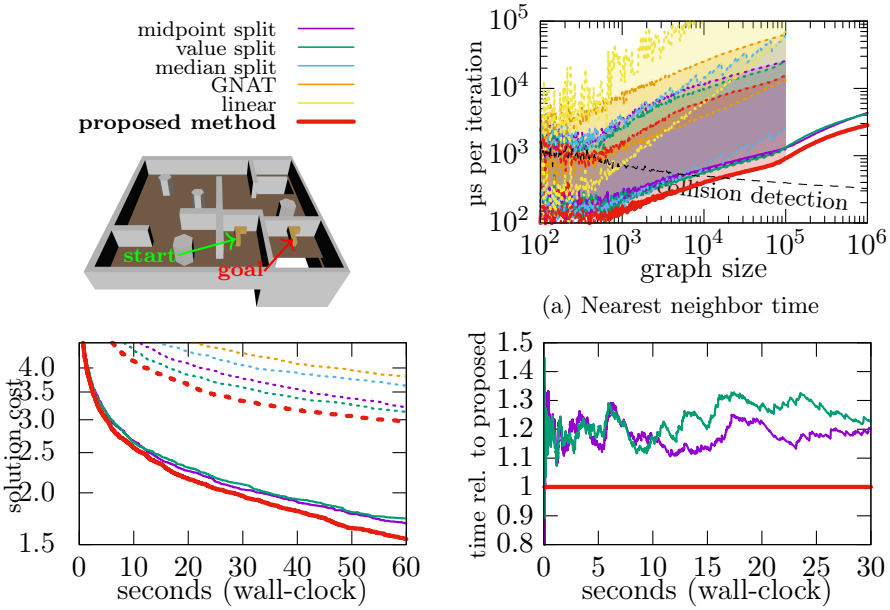


(a) Nearest neighbor time

(b) PRRT* convergence

(c) PRRT* relative solve time

Figure 4.4: **The proposed data structure speeds up parallelized motion planning in the "Home" SE(3) scenario from OMPL**. In this scenario, the motion planner finds a path for the red table to move from the lower right to the upper right configuration. The graph in (a) shows the time in microseconds spent performing nearest neighbor operations (insert, nearest, and k-nearest) relative to the size of the nearest neighbor structure. To illustrate relative impact on overall planner performance, the graph also shows the time spent in collision detection, which is typically the other dominant time consumer in sampling-based motion planners. For the locked versions of the nearest-neighbor structures, the time spent waiting for the lock is shown in the shaded area—the lower boundary of the region is the time spent performing a nearest neigbor operation, and the height of the region is the time the planner must wait for the nearest neighbor operation including the lock. Locked structures (dotted lines) become prohibitively expensive to benchmark past a graph size of 10^5. The graph in (b) shows the average path cost relative to the estimated optimal path cost as it converges over wall-clock time. The graph in (c) shows the time relative to the proposed method to compute the same solution cost—the proposed method finds the same solution 10% to 30% faster than previous lock-free methods.



(a) Nearest neighbor time

(b) PRRT* convergence

(c) PRRT* relative solve time

Figure 4.5: **Speeding up planning on OMPL's "Cubicles" scenario**. See description in Fig. 4.4.

and planner implementations use the standard C++ atomic library [55] for memory operations that require release and acquire semantics. PRRT* uses the proposed data structure for concurrent insert, nearest, and k-nearest operations. We have PRRT* compute motion plans in two SE(3) rigid-body scenarios from OMPL [112] on a computer with four Intel x7550 2.0-GHz 8-core Nehalem-EX processors, using all 32 cores.

The experiments compare both concurrent and locked nearest neighbor data structures to show the benefit of using data structures designed for concurrency. Locking on the data structure makes use of an efficient reader/writer lock, under the observation that insertions are relatively fast and infrequent compared to time spent nearest neighbor searching. Thus the locked version of the data structure is exclusively write-locked when inserting, and shared read-locked when searching. This prevents searches from traversing an inconsistent data structure that would result from partial mutations and reordered memory writes of a concurrent insert. It also allows multiple concurrent searches that only block when there is a concurrent insert.

We compare our proposed method to the linear (brute-force) implementation included in OMPL, the GNAT implementation included in OMPL, the dynamically rebalanced median-split kd-tree from prior work [46], and the original lock-free kd-trees in PRRT*. The OMPL methods and the median-split kd-tree method are read/write locked. The concurrent methods are also evaluated

in locked form. The implementations of the locked versions of the kd-trees do not make use of memory-ordering operations, and thus run slightly faster in the absence of concurrency. In all experiments, the leaf nodes of the proposed method are configured to have a capacity of 8.

Figures 4.4 and 4.5 show two evaluated scenarios involving motion planning for a robot in SE(3). In both scenarios the motion planner must find, and asymptotically optimize, a path for a rigid body robot through a 3D environment. The topological space for nearest neighbor searching is thus $\mathbb{R}^3 \times \mathrm{SO}(3)$. We set the SO(3) distance scale factor to $\alpha_{\mathrm{SO}(3)} = 100$, and leave the $\alpha_{\mathbb{R}^3} = 1$ (the default). The $\mathbb{R}^3$ space extends for hundreds of units, so this makes the two sub-topologies approximately evenly weighted. This weighting has two effects: (1) it makes rotations more expensive, thus as the motion planner converges, the robot rotates less freely than otherwise, and (2) it ensures that the kd-tree splits both $\mathbb{R}^3$ and SO(3) axes.

The figures 4.4 (a) and 4.5 (a) show the time spent in nearest neighbor operations (both inserts and searches) per sampling iteration based upon the size of the PRRT* graph (which is equivalent to the number of points in the nearest neighbor data structure). These graphs show both the time spent searching (bottom line of shaded regions) and the time spent waiting on a lock (shaded regions). We generate a data structure sizes up to 10^5 with the locked versions, stopping then because it becomes too time consuming to continue to the next order of magnitude. The concurrent versions of the kd-tree continue to 1 million. In both graphs we observe that our proposed method performs better than alternatives, even under high concurrency, with roughly half the time (the graph is log scaled) spent compared to the best alternatives.

To demonstrate the relative impact on the motion planner, the graph includes the time spent in collision detection—which typically is the other most time consuming part of a sampling-based motion planner. From the graphs, we observe the time spent in collision detection shrinks as its computation time is a function of shrinking expected distance between random samples. We observe that nearest neighbor operations eventually dominate the per-iteration time.

The figures 4.4 (b) and 4.5 (b) show the overall effect on convergence rate of the asymptotically optimal sampling-based planner. Due to the acceleration of each iteration, the motion planner is able to find lower-cost paths faster. The alternate presentation of the same data in 4.4 (c) and 4.5 (c), shows that the proposed method results in approximately 20% to 30% faster convergence of PRRT*.

4.6 Conclusion

This chapter presents and evaluates an exact nearest neighbor data structure that handles concurrent inserts and queries. Based on a kd-tree, the data structure supports searching nearest neighbors on topologies relevant to robotics. Building on the advancements in chapter 3, this chapter described how the concurrent data structure supports Cartesian products of an arbitrary number of Euclidean and SO(3) spaces with a distance metric that is the weighted sum of sub-topology components within the concurrent data structure.

In evaluation, the parallelized asymptotically optimal sampling-based motion planner from chapter 2 uses the proposed data structure from this chapter to further accelerate motion planning. Furthermore, the faster performance relative to the lock-based alternatives demonstrates the importance of having a concurrent data structure in parallel processing algorithms such as sampling-based motion planners that depend heavily on nearest-neighbor searching.

The fast and concurrent nearest neighbor data structure from this chapter embedded in a parallel sampling-based motion planner from chapter 2 introduces a challenge not typically present in sampling-based planners. The large number of samples the motion planner is able to rapidly generate causes the data structures to exceed the size of CPU's fast memory caches sooner. When the cache size is exceeded, the motion planner generates samples at a slower pace. This problem will be addressed in chapter 5.

CHAPTER 5

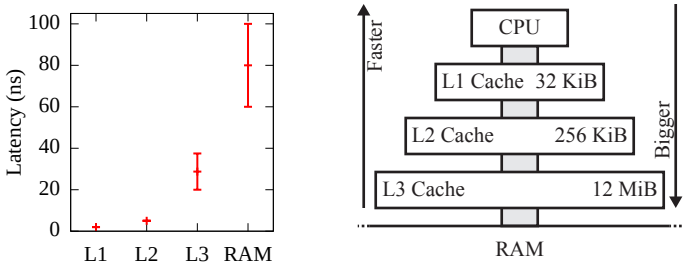
Cache-Aware Sampling-Based Motion Planning

In previous chapters we sped up incremental sampling-based motion planning through lock-free operation, faster and concurrent nearest neighbor data-structures, and scalable multi-core parallel processing. With this faster motion planning, incremental sampling-based algorithms are able to rapidly generate data structures that exceed the size of the CPU's cache—and when that happens their computation rate begins to slow down. In this chapter, we introduce CARRT*, "Cache-Aware Rapidly-exploring Random Tree (Star)," an asymptotically optimal sampling-based motion planner that significantly reduces motion planning computation time by effectively utilizing the cache memory hierarchy of modern central processing units (CPUs).

Modern CPUs can perform hundreds of computation instructions in the time that it takes to access a single value in memory (RAM) [78]. To reduce this disparity, CPUs have multiple levels of small and fast cache memories for storing frequently accessed data and avoiding they costly access time of RAM. When the CPU finds data in the cache (a cache hit), it uses the value from the cache and saves time by not accessing RAM. When the CPU does not find data in the cache (a cache miss), it stalls while waiting for the value in RAM and then populates the cache with the value for future use. Fig. 5.1 shows a typical modern CPU with three levels of cache: its L1 cache is the smallest and fastest (30–50× faster than RAM), L2 is bigger and not as fast (12–20× faster than RAM), and L3 is largest but slowest cache (though still 2–5× faster than RAM).

CARRT* is an asymptotically optimal sampling-based motion planner that is *cache-aware*—it takes into account the size of the cache to organize its computations in a manner that significantly increases the number of cache hits. We focus on two portions of the algorithm that have increasing memory complexity as the algorithm iterates: nearest neighbor searching and graph rewiring.

Nearest neighbor searching is a critical component of sampling-based motion planning, and the computational complexity grows with the number of sampled configurations in the motion planning graph. As the number of sampled configurations rises, the nearest neighbor search data structure



(a) CPU cache latency (b) CPU cache size

Figure 5.1: Example cache hierarchy a typical modern CPU—the same as used in Section 5.4 results. (a) Cache hit latency timings for different levels of the CPU cache hierarchy. (b) The cache levels are depicted graphically.

exceeds the capacity of the CPU's cache levels. The result is *cache misses* where the cache does not contain a requested value. As shown in Fig. 5.2, the impact of cache misses is significant; nearest-neighbor search times diverge from the trend seen when the data structure fits completely in L2 cache.

Rather than exploring anywhere in configuration space in every iteration as in RRT*, CARRT* focuses on exploring in distinct smaller regions of the configuration space for short periods of time. As CARRT* adds more configurations, it progressively subdivides regions to keep the working dataset under a preconfigured limit. By tuning the region size limit to match the characteristics of the problem and the CPU cache size, CARRT* works with a dataset that fits in the cache. Computation times thus become closer to what would be possible if RAM operated as fast as the cache, enabling significant improvements in motion planning performance.

RRT* and CARRT* incrementally converge towards optimality by *rewiring* the planning tree around configurations as they add them. Because CARRT* samples in regions, it would take longer for rewiring to have a global impact were it to follow the same rewiring approach of RRT*. We thus develop a rewiring strategy compatible with cache-aware region-based sampling and that accelerates computation of high quality motion plans.

We evaluate CARRT* in scenarios involving a point robot as well as the Rethink Robotics Baxter robot [99]. Our results show that the cache-aware approach of CARRT* outperforms non-cache-aware RRT*.

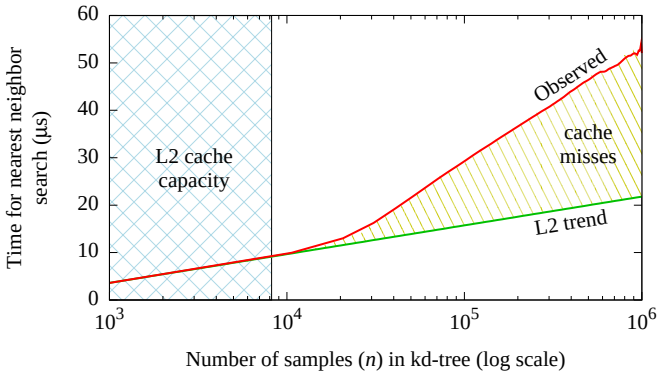


Figure 5.2: Nearest neighbor searching is a critical component of sampling-based motion planning. Proper use of the CPU's cache can lead to significantly faster nearest neighbor searches. As the number of configurations in the space rises, the memory required to store the nearest neighbor search data structure (e.g., a kd-tree) exceeds the capacity of the CPU's L2 cache. This results in L2 cache misses, and the associated latency causes the observed nearest neighbor search times to diverge from the trend seen when the data fits in L2 cache. In this chapter, we present a motion planner that is *cache-aware*—with a simple tunable parameter, it keeps its working dataset in the CPU cache. This results in computation times closer to the L2 cache trend line (in green) than the observed red line, enabling significant improvements in motion planning performance.

5.1 Related Work

CARRT* uses a cache-aware region-based sampling strategy. Non-uniform sampling in a sampling-based planner has been a subject of considerable research. Hsu et al. provide an overview of many sampling strategies in their approach that adaptively chooses among several samplers [42].

Sampling within a bounded region of the configuration space has been used to varying effects. RESAMPL [101] uses sampling to classify regions and then refine sampling within the regions based upon their classification to help solve difficult planning problems such as narrow passages. PRRT* [48] from chapter 2 uses a simple partitioning scheme to split computation across multiple cores and achieve superlinear speedup of RRT*. The fixed sampling region size from chapter 2 makes use of the fact that each core has some amount of independent low-level cache—by splitting sampling across cores, the net effect is that PRRT* multiplies the effective size of low-level cache by the number of cores in use. However, this effect only delays the cache-effects until that multiple runs out. This chapter shows how to keep the cache-based effect indefinitely—even in single-core operation. Jacobs et al. radially partition the space into regions to construct portions of the planning tree in parallel and increase the locality of the computation [58]. C-FOREST [93] samples from a bounded region defined by the length of the best known path and cost metric for which the triangle inequality

holds. This effective heuristic allows C-FOREST to only generate samples that have the possibility of improving the solution. KPIECE [114] prioritizes cells in a discretized grid for sampling based upon a notion of a cell's importance to solving a difficult portion of the planning problem. The planner of Burns et al. [18] biases samples towards regions of complexity, as defined by a locally weighted regression and active learning, to improve its ability to navigate narrow passages and other complex regions. Akgun et al. [3] use biased sampling to improve convergence towards optimality.

Varadhan et al. [118] eschew random sampling in favor of a deterministic recursive subdivision of free space into star-shaped partitions, which are then used to generate the roadmap. They use a recursive subdivision of space similar to that of a kd-tree [13], which is used in our method.

Sampling-based planners search nearest-neighbor data structures to find connection points for new samples. Cache-efficient data structures have been an area of active research for many years. Both [2] and [9] discuss the construction of a cache-efficient kd-tree for nearest-neighbor searches. They perform a one-time (i.e., "static") construction of the tree using a van Emde Boas layout [29] which preserves locality in hierarchical traversals (e.g., searches) of the tree. Our method requires the tree to be constructed and queried on-the-fly (i.e., "dynamic"), and methods like [14] can be used to convert static trees to dynamic. Yoon et al. [125] apply cache-efficient construction to bounding volume hierarchies (BVH) and describe how the BVH approach can be extended to kd-trees. They, too, use static construction of van Emde Boas layout and exploit access pattern localities typical of BVH applications (e.g., collision detection and ray tracing) and achieve from good to exceptional (26%–2600%) performance boost based upon the cache-efficient layout. Such methods create cache-efficient layouts for generalized searches whereas CARRT* gains cache-efficiency by constraining searches to a region of a kd-tree.

5.2 Problem Formulation

Let $\mathcal{C}$ be the bounded d-dimensional configuration space of a robot, and let $\mathcal{C}_{\text{free}} \subseteq \mathcal{C}$ be the subspace of $\mathcal{C}$ that is not in collision with any obstacle in the environment. Let $\mathbf{q} \in \mathcal{C}$ denote a configuration of the robot. The inputs $\mathbf{q}_{\text{init}} \in \mathcal{C}_{\text{free}}$ and $\mathcal{Q}_{\text{goal}} \subseteq \mathcal{C}_{\text{free}}$ are the robot's starting configuration and set of goal configurations, respectively.

The objective of the motion planner in this chapter is to compute a collision-free path through the configuration space that reaches the goal region while minimizing a user-specified cost function. We define the path as $\Pi : (\mathbf{q}_{\text{init}}, \mathbf{q}_1, \mathbf{q}_2, \ldots, \mathbf{q}_{\text{end}})$ through $\mathcal{C}_{\text{free}}$ where $\mathbf{q}_{\text{end}} \in \mathcal{Q}_{\text{goal}}$.

The computing platform is a CPU with a cache of limited size that provides low latency access to recently used values from RAM. When the CPU finds a value in the cache, it is a *cache hit*. Conversely, when the CPU does not find the value in the cache, it is a *cache miss*. The difference in latency between a cache hit and a cache miss is called the *cache miss penalty*. A performance objective of the planner is to minimize cache miss penalties by maintaining a working dataset that fits in the cache. Fig. 5.1 shows the sizes and latencies of the cache levels on a typical modern CPU.

As with other sampling-based motion planners, we require several functions as an input to define the planning problem. The function `STEER`$(\mathbf{q}_1, \mathbf{q}_2)$ returns a new configuration that would be reached when moving from $\mathbf{q}_1$ toward $\mathbf{q}_2$ up to some specified maximum distance. The function `FEASIBLE`$(\mathbf{q}_1, \mathbf{q}_2)$ returns `false` if the local path from $\mathbf{q}_1$ to $\mathbf{q}_2$ collides with an obstacle or violates a motion constraint and `true` otherwise. The function `COST`$(\mathbf{q}_1, \mathbf{q}_2)$ defines the cost associated with moving from $\mathbf{q}_1$ to $\mathbf{q}_2$ and can represent control effort, Euclidean distance, or any problem-specific cost function that can be used with RRT* [60].

5.3 The CARRT* Algorithm

At a high level, CARRT* is an iterative algorithm that builds a motion planning tree with a similar strategy to RRT* [60]. The key difference is that, rather than exploring anywhere in configuration space in every iteration, CARRT* focuses on exploring in distinct smaller regions of the configuration space for short periods of time so as to keep the working dataset small enough to fit in the CPU caches. We call the region being sampled the *active sampling region*.

The planner starts by queuing up a sampling region equal to the problem's configuration space bounds. It then dequeues the active sampling region and samples within the region. Once the region reaches a threshold number of configurations, the planner splits the region in half, queues up the two smaller regions, and repeats the process. The region threshold is tuned to keep the working dataset for a region within the CPU's cache.

CARRT*'s approach to repeatedly splitting configuration space regions in half to create smaller regions naturally synergizes with the kd-tree nearest neighbor search data structure. As such, the planner uses a kd-tree that is explicitly integrated with the region-based sampling. Each active and queued sampling region represents the root of a subtree in the kd-tree—the same subtree that will be explored and expanded during sampling.

CARRT* builds a motion planning tree $G = (V, E)$ with a similar strategy to RRT*. The tree is rooted at the robot's initial configuration. The set of vertices V corresponds to feasible configurations. The directed edge list E defines a tree with the best known feasible paths from the initial configuration to the configurations in V. Each iteration of CARRT* randomly samples a configuration from the active sampling region, and if `FEASIBLE`, adds the sample to V and an edge to E. Then, within a radius around the new sample, the planner rewires edges in E, replacing longer edges with shorter ones while maintaining the above invariants.

Our planner maintains a second graph $G' = (V, E')$ which shares V from the tree in G and has an *undirected* edge list E' of nearest neighbors of each configuration. This graph is used in the rewiring step discussed in Section 5.3.4.

5.3.1 Sampling Region Queue

CARRT*'s outer loop is shown in Algorithm 15. It starts by initializing the data structures and setting the root of the tree to the robot's initial configuration $\mathbf{q}_{\text{init}}$ (line 1). We initialize the sampling region queue `Q` in line 3 to have a single region with the bounds of the configuration space $[\mathbf{C}_{\min}, \mathbf{C}_{\max}]$.

The priority queue `Q` ensures even sampling coverage by defining the highest priority region as the region with the lowest *sample density*:

$$\texttt{Density}(\mathbf{r}) = \frac{(\text{samples considered in region } \mathbf{r})}{(\text{volume of region } \mathbf{r})}.$$

In the outer loop, the planner removes the highest priority region from the queue to make it the active sampling region $\mathbf{r}$ (line 5). Using the function `PlanRegion`($\mathbf{r}$) (Sec. 5.3.3), the algorithm samples and extends the active sampling region for a short period of time. CARRT* then determines if $\mathbf{r}$ exceeds the threshold tied to the CPU cache size (line 7). If `PlanRegion`($\mathbf{r}$) terminated before exceeding the threshold, the planner re-queues the region with its increased sample count and thus lower priority (line 8). Otherwise, $\mathbf{r}$ grew to exceed the region limit, and the planner splits it along an axis shared by the kd-tree (Sec. 5.3.2) and adds each new region to the queue (lines 10–13). Since CARRT* uniformly samples within a region, we assign half the sample count in $\mathbf{r}$ to each of the new regions. With half the samples, and half the volume, the new regions have the same sample density as $\mathbf{r}$. If, after splitting a region, the resulting child regions still have the highest priority, the planner

Algorithm 15 CARRT*

1: $V \leftarrow \{\mathbf{q}_{\text{init}}\}, E \leftarrow \varnothing, E' \leftarrow \varnothing$
2: $\texttt{Q} \leftarrow$ empty priority queue
/* "Q top" is the region with highest priority in Q */
3: add initial region $[\mathbf{C}_{\min}, \mathbf{C}_{\max}]$ to Q
4: **while not** done **do**
5: $\quad \mathbf{r} \leftarrow$ remove Q top
6: $\quad \texttt{PlanRegion}(\mathbf{r})$
7: $\quad$ **if** $\texttt{ConfigCount}(\mathbf{r}) <$ (region config limit) **then**
8: $\quad\quad$ add $\mathbf{r}$ back to Q
9: $\quad$ **else**
10: $\quad\quad (\mathbf{r}^{\text{left}}, \mathbf{r}^{\text{right}}) \leftarrow$ split $\mathbf{r}$ region in half along $\mathbf{r}_{\text{axis}}$
11: $\quad\quad \mathbf{r}^{\text{left}}_{\text{sample\_count}} \leftarrow \frac{1}{2}\mathbf{r}_{\text{sample\_count}}$
12: $\quad\quad \mathbf{r}^{\text{right}}_{\text{sample\_count}} \leftarrow \frac{1}{2}\mathbf{r}_{\text{sample\_count}}$
13: $\quad\quad$ add $\mathbf{r}^{\text{left}}$ and $\mathbf{r}^{\text{right}}$ to Q

immediately dequeues one of the new regions and avoids the cache miss penalties that would result from moving to a different region.

Each iteration of the outer loop removes one region from the queue and adds one or two new regions back. Hence, the queue will never be empty at the beginning of each iteration.

We note that priority queues are not well known for being cache efficient. Their use in CARRT* however, coincides with when the motion planner has filled the cache and thus would be expected to experience a few cache misses. Their use is also a small portion of the overall compute time, and is thus bounded at each top-level iteration to a few $O(\log n)$ operations. This property suggests that the motion planning algorithm should be tuned operate within a region for as long as possible before moving to another sampling region, in order to maximize the cache-based performance benefit while avoiding the periodic cache misses induced by moving to another region.

5.3.2 Integrated KD-Tree

For efficient nearest neighbor searches, CARRT* uses a kd-tree that is integrated with the region-based sampling. A kd-tree is a hierarchical space-partitioning data structure in which branch nodes successively subdivide regions of space by hyperplanes [13, 81]. The subdivisions on the path from the root to any node in the kd-tree define an implicit bounding box for a node. In CARRT*, a kd-tree node's bounding box also represents a sampling region of $\mathcal{C}$-space—it may be the active sampling region, a queued sampling region, a previously split region, or a region that may be queued in the future.

Algorithm 16 adds a configuration $\mathbf{q}$ to the kd-tree. The kd-tree nearest neighbor search (`Nearest`($\mathbf{q}$)) and fixed-radius nearest neighbor search (`Near`($\mathbf{q}, \mathtt{r}$)) follow a similar traversal strategy.

The bounds of each node are implicitly defined by the bounds of $\mathcal{C}$ and the node's position in the tree. Line 1 copies bounds of $\mathcal{C}$ into $[\mathbf{c}_{\min}, \mathbf{c}_{\max}]$ defining the bounds of the root node. The loop (line 3) traverses one level deeper in the kd-tree at every iteration, each time dividing the bounding box in $\mathbf{c}$ in half by a hyperplane defined along an implicit axis (lines 5, 6). After determining which side of the split to follow (line 7), the algorithm updates the bounding box (lines 11, 13) to reflect the split.

The axis and the split point are defined to be consistent with the splitting done in Algorithm 15. In our approach the axis is (depth of the node) modulo (dimensions of $\mathcal{C}$-space), and the split is at the midpoint of the node/region's bounding box (line 6).

The traversal loop stops once it has found a node in the tree without a configuration (line 3). The algorithm then adds the configuration $\mathbf{q}$ to the tree (line 14) before returning. The terminal node can be generated in one of two places: (1) the `KD_Insert` algorithm when the left or right child node to traverse is `nil` (lines 8, 13), or (2) in Algorithm 15, when a sampling region is split and a region is empty.

The kd-tree tracks the number of configurations in each subtree (line 4) as configurations are added to it. Algorithm 15 uses the subtree's size (and thus the sampling region's size) to determine when a sampling region needs to be split.

5.3.3 Planning Within a Region

CARRT* samples the active region using the inner loop of RRT* modified to run in a cache-aware manner, as shown in Algorithm 17. The notable changes from RRT* are: (1) it has additional loop termination conditions necessary to keep the working dataset small enough to fit in the CPU's cache (line 1); (2) it generates samples from a region of the sampling space (line 2); (3) the nearest neighbor ball radius computation uses a region-based approximation of the sample count (lines 7–8); (4) it tracks the sample count (line 3) to compute the sample-density metric used in the priority queue; and (5) the rewiring strategy accounts for samples being added in regions.

The stopping conditions are specified in line 1. The first criterion ("done") represents typical planning termination checks, e.g., a computation time limit or desired plan cost achieved. The second termination criterion of "`ConfigCount`($\mathbf{r}$) < (region config limit)" checks that the number

Algorithm 16 `KD_Insert`($\mathbf{q}$)

```
1: [c_min, c_max] ← [C_min, C_max]
2: n ← kd_root
3: while n_config is not nil do
4:    n_size ← n_size + 1
5:    axis ← next axis
6:    split ← ½(c_min[axis] + c_max[axis])
7:    if q[axis] < split then
8:       if n_left is nil then
9:          n_left ← new node with config = nil
10:      n ← n_left
11:      c_max[axis] ← split
12:   else
13:      follow n_right similar to n_left above,
         updating c_min instead
14: n_config ← q
```

of configurations in the subgraph contained within the region is smaller than the cache-based limit. The third criterion, "not out of time", sets up a time limit to ensure that CARRT* does not work indefinitely in obstructed or disconnected regions as such regions might otherwise never meet the second stopping condition. In the results section, "out of time" limits the number of samples considered in a region to 1024 (8× the region configuration limit), although other criteria, such as elapsed time, may be used.

In line 2, CARRT* generates a sample in the active sampling region—localizing the computation to the region. The planner finds the random sample's nearest neighbor, and computes $\mathbf{q}_{\text{new}}$ as the result of `STEER`ing towards the random sample (line 5). If the path between $\mathbf{q}_{\text{new}}$ and the nearest neighbor is feasible, CARRT* searches for samples in a ball-radius of $\mathbf{q}_{\text{new}}$. The ball-radius from [60] is computed using the dimensionality of the space d, two tunable parameters γ and η, and the number of configurations in the motion planning graph $|V|$. As CARRT* updates different regions of the space at different times, $|V|$ may be inconsistent with the portion of the graph in the active sampling region. In line 7, the algorithm computes an approximation of $|V|$ in the current region based upon the full motion graph size scaled by the volume ratio of the region to the volume of $\mathcal{C}$. The resulting approximation is fed into the ball radius computation (line 8) in place of the full RRT* graph size.

Algorithm 17 `PlanRegion`($\mathbf{r}$)

1: **while not** done
   and `ConfigCount`($\mathbf{r}$) < (region config limit)
   and not out of time **do**
2: $\mathbf{q}_{\text{rand}} \leftarrow$ random sample from $\mathbf{r}$
3: $\mathbf{r}_{\text{sample\_count}} \leftarrow \mathbf{r}_{\text{sample\_count}} + 1$
4: $\mathbf{q}_{\text{nearest}} \leftarrow \texttt{Nearest}(\mathbf{q}_{\text{rand}})$
5: $\mathbf{q}_{\text{new}} \leftarrow \texttt{STEER}(\mathbf{q}_{\text{nearest}}, \mathbf{q}_{\text{rand}})$
6: **if** $\texttt{FEASIBLE}(\mathbf{q}_{\text{nearest}}, \mathbf{q}_{\text{new}})$ **then**
7: $\mathtt{n}_{\text{approx}} \leftarrow \texttt{ConfigCount}(\mathbf{r}) \times \frac{\texttt{Volume(root)}}{\texttt{Volume}(\mathbf{r})}$
8: $N \leftarrow \texttt{Near}(\mathbf{q}_{\text{new}}, \min\{\gamma \left(\frac{\log \mathtt{n}_{\text{approx}}}{\mathtt{n}_{\text{approx}}}\right)^{1/d}, \eta\})$
9: $N_{\texttt{feasible}} \leftarrow \{\mathbf{q} \mid \mathbf{q} \in N \wedge \texttt{FEASIBLE}(\mathbf{q}, \mathbf{q}_{\text{new}})\}$
10: $\mathbf{q}_{\min} \leftarrow \underset{\mathbf{q} \in N_{\texttt{feasible}}}{\operatorname{argmin}} \ \texttt{PathCost}(\mathbf{q}) + \texttt{COST}(\mathbf{q}, \mathbf{q}_{\text{new}})$
11: $E \leftarrow E \cup (\mathbf{q}_{\text{new}}, \mathbf{q}_{\min})$
12: **for all** $\mathbf{q}_{\text{near}} \in N_{\texttt{feasible}} \setminus \mathbf{q}_{\min}$ **do**
13: $\mathtt{c}'_{\text{near}} \leftarrow \texttt{PathCost}(\mathbf{q}_{\text{new}}) + \texttt{COST}(\mathbf{q}_{\text{new}}, \mathbf{q}_{\text{near}})$
14: **if** $\mathtt{c}'_{\text{near}} < \texttt{PathCost}(\mathbf{q}_{\text{near}})$ **then**
15: $E \leftarrow E \setminus (\mathbf{q}_{\text{near}}, \texttt{Parent}(\mathbf{q}_{\text{near}}))$
16: $E \leftarrow E \cup (\mathbf{q}_{\text{near}}, \mathbf{q}_{\text{new}})$
17: $\texttt{CARRT*Update}(\mathbf{q}_{\text{near}})$
18: $E' \leftarrow E' \cup \{\{\mathbf{q}_{\text{new}}, \mathbf{q}_{\text{near}}\} \mid \mathbf{q}_{\text{near}} \in N_{\texttt{feasible}}\}$
19: $V \leftarrow V \cup \mathbf{q}_{\text{new}}$
20: $\texttt{KD\_Insert}(\mathbf{q}_{\text{new}})$

CARRT*, like RRT*, adds the new configuration to G by linking it to the configuration in the ball radius that produces the shortest path (line 10). The planner then rewires the other configurations in the ball-radius through the new configuration if the rewired path is shorter and feasible (lines 11–17).

5.3.4 Rewire Update Strategy

Rewiring in RRT* only considers neighboring configurations in the ball-radius of the new sample $\mathbf{q}_{\text{new}}$. When a neighbor $\mathbf{q}_{\text{near}}$ is rewired through $\mathbf{q}_{\text{new}}$, it can create an opportunity for a neighbor of $\mathbf{q}_{\text{near}}$ to be rewired as well (and of the neighbors' neighbors and so on). RRT* will efficiently propagate such a cascade with future random samples generated from $\mathcal{C}$. If CARRT* followed the same update strategy, the cascade would only be percolated after sampling from a sequence of regions, and thus produce a slower convergence to optimality.

CARRT* takes a different rewiring approach than RRT* to account for this cascade behavior, shown in Algorithm 18. This algorithm is invoked from the main sampling loop of CARRT* (see Algorithm 17) every time it rewires an existing node in the RRT* tree to a better path. It performs

Algorithm 18 `CARRT*Update`($\mathbf{q}$)

```
1: children ← queue with q
2: while not children is empty do
3:    q ← remove first from children
4:    for all q_near | {q_near, q} ∈ E′ do
5:       c′ ← PathCost(q) + COST(q, q_near)
6:       if c′ < PathCost(q_near) then
7:          E ← E \ (q, Parent(q))
8:          E ← E ∪ (q, q_near)
9:          append q_near to children
```

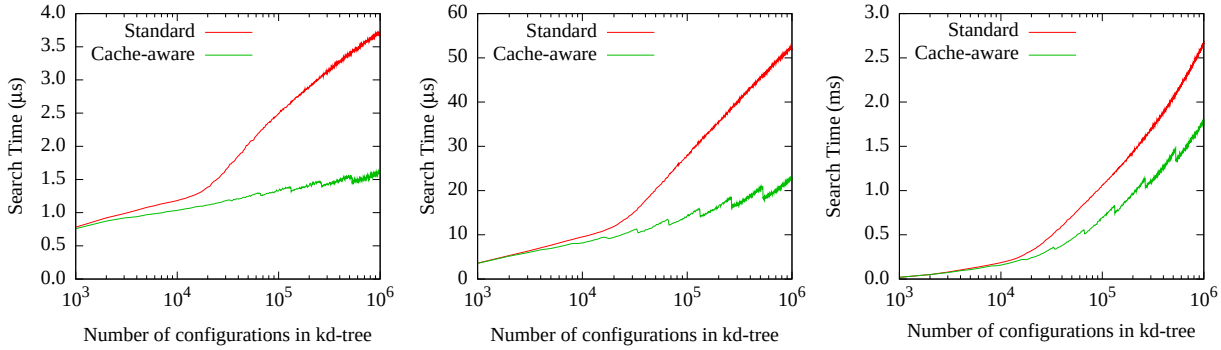
a breadth-first traversal of the subtree rooted in the rewired node, rewiring as it goes. The traversal is managed by a FIFO queue, initialized to contain only the root of the rewired subtree (line 1). It then repeatedly dequeues the first node until the queue is empty (line 2, 3). For every configuration $\mathbf{q}$ visited by the traversal, the algorithm visits all of $\mathbf{q}$'s previously computed nearest neighbors as stored in E' (line 4). If the neighboring child's path through $\mathbf{q}$ is shorter than its existing path (line 5, 6), it is rewired (line 7, 8) and added to the queue (line 9) to continue the process of percolating the updates through the subtree.

5.4 Results

We first evaluate the performance impact on nearest neighbor searches using CARRT*'s region-based sampling in an obstacle-free environment. We then compare CARRT* to RRT* in scenarios involving a point robot and the Rethink Robotics Baxter robot performing a task using 7 degrees of freedom (DOF). Plans are computed on an Intel X5670 2.93 GHz 6-core Westmere processor. Each processing core has a 32 KiB L1 data cache, 256 KiB private L2 cache, and 12 MiB shared L3 cache. The cache-line size is 64 bytes. CARRT* as presented in this chapter is not multi-threaded in order to demonstrate the cache-based benefits can apply to on single-core systems as well, and thus only utilizes 1 core of the processor.

5.4.1 KD-Tree Cache Impact

We first evaluate the performance impact of the planner's cache-aware sampling strategy on nearest neighbor searches. We create obstacle-free environments for a point robot in 3, 7, and 14 dimensional space. We compute the average time for a nearest neighbor search with $n = 10^3$ to 10^6 configurations in the kd-tree and plot the results in Fig. 5.3.



(a) 3 DOF kd-tree (b) 7 DOF kd-tree (c) 14 DOF kd-tree

Figure 5.3: Average time for a single nearest neighbor search with increasing kd-tree size (n). We search kd-trees using CARRT*'s *cache-aware* region-based sampling and using *standard* uniform random sampling. The kd-tree is 3, 7, and 14 DOF in (a), (b), and (c) respectively, showing the effect of dimensionality on performance. The divergence between $n = 10,000$ and $20,000$ occurs as the tree exceeds the size of the L2 cache.

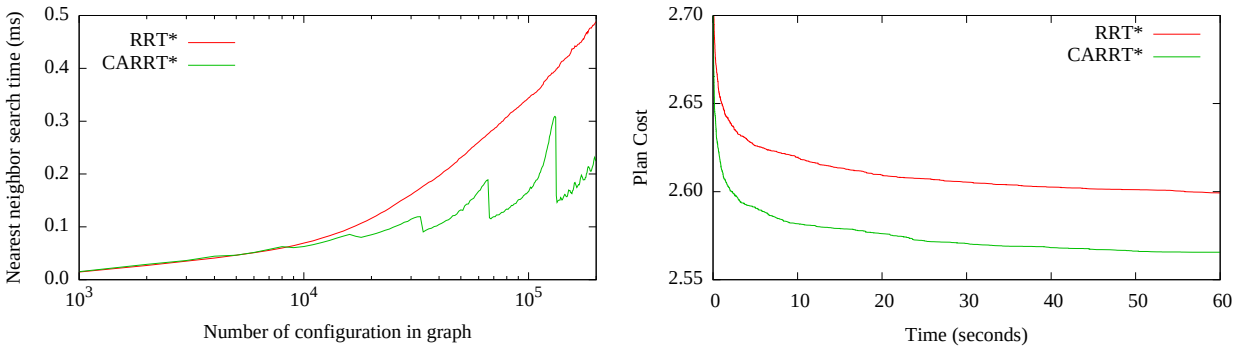
With a log scale x-axis and the theoretic $O(\log n)$ performance of searches, we expect to see a straight-line trend on the graph. In the three plots we observe the non-cache-aware approach has an approximately straight-line trend up to n_t=10,000–20,000, and then a steeper straight-line trend after. In our implementation, the kd-tree node occupies 32 bytes, and with a 256 KiB L2 cache, the cache can hold 8192 kd-tree nodes. As the height of the tree also grows logarithmically with n, we expect to see a change in the performance trend at twice the L2 cache capacity. The observed n_t matches this expectation.

At lower dimensions (Fig. 5.3 (a)), the cache-aware approach of CARRT* roughly follows the trend line established before the capacity of L2 cache is exceeded—a nearly ideal result. This enables a 3× performance improvement at $n = 10^6$. The cache-aware approach retains an improvement, though diminishing, for higher dimensions (Fig. 5.3 (b)-(c)).

5.4.2 7 DOF Ball Obstacle

We consider a scenario in which a point robot must move from one corner of a 7 dimensional cube to the opposite corner while avoiding a spherical obstacle placed at the center of the cube. We run both CARRT* and standard RRT* for comparison. The spherical obstacle implies that an optimal plan can only be found in the limit.

In Fig. 5.4 (a), we show the average time to run a single nearest neighbor search for a given number of samples in CARRT*'s roadmap. We observe that at approximately 8,000 samples, the



(a) Nearest neighbor search time vs. number of configurations in the kd-tree

(b) Plan cost vs. CARRT* and RRT* run times

Figure 5.4: **CARRT* and RRT* compute plans for the 7 DOF ball obstacle scenario**. The average time to complete a single nearest neighbor search is shown in (a). The average plan cost computed with a given wall-clock runtime is shown in (b).

performances of RRT* and CARRT* diverge. CARRT*'s nearest neighbor search time always remains below the non-cache-aware RRT* approach.

In Fig. 5.4 (b), we show the average path cost obtained after running the algorithm a given amount of wall-clock time. On average, CARRT* finds a lower cost plan than RRT* at all times. When viewing Fig. 5.4 (b) from the perspective of time to reach the same path cost, CARRT* finds a plan at 2.3 s of comparable cost to the plan RRT* finds at 60 s—approximately 26 times faster.

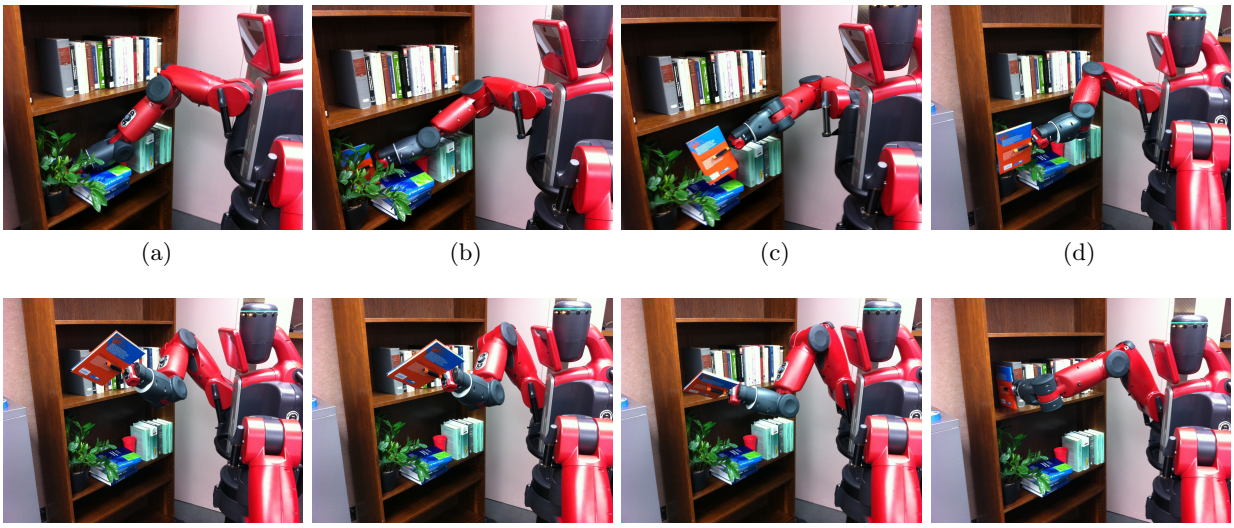
5.4.3 Baxter Robot 7 DOF Task

We give a Rethink Robotics Baxter robot the task of moving a book from behind a plant on a shelf to its proper spot on the shelf above, as shown in Fig. 5.5. The scenario requires the Baxter to move its 7 DOF arm through narrow passages both at the beginning of the task and at the end.

We ran CARRT* and RRT* on the Baxter robot 7 DOF scenario. Fig. 5.6(a) plots the average nearest neighbor search time as a function of number of states in the graph, with the x-axis on a log scale. Both CARRT* and RRT* initially start on the same trend line. Between 4,000 and 6,000 samples, RRT* diverges to a slower trend, whereas CARRT* more closely follows the original trend. Fig. 5.6(b) shows that CARRT* produces lower cost plans faster. CARRT* produces the same plan cost at approximately 90 s as RRT* produces in 180 s, a 2× improvement.

5.5 Conclusion

In this chapter, we presented CARRT* (Cache-Aware RRT*), a cache-aware sampling-based asymptotically optimal motion planner. By progressively partitioning the sampled space into regions



(a) (b) (c) (d)

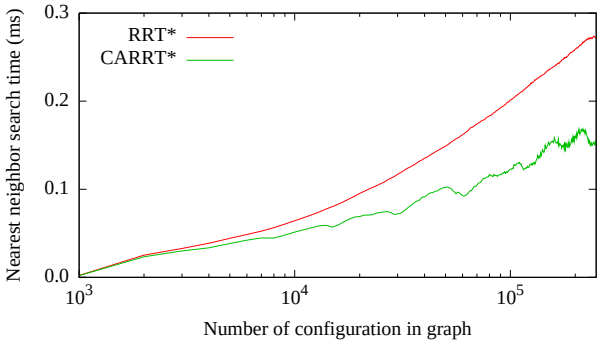
(e) (f) (g) (h)

Figure 5.5: The Baxter robot moves a book located behind a plant to its proper place on the shelf above while avoiding obstacles in the cluttered environment. This is a 1-arm, 7 DOF task with a narrow passage to remove the book from behind the plant and another narrow passage to place the book between two books on the shelf above.

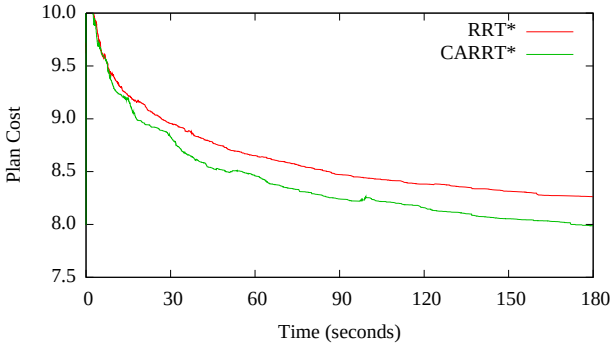
that fit into the CPU's cache, CARRT* is able to keep its working dataset for nearest neighbor searches in the CPU cache and avoid delays associated with cache miss penalties. CARRT* also rewires the motion planning graph in a manner that complements the cache-aware subdivision strategy to more quickly refine the motion planning graph toward optimality. We demonstrated the performance benefit of our cache-aware motion planning approach for scenarios with a point robot and the Rethink Robotics Baxter robot.

While this chapter focused on making a RRT* sampling-based motion planner cache-aware, the benefits can likely be extended to many other sampling-based motion planning algorithms. Since the underlying sampling region data-structure is based on a kd-tree, CARRT* can be integrated with the approaches from chapters 3 and 4 which would extend the benefits to rotational spaces and multi-core concurrent motion planners such as that in chapter 2.

With the tools from these chapters, we are able to generate motion plans quickly, but are limited by the amount of computing power on the robot or connected to the robot by a fast, low-latency network. In the next chapter, we overcome this limitation by moving motion planning to the vast computing power available in the cloud while overcoming the associated network bottlenecks.



(a) Nearest neighbor search time vs. number of configurations in the kd-tree



(b) Plan cost vs CARRT* and RRT* run times

Figure 5.6: CARRT* and RRT* compute plans for the Baxter 1-arm 7 DOF scenario. The average time to complete a single nearest neighbor search is shown in (a). The plan cost after a given wall-clock runtime is shown in (b).

CHAPTER 6

Cloud-based Motion Planning in Dynamic Environments

Robots operating in dynamically evolving environments with moving obstacles and changing goals need to be able to compute motion plans rapidly in order to continually avoid obstacles while moving toward their goal. In previous chapters, we sped up motion planning using lock-free operations, fast and concurrent nearest neighbor data structures, cache-aware operation, and parallel multi-core operations. Even with these advancements, the CPUs on board some robots, due to the robot's degrees of freedom, physical size, and/or power source, may not be capable of computing motion plans fast enough to interact effectively and safely with a dynamically evolving environment. When this is the case, we can look to offloading some or all of the motion planning computation to a network-attached high-performance computer. One such source of computing, with many economic benefits (as described in chapter 1), is the cloud. The cloud, however is accessed through a network that introduces bottlenecks on communication in the form of round-trip latency and bandwidth limits. These network bottlenecks must be taken into account when computing motion plans, especially when operating in dynamic environments where reaction time is critical. In this chapter, we present algorithms for a robot and a computer in the cloud that allow a robot to effectively utilize the cloud in order to dramatically improve its capabilities when operating in a dynamic environment.

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 one second of 360 cores of computation can be less than $0.0047 [5]. This implies that with an embarrassingly parallel algorithm [4], 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 one core for 360 seconds, or 360 cores for one 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 [98], and as such, if the mobile robot has more than a few degrees of freedom, its computational demands for motion planning can quickly exceed its available onboard 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 chapter 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.

6.1 Related Work

The NIST definition of cloud computing [84], 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

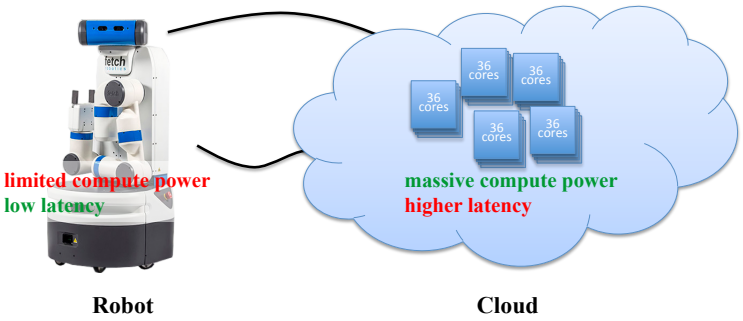


Figure 6.1: Comparison of robot only and cloud computing for robot motion planning. The robot has limited computing power in order to reduce weight and increase battery life, however it has low latency access to its sensors and actuators. The cloud-computing has vast amounts of on-demand computing power available, but has a higher latency access to the robot and the information it sends.

robot learning, and remote human interaction. Kehoe et al. provides an excellent survey of cloud-robotics in [62].

In this chapter we focus on cloud-computing as an on-demand high-performance computing platform to accelerate motion planning. Bekris et al. [12] 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 [64], 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 [63], 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× speedup.

Parallel processing has been successfully used to accelerate motion planning computations. In [4], 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 chapter 2 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 [19] 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 [93] 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 [102] 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) [61] 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* [60] method improves upon PRM by defining a connectivity level (k) needed to guarantee asymptotic optimality.

Sparse roadmaps and roadmap spanners such as SPARS [25] 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 [82] 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* [79] and Anytime D* Lite [80] algorithms use an inadmissible heuristic in A* to find a path quickly, then incrementally improve the plan in subsequent iterations.

6.2 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 $\boldsymbol{\tau} = \left(\left[\mathbf{q}_0^\mathsf{T}\ t_0\right]^\mathsf{T}, \left[\mathbf{q}_1^\mathsf{T}\ t_1\right]^\mathsf{T}, \ldots, \left[\mathbf{q}_n^\mathsf{T}\ t_n\right]^\mathsf{T} \right)$, 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.

6.3 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 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.

Algorithm 19 Robot Computation

Require: 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}

6.3.1 Roadmap-Based Robot Computation

The robot's algorithm is shown in Alg. 19. 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. 19 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 [80], with a time component as described in [117]. 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. 19 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.

6.3.2 Roadmap-Based Cloud Computation

Cloud-based computation in out 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

Algorithm 20 Cloud Planning Session

1: $\mathbf{G} = (\mathbf{V}, \mathbf{E}_\text{s} \subseteq \mathbf{E}) \leftarrow (\varnothing, \varnothing)$
2: $\mathbf{G}_\text{robot} = (\mathbf{V}_\text{robot}, \mathbf{E}_\text{robot}) \leftarrow (\varnothing, \varnothing)$
3: $\mathcal{W} \leftarrow \varnothing$ {static obstacles}
4: **loop**
5: **recv** `plan_req`$(t_\text{req}, \mathbf{q}_\text{req}, \mathcal{W}, \mathbf{Q}_\text{goal}) \Leftarrow$ robot {blocking wait for next request}
6: $\mathbf{V} \leftarrow \{\mathbf{q} \in \mathbf{V} \cup \{\mathbf{q}_\text{req}\} \mid \forall w \in \mathcal{W} : \texttt{clear}(\mathbf{q} \mid w)\}$
7: $\mathbf{E} \leftarrow \{(\mathbf{q}_a, \mathbf{q}_b) \in \mathbf{E} \mid \forall w \in \mathcal{W} : \texttt{link}(\mathbf{q}_a, \mathbf{q}_b \mid w)\}$
8: **while** $t_\text{now} < t_\text{req} - t_\text{res}$ **and not** satisfactory solution **do**
9: update $\mathbf{G}$ and $\mathbf{q}_\text{goal}$ using k-PRM*+IRS on $\mathcal{W}$ and $\mathbf{Q}_\text{goal}$
10: $(\mathbf{V}'_\text{robot}, \mathbf{E}'_\text{robot}) \leftarrow$ `serialize_graph`$(\mathbf{G}, \mathbf{q}_\text{req}, \mathbf{Q}_\text{goal}, \mathbf{G}_\text{robot})$
11: **send** `plan_res`$(\mathbf{V}'_\text{robot} \setminus \mathbf{V}_\text{robot}, \mathbf{E}'_\text{robot} \setminus \mathbf{E}_\text{robot}) \Rightarrow$ robot
12: $(\mathbf{V}_\text{robot}, \mathbf{E}_\text{robot}) \leftarrow (\mathbf{V}'_\text{robot}, \mathbf{E}'_\text{robot})$

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. 20 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. 6.3.3) 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. 20 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.

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. 19 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 6.3.4, 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

Algorithm 21 Lock-free Parallel k-PRM* IRS Thread

Require: $G = (V, E)$ is an initialized graph shared between threads, $\exists v \in V$: `is_goal`(v)

```
1: while not done do
2:    v_rand ← new vertex with random sample and connected component C_rand
3:    C_rand.goal ← is_goal(v_rand)
4:    if clear(v_rand) then
5:       for all v_near ∈ k_nearest(V, v_rand, {k =}⌈log (|V| + 1) * k_RRG⌉) do
6:          if link(v_rand, v_near) then
7:             sparse ← shortest_path_dist(v_rand, v_near) < w_stretch * dist(v_rand, v_near)
8:             add_edge(v_rand, v_near, sparse)
9:             add_edge(v_near, v_rand, sparse)
10:            solved ← solved or merge_components(v_rand.cc, v_near.cc)
11:      V ← V ∪ v_rand
```

the cloud computer may continue to update the roadmap in the background until it receives another `plan_req` from the robot.

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

The cloud-based service computes a roadmap using k-PRM* [60] with the Incremental Roadmap Spanner (IRS) [82], sped up by a lock-free parallelization construction we introduce in this section, and based on the concepts in chapters 2 and 4. 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 ($\mathbf{E}_\mathrm{s} \subseteq \mathbf{E}$).

The server computes k-PRM*+IRS using a parallel lock-free algorithm in which all provisioned cores run Alg. 21 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: (1) nearest neighbor searching is fast and non-blocking due to the use the lock-free kd-tree described in [49], (2) graph edges are stored in lock-free linked lists (Alg. 22), and (3) progress towards a solution is tracked via connected components that are stored in lock-free linked trees (Alg. 23). As with k-PRM*, in each iteration this algorithm generates a random robot configuration and searches for its k-nearest neighbors using k from [60]. 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 22 `add_edge`($v_{\text{from}}, v_{\text{to}}$, sparse)

```
1: e_dense ← new edge to v_to with e_dense.next = v_from.dense_list_head
2: while not CAS(v_from.dense_list_head, e_dense.next, e_dense) do
3:    e_dense.next ← v_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 ≠ nil do
7:    v_from.cc ← v_from.cc.parent {Lazy update of vertex's connected component}
```

The algorithm adds edges to the graph using Alg. 22. 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.

Algorithm 23 `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
```

The algorithm tracks progress towards a solution by maintaining information on each connected component ("cc" in Alg. 22) in the roadmap. When it adds an edge between two vertices, it also merges the connected components associated with the vertices (Alg. 23). 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

Algorithm 24 `serialize_graph`$(\mathbf{G}, \mathbf{q}_{\text{req}}, \mathbf{Q}_{\text{goal}}, \mathbf{G}_{\text{robot}})$

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

1: $(\mathbf{V}'_{\text{robot}}, \mathbf{E}'_{\text{robot}}) \leftarrow (\mathbf{V}_{\text{robot}}, \mathbf{E}_{\text{robot}})$
2: $\mathbf{V}_{\text{frontier}} =$ `forward_frontier`$(\mathbf{q}_{\text{req}}, \mathbf{E})$
3: $p(\cdot) \leftarrow$ `path_to_frontier`$(\mathbf{Q}_{\text{goal}}, \mathbf{V}_{\text{frontier}}, \mathbf{E})$
4: $\mathbf{V}'_{\text{robot}} \leftarrow \mathbf{V}'_{\text{robot}} \cup \mathbf{V}_{\text{frontier}}$
5: $\mathbf{Q} \leftarrow \{\mathbf{q} \in \mathbf{V}_{\text{frontier}}\}$ {populate FIFO queue}
6: **while** $|\mathbf{Q}| > 0$ **do**
7: $\quad \mathbf{q}_i \leftarrow$ remove head from $\mathbf{Q}$
8: $\quad$ **for all** $(\mathbf{q}_i, \mathbf{q}_s) \in \mathbf{E}_{\text{s}} : \mathbf{q}_s \notin \mathbf{V}'_{\text{robot}}$ **do**
9: $\quad\quad (\mathbf{V}'_{\text{robot}}, \mathbf{E}'_{\text{robot}}) \leftarrow (\mathbf{V}'_{\text{robot}} \cup \{\mathbf{q}_s\}, \mathbf{E}'_{\text{robot}} \cup \{(\mathbf{q}_i, \mathbf{q}_s)\})$
10: $\quad\quad$ append $\mathbf{q}_s$ to $\mathbf{Q}$
11: $\quad$ **if** $p(\mathbf{q}_i) \neq$ `nil` **and** $(\mathbf{q}_i, p(\mathbf{q}_i)) \notin \mathbf{E}'_{\text{robot}}$ **then**
12: $\quad\quad$ **if** $p(\mathbf{q}_i) \notin \mathbf{V}_{\text{robot}}$ **then**
13: $\quad\quad\quad$ append $p(\mathbf{q})$ to $\mathbf{Q}$
14: $\quad\quad\quad \mathbf{V}'_{\text{robot}} \leftarrow \mathbf{V}'_{\text{robot}} \cup p(\mathbf{q}_i)$
15: $\quad\quad \mathbf{E}'_{\text{robot}} \leftarrow \mathbf{E}'_{\text{robot}} \cup (\mathbf{q}_i, p(\mathbf{q}_i))$
16: **return** $(\mathbf{V}'_{\text{robot}}, \mathbf{E}'_{\text{robot}})$

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.

6.3.4 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. 24 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 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

Algorithm 25 `path_to_frontier`$(\mathbf{q}_{\text{goal}}, \mathbf{V}_{\text{frontier}}, \mathbf{E})$

```
1: g(q_goal) ← 0 {cost to goal}
2: p(q_goal) ← nil {forward pointers}
3: U ← {q_goal} {priority queued ordered by g(·)}
4: while |V_frontier| > 0 do
5:    q_min ← remove (min U) from U
6:    V_frontier ← V_frontier \ {q_min}
7:    for all (q_from, q_min) ∈ E do
8:       d ← g(q_min) + cost(q_from, q_min)
9:       if q_from ∉ U or d < g(q_from) then
10:         g(q_from) ← d
11:         insert/update q_from in U
12:         p(q_from) ← q_min
13: return p(·)
```

Alg. 25, 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. 24, 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. 24 line 11).

6.4 Results

We evaluate our algorithm on a Fetch robot [31] 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 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 $t_{L_{\text{sim}}}$ subject to noise sampled from a Gaussian distribution with standard deviation of $0.16\, t_{L_{\text{sim}}}$.

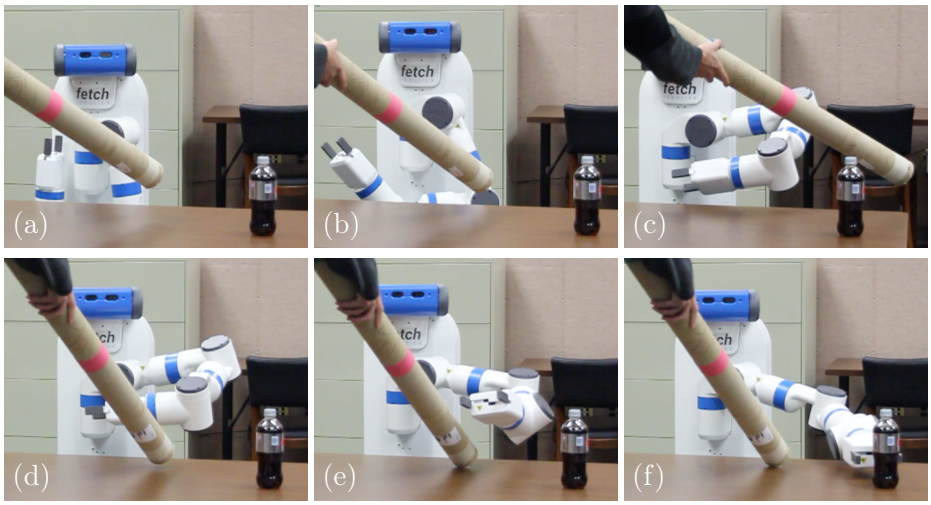
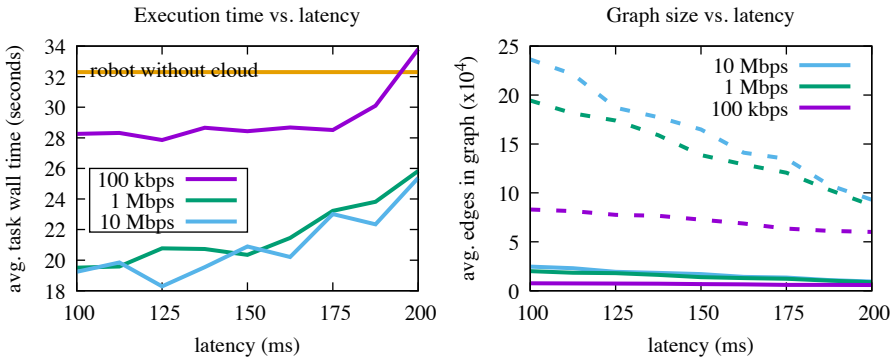


Figure 6.2: The Fetch robot using our cloud-based motion planning for the task of grasping the bottle resting on the table while avoiding both the static obstacles (e.g., table) and the dynamic obstacle (a tube sensed via an RGBD camera). In frame (a) after the Fetch approaches the table with its arm 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).

We implemented our algorithm as a web-service accessible via HTTP [32]. 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. 6.2(a)) to a pre-grasp configuration over a table (Fig. 6.2(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.



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

Figure 6.3: 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.

The sequence in Fig. 6.2 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 four 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. 6.3(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. 6.3(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. 6.4 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. 6.4 (a)). From Fig. 6.4 (b), we see that if $t_{\max}$ is too small, the robot is slower to find a collision-free path past

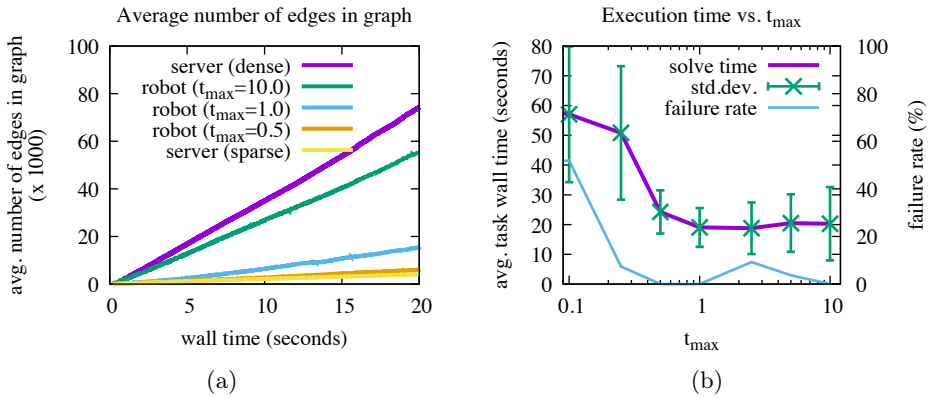


Figure 6.4: The serialization parameter $t_{\max}$ affects the size of the graph on the robot and the robot's task completion wall time. In these graphs the simulated network is fixed at $R_{\text{sim}} = 1$ Mbps and $T_{L_{\text{sim}}} = 200$ ms and the server solve time is 250 ms. Graph (a) shows that 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 two minutes), while having it too large increases the variance of the execution time.

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.

6.5 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. 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.

CHAPTER 7

Efficient Motion Planners via Templates

Planning motions for battery-powered robots with many degrees of freedom using their on-board computers often presents a difficult problem. The problem is difficult due to the computationally demanding nature of motion planning [98], which involves computing a sequence of robot actions that take the robot to a goal state while avoiding obstacles and satisfying task-specific constraints. The difficulty is then compounded when the robot's size is measured in the tens of centimeters, as its form factor and battery-life constraints only allow for low-power CPUs. While a wealth of planning algorithms aim to address the problem of motion planning [20], it is typically left to developers to implement these algorithms for low-power CPUs with fast robot-specific code. To address this requirement for a broad class of robots with low-power CPUs, we introduce Motion Planning Templates (MPT)[1], a system that generates robot-specific code from a set of motion planning algorithms.

MPT is a C++ software library that reduces the algorithm and data structure advances from previous chapters to practice. It is designed to be reusable for a wide variety of robots and tasks, while not sacrificing performance one might get with a custom-coded motion planning algorithm. With the design behind MPT, the performance gains over other paradigms for reusable motion planning libraries can be significant. When we add in the gains based upon implementations of the algorithms and data structures from previous chapters, MPT is able to compute motion plans in a fraction of the time of competing paradigms. These performance gains are initially targeted towards application on low-power single- and multi-core CPUs that one might find onboard a small robot—with the idea that these robots will be able to leverage, or be augmented by, the cloud-based computation from chapter 6 as needed.

[1]MPT is available at `https://robotics.cs.unc.edu/mpt`

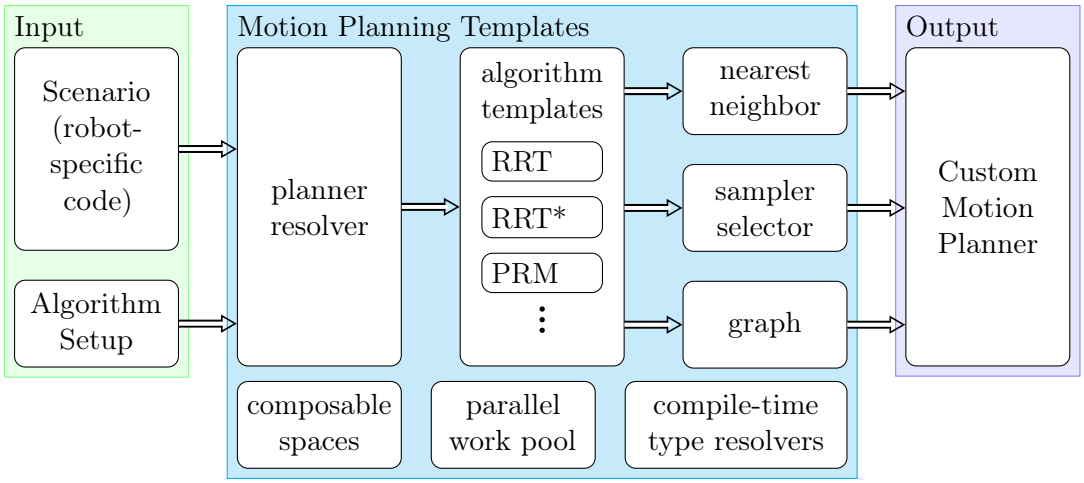


Figure 7.1: The process flow of Motion Planning Templates (MPT) starts with a developer supplying a robot's motion planning problem scenario and selecting an algorithm setup. At compile time, the template system of MPT generates code for a robot-specific implementation of a motion planning algorithm. This system trades off runtime flexibility (algorithms and their data structures cannot be changed without recompiling) in favor of improved performance and reduced memory utilization, both of which are critical to battery-powered small robots that use their on-board low-power CPU to perform motion planning.

The key philosophy behind MPT is that it *generates* robot-specific motion planning code. This means that a software developer writes code specific to the robot and the scenario, and then, through the compile-time constructs of MPT, a C++ compiler generates the code and data structures for a custom implementation of a motion planning algorithm. The resulting implementation will have performance competitive with hand-written implementations of the same motion-planning algorithm that use robot-specific data structures. The system behind MPT's code generation is C++ templates, which is a Turing-complete [119] compile-time polymorphic system—which is a fancy way of saying that C++ templates are programs that write code.

In order to eke out as much performance as possible from low-power embedded processors, MPT is also multi-core ready—which allows MPT to take advantage of multi-core parallel processing increasingly available on low-power CPUs. This parallelism can be exploited in a complete robot system to allow robots to take on multiple computational tasks simultaneously (e.g., sensor processing, actuation, etc.) or to tackle computationally demanding tasks such as motion planning. As available parallelism and demands on computation can vary from robot to robot, MPT can be set to use as little or as much parallelism as desired. When parallelism is enabled, MPT's parallelized motion planning algorithms make use of concurrent data structures for nearest neighbors searching [45]

and motion planning graphs. But concurrent data structures do not come for free—in order to ensure correct operation, they must use locks and ordered memory operations [41] that can result in decreased per-thread performance and increased memory usage. When parallelism is disabled, MPT generates code without locks or ordered memory operations, to maximize single-threaded performance.

This chapter presents MPT, the design principles behind it, background on its compile-time polymorphic system, how to use it, and examples from applications in our own lab using low-powered processors that one finds, or might find, in small battery-powered robots.

7.1 Design Principles

The design principles behind MPT help differentiate it from related and complementary libraries. This section describes those principles.

7.1.1 Performance over runtime flexibility

MPT started with the design decision that performance of robot-specific motion planners in small battery-powered robots is more important than runtime flexibility. For example, an articulated robot does not need the flexibility to compute motion plans for a wheeled robot or aerial drone. Thus MPT uses compile-time algorithms to generate robot-specific motion planners instead of using a flexible runtime system.

7.1.2 Floating-point precision selection

Robots with low-power CPUs may have performance and memory requirements that benefit from using single-precision (32-bit) floating-point arithmetic. Conversely, some robots must plan motions with accuracy and thus require double-precision (64-bit) arithmetic or better. MPT allows the selection of floating point precision at compile time.

7.1.3 Custom state and trajectory data types

Motion planners must inter-operate with other robot software components, and thus MPT should generate and operate on graph structures with robot-specific data types that do not require runtime translation. For example, a robot with a ROS [102] interface to its actuators could compute trajectories in the native ROS message type and then store the trajectories directly in the motion graph. This would add efficiency by removing a translation between data types (e.g., when the robot sends the trajectory to the actuators). In an example from a robot with complex forward

kinematics, it may be helpful or necessary to carry additional information within states to help speed up kinematic computations in the local planner; with a custom state data type stored directly in the motion graph, the extra information would be made available to the local planner method, thus allowing for faster computation.

7.1.4 (De-)Composable Metric Spaces

Some motion planners (e.g., KPIECE [113]) and nearest neighbor data structures (e.g., kd-trees [123, 46]) benefit from the ability to decompose the state space into its constituent components. Complex metric state spaces in MPT can be composed from simpler metric spaces and decomposed at compile-time to select and construct state-space specific implementations of motion planners and data structures.

7.1.5 Multi-core Ready

CPUs are trending towards increased multi-core parallelism. However, many low-power CPUs are still single-core, and robots with multi-core CPUs may wish to use only a single-core for motion planning. Since multi-core parallelism requires additional overhead and is not always necessary, MPT can switch between generating multi-core parallel and single-core planners.

7.1.6 C++ 17 Header-only Library

The latest C++ standard [55] provides a wealth of capabilities that eases development of template-based programs while remaining compatible with existing C and C++ software libraries. A header-only library means that none of the code is compiled until an application makes use of it, which can ease deployment.

7.2 Related Work

The Open Motion Planning Library (OMPL) [112] is an actively developed, well-maintained, and popular motion planning library. It implements a wide variety of motion planning algorithms using an architecture that allows for maximum flexibility at runtime. The architecture is based upon virtual classes and methods which are popular and well-studied, thus OMPL provides many with a familiar development environment and a relatively gentle learning curve. MPT does not use virtual classes and methods and is thus less flexible at runtime and instead uses templates to generate robot-specific motion planners. Since templates are resolved at compile-time, MPT gains the ability to detect detect return types and alter data structures accordingly. For example, when MPT detects that the

collision detection routine returns a type that is not a boolean, MPT will generate a graph data structure in which the return value is stored in the graph's edges. Similar behavior is not possible in a runtime-polymorphic system such as used by OMPL, since the collision detection routine's return type is fixed by the virtual class hierarchy. MPT's reliance on templates likely introduces a steeper learning curve since template-based programming is less thoroughly covered in many university courses. OMPL provides mostly single-core motion planners, with some notable multi-core ready exceptions (e.g., C-FOREST [93]). In contrast, all of MPT planners support multi-core parallel processing as well as the ability to turn off parallel processing when a single-core planner is desired; additionally, MPT provides data structures and frameworks for parallel multi-core motion planning. OMPL will likely be the first choice of anyone learning motion planning or exploring a specific motion problem, whereas MPT aims to replace hand-writing custom motion planners once the planning problem is understood and needs to eke out as much performance as possible on small battery-powered robots.

OpenRAVE [24] integrates motion planning, perception, and control algorithms into a runtime-configurable system. The architecture allows developers to add functionality using plugins and uses virtual classes for maximum runtime flexibility, but as a result may not perform motion planning as fast as a robot-specific planner. MPT could generate motion planners that run as OpenRAVE plugins, allowing robots to benefit from the best of both systems.

Robotics Library (RL) [100] provides a large collection of robot planning and control software in one coherent whole. This library includes a collection of sampling-based planners, including RRT [76] and PRM [61]. RL makes some use of templates but largely depends on virtual classes and methods to adapt different robot systems.

MoveIt! [111, 22] is an open-source tool for mobile manipulation built on top of ROS and OMPL. It aims to automate the setup of motion planning integrated with perception and control. MPT automates less of the motion planning setup process, but instead aims to provide greater efficiency for battery-powered small robots.

Robot Operating System (ROS) [102] is a popular software framework that aims to provide a complete system to operate a robot. It includes modules (e.g., OMPL and MoveIt!) for motion planning. MPT could similarly integrate with ROS, providing motion planners specific to the robot on which it runs and operating directly on ROS data types.

Murray et al. show that another route for low-power and fast motion plan computation is through the use of programmable circuitry [90]. But these methods require specialized hardware that is not always available on robot systems. The software-based approach of MPT aims to be compatible with readily available low-power CPUs.

7.3 Background

This section formally defines the motion planning problem, and provides background on tools MPT uses: compile-time polymorphism and C++ template metaprogramming.

7.3.1 Motion Planning Problem

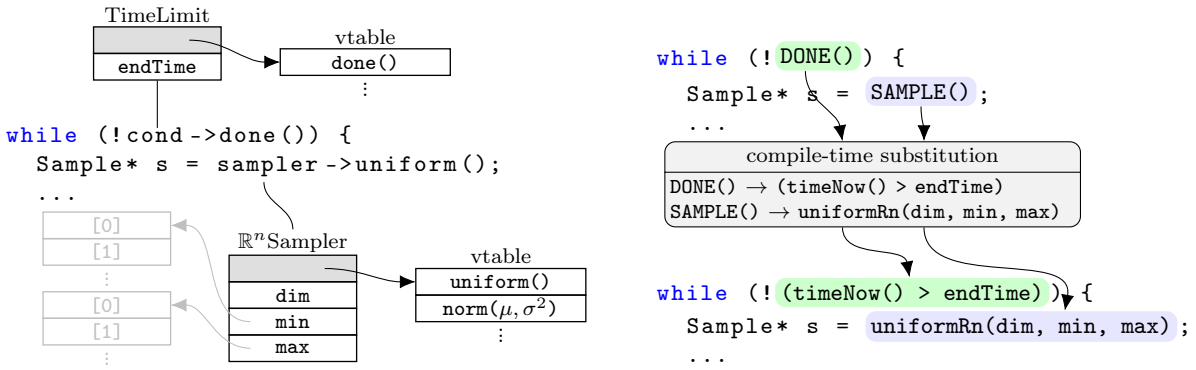
Robot motion planning algorithms compute a sequence of states that takes a robot from an initial state to a goal state while avoiding obstacles and staying within task-specific constraints. The set of robot states is the state-space $\mathcal{X}$. Within the subset $\mathcal{X}_{\text{free}} \subseteq \mathcal{X}$, the robot does not collide with any obstacle and does not violate any constraint. Thus the input to the motion planning problem is: the initial state $\mathbf{x}_0 \in \mathcal{X}_{\text{free}}$, the set of goal states $\mathcal{X}_{\text{goal}} \subseteq \mathcal{X}_{\text{free}}$, and $\mathcal{X}_{\text{free}}$. The output is a path $\tau = (\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_n)$, where $\forall i : \mathbf{x}_i \in \mathcal{X}_{\text{free}}$, and $\mathbf{x}_n \in \mathcal{X}_{\text{goal}}$. When $\mathcal{X}$ is continuous, the output path τ must also satisfy the condition

$$\forall i \in \{1, 2, \ldots, n\}, t \in [0, 1] : L(t; \mathbf{x}_{i-1}, \mathbf{x}_i) \in \mathcal{X}_{\text{free}},$$

where $L(t; \mathbf{x}_{\text{a}}, \mathbf{x}_{\text{b}}) : [0, 1] \to \mathcal{X}$ is a problem-specific local planner that continuously interpolates the robot's state as parameterized by two states, with $L(0; \mathbf{x}_{\text{a}}, \mathbf{x}_{\text{b}}) = \mathbf{x}_{\text{a}}$ and $L(1; \mathbf{x}_{\text{a}}, \mathbf{x}_{\text{b}}) = \mathbf{x}_{\text{b}}$.

The various sampling-based motion planners of MPT require problem-specific definitions of functions in order to explore $\mathcal{X}_{\text{free}}$ and build a graph of valid motion. Many sampling-based motion planners, when a bounded region of $\mathcal{X}$ is not implied by its topology, require a sampling region or function. For many sampling-based motion planners, the full definition of a problem-specific L is not required; instead, it is often sufficient to define a problem-specific function $L_{\text{free}}(\mathbf{x}_{\text{a}}, \mathbf{x}_{\text{b}}) = \forall t \in [0, 1] : L(t; \mathbf{x}_{\text{a}}, \mathbf{x}_{\text{b}}) \in \mathcal{X}_{\text{free}}$, that checks if there exists valid motion between two states. Additionally, some motion planners require a distance function $d : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ in order to operate efficiently and/or to minimize the resulting path length $\sum_{i=1}^{n} d(\mathbf{x}_{i-1}, \mathbf{x}_i)$.

In summary, MPT requires the following definitions in order to generate a problem-specific motion planner: the topology of $\mathcal{X}$ (which corresponds to the data type of a state), a sampling region or



(a) Runtime polymorphism and virtual tables

(b) Compile-time polymorphic calls

Figure 7.2: **Comparison of runtime polymorphic calls to compile-time polymorphi calls**. In runtime polymorphism (a), calls to virtual method require a lookup into a virtual table (vtable). The vtable introduces a level of indirection that provides the flexibility to swap in different object types to get different behaviors. With template-based compile-time polymorphism (b), the compiler substitutes placeholders with direct function calls. In contrast to runtime polymorphism, flexibility to change the termination condition at runtime is lost, but execution time is sped up. In this example, the time-limit termination condition and sampler in (a) can be changed by passing in objects of different types. While in (b), the termination condition and sampler can only be changed at compile time with a different substitution. In practice, these rarely change. Thus, the vtable lookup in (a) provides flexibility, but also introduces a repeated delay. In (b) speedup comes from saving a level of indirection, and giving the compiler the ability to perform additional optimizations since it knows which code will be called. Refer to Sec. 7.3.2 for additional details.

function, a definition of L_{free}, and a distance function. At runtime, the motion planner generated by MPT takes the inputs $\mathbf{x}_0 \in \mathcal{X}_{\text{free}}$ and $\mathcal{X}_{\text{goal}}$, and computes either a valid path, or a graph $G = (V, E)$, in which vertices $V \subseteq \mathcal{X}_{\text{free}}$, and for each edge's vertex pair $(\mathbf{x}_i, \mathbf{x}_j) \in E$, $L_{\text{free}}(\mathbf{x}_i, \mathbf{x}_j)$ is true.

7.3.2 Compile-time Polymorphism

Polymorphism, from the Greek meaning "many forms", refers to the ability of a single code interface to provide many different implementations [110]. In practice this means that the data and code behind a name can be changed without changing the code that refers to that name. When the executed code can be changed while the program is running, it uses *runtime polymorphism*, a concept that is likely familiar to people with experience with class-based object oriented programming in languages such as Java, Python, and C++. In runtime polymorphism, when code invokes a virtual method, it finds the the concrete implementation through a virtual table (vtable) lookup. Fig. 7.2 (a) shows an example of a sampling-based motion planner's outer loop using runtime polymorphism to change its behavior. The loop continues until the `done()` method returns `true`—the exact meaning of

`done()` is dependent on the `cond` object's concrete type. Similarly, the loop can work in any state space using `sampler` object of the appropriate concrete type.

Compile-time polymorphism, also called static polymorphism, operates on a similar principle, but instead resolves implementations when the code is compiled, so it does not need a virtual table. Fig. 7.2 (b) shows a compile-type polymorphic equivalent of Fig. 7.2 (a). In this case, the behavior cannot be changed at runtime, and as a result, can run faster than the vtable-based approach. Virtual calls are an important enough performance consideration that researchers have put effort into devirtualizing calls at runtime [54]. The loss of runtime flexibility in this example is likely to be acceptable for the performance gained by the robot-specific motion planner.

7.3.3 C++ Template Metaprogramming

MPT uses compile-time polymorphism based on C++ templates. Templates are like functions that run in the compiler that take data types and constants as parameters and generate code that will be executed. Template data type parameters can be arbitrarily complex structures, which allows seemingly simple template substitutions to transitively lead to complex results—e.g. robot-specific motion planners.

C++ templates can also be *specialized* to allow for specific substitutions based upon a template parameter matching a condition. As an example, specialization can select an appropriate nearest neighbor data structure depending on whether or not the distance function is symmetric.

Templates are defined using a `template` keyword, followed by parameter declaration within `<` angle `>` brackets, followed by the class or method template. Template substitution occurs when the compiler encounters the template name followed by parameters within angle brackets.

7.4 Approach

This section describes MPT's design from the users' perspective. All motion planners in MPT are available through a single `mpt::Planner` template, which takes two type parameters: the *Scenario* and the *Algorithm*. The user provides the Scenario and selects the algorithm, and MPT provides the algorithm's implementations and the building blocks to make a scenario.

7.4.1 Scenario Specification

In MPT, a *Scenario* is a user-provided C++ class whose member types and methods define a robot-specific motion planning problem (i.e., $\mathcal{X}$, $\mathcal{X}_{\text{free}}$, L_{free}, etc.). To give a high-level overview of

```
1  template <typename Scalar = double>
2  struct ExampleScenario {
3    using Space = mpt::SE3Space<Scalar>;
4    using State = typename Space::State;
5    using Goal = mpt::GoalState<State>;
6    using Bounds = mpt::BoxBounds<Scalar, 3>;
7
8    Space space();
9    Bounds bounds();
10   Goal goal();
11
12   bool validState(State q);
13   bool validMotion(State a, State b);
14 };
```

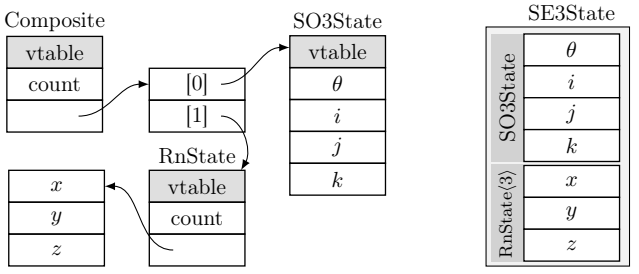
Listing 7.1: Minimal definition of a scenario

how this is done and to show some of the capabilities of MPT, we will walk through the example scenario shown in listing 7.1. For brevity, the listing does not include `const` and reference modifiers, nor does it include implementation code.

A scenario definition starts with the declaration of a (template) class, as shown in lines 1 and 2. There is *no* base class from which to inherit members, instead Scenario classes must conform to a few requirements. The scenario defines the state space ($\mathcal{X}$) as a type alias called `Space` (line 3). In the example, it will plan for a robot that can translate and rotate in 3D space, and thus it uses the SE(3) state space. The `Scalar` type parameter allows the scenario to switch between single-precision and double-precision (the latter being the default). The `Space` defines both the metric and the C++ data type (more details in Sec. 7.4.2). For SE(3), the state type is a class with a quaternion for rotation and a 3-element vector for translation (see Fig. 7.3 (b)). Line 4 creates an alias for the state data type used later. Since some spaces carry data members to implement their metric (e.g., a weighting components in a Cartesian space), MPT requires a `space()` method (line 8) to return a `Space` object.

Sampling-based motion planners require a mechanism to generate random states from $\mathcal{X}$. Were this class to define a `sample()` method, MPT would use it to generate samples. This scenario instead has MPT use uniform sampling by defining the sampling bounds (lines 6 and 9).

The scenario defines the problem's goal set ($\mathcal{X}_{\text{goal}}$) as a type (line 5) and method (line 10) pair. The goal type provides an indicator function that checks if a state is in $\mathcal{X}_{\text{goal}}$. Motion planners and goal types that support goal-biased sampling make use of template specialization to obtain biased samples from $\mathcal{X}_{\text{goal}}$.



(a) Runtime construction (b) Compile-time

Figure 7.3: An SE(3) state is constructed by combining and reusing state definitions for SO(3) and $\mathbb{R}^n$. Using run-time polymorphism (a) requires the composite state to carry an array of sub-states, each of which is dynamically allocated and addressed through pointers—this allows for maximum flexibility as composite states can vary in number of sub-states, and $\mathbb{R}^n$ state can vary in number of components. In contrast, compile-time polymorphism (b) defines a single composite state type at compile-time, reducing the amount of memory and objects required at runtime. In this example system, the runtime polymorphic system requires 2× the memory and 5× the objects of the compile-time polymorphic system.

The scenario defines $\mathcal{X}_{\text{free}}$ and L_{free} using the indicator functions `validState()` (line 12) and `validMotion()` (line 13) respectively. For some robots, testing $L_{\text{free}}(\mathbf{x}_a, \mathbf{x}_b)$ may require complex and time-consuming forward-kinematics computation of $L(\cdot; \mathbf{x}_a, \mathbf{x}_b)$. As such, it may be desirable to save the result of the computation to avoid regenerating it later. MPT detects when `validMotion()` returns something other than a boolean, and changes the motion graph definition to store the result for later retrieval. For example, given the method declaration

```
std::optional<Trajectory> validMotion(...);
```

MPT stores a `Trajectory` value in each graph edge. Similarly, changing the return type of `validState` allows additional information to be stored in each vertex of the graph.

7.4.2 (De-)Composable Metric Spaces

While MPT supports arbitrary data-types and metric combinations, it provides special handling for metric spaces commonly found in motion planning, including L^p (with $p \geq 1$), SO(2), SO(3), and weighted Cartesian products thereof. A mathematical metric space is an ordered combination of a set $\mathcal{X}$ and metric d. In MPT a metric space is expressed as an ordered pair of state data types (e.g., a vector of floats), and a metric tag type (e.g. L2). Using specialization, MPT provides support for a variety of common C++ data types available in the standard library and in the popular Eigen [56] linear algebra library. Using the built-in spaces allows MPT to inspect the space in order to make an informed selection of data structures and planning algorithm behaviors.

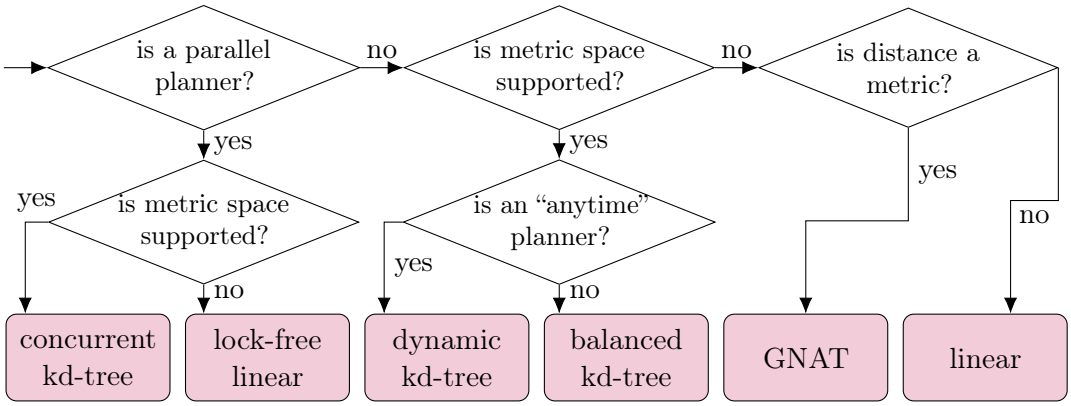


Figure 7.4: Using a compile-time algorithm, MPT automatically selects and generates the planner's nearest neighbor data structure based upon the requirements of the scenario and planning algorithm.

MPT allows easy setup of supported metric space to match the data types in the rest of the robot's system. For example, to use a Euclidean metric on $\mathbb{R}^3$ using a custom vector type `Vec3d`, the syntax is: `MetricSpace<Vec3d, LP<2>>`. It is also possible to create weighted Cartesian metric spaces. For example, to create an SE(3) space that combines translations in $\mathbb{R}^3$ with rotations in SO(3), the syntax is:

```
using R = MetricSpace<Quaternion, SO3>;
using T = MetricSpace<Vec3d, LP<2>>;
using SE3 = CartesianSpace<R, T>;
```

Assuming `Quaternion` and `Vec3d` are appropriately defined, the above code is equivalent to:

```
using SE3 = MetricSpace<
  std::tuple<Quaternion, Vec3d>,
  Cartesian<SO3, L2>>;
```

The result of this construction is that the Cartesian state space is flexibly defined at compile-time and its state data type is compact at runtime. Fig. 7.3 (b) shows the resulting state type as it will be stored in memory. A similar flexibility is possible in a runtime-polymorphic system and is shown in Fig. 7.3 (a), but requires significantly more overhead since the states must be assembled as object graphs at runtime. While it is possible to avoid this overhead with a custom implementation, such an approach would lose the benefit of code reuse.

7.4.3 Nearest Neighbors

Nearest neighbor searching is a fundamental building block for many motion planning algorithms. The performance of nearest neighbor searching can dramatically affect the performance of a planning

Figure 7.5: The Nao robot uses a low-power Intel Atom CPU to solve a 10 DOF motion planning problem (from [49]) that avoids obstacles in order to drop an effervescent tablet into a glass while not spilling in the process.

algorithm [123, 67]. MPT thus uses a *compile-time* algorithm to select and define a nearest neighbor data structure that best matches the needs of the scenario and planner. This algorithm is shown in Fig. 7.4.

The nearest neighbor searching data structures in MPT are: kd-tree that supports concurrent inserts and queries [45] (ideal for parallelized motion planners) and a non-concurrent variant of it, a (non-concurrent) kd-tree that maintains near optimal balance [14] at the expense of periodic rebalancing (ideal for non-parallel, long running motion planners), GNAT [17], and linear searching for custom metric and non-metric spaces. When the scenario uses an MPT-supported metric space, MPT can decompose it at compile-time to generate a custom implementation of a kd-tree.

7.4.4 Planner Algorithm Selection

In a compile-time algorithm that is similar to, though more involved than Fig. 7.4, MPT uses a template argument to determine the motion planner implementation to generate. The process starts with the creation of a `mpt::Planner<Scenario, Algorithm>` object, where the `Scenario` is defined in a similar manner to Listing 7.1, and `Algorithm` is an MPT-provided algorithm selection tag, such as `mpt::RRT<>`. Under the hood, MPT uses a cascade of template specializations to resolve a final algorithm implementation. The planning algorithms included in MPT's initial release are parallel lock-free [49] versions of RRT [76], RRT* [60], PRM [61], PRM* [60], and IRS [82].

7.5 Applications

In this section we demonstrate MPT's performance on an articulated robot and in OMPL's SE(3) rigid-body planning benchmarks. We compare to OMPL as it is an example of a well-designed flexible motion planning library that uses runtime polymorphism. To the extent possible, we

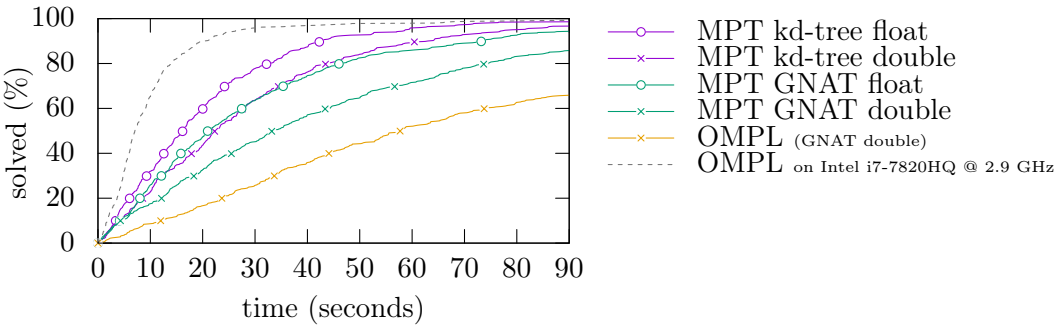


Figure 7.6: **RRT probability vs. compute time**. An Intel Atom CPU computes a solution to a 10 DOF problem for the Nao robot using RRT. The compile-time polymorphism in MPT more efficiently computes samples which results in finding solutions sooner. The dashed gray line shows the same OMPL benchmark as the orange line, but run instead on an Intel i7-7820HQ @ 2.9 GHz (a high-end laptop CPU)—showing the CPU performance difference that MPT aims to address. The low power processor is much slower than the laptop processor, but with MPT, the low-power processor on the robot can begin to compete with a runtime polymorphic system running on a much faster laptop processor.

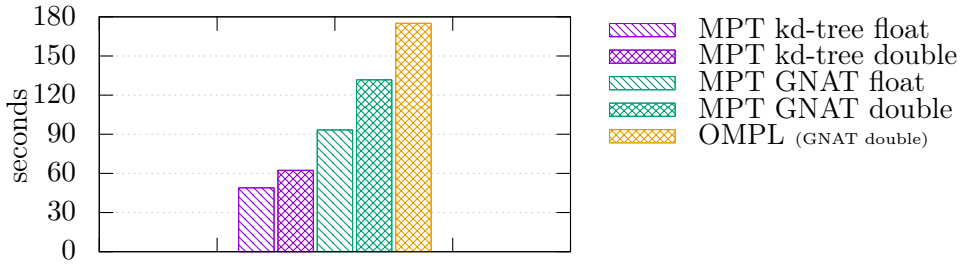
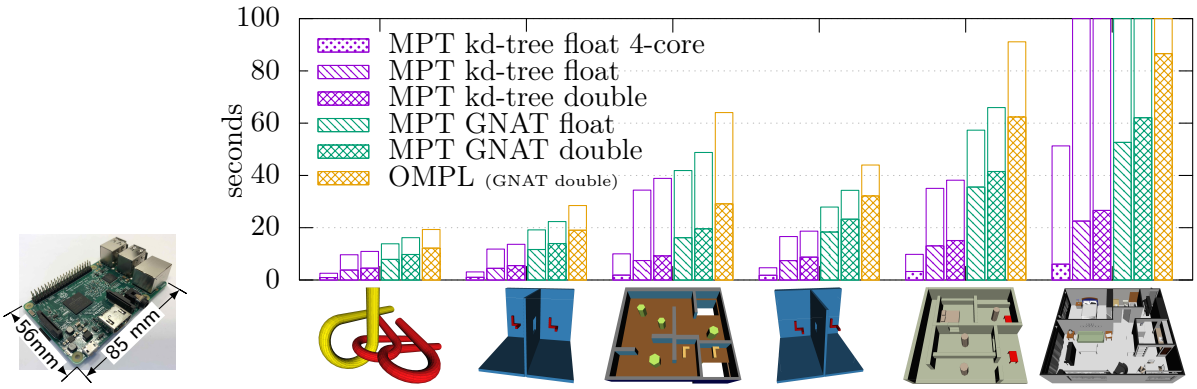


Figure 7.7: **Nao computes a 5 000 vertex RRT* graph** for a 10 DOF task using MPT and OMPL running on an Intel Atom processor.

set up corresponding motion planners from MPT and OMPL to run identical algorithms. The performance benefit of MPT over OMPL thus comes from the MPT's compile-time data-structure and algorithm selections, compact state representation, non-virtual methods, and affordances that allow the compiler to inline and vectorize code. This does however come at the cost of losing runtime flexibility and a potentially steeper learning curve. We run MPT with both single ("float") and double precision arithmetic. OMPL only supports double precision arithmetic. OMPL uses GNAT for nearest neighbor searching so we compare to MPT using GNAT. We also compare against MPT's automatic selection of kd-trees for nearest neighbor searching.



(a) Raspberry Pi 3 B (b) alpha-1.5 (c) Easy (d) cubicles (e) Twistycool (f) Home (g) Apartment

Figure 7.8: **The Raspberry Pi 3 computes a 10 000 vertex RRT* graph** for SE(3) rigid body motion planning problems from OMPL [112]. To the extent possible, MPT and OMPL are set up to run identically. Collision detection, which is *not* provided by MPT or OMPL, is plotted in the unfilled blocks.

7.5.1 Small Humanoid Motion Planning using an Intel Atom

We use MPT to solve a 10 degree of freedom (DOF) task on a SoftBank Nao small humanoid robot shown in Fig. 7.5. This robot has a low-power (2 to 2.5 W) Intel Atom Z530 @ 1.6 GHz CPU. To avoid taxing our robot, we run hundreds of simulations on a more recent Atom N270 @ 1.6 GHz, noting that the CPUs perform similarly in benchmarks [107].

The Nao simulation uses an RRT [76] motion planner that terminates as soon as it finds a feasible plan. We plot the observed solution probability given the wall-clock time spent computing. As the graph in Fig. 7.6 shows, MPT's custom generated motion planner solves the planning problem in less than half the time of a runtime polymorphic system.

We also run the asymptotically optimal RRT* [60] motion planner until it creates a 5 000 vertex motion graph. Over 50 runs all implementations of the planner require approximately the same number of iterations and generate paths of similar cost distribution, confirming the planners implement nearly identical algorithms. Fig. 7.7 shows the wall-clock time to compute the graph, showing the performance impact of having a custom generated planner, using single-precision floats, and using kd-trees for nearest neighbor.

7.5.2 Rigid Body Motion Planning using a Raspberry Pi

We use a Raspberry Pi 3 Model B v 1.2 (Fig 7.8 (a)) to compute RRT* solutions to SE(3) rigid-body planning problems from OMPL (Fig. 7.8 (b)–(g)). The Pi is a low-power (2 to 3 W)

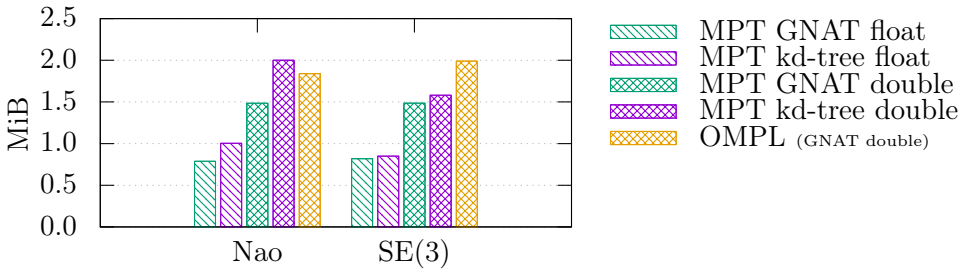


Figure 7.9: **Memory usage with 10 000 RRT* graph vertices** as generated by MPT and OMPL for the Nao and SE(3) problems on 32-bit CPUs.

4-core ARM-architecture processor which would make a suitable processor for a battery-powered small robot due to its low power consumption and small form factor. Fig. 7.8 shows the wall-clock time elapsed when computing a 10 000 vertex graph. In this setup, we also show the benefit of the parallelism included in MPT by running PRRT* [49], a parallelized version of RRT*, running on all 4-cores.

7.5.3 Reduced Memory Usage

We measure and compare the mean memory usage of RRT* runs from the Nao and SE(3) scenarios. The results in Fig. 7.9 show the impact of the compact memory representation and the choice of nearest neighbor structures. The difference between MPT's GNAT and kd-tree data structures shows that GNAT is more memory efficient. This implies some MPT users will have to choose between the speed of a kd-tree vs. the lower memory usage of GNAT. The comparison between MPT's GNAT using double-precision and OMPL shows the impact of the compact data-structures that MPT is able to generate. The difference in improvements between the Nao and SE(3) highlights the impact of compile-time state composition since the complex object graph for SE(3) states (Fig. 7.3) incurs more overhead than the Nao scenario's relatively simple $\mathbb{R}^{10}$ state. Finally, the figure shows the significant impact of changing floating-point precision—when the loss of precision is acceptable, MPT may enable planners to run on systems where memory usage comes at a premium.

7.6 Conclusions and Future Work

We presented Motion Planning Templates, a framework based upon the compile-time polymorphic system of C++ templates for building motion planners for robots with low-power CPUs. MPT's template system generates custom planning code specific to the robot and a set of tasks encompassed by a concept of a scenario.

In benchmarks on a small humanoid robot and synthetic benchmarks on rigid body motions, the template-based architecture behind MPT allows it to generate planners that demonstrate better performance and lower memory usage than planners based upon runtime polymorphism. While this approach loses the flexibility of runtime polymorphism and introduces a potential learning curve to developers more familiar with runtime polymorphic systems, the trade-off may be worth the cost, especially in small low-powered robots where every CPU cycle counts. Additionally, the implementation of the lock-free parallel motion planners from chapter 2, SO(3)/SE(3) nearest-neighbor partitioning from chapter 3, and concurrent nearest-neighbor data structure from chapter 4, allow for additional performance gains over generalized and single-threaded approaches found in other runtime-polymorphic motion planning libraries.

MPT is an evolving project under active development. While the initial release is focused on creating robot-specific planners, with additional work it will be extended to support the infrastructure needed for cloud-based motion planning from chapter 6.

CHAPTER 8

Conclusion and Future Work

The computational demands of robot motion planning for a future in which robots are increasingly a part of everyday life will require planning algorithms that can effectively utilize available computing power. In the past, single-threaded computing power grew exponentially as new processors were released over time, but this trend no longer continues. Instead, computing power is now exponentially growing through increasing multi-core parallelism. Making use of this parallelism has been our focus—whether through the robot's onboard CPU, or through a tandem computation with the vast computing power in the cloud.

In this dissertation, we addressed the following thesis statement:

Robot motion planning algorithms using multi-core parallelism, concurrent data structures, and cache-awareness can demonstrate superlinear speedup. With this speedup, robots can solve complex motion planning problems sooner and converge towards optimal motion plans faster. The resulting faster motion planning can enable robots to effectively operate in dynamically evolving scenarios, including cases in which a robot with a low-power CPU gains access to faster motion planning through computers deployed in the cloud.

To support this thesis, we introduced lock-free sampling-based motion planning, a partitioning approach to SO(3) for fast nearest neighbor searching, a concurrent nearest neighbor searching data structure with provable guarantees, cache-aware motion planning, and cloud-based motion planning for dynamic environments. With the lock-free sampling-based motion planning we sped up motion planning to scale linearly with additional multi-core parallelism, and sometimes observed superlinear speedup. With SO(3) nearest neighbor partitioning, we further sped up motion planning for robots that must plan for rotations in 3D. With the concurrent nearest neighbor searching data structure, we provide fast nearest neighbor searching with provably correct and asymptotically wait-free operation. With the cache-aware sampling-based motion planning, motion planning is able avoid the slowdown due to cache misses, and to run faster for longer. With the cloud-based motion

planning, robots can react to changing environments, even when their on-board computing abilities would be insufficient to do it alone. Finally, to facilitate the use of this dissertation's contributions in practice, we developed an open-source library for fast motion planning using scalable multi-core parallelism.

8.1 Future Work

The contributions presented in this dissertation open up several avenues for continuing research. Some of the immediate avenues relate to distributed parallel algorithms, metric spaces for nearest neighbor searching, approximate algorithms to increase motion planning convergence rate, cloud-based generation of roadmaps constrained to hard real-time limits, and integration with the optimization-based motion planning paradigm.

Distributed memory parallelism The algorithms and data structures presented in this dissertation focus on shared-memory parallelism, in which multiple computing cores share common data structures, such as the motion planning graph and nearest neighbor data structure. While computing trends indicate that future multi-core CPUs will continue to gain increasing core counts, it is not guaranteed. Additionally, some motion planning problems may require more computing power than even the fastest cloud-based computers can provide. As such, it may be desirable to gain additional computing power through multiple networked multi-core computers. In this case, the memory is distributed (no longer shared), which introduces additional computational bottlenecks. While a wealth of research has explored different avenues for distributed motion planning, to the best of our knowledge, none have integrated the shared memory approach presented in the dissertation with distributed memory computations.

Additional metric spaces for concurrent nearest neighbor searching The nearest neighbor data structures we presented support searching on weighted Cartesian products of Minkowski, 2D rotational, and 3D rotational spaces—and while this covers a broad class of robot motion planning problems, it would be beneficial to support addition metric spaces or even general metric spaces. GNAT, while slower than kd-trees, has the benefit of supporting arbitrary metric spaces. Thus one avenue of research could be creating a concurrent data structure based upon GNAT.

Approximate nearest neighbor search and near-optimal motion planning Recently there has been a surge in exploring motion planners that are asymptotically "near" optimal. In these planners, the optimality constraint is relaxed in favor of decreased computation time. When planning for dynamic environments using robot-based or cloud-based multi-core parallelism, it may be possible to gain additional speedup by using nearest-neighbor approximations that would result in an asymptotically near-optimal planner.

Generating roadmaps which have timing guarantees In the cloud-based motion planning contribution, the cloud generates a roadmap (graph) that the robot uses to avoid collisions with dynamic obstacles. It is left as an engineering exercise to ensure that the robot can search the roadmap rapidly enough to avoid collision—perhaps relying on a wealth of graph-based search algorithm research to perform the task. A future avenue of research would be to look into how to make cloud-based roadmap generation produce a graph which matches the robot's computational capabilities—thus, for exampling, guaranteeing that the robot's worst-cased execution time on the graph search would always meet a safety-critical deadline.

Integration with optimization-based motion planning The work of this dissertation focuses on sampling-based motion planning as these types of motion planners can have the favorable properties of providing probabilistic completeness and/or asymptotic optimality. Unfortunately, as a consequence of the sampling-based approach, the produced motion plans often require a post-processing step to produce a smooth motion. On the other hand, optimization-based motion planners often produce smooth motions, but without the probabilistic completeness and asymptotic optimality guarantees. To address the lack of probabilistic completeness, one approach is to repeatedly attempt optimizations by varying the initial conditions. This suggests that future research could explore using the parallel multi-core planners of this dissertation to rapidly seed the initial conditions of multiple optimization-based planners run in parallel.

REFERENCES

[1] Anant Agarwal, John Hennessy, and Mark Horowitz. "An analytical cache model". In: *ACM Transactions on Computer Systems (TOCS)* 7.2 (1989), pp. 184–215.

[2] Pankaj K Agarwal et al. "Cache-oblivious data structures for orthogonal range searching". In: *Proceeding of Annual Symposium on Computational Geometry*. 2003, pp. 237–245.

[3] Baris Akgun and Mike Stilman. "Sampling heuristics for optimal motion planning in high dimensions". In: *Proceedings IEEE/RSJ Int. Conf. Intelligent Robots and Systems (IROS)*. 2011, pp. 2640–2645.

[4] 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.

[5] Amazon. *EC2 Instance Pricing*. `https://aws.amazon.com/ec2/pricing/`. Accessed: 2016-07.

[6] Gene M Amdahl. "Validity of the single processor approach to achieving large scale computing capabilities". In: *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*. ACM. 1967, pp. 483–485.

[7] Alexandr Andoni and Piotr Indyk. "Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions". In: *Foundations of Computer Science, 2006. FOCS'06. 47th Annual IEEE Symposium on*. IEEE. 2006, pp. 459–468.

[8] Apple. *Environment - Answers*. `http://web.archive.org/web/20170612120817/https://www.apple.com/environment/answers/`. Accessed 2017-06-12.

[9] Lars Arge, G Brodal, and Rolf Fagerberg. "Cache-oblivious data structures". In: *Handbook of Data Structures and Applications* 27 (2005).

[10] Sunil Arya et al. "An optimal algorithm for approximate nearest neighbor searching fixed dimensions". In: *J. ACM* 45.6 (1998), pp. 891–923.

[11] Jeffrey S Beis and David G Lowe. "Shape indexing using approximate nearest-neighbour search in high-dimensional spaces". In: *Computer Vision and Pattern Recognition, 1997. Proceedings., 1997 IEEE Computer Society Conference on*. IEEE. 1997, pp. 1000–1006.

[12] Kostas Bekris et al. "Cloud automation: Precomputing roadmaps for flexible manipulation". In: *IEEE Robotics & Automation Magazine* 22.2 (2015), pp. 41–50.

[13] Jon Louis Bentley. "Multidimensional binary search trees used for associative searching". In: *Commun. ACM* 18.9 (Sept. 1975), pp. 509–517. ISSN: 0001-0782. DOI: `10.1145/361002.361007`.

[14] Jon Louis Bentley and James B Saxe. "Decomposable searching problems I. Static-to-dynamic transformation". In: *J. Algorithms* 1.4 (1980), pp. 301–358.

[15] Alina Beygelzimer, Sham Kakade, and John Langford. "Cover trees for nearest neighbor". In: *Proceedings International Conference on Machine Learning*. ACM. 2006, pp. 97–104.

[16] Joshua J Bialkowski, Sertac Karaman, and Emilio Frazzoli. "Massively parallelizing the RRT and the RRT*". In: *Proceedings IEEE/RSJ Int. Conf. Intelligent Robots and Systems (IROS)*. San Francisco, CA, Sept. 2011, pp. 3513–3518.

[17] Sergey Brin. "Near neighbor search in large metric spaces". In: *Proceedings International Conference on Very Large Databases*. 1995.

[18] Brendan Burns and Oliver Brock. "Sampling-based motion planning using predictive models". In: *Proceedings IEEE Int. Conf. Robotics and Automation (ICRA)*. 2005, pp. 3120–3125.

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

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

[21] Paolo Ciaccia, Marco Patella, and Pavel Zezula. "M-tree: An Efficient Access Method for Similarity Search in Metric Spaces". In: *Proceedings International Conference on Very Large Databases*. 1997, p. 426.

[22] David Coleman et al. "Reducing the barrier to entry of complex robotic software: a moveit! case study". In: *arXiv preprint arXiv:1404.3785* (2014).

[23] Didier Devaurs, T Siméon, and J Cortés. "Parallelizing RRT on large-scale distributed-memory architectures". In: *IEEE Transactions on Robotics* 29.2 (Apr. 2013), pp. 767–770.

[24] Rosen Diankov and James Kuffner. "OpenRAVE: A planning architecture for autonomous robotics". In: *Robotics Institute, Pittsburgh, PA, Tech. Rep. CMU-RI-TR-08-34* 79 (2008).

[25] Andrew Dobson and Kostas E Bekris. "Sparse roadmap spanners for asymptotically near-optimal motion planning". In: *The International Journal of Robotics Research* 33.1 (2014), pp. 18–47.

[26] Lester E Dubins. "On curves of minimal length with a constraint on average curvature, and with prescribed initial and terminal positions and tangents". In: *American Journal of mathematics* 79.3 (1957), pp. 497–516.

[27] Erik Elmroth et al. "Recursive blocked algorithms and hybrid data structures for dense matrix library software". In: *SIAM review* 46.1 (2004), pp. 3–45.

[28] Chris Ely. *The Life Expectancy of Electronics*. Consumer Technology Association, Sept. 2014. URL: `https://www.cta.tech/News/Articles/2014/September/The-Life-Expectancy-of-Electronics.aspc`.

[29] Peter van Emde Boas. "Preserving order in a forest in less than logarithmic time and linear space". In: *Information Processing Letters* 6.3 (1977), pp. 80–82.

[30] Steven K Esser et al. "Convolutional networks for fast, energy-efficient neuromorphic computing". In: *Proceedings of the National Academy of Sciences* (2016), pp. 11441–11446.

[31] Fetch Robotics. *Fetch Research Robot*. `http://fetchrobotics.com/research/`.

[32] Roy Fielding et al. *Hypertext transfer protocol–HTTP/1.1*. RFC 2616. RFC Editor, June 1999. URL: `https://www.rfc-editor.org/rfc/rfc2616.txt`.

[33] Raphael A. Finkel and Jon Louis Bentley. "Quad trees a data structure for retrieval on composite keys". In: *Acta informatica* 4.1 (1974), pp. 1–9.

[34] Dexter Ford. "As Cars Are Kept Longer, 200,000 Is New 100,000". In: *New York Times* (Mar. 2012). URL: `http://www.nytimes.com/2012/03/18/automobiles/as-cars-are-kept-longer-200000-is-new-100000.html`.

[35] Mark Foskey et al. "A Voronoi-Based Hybrid Motion Planner". In: *Proceedings IEEE/RSJ Int. Conf. Intelligent Robots and Systems (IROS)*. Oct. 2001, pp. 55–60.

[36] Jerome H Friedman, Jon Louis Bentley, and Raphael Ari Finkel. "An algorithm for finding best matches in logarithmic expected time". In: *ACM Transactions on Mathematical Software (TOMS)* 3.3 (1977), pp. 209–226.

[37] Matteo Frigo et al. "Cache-oblivious algorithms". In: *Foundations of Computer Science, 1999. 40th Annual Symposium on*. IEEE. 1999, pp. 285–297.

[38] Ananth Grama et al. *Introduction to parallel computing*. Pearson Education, 2003.

[39] William Rowan Hamilton. "On quaternions; or on a new system of imaginaries in algebra". In: *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science* (1844–1850).

[40] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[41] Maurice Herlihy and Nir Shavit. *The art of multiprocessor programming*. Morgan Kaufmann, 2011.

[42] David Hsu, Gildardo Sánchez-Ante, and Zheng Sun. "Hybrid PRM sampling with a cost-sensitive adaptive strategy". In: *Proceedings IEEE Int. Conf. Robotics and Automation (ICRA)*. 2005, pp. 3874–3880.

[43] W.M.W. Hwu. *GPU Computing Gems Jade Edition*. Applications of GPU Computing Series. Elsevier Science & Technology, 2011. ISBN: 9780123859631.

[44] Jeffrey Ichnowski. *Configuration Space Visualization of 2-D Robotic Manipulator*. `https://cs.unc.edu/~jeffi/c-space/robot.xhtml`. 2010.

[45] Jeffrey Ichnowski and Ron Alterovitz. "Concurrent Nearest-Neighbor Searching for Parallel Sampling-based Motion Planning in SO(3), SE(3), and Euclidean Spaces". In: *Algorithmic Foundations of Robotics (Proceeding of WAFR)*. Springer, 2018.

[46] Jeffrey Ichnowski and Ron Alterovitz. "Fast Nearest Neighbor Search in SE(3) for Sampling-Based Motion Planning". In: *Algorithmic Foundations of Robotics XI*. Springer, 2015, pp. 197–214.

[47] Jeffrey Ichnowski and Ron Alterovitz. "Motion Planning Templates: A Motion Planning Framework for Robots with Low-power CPUs". In: *Proceedings IEEE Int. Conf. Robotics and Automation (ICRA)*. May 2019.

[48] Jeffrey Ichnowski and Ron Alterovitz. "Parallel sampling-based motion planning with super-linear speedup". In: *Proceedings IEEE/RSJ Int. Conf. Intelligent Robots and Systems (IROS)*. Oct. 2012, pp. 1206–1212.

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

[50] 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.

[51] Jeffrey Ichnowski, Jan Prins, and Ron Alterovitz. "Cloud-based Motion Plan Computation for Power-Constrained Robots". In: *Algorithmic Foundations of Robotics (Proceeding of WAFR)*. Springer, 2016.

[52] Masayuki Inaba et al. "A platform for robotics research based on the remote-brained robot approach". In: *The International Journal of Robotics Research* 19.10 (2000), pp. 933–954.

[53] Piotr Indyk. "Nearest neighbors in high-dimensional spaces". In: *Handbook of Discrete and Computational Geometry*. 2nd ed. Chapmann & Hall/CRC, 2004. Chap. 39.

[54] Kazuaki Ishizaki et al. "A study of devirtualization techniques for a Java Just-In-Time compiler". In: *ACM SIGPLAN Notices*. Vol. 35. 10. ACM. 2000, pp. 294–310.

[55] ISO/IEC. *ISO International Standard ISO/IEC 14882:2017(E)—Programming languages—C++*. Standard. Geneva, Switzerland: International Organization for Standards (ISO), Dec. 2017.

[56] Benoît Jacob, Gaël Guennebaud, et al. *Eigen*. `http://eigen.tuxfamily.org`. 2018. (Visited on 09/01/2018).

[57] Sam Ade Jacobs 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.

[58] Sam Ade Jacobs 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.

[59] Norman P Jouppi et al. “In-datacenter performance analysis of a tensor processing unit”. In: *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ACM. 2017, pp. 1–12.

[60] 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.

[61] L E Kavraki et al. “Probabilistic roadmaps for path planning in high dimensional configuration spaces”. In: *IEEE Trans. Robotics and Automation* 12.4 (1996), pp. 566–580.

[62] Ben Kehoe et al. “A survey of research on cloud robotics and automation”. In: *IEEE Transactions on Automation Science and Engineering* 12.2 (2015), pp. 398–409.

[63] Ben Kehoe et al. “Cloud-based grasp analysis and planning for toleranced parts using parallelized Monte Carlo sampling”. In: *IEEE Transactions on Automation Science and Engineering* 12.2 (2015), pp. 455–470.

[64] Ben Kehoe et al. “Cloud-based robot grasping with the google object recognition engine”. In: *Proceedings IEEE Int. Conf. Robotics and Automation (ICRA)*. IEEE. 2013, pp. 4263–4270.

[65] Charles Pisula Kenneth et al. “Randomized Path Planning for a Rigid Body Based on Hardware Accelerated Voronoi Sampling”. In: *Algorithmic Foundations of Robotics (Proceeding of WAFR)*. 2000.

[66] Joseph T. Kider Jr. et al. “High-dimensional planning on the GPU”. In: *Proceedings IEEE Int. Conf. Robotics and Automation (ICRA)*. 2010, pp. 2515–2522. DOI: `http://dx.doi.org/10.1109/ROBOT.2010.5509470`.

[67] Michal Kleinbort, Oren Salzman, and Dan Halperin. “Collision detection or nearest-neighbor search? On the computational bottleneck in sampling-based motion planning”. In: *Algorithmic Foundations of Robotics (Proceeding of WAFR)*. Springer, 2016.

[68] Michal Kleinbort, Oren Salzman, and Dan Halperin. “Efficient high-quality motion planning by fast all-pairs r-nearest-neighbors”. In: *Proceedings IEEE Int. Conf. Robotics and Automation (ICRA)*. IEEE. 2015, pp. 2985–2990.

[69] Donald E. Knuth. *The art of computer programming, volume 3: (2nd ed.) sorting and searching*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 1998. ISBN: 0-201-89685-0.

[70] Jack B Kuipers et al. *Quaternions and rotation sequences*. Vol. 66. Princeton university press Princeton, 1999.

[71] HT Kung and Philip L Lehman. "Concurrent manipulation of binary search trees". In: *ACM Transactions on Database Systems (TODS)* 5.3 (1980), pp. 354–382.

[72] Eyal Kushilevitz, Rafail Ostrovsky, and Yuval Rabani. "Efficient search for approximate nearest neighbor in high dimensional spaces". In: *SIAM J. Computing* 30.2 (2000), pp. 457–474.

[73] Anthony LaMarca and Richard E Ladner. "The influence of caches on the performance of sorting". In: *Journal of Algorithms* 31.1 (1999), pp. 66–104.

[74] S M LaValle and J J Kuffner. "Rapidly-exploring random trees: Progress and prospects". In: *Algorithmic and Computational Robotics: New Directions*. Ed. by Bruce R Donald et al. Natick, MA: AK Peters, 2001, pp. 293–308.

[75] Steven M. LaValle. *Planning Algorithms*. Cambridge, U.K.: Cambridge University Press, 2006.

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

[77] Jed Lengyel et al. "Real-Time Robot Motion Planning Using Rasterizing Computer Graphics Hardware". In: *Proceedings ACM SIGGRAPH*. 1990, pp. 327–335.

[78] David Levinthal. *Performance Analysis Guide for Intel® Core™ i7 Processor and Intel® Xeon™ 5500 processors*. `http://software.intel.com/sites/products/collateral/hpc/vtune/performance_analysis_guide.pdf`. 2009.

[79] Maxim Likhachev, Geoffrey J Gordon, and Sebastian Thrun. "ARA*: Anytime A* with provable bounds on sub-optimality". In: *Advances in neural information processing systems*. 2004, pp. 767–774.

[80] Maxim Likhachev et al. "Anytime Dynamic A*: An Anytime, Replanning Algorithm." In: *ICAPS*. 2005, pp. 262–271.

[81] Songrit Maneewongvatana and David M. Mount. "It's Okay to Be Skinny, If Your Friends Are Fat". In: *Center for Geometric Computing 4th Annual Workshop on Computational Geometry*. 1999.

[82] James D. Marble and Kostas E. Bekris. "Asymptotically near optimal planning with probabilistic roadmap spanners". In: *IEEE Transactions on Robotics* 29.2 (2013), pp. 432–444.

[83] Paul E McKenney. "Memory barriers: a hardware view for software hackers". In: *Linux Technology Center, IBM Beaverton* (2010).

[84] Peter Mell and Tim Grance. *The NIST definition of cloud computing*. `http://dx.doi.org/10.6028/NIST.SP.800-145`. 2011.

[85] Maged M Michael and Michael L Scott. "Simple, fast, and practical non-blocking and blocking concurrent queue algorithms". In: *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*. ACM. 1996, pp. 267–275.

[86] Gordon E Moore. "Cramming more components onto integrated circuits". In: *Proceedings of the IEEE* 86.1 (1998), pp. 82–85.

[87] Marco Morales et al. "A machine learning approach for feature-sensitive motion planning". In: *Algorithmic Foundations of Robotics VI*. Springer, 2005, pp. 361–376.

[88] David M Mount. *ANN programming manual*. Tech. rep. Dept. of Computer Science, U. of Maryland, 1998.

[89] Marius Muja and David G. Lowe. "Fast Approximate Nearest Neighbors with Automatic Algorithm Configuration". In: *Int. Conf. Computer Vision Theory and Application (VISSAPP)*. INSTICC Press, 2009, pp. 331–340.

[90] Sean Murray et al. "Robot Motion Planning on a Chip." In: *Proceedings Robotics: Science and Systems*. 2016.

[91] Michal Nowakiewicz. "MST-based method for 6DOF rigid body motion planning in narrow passages". In: *Proceedings IEEE/RSJ Int. Conf. Intelligent Robots and Systems (IROS)*. IEEE. 2010, pp. 5380–5385.

[92] Joe Olivas, Mike Chynoweth, and Tom Propst. *Benefitting Power and Performance Sleep Loops*. `https://software.intel.com/en-us/articles/benefitting-power-and-performance-sleep-loops`. 2015.

[93] 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`.

[94] Michael Otte and Nikolaus Correll. *Path Planning with Forests of Random Trees: Parallelization with Super Linear Speedup*. Tech. rep. CU-CS 1079-11. Department of Computer Science University of Colorado at Boulder, Apr. 2011.

[95] Jia Pan, Christian Lauterbach, and Dinesh Manocha. "g-Planner: Real-time motion planning and global navigation using GPUs". In: *AAAI Conference on Artificial Intelligence (AAAI-10)*. July 2010, pp. 1245–1251.

[96] E. Plaku and L. E. Kavraki. "Distributed Sampling-Based Roadmap of Trees for Large-Scale Motion Planning". In: *Proceedings IEEE Int. Conf. Robotics and Automation (ICRA)*. Apr. 2005, pp. 3879–3884.

[97] Erion Plaku and Lydia E Kavraki. "Quantitative analysis of nearest-neighbors search in high-dimensional sampling-based motion planning". In: *Algorithmic Foundation of Robotics VII*. Springer, 2008, pp. 3–18.

[98] John H. Reif. "Complexity of the Mover's Problem and Generalizations". In: *20th Annual IEEE Symp. on Foundations of Computer Science*. Oct. 1979, pp. 421–427.

[99] Rethink Robotics. *Baxter Research Robot*. http://www.rethinkrobotics.com/product/baxter-research-robot/. 2013.

[100] Markus Rickert and Andre Gaschler. "Robotics library: An object-oriented approach to robot applications". In: *Proceedings IEEE/RSJ Int. Conf. Intelligent Robots and Systems (IROS)*. IEEE. 2017, pp. 733–740.

[101] Samuel Rodriguez et al. "RESAMPL: A region-sensitive adaptive motion planner". In: *Algorithmic Foundation of Robotics VII*. Springer, 2008, pp. 285–300.

[102] ROS.org. *Robot Operating System (ROS)*. http://ros.org. 2012.

[103] Jan Rosell, Carlos Vázquez, and Alexander Pérez. "C-space decomposition using deterministic sampling and distance". In: *Proceedings IEEE/RSJ Int. Conf. Intelligent Robots and Systems (IROS)*. IEEE. 2007, pp. 15–20.

[104] Chris Sherlock, Andrew Golightly, and Daniel A Henderson. "Adaptive, delayed-acceptance MCMC for targets with expensive likelihoods". In: *Journal of Computational and Graphical Statistics* 26.2 (2017), pp. 434–444.

[105] Ken Shoemake. "Animating rotation with quaternion curves". In: *Proceedings ACM SIGGRAPH* 19.3 (1985), pp. 245–254.

[106] Chanop Silpa-Anan and Richard Hartley. "Optimised KD-trees for fast image descriptor matching". In: *Computer Vision and Pattern Recognition, 2008. CVPR 2008. IEEE Conference on*. IEEE. 2008, pp. 1–8.

[107] Inc. Snapsort. *Intel Atom Z530 vs N270*. http://cpuboss.com/cpus/Intel-Atom-Z530-vs-Intel-Atom-N270. 2018. (Visited on 09/01/2018).

[108] SoftBank Robotics. *NAO the humanoid robot*. https://www.softbankrobotics.com/emea/en/nao. 2018.

[109] Robert F Sproull. "Refinements to nearest-neighbor searching in k-dimensional trees". In: *Algorithmica* 6.1-6 (1991), pp. 579–589.

[110] Bjarne Stroustrup. *The C++ programming language*. Pearson Education, 2013.

[111] Ioan A. Şucan and Sachin Chitta. *Moveit!* http://moveit.ros.org. 2013.

[112] Ioan A. Şucan, Mark Moll, and Lydia E. Kavraki. "The Open Motion Planning Library". In: *IEEE Robotics and Automation Magazine* 19.4 (Dec. 2012), pp. 72–82. URL: http://ompl.kavrakilab.org.

[113] 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.

[114] Ioan Şucan and Lydia E Kavraki. "A sampling-based tree planner for systems with complex dynamics". In: *IEEE Transactions on Robotics* 28.1 (2012), pp. 116–131.

[115] Tiffany Trader. *TOP500 Reanalysis Shows 'Nothing Wrong with Moore's Law'*. `https://www.hpcwire.com/2015/11/20/top500/`. Nov. 2015.

[116] John D. Valois. "Lock-Free Linked Lists Using Compare-and-Swap". In: *Proceedings ACM Symposium on Principles of Distributed Computing*. 1995, pp. 214–222.

[117] Jur Van Den Berg, Dave Ferguson, and James Kuffner. "Anytime path planning and replanning in dynamic environments". In: *Proceedings IEEE Int. Conf. Robotics and Automation (ICRA)*. 2006, pp. 2366–2371.

[118] Gokul Varadhan and Dinesh Manocha. "Star-shaped Roadmaps - A Deterministic Sampling Approach for Complete Motion Planning". In: *Proceedings Robotics: Science and Systems*. 2005.

[119] Todd L. Veldhuizen. *C++ Templates are Turing Complete*. Tech. rep. Indiana University, 2003.

[120] Markus Waibel et al. "Roboearth". In: *IEEE Robotics & Automation Magazine* 18.2 (2011), pp. 69–82.

[121] Boris Yamrom. *Alpha Puzzle*. `https://parasol.tamu.edu/dsmft/benchmarks/mp/`. GE Corporate Research & Development Center.

[122] Anna Yershova and Steven M LaValle. "Deterministic sampling methods for spheres and SO(3)". In: *Proceedings IEEE Int. Conf. Robotics and Automation (ICRA)*. 2004, pp. 3974–3980.

[123] Anna Yershova and Steven M. LaValle. "Improving Motion-Planning Algorithms by Efficient Nearest-Neighbor Searching". In: *IEEE Transactions on Robotics* 23.1 (2007), pp. 151–157.

[124] Peter N Yianilos. "Data structures and algorithms for nearest neighbor search in general metric spaces". In: *Proceedings of the fourth annual ACM-SIAM Symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics. 1993, pp. 311–321.

[125] Sung-Eui Yoon and Dinesh Manocha. "Cache-Efficient Layouts of Bounding Volume Hierarchies". In: *Computer Graphics Forum*. Vol. 25. 2006, pp. 507–516.