TOWARD DYNAMICALLY SCALABLE OPEN-SOURCE MOTION PLANNING ON THE MOBILE EDGE AND IN THE CLOUD

BY

EMMANUEL GALLEGOS

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois Urbana-Champaign, 2023

Urbana, Illinois

Adviser:

Professor Nancy Marie Amato

ABSTRACT

This work presents a platform capable of deploying massively scalable and portable high performance software for robotics applications by utilizing containerization technologies. To this end, we employ a fully automated method for creating a swarm of containers among machines with homogeneous or heterogeneous computing power and operating systems that can work in tandem to perform large-scale distributed computation, including the ability to leverage robots as mobile edge devices. We demonstrate that this method can also be massively scaled on a large, shared HPC cluster, and evaluate the overhead associated with containerization by performing highly distributed motion planning calculations. For each of these platforms, we compare bare-metal and containerized runtime and scalability and show that our containerized platform is capable of fully utilizing the resources of the host machines while achieving the same nearly-linear scalability for parallel sampling-based motion planning algorithms that is possible on bare-metal implementations.

To my father, for working harder than anyone I know, all to give his family a better life.
To my mother, for instilling within me her life-long passion to learn and grow.
(She works really hard, too.)

ACKNOWLEDGMENTS

I would like to express my sincere gratitude towards my advisor, Dr. Nancy M. Amato, for her patience and support throughout the various stages of my academic career while at the University of Illinois Urbana-Champaign, and for her unwavering support for me and my work, even through the hard times.

I would also like to acknowledge the contributions from all the researchers hard at work on updating the Parasol Planning Library and the Standard Template Adaptive Parallel Library, toiling to get these libraries ready for large-scale open-source distribution. The work being done in the Parasol Lab is truly exemplary. In particular, I would like to acknowledge the contributions from PhD candidate Francisco Coral (Texas A&M), for his invaluable patience and generosity in sharing from his wealth of high performance computing knowledge and PhD student Victor Murta, for his help in getting our platform running on the Illinois Campus Cluster.

TABLE OF CONTENTS

CHAPTER 1 INTRODUCTION 1
1.1 Contribution 3
1.2 Overview 4

CHAPTER 2 PRELIMINARIES AND RELATED WORK 5
2.1 Motion Planning 5
2.2 Parallel Computing 9
2.3 Cloud Computing 12
2.4 Edge Computing 13
2.5 Parallel Motion Planning Methods 14
2.6 Leveraging the Cloud for Robotics 16
2.7 Leveraging Robots as Mobile Edge Devices 17
2.8 The Case for Open Source 18
2.9 Open-Source Libraries for Robotics 19

CHAPTER 3 METHOD OVERVIEW 23
3.1 Algorithm Overview 23
3.2 Containerization Platform Overview 27

CHAPTER 4 EXPERIMENTAL DESIGN 30
4.1 Experiment Overview 30
4.2 Robot Environments 31
4.3 Hardware Platforms 31
4.4 Software Platforms 32

CHAPTER 5 EXPERIMENTAL EVALUATION 33
5.1 Experiment 1 - Docker for Planning on One Machine 33
5.2 Experiment 2 - Docker Swarm for Planning with Homogeneous Cluster . . . 33
5.3 Experiment 3 - Singularity for Planning on a Single HPC Node 35
5.4 Experiment 4 - Singularity for Planning on Shared HPC's 36
5.5 Experiment 5 - Docker Swarm for Planning with Heterogeneous Cluster . . . 37

CHAPTER 6 CONCLUSION 38

REFERENCES 39

CHAPTER 1: INTRODUCTION

With the wide variety of distributed computing platforms now available to roboticists, there is an open-question of how to most efficiently utilize these varied systems throughout the various components of the robotic computational pipeline. Private servers, edge devices, GPU's, scalable cloud infrastructure, and private and shared computing clusters are only some of the computing resources that are now commonly available in both academia and industry. Meanwhile, sensing, perception, controls, and planning can all be highly computationally expensive tasks.

In order to take advantage of these various distributed computational modalities to solve these computationally intensive tasks, we require scalable algorithms, software to design, implement, and test those algorithms, and a means of porting this software to a wide variety of platforms for testing and deployment.

A Simple Motivating Example Consider a large, modern warehouse containing goods for a large-scale online retailer. Within the warehouse, dozens of autonomous robots are roaming around the floor, loading and unloading inventory, cleaning, and performing other tasks to keep operations running. Human workers are also moving through the space, performing maintenance, managing inventory and monitoring general operations to ensure important deadlines are met.

A central controller in the warehouse is passing data from the onboard sensors on the robots and from sensors in the environment to a small computing cluster in the cloud, which then generates task and motion plans for the robots. The robots themselves split their computing power between low-level trajectory planning, in order to execute the larger motion plans passed to them, and performing simultaneous location and mapping (SLAM), as the environment is frequently changing and needs to be constantly mapped. They also pass their data back to the controller to continue the cycle of sensing, computing, and acting that is needed to keep operations running smoothly.

Suddenly, a laptop battery inside of a pallet that was improperly packed is punctured and spontaneously ignites, starting a chemical fire in the center of the warehouse. Sensors in the environment quickly detect the fire's existence, and communicate this information to the central controller. The controller then rapidly performs several actions:

1. The controller sounds the alarm, letting the humans in the warehouse know to evacuate in a safe and orderly manner.

2. The controller scales up the size of the virtual computing cluster located in the cloud that performs the day-to-day task and motion planning, as well as the number of GPUs available.

3. The controller executes a modal switch that shifts the way computing tasks are distributed among the available resources. The additional cloud-based GPU resources are enlisted to perform collision-checking [1] for massively-parallelized decoupled PRM calculations [2] with inter-robot collisions resolved using a parallelized hierarchical conflict-based search [3]. The robots that are equipped with firefighting capabilities are instructed to move to an open area near the walls, turn off all non-critical functions, and join a Docker swarm comprised of all the in-house robots, servers, and possibly personal computers, shifting motion planning calculations like nearest neighbors distance calculations out of the cloud and onto local machines, leveraging the robots as mobile edge devices to reduce latency.

Through intelligent planning and perfectly leveraging the distributed sensing and computing resources available, the robots manage to execute a plan to put out the fire while at no point obstructing any people from leaving. The company reports no casualties, and only minimal losses in property damage.

For this heroic feat of engineering to be accomplished, several key pieces of infrastructure need to have been in place.

- Firstly, each robot needed to be able to operate as a mobile edge device, capable of sensing and working together to perform important real-time calculations.

- Secondly, a scalable cloud-based computing service needed to be readily available to handle a large increase in computing requirements.

- Thirdly, a policy needed to be implemented to determine when and how to dynamically switch computing models to address the needs of the specific task.

- Fourthly, scalable and efficient parallel algorithms needed to be available to take advantage of the increased computing resources available.

- Lastly, all of these algorithms needed to be readily accessible through portable software that could run on a variety of different computing platforms.

Current Solutions While all of these components individually exist, and have been implemented to solve a variety of use cases, there have been few attempts to combine them in a single ecosystem. This is partly just a natural result of ubiquitous cloud-based computing services being a relatively recent technology, but also in large part stems from the lack of a unified, portable, and open-source motion planning software suite. Libraries like the Open Motion Planning Library (OMPL) [4] have made great strides in bringing open-source motion planning software to the masses, but its use cases are limited relative to the grand scale of potential motion planning problems, with little to no support for multiple agents or parallel planning algorithms. Other autonomous robot software suites, like Baidu Apollo, have admirably brought autonomous vehicle software and hardware development platforms (including motion planning algorithms) to open-source distribution with support for cloud services, but are limited to the realm of autonomous vehicles [5].

Our Solution This work presents the Parasol Planning Library (PPL) as a promising general-purpose and open-source task and motion planning library with support for advanced parallel methods using the underlying Standard Template Adaptive Parallel Library (STAPL), which, combined with popular open-source containerization technologies and a scalable cloud-based computing platform, can advance the field of robotics toward dynamically scalable motion planning on the mobile edge and in the cloud.

To this end, this work proposes, implements, and evaluates a containerized platform capable of solving large-scale distributed motion planning algorithms using the Parasol Planning Library that is able to run on a distributed cluster of machines with heterogeneous computing power and operating systems, and presents a fully automated method for deploying this technology using Docker Swarm. This work also presents a simple protocol to transform this container technology to run on a large, shared computing cluster using Singularity.

Finally, this work evaluates the runtime and scalability of this distributed containerized platform as compared to bare-metal implementations, and demonstrates that this containerized platform is capable of achieving the same nearly-linear scalability when solving large-scale parallel sampling-based motion planning algorithms as traditional bare-metal implementations on a variety of computing platforms.

1.1 CONTRIBUTION

Key contributions of this thesis include:

- Design of a containerized platform to solve distributed motion planning algorithms that can be easily ported to any computing platform that supports containerization technologies (i.e. servers, PC's, cloud, HPC clusters).

- Demonstration of the portability and scalability of this method by testing it on a variety of different platforms including a single PC, a cluster of PC's with varied operating systems and architectures working in tandem, and a large-scale shared HPC cluster.

- Analysis of the inherent overhead associated with containerization technologies as it pertains to solving distributed motion planning algorithms by comparing container vs bare-metal performance for each of these computing modalities.

1.2 OVERVIEW

The rest of this work will be organized as follows:

Chapter 2 will provide a primer on the main technologies that this work posits should be incorporated into a single open-source motion planning software suite with the aim of standardizing motion planning techniques with access to varying computing platforms and sensing modalities. These technologies are parallel computing (CPU, GPU, fpga, etc), cloud computing, edge computing, and distributed sensing. It will evaluate the state of the art of these methods as they relate to motion planning and robotics as a whole. Finally, it will introduce the prior works that serve as a basis for our platform, as well as other related works and libraries that attempt to solve these challenges.

Chapter 3 will then formally introduce our platform and give in depth explanations of the algorithm we use to test our platform as well as a system overview of how the containerized platform operates on different computing modalities.

Chapter 4 will introduce our experimental design, in which we compare bare-metal and containerized runtime and scalability for large-scale distributed motion planning problems. It will also introduce an experiment to evaluate the scalability of a distributed computing platform utilizing computers with heterogeneous computing power and operating systems.

Chapter 5 will evaluate the results of the experiments presented in Chapter 4.

Chapter 6 will summarize the primary contributions of this work, its implications to the field of motion planning, and propose future work.

CHAPTER 2: PRELIMINARIES AND RELATED WORK

2.1 MOTION PLANNING

2.1.1 Definition

Jean-Claude Latombe [6] gives an excellent summary of the history and problem definition inherent to the field of motion planning. Simply put, the goal of motion planning is to find a *motion plan*, that is, a set of valid paths for a set of agents from a set of starting positions, or configurations, within an environment to a set of goal configurations in the same environment. As the purpose of this work is to explore the problem of motion planning in the context of applications relating to robotics, the term 'agent' and 'robot' may occasionally be used interchangeably, though it should be noted that there are several other applications for motion planning aside from robotics, such as in computational biology [7, 8], animation, and video gaming. In some examples, there may also be agents in the environment that are not robots, such as pedestrians or vehicles, though in describing these cases, more specific language will be used to avoid ambiguity.

2.1.2 A Simple Example

In the most basic version of the motion planning problem, there is only a single agent contained within a static, bounded environment, and the planner must generate a valid path from a single start configuration to a single goal configuration.

What is meant by 'valid' in this definition can vary, but most commonly refers to 'collision-free,' meaning the robot is at no point in collision with itself, the environmental boundaries, or any other agents or obstacles contained within the environment. The agent's 'configuration' refers to the complete and minimal description of its degrees of freedom.

For example, for a fixed-base 2-link robot in a 2-dimensional, rectangular workspace, as shown in Figure 2.1, the robot has two degrees of freedom, corresponding to the angle measurements of its two joints respectively. We can then define any configuration of this robot succinctly as a vector, $v = [\theta, \phi]$ and describe the domain of the agent's possible configurations as the set C:

$$C : \big[\theta, \phi \mid \theta \in [0, 2\pi), \phi \in [0, 2\pi)\big] \tag{2.1}$$

This is often referred to as the *configuration space* or *C-space* of the agent. Note that in this definition, not all vectors $v = [\theta, \phi]$ in C are valid configurations. The environment

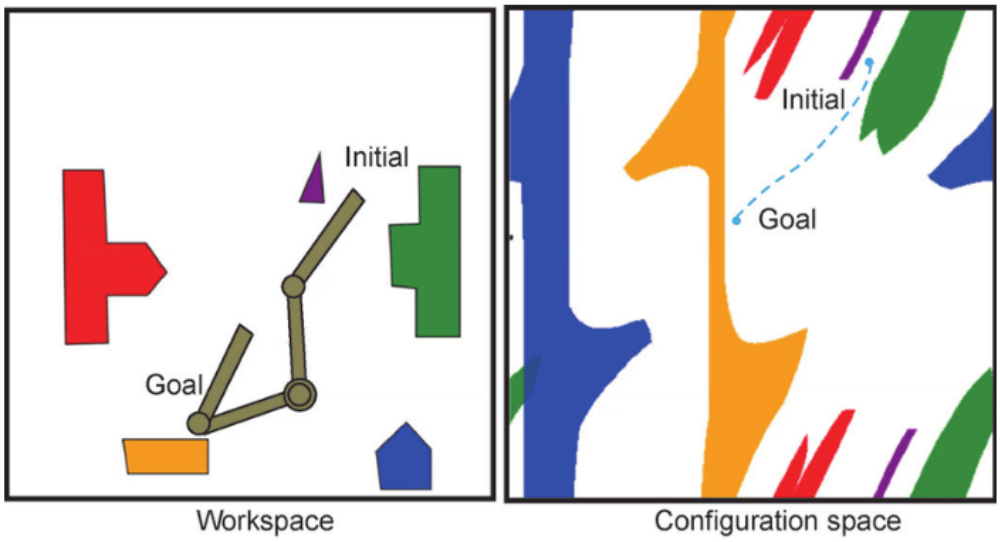


Figure 2.1: An example of a 2-linked robot with start and goal positions in the workspace (left) and corresponding start and goal configurations in the configuration space (right) [9, Fig. 5].

contains boundaries and obstacles, and some configurations would place the links of the robot in collision with each other. The invalid subset of the configuration space is known as the *obstacle-space* while its complement is the *free-space*.

The problem of motion planning can then be reduced to generating a set of valid configurations between the start and goal configurations.

2.1.3 Sampling Based Motion Planning Methods

While there are many methods for solving motion planning problems, many motion planning methods largely revolves around *sampling-based* methods. A brief exploration of other planning methods can be found in [6], such as those that rely on visibility graphs, approximate cell decomposition, or potential fields.

Sampling based methods usually rely on generating a subset of the configuration space that models the connectivity of the C-space reasonably well and can be efficiently searched for optimal paths. A graph-based data structure is commonly used to represent this subspace.

The Probabilistic Roadmap The Probabilistic Roadmap [10] (PRM) is a common implementation of the sampling-based motion planning strategy. In this strategy, the start and goal configurations of the robot are first added as vertices to a graph data structure called the *roadmap graph*. Then, configurations are randomly sampled from the whole configuration space. Configurations that are valid are added to the roadmap graph. Then, a *local planner*, which is capable of finding local paths between configurations in the roadmap, is used

to connect the sampled configurations to other configurations in the roadmap graph with edges. An algorithm like Dijkstra's or A* can then be used to query if a path exists between the start and goal configurations through the sampled configurations in the roadmap graph and return that path. If a path is not found, then the algorithm can repeat the process of sampling and connecting vertices until either a path is found, or some other stop condition is met, such as a time limit. This algorithm is said to be *probabilistically resolution complete*, in that it is guaranteed to find a solution if one exists given enough time and given that the resolution of the local planner is sufficient to never falsely classify an edge between two configurations as valid.

The Rapidly Exploring Random Tree The Rapidly Exploring Random Tree [11] (RRT) is another common strategy used to solve the sampling-based motion planning problem. In this method, an iterative approach is used where a random configuration is sampled, then the nearest vertex in the roadmap to the new sample is selected, and a new vertex is created by extending the nearest vertex toward the randomly generated sample. If the new sample is valid, it is added to the tree along with an edge connecting from the nearest vertex. This allows the tree to efficiently explore the configuration space.

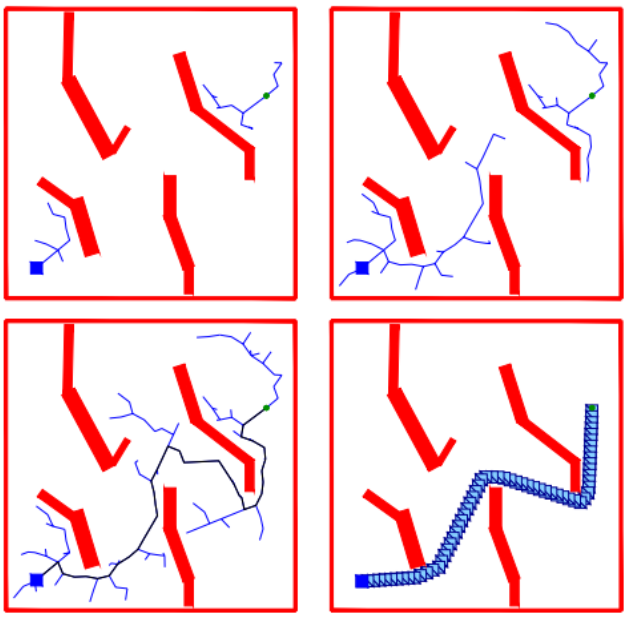


Figure 2.2: An example of two RRTs growing toward each other to find a path between a start and goal configuration [12, Fig. 6].

For a given start and goal configuration, the algorithm can be modified to alternate between expanding two separate RRTs [12] toward one another, one initialized at the start and one at the goal, as shown in Figure 2.2. One tree samples a random intermediary

configuration and tries to grow in that direction. It adds the farthest valid configuration generated toward that intermediary goal configuration (including the configuration itself if it was reached), then the other tree tries to extend outward to connect to the new configuration as well. If it succeeds, then the trees are connected and a path can be returned between the start and goal configurations. If it does not succeed, the trees switch roles and repeat the process. Like PRM, this process can iterate until a path is found or some other metric has been satisfied, like time elapsed, number of nodes generated, etc.

Both the PRM and RRT algorithms have been extended and modified in numerous ways, including the well known variations PRM* and RRT* [13] that provide asymptotic optimality.

2.1.4 Additional Complications

It should be evident that motion planning problems could be made arbitrarily more complicated in a variety of ways. The robot's geometry could be significantly more complex, for example, with many degrees of freedom. Additional constraints could be put on the robot such as velocity constraints or time constraints related to the dynamics of the robot and obstacles [14]. A robot living in three dimensions could have three degrees of freedom representing its x, y, and z coordinates, but may only be able to travel forward or backward, and cannot arbitrarily translate its position from side to side without turning [15]. Additionally, the problem could involve creating motion plans for tens or hundreds of robots [16]. The last case, where a plan must be created for multiple robots is commonly referred to as *multi-agent path finding* (MAPF) or multi-agent motion planning.

2.1.5 Multi-Agent Motion Planning

Multi-agent path finding is a well-defined problem in the field of motion planning [17]. In brief, multi-agent motion plans are often evaluated by either the makespan of the agents' individual path costs (the maximum cost among all paths) or the sum of costs of the individual paths, though other metrics can be used as well, particularly those that attempt to minimize other qualities, like energy or fuel expenditure. Some variants, called "Anonymous MAPF" of the problem generalize the problem so that rather than a specific start and goal pair of configurations for each robot, there are merely two sets of starts and goals, and which robot reaches which goal is not important. Other variants generalize to task planning, where there are multiple tasks that must be accomplished, and once a single task is completed, the robot can move onto another task [17]. This can be combined with anonymous MAPF to create task and motion plans like in the scenario introduced in chapter 1.

Multi-agent motion planning methods broadly fall into three categories:

1. **Coupled Methods**: These methods combine the degrees of freedom of all the agents into a single system, and plan in the joint configuration space. These methods are probabilistically resolution complete. However, these methods get significantly harder to solve as more robots are added and the dimensionality of the space increases. Many variations can also guarantee optimality [13].

2. **Decoupled Methods**: These methods plan for each robot individually within their own configuration spaces, and then resolve conflicts between the individual robot paths after [18]. This resolution can involve planning for each robot sequentially, imposing constraints on the next robot's plan, which compound with each additional robot. It can also involve planning the robots' paths agnostically of each other and attempting to resolve conflicts after. These methods are often more practical, as planning in lower dimensional spaces is significantly easier, but usually cannot guarantee optimality or completeness.

3. **Hybrid Methods**: These methods seek to combine the strengths of coupled and decoupled methods by attempting to primarily plan in decoupled spaces and only planning in coupled spaces when necessary [3, 19]. These methods ensure optimality and try to leverage the speed of decoupled planning as much as possible.

Distributed vs Centralized These algorithms can also be separately classified as being distributed or centralized. Distributed algorithms rely on computation being distributed among the various agents, and communicating with each other to find such a final solution. Centralized algorithms, on the other hand, assume one controller that computes the motion plan for each agent [20]. This terminology can be a bit obfuscatory, however, because that central controller is free to leverage distributed computing technologies such as cloud computing, or edge computing — even distributing computation among the robots themselves!

2.2 PARALLEL COMPUTING

Parallel Computing refers to computation involving multiple computing units. A larger problem is broken down into smaller sub-problems, and these sub-problems are solved in tandem by multiple processing elements.

Sometimes the tasks allocated to the processing elements are heterogeneous. For example, in a word processor application, one thread may be responsible for constantly running a spell checker, one thread may run a grammar checker, one or more threads may manage the user interface, while another thread manages user input [21].

For other applications, the sub-problems may be virtually identical, but with different inputs. For example, GPU's, with their huge number of cores, are exceptionally good at executing many similar instructions in parallel. This makes them excellent for computing solutions to problems like graphical rendering, which involve copious simple but repetitive calculations on large data sets [22].

Some popular libraries for high performance computing seek to take advantage of shared-memory architectures at the thread level [23], while others work in distributed memory across a large number of computing nodes [24].

Inherently Sequential Algorithms For some computing problems, the most popular and effective algorithms that solve these problem are *inherently sequential*, meaning the algorithm cannot be divided into independent tasks that can be executed concurrently by multiple processors or threads. This could be because of data dependencies in the sub-problems, for example, where each sub-problem depends on the results of prior sub-problems in a serial manner [25].

In order to create a parallelizable solution to these problems, different algorithms must be used that are able to reduce or eliminate the serial dependencies in the sub-problems. Finding these algorithms often requires analyzing the structure of the problem and finding clever ways to sub-divide the problem into parallelizable sub-problems so a parallel algorithm can be used.

Unfortunately, many times the parallelizable algorithms that solve these problems can have worse space or time complexities than the serial versions, or otherwise introduce additional computation that was not originally required in the serial version. In these cases, it is worth carefully considering if a sequential or parallel algorithm should be used. The "best" answer will usually depend on the expected inputs to the problem, the technical specifications of the distributed computing platform (software and hardware) [26], the relative programming complexity of the two algorithms [27], and the overhead cost of communication relative to the speedup attained by parallelizing the computation in the algorithm [28].

2.2.1 Challenges

Complexity Parallel algorithms are generally more complex than sequential algorithms because they involve multiple processes or threads executing simultaneously and coordinating their work. This introduces new challenges that do not exist in sequential algorithms, such as:

- Synchronization: In a parallel algorithm, multiple processing elements may need to access the same data or resources simultaneously, which can result in conflicts or race

conditions [29]. Synchronization mechanisms such as locks, semaphores, or communication and execution barriers must be used to ensure that the processes or threads cooperate properly.

- Load balancing: In a parallel algorithm, workload distribution can be a challenge. If one processing elements finishes its work much faster than the others, it may have to wait for the slower processing elements to catch up. Load balancing techniques must be used to ensure that the workload is distributed evenly among processing elements to minimize downtime [30].

- Communication: In a parallel algorithm, processing elements may need to communicate with each other to exchange data or coordinate their work. This introduces additional complexity, as communication can be a bottleneck and may require specialized communication patterns and algorithms [28].

- Computing Scalability: Parallel algorithms must be designed to scale efficiently as the number of available computing units increases. This requires careful consideration of all of the above factors, and more.

Addressing the Challenges of Parallel Computing Because of the challenges in parallel computing described above, managing large-scale, object-oriented software that seeks to take advantage of parallel algorithms can be a difficult task, requiring specialized parallel programmers. To solve this problem, protocols like the Message Passing Interface (MPI) [24] have been developed to provide a standard mechanism to allow for communication between processing elements in a distributed program. Similarly, high performance computing libraries like Open Multi-Processing (OpenMP) [23] offer convenient mechanisms for implementing parallelism while abstracting away more complex thread-level logic.

A popular lower-level library for parallelism in Unix-based systems is POSIX threads (Pthreads) [31]. Another popular parallel computing library, Compute Unified Device Architecture (CUDA) [32], provides support for parallel computing on NVIDIA GPU's. Another library, the Open Computing Language (OpenCL) [33], introduces support for parallel computing on heterogeneous systems, and can leverage many types of accelerators such as GPU's, CPU's, and FPGA's. Charm++ provides an excellent framework with an adaptive runtime system for parallelism in C++ [34].

In our work, we leverage the Standard Template Adaptive Parallel Library (STAPL) to integrate parallelism into the Parasol Planning Library.

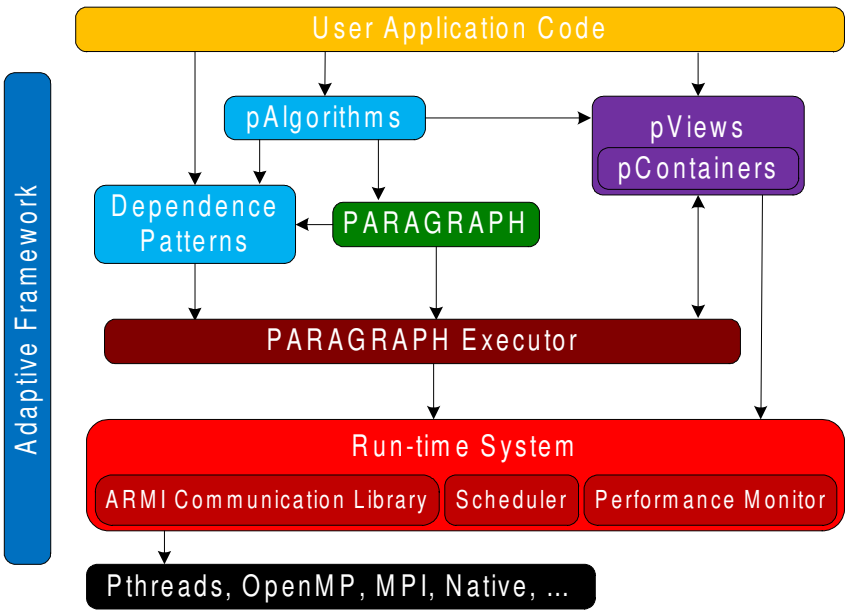


Figure 2.3: STAPL framework diagram, showing the major components included in the library [2, Fig. 2].

2.2.2 The Standard Template Adaptive Parallel Library (STAPL)

The Standard Template Adaptive Parallel Library (STAPL) is a framework for the development of parallel code in C++. It is comprised of parallel algorithms (pAlgorithms), parallel data structures (pViews [35] and pContainers [36]), dependence patterns (Skeleton framework [37]), parallel task graph scheduler and executor (PARAGRAPH), runtime system (STAPL-RTS [38]), and a communication library (ARMI [39]), as shown in figure 2.3.

The STAPL parallel data structures are considered *p_objects* by the runtime system, distributing them into STAPL *locations*. A STAPL *location* is comprised of a processing element (PE) combined with a virtual address space. For the purpose of this work, when we reference a unique STAPL location, we will be referring to an individual process with a unique MPI rank bound to a unique computing unit with its own virtual address space.

2.3 CLOUD COMPUTING

The National Institute of Standards and Technology provides an excellent definition of cloud computing: "Cloud computing is a model for enabling convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction" [40].

Some of the major advantages of cloud computing involve its dynamically scalable nature. Cloud computing allows for dynamic scalability of the amount of computing resources available, whether it's scalable storage, memory, or computational units (GPUs, CPUs, etc). It can also offer high levels of flexibility, in that cloud computing allows for offloading the complexity of systems like network infrastructure and memory management to the cloud service provider. Modern cloud service providers offer large amounts of services that provide out-of-the-box functionality that allows developers to focus on the business logic of their particular application [41].

2.3.1 Massively Scalable Distributed Parallel Computing

In [42], a method was proposed to create scalable containers for HPC systems. They are able to achieve performance and scalability with their containers equal to bare-metal implementations by using an internal tool within the container that swaps the version of MPI used in their container with the version used on the host HPC that has been optimized for performance on large-scale HPC's. These containers are then able to leverage the host machine's hardware acceleration capabilities. It has still not found widespread adoption, but serves as proof-of-concept that containerization technologies are capable of achieving bare-metal performance on HPC's.

In [43], a method for employing Docker containers for HPC applications was introduced. This method utilizes Docker's native Swarm mode in order to orchestrate distributed containers while using MPI as the underlying communication interface. Once all the available machines are manually added to the Docker swarm, a simple script can be used to dynamically spin up any number of Docker containers among the available nodes in the Swarm. This number can also be dynamically scaled at runtime.

This method serves as the basis of the distributed container orchestration for our platform, with additional modifications to automate the connection process of the Swarm, and to allow the platform to be compatible with Docker 2 and other Linux distributions within Docker aside from Alpine, the operating system used within the containers in the original work.

2.4 EDGE COMPUTING

Edge computing seeks to provide a means to bring remote computation closer to the user and source of data as compared to traditional cloud computing [44].

Some advantages of edge computing include improved network bandwidth utilization, reduced latency and reduced network congestion. Edge computing allows data to be processed closer to the source, reducing the amount of time it takes to transmit data to a centralized location and receive a response. This can be particularly important for applications that

require real-time processing, such as IoT sensors or autonomous vehicles. In addition, by processing data locally, edge computing can reduce the amount of data that needs to be transmitted over the network, which can help to optimize bandwidth utilization and reduce network congestion. Edge computing also allows for greater flexibility in terms of where and how data is processed, enabling developers to adapt to changing needs and requirements, and can reduce operation costs.

2.5 PARALLEL MOTION PLANNING METHODS

Critical to dynamically scalable motion planning algorithms among varied computational resources is the ability to apply parallel-programming based high performance computing techniques to motion planning problems.

Parallel Sampling-Based Motion Planning with Workspace Subdivision In [2], a method was proposed to parallelize any sampling-based motion planning method by subdividing the workspace into regions according to a region graph, and then assigning each processor the task of constructing a roadmap within one of those regions. The particular sampling based method used within each region can be interchanged. Then, attempts were made to connect candidate nodes from connected component within each region to candidate nodes in adjacent regions, with adjacency defined by the region graph. These could be used to construct the large-scale roadmap similar to those created by the original PRM algorithm.

This work was extended with a custom radial subdivision C-space subdivision strategy for optimizing RRT calculations in parallel [45]. This strategy produced nearly linear scaling with hundreds of processors.

The method in [2] is implemented, with some adaptations, in the Parasol Planning Library, and serves as the primary algorithm for evaluation of our proposed containerized platform.

Parallel Search-Based Motion Planning In [46], a method was proposed to perform a multi-agent A* search for parallel and distributed systems, which can help find optimal paths for loosely-coupled multi-agent path finding problems with super-linear speedups on distributed architectures.

In [47], the authors propose a massively parallelized lazy search algorithm that, while not guaranteed to produce optimal solutions, leverages asynchronous lazy edge evaluations leading to a significant improvement in planning time. For many real-time use cases, speed may be preferable to optimality.

Some methods leverage a combination of motion planning techniques, like using sampling based motion planning to calculate initial motion plans with static obstacles, along with

parallelized trajectory search-based planning for generating smooth motion plans in the presence of dynamic obstacles [48].

GPU and FPGA Acceleration for Motion Planning Some parts of the robot motion stack are natural candidates for acceleration using GPUs and other non-CPU computing units. Any methods involving computer vision or machine learning can often leverage these devices. Within the realm of motion planning in particular, there is less research in this area, though there is certainly still some interesting research being done.

For example, in [49], algorithms are proposed to leverage GPUs for several components of motion planning, such as sampling, nearest neighbor calculations, local planning, collision detection, and graph searching. The system is only tested up to 6 degree-of-freedom systems, so it is unclear if this particular implementation would be suitable for the large-scale type motion planning problems described in Chapter 1, however, it is important to note that research is being done in this area.

In [1], massive parallelization of RRT and RRT* is performed by leveraging NVidia CUDA GPUs for parallelizing collision-checking.

In [50], FPGAs or field programmable gate arrays are leveraged for a variety of motion planning tasks involved in PRM construction. [50] demonstrates that the reconfigurable processor is capable of serving as a dedicated motion planning processor, a co-processor (for distributed experiments), and as a dedicated collision detection processor, like in [1]. FPGAs for collision detection is further optimized in [51] by leveraging a caching system using SRAM. FPGA based motion planning using the search based A* method is shown to allow for motion planning for dual-armed robot manipulators in [52].

Synthesis of Parallel Motion Planning Techniques While all of these techniques attempt to solve the motion planning problem, or a part of the motion planning problem in parallel, there are deficiencies in each technique. Some techniques cannot guarantee optimal plans [47, 48], while others do not obviously scale well into the multi-agent motion planning domain without substantial modification [2, 46]. Some are able to provide excellent scalability for pieces of the motion planning problem [1, 49, 50, 51, 52]. Finding generalizable distributed motion planning techniques that guarantee optimal solutions and scale well with regards to both the size of the problem (soft parallel scaling) and the number of computing units available given a fixed problem specification (hard parallel scaling) is still an open problem in the field of motion planning, and one that is of critical importance in order to effectively leverage cloud and edge based computing to solve motion planning problems.

2.6 LEVERAGING THE CLOUD FOR ROBOTICS

As one of the primary goals of this research is to create a distributed motion planning platform capable of running in the cloud, of interest to this work is examining the state of the art of other cloud-based motion planning methods. Existing methods that seek to use cloud architectures for motion planning vary greatly, from moving highly specific tasks to the cloud, like in [53], where motion planning computation is done in the cloud to satisfy a low-power constrained robot, to shifting virtually all robot control into an all-in-one, unified cloud service for robot planning and control [54], [55]. Other methods attempt to pre-compute roadmaps in the cloud to aid in real-time queries [56].

Cloud-Based Robotics as a Service In [54], an open service-based framework for robotics in the cloud is proposed and tested for a single robotic manipulator. This work also analyzes the trade-offs in improving computation time while creating additional latency as a result of moving various operations into the cloud. Of note is the result that motion planning in particular could be made significantly faster, outweighing the latency introduced by the communication overhead.

In [55], another full-blown service-based framework for robotics in the cloud is proposed for automatic data center monitoring. This system utilizes the Robot Operating System framework (explored more in depth in chapter 2.9.1) for motion planning in the cloud to plan for a single robot. It also briefly discusses the trade-off between local and cloud computing for computing the motion plan.

Both of these services, while promising, are still highly theoretical and have only been implemented to solve simple, single-robot motion plans. These methods are also specific to cloud computing and do not generalize to other computing modalities.

Serverless Cloud Operations (Function-as-a-Service) Some works, like [57], leverage dynamically scalable cloud-based lambda functions (Function-as-a-Service), where a controller can request an arbitrary number of instances to try to solve RRT or RRT* operations until one returns a solution. Based on the needs of the specific task, the number of function calls can be scaled arbitrarily, and because these solutions do not need to communicate with each other, the system is agile and relatively easy to implement. Similarly, the method used in [58] leverages a dynamic number of lambda instances to each compute PRM's over the whole C-space, which are then combined in a fashion similar to that described in [2]. This latter method, however, ran into several bottlenecks, including redundant computing between instances that did not generally help improve connectivity.

Combining the subdivision methods proposed in [2] and [45] with the serverless lambda operations as described above may actually provide an interesting avenue of future research,

though work would still need to be done to efficiently generalize the problem to the large-scale multi-agent scenario.

2.6.1 Synthesis of Prior Work for Cloud Based Motion Planning

While many of the above systems provide possible solutions to creating a unified cloud-based motion planning platform, with some going so far as to touch on some of the core concepts of this work, none are able to offer a platform that can generalize to employing any parallel motion planning strategy in the cloud or adapt to other non-cloud based computing modalities.

This is not to say that their contributions are not valuable; on the contrary, they offer practical solutions to various use cases and provide excellent solutions to many parts of the robot stack, in addition to laying important groundwork for future research to come.

However, of important note to all cloud operations is the potential latency introduced in cloud-based computation [59, 60, 61, 62]. Due to the ubiquitous nature of these latencies in multiple domains of cloud computing, a growing area of research is evolving to study the trade-offs between cloud, fog, and edge computing and determining optimal policies for when and how to use these services to optimize for low-latency [63, 64] and low-cost [65, 66].

2.7 LEVERAGING ROBOTS AS MOBILE EDGE DEVICES

A large body of methods require robots to do heavy calculations for motion planning, controls, and sensor data processing. There are many methods that attempt to offload these calculations to edge devices or cloud resources. However, of interest to this work is the ability to treat robots themselves as roaming edge devices.

In one sense, any of the motion planning algorithms presented so far that are meant to run locally on the robots themselves can be thought of as leveraging edge computing if there is a central controller coordinating the planning on a larger scale. Generally, however, the method can be characterized by having a central controller which uses mobile robots to offload specific computations, such as motion planning, trajectory planning, SLAM, or visual odometry. In particular, leveraging the distributed sensing mechanisms present in mobile robots is of interest within this domain.

For example, in [67], the primary objective is to utilize the aggregated computational power of robots in a distributed robot system to perform DNN-based recognition in real-time. This collaboration enables robots to take advantage of the collective computing power of the robot-group to understand the consolidated raw data, while minimizing each robot's individual energy consumption. Although such collaboration could be extended to a variety

of systems, limited computing power and memory, scarce energy resources, and tight real-time performance requirements make this an interesting challenge within the field of robotics.

In [68], the use of robotics in IoT as edge devices is discussed at length, classifying robots as a special type of dynamic actuated edge device which can serve as a mobile sensor, actuator, or signal relay for short-range communication technologies like Bluetooth or Wi-FI.

2.8 THE CASE FOR OPEN SOURCE

Generally speaking, open source code refers to software that is made available to the public with its source code released under a license that allows users to view, modify, and distribute the code.

Some of the major advantages of open source software include community-driven development, cost-effectiveness, and transparency. Conversely, open source code can sometimes suffer from a lack of support, compatibility issues, and limited features. There are also positive and negative effects to security, as more eyes watching a code base makes it harder for a hole in security to slip by, though open source code is also freely available for malicious agents to inspect, and malicious agents can also try to inject flaws in security that may end up unnoticed by others.

2.8.1 Open-Source Motion Planning

In the realm of academia, open source software has undoubtedly had a great positive impact [69], allowing for researchers to share access to common methods and tools.

For example, a simple search for "TensorFlow" in Google Scholar, yielded only about 210,000 results, relative to about 2,300,000 found results for "Neural Network," anecdotally suggesting that roughly 10% of all academic literature relating to neural networks employed or referenced TensorFlow. Comparatively, "Open Motion Planning Library" generates roughly 2000 results, compared to 280,000 results for "Motion Planning," suggesting less than 1% of academic literature relating to motion planning mentions the Open Motion Planning Library.

While these search results are certainly a poor metric for precisely quantifying the prevalence of various open-source software libraries in their respective bodies of academic literature, they do highlight the simple fact that many motion planning researchers and specialists are likely employing a wide array of software platforms, many of which may be specialized proprietary systems, to solve motion planning problems which share much of the same fundamental structure.

2.9 OPEN-SOURCE LIBRARIES FOR ROBOTICS

As one of the primary concerns of this work is to move toward open-source scalable motion planning, we provide an overview of some popular open-source motion planning libraries and platforms.

2.9.1 The Robot Operating System (ROS)

The Robot Operating System (ROS) was introduced by researchers in 2009 by [70]. For a full explanation of the design goals and implementation strategy of ROS, this work recommends reading their full proposal. In brief, rather than operating as an operating system in the traditional low-level sense, the Robot Operating System seeks to provide a structure communications layer that allows for abstraction and modularization of various components of the robot operating stack. The open-source system is widely used both in industry and in academia, and in fact many of the papers cited in this work actually leverage ROS for deploying and testing their systems.

In brief, the Robot Operating System is broken down into four primary components: Nodes, Messages, Topics and Services.

- Nodes - These act as the core processing building blocks, or software modules. An example may be a node dedicated to receiving feedback from a camera module and doing basic pre-processing on the data, before publishing it to a topic.
- Topics - A topic is simply a string representing a name that can be referenced from other nodes. Multiple nodes can subscribe to or publish to a given topic. This can be thought of as a broadcast scheme allowing any node to broadcast a message to any other node.
- Messages - A message is a strictly typed data structure that is used for nodes to communicate with each other.
- Services - Where the broadcasting method of topic subscriptions and publishing is not appropriate, directed messages can be sent between nodes, called 'services' which operate similarly to requests. Services strictly belong to a single node, though many nodes can be serviced by that node.

There are several open-source motion planners available to use as ROS nodes, including popular motion planners built using the Open Motion Planning Library, described further in 2.9.2.

Deploying ROS in the Cloud In [71], the researchers propose ROSLink, which seeks to distribute functionality between ROS-enabled robots with the Internet of Things, using a proxy cloud server to link the ROS-enabled robots. A similar approach, Robot-Cloud, was introduced in [72], particularly targeting heterogeneous robots to support Software, Platform, and Infrastructure-as-a-Service applications.

Excitingly, in [73], the researchers propose a FogROS, a framework for automating fog robotics deployment by extending ROS to easily scale into the cloud to solve tasks such as SLAM, GPU-based grasp planning, and multi-core motion planning. Of most note to this work is their ability to speed up their motion planning by over 30 times, while only introducing a 0.5s delay due to network communication overhead. In particular, they leveraged the scalable motion planning strategy introduced in [53].

This relatively recent proposal, as of when this work was written, offers a hopeful insight into a future where ROS can easily be deployed and scaled in the cloud without requiring substantial knowledge of cloud infrastructure on the back-end.

2.9.2 The Open Motion Planning Library (OMPL)

Introduced in [4], the Open Motion Planning Library, is an open-source sampling-based motion planning library which contains many common planning algorithms ready to use out-of-the-box. It is an excellent tool as a teaching aid, and can interface with the Robot Operating System through the *MoveIt!* framework [74], allowing its algorithms to be deployed to actual robots with relative ease. It also includes a graphical user interface to allow for ease of use, particularly as a teaching aid.

One of the great strengths of this library is their willingness to accept contributions from other researchers. They understand that the success of the project is dependent on the library's ability to integrate new motion planning methods as they become standardized in the robotics industry.

Unfortunately, the OMPL does not offer robust methods for multi-agent motion planning, requiring significant hacks and workarounds just to implement a basic 2-manipulator system. It also only supports sampling-based methods. Additionally, it does not generally support multi-processing libraries on the back-end, such as parallel computing libraries like OpenMP or MPI or GPU/FPGA acceleration, and does not generalize to cloud-based computing infrastructure.

It does, however, support integration with simulating and planning environments such as OpenRAVE [75], CoppeliaSim, and MORSE [76].

2.9.3 Open-Source AV Platforms

Because this work is concerned with open-source motion planning systems, it is worth mentioning some of the large platforms for the autonomous vehicle stack that also include motion planning as part of their ecosystems. It should be noted, however, that none of these systems currently generalize to large-scale planning and vehicle routing, which is still a budding avenue of research. The analogous problem of planning in smaller-scale settings, like factory or warehouse floors, does not map perfectly to the large-scale problem of AV routing in city-scale environments, though the general organization of cloud, edge, and local computing resources operating in concert to solve dynamic AV routing problems is similar in nature to the problem introduced in chapter 1.

Apollo, also called Apolong, or Baidu-Apollo, is an open source platform for the entire software and hardware stack for autonomous vehicles. It includes integration of a cloud service platform, an open software platform, a hardware development platform, and a vehicle certification platform. Each of these platforms is modularized into smaller software modules that can work together to provide autonomous vehicle implementation solutions.

In [5], the researchers proposed and implemented a motion planning system for autonomous vehicles built using the Apollo project for inter and intra-lane driving. They were able to test the system by deploying it to dozens of autonomous vehicles for over 40,000 miles of drive time. Many other researchers have also leveraged the open-source software suite to design, implement, and test several other motion planning strategies, such as those found in [77, 78, 79].

There are also many other popular open-source platforms in the AV space that are worth noting as well, such as the CARLA simulator, Comma.ai, and Project Aslan. However, none of these libraries offer city-scale solutions for large-scale multi-agent routing, the analog to the smaller-scale problem this work is concerned with. Thus, this highlights the field-wide gap in knowledge present of how to best combine scalable algorithms with cloud and edge computing resources in order to solve planning problems in dynamic environments.

2.9.4 The Parasol Planning Library (PPL)

The Parasol Planning Library is an open source C++ task and motion planning library developed by the Parasol Lab. It serves as a platform for designing, implementing, and testing novel task and motion planning algorithms. The library is built in a modular fashion to allow for quick utilization of different motion planning components such as local planners, validity checkers, and distance metrics. It also includes a simulator and other visualization

tools to view roadmaps, paths, and more. For handling parallel planning algorithms, PPL utilizes the STAPL library. It is also constantly adding new features.

This work proposes the Parasol Planning Library, coupled with the high performance computing library STAPL, as a possible solution to the central problem of this work: implementing computing strategies in open-source software that can leverage heterogeneous distributed computing platforms to solve the motion planning problem in a dynamically scalable fashion.

CHAPTER 3: METHOD OVERVIEW

As explained in Chapter 1, the primary goal of this work is to develop a scalable and portable motion planning platform capable of engaging heterogeneous computing modalities, such as private servers, edge devices (including robots themselves), scalable cloud infrastructure, and private and shared computing clusters.

Such a platform would also need to be capable of dynamically shifting computing modalities between the available computational resources in order to fit the needs of the particular motion planning problem at hand. This could involve scaling up or down the cloud resources available, moving work from the cloud to local edge devices to reduce latency, engaging a high performance cluster to aid in times of heavy computational need, or any combination of these modalities.

To this end, this chapter presents an overview of our containerized platform which is capable of leveraging all of these computing modalities, including heterogeneous systems made up of machines with highly varied computing capabilities. We also present a thorough overview of the primary algorithm we use to evaluate our platform in Chapters 4 and 5.

3.1 ALGORITHM OVERVIEW

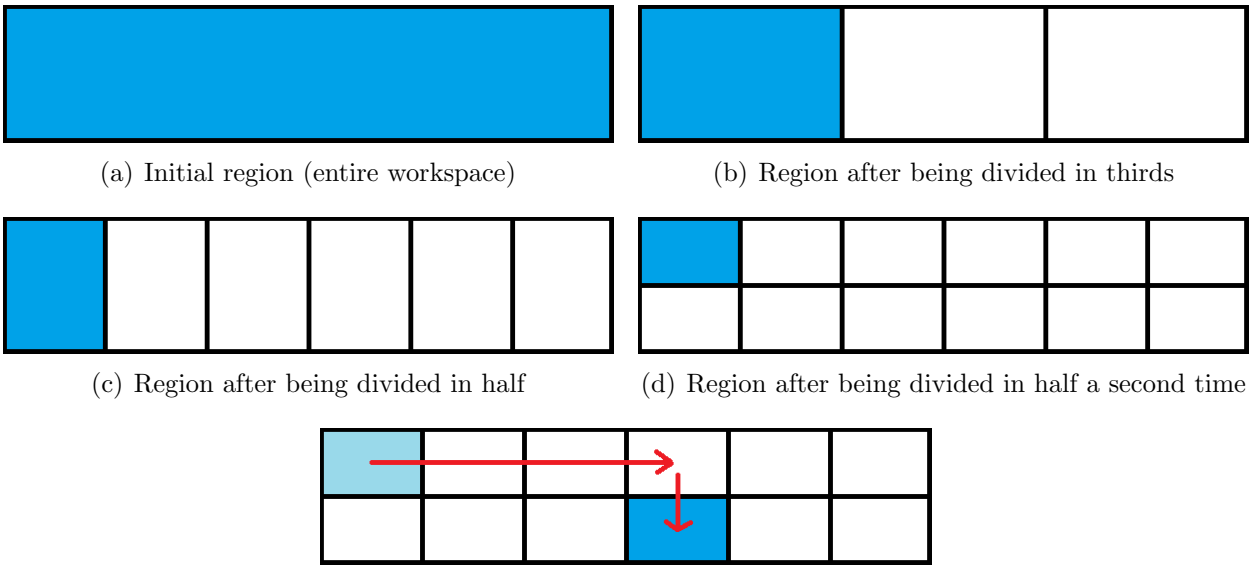
In this section, we describe the parallel algorithm used to evaluate our platform and include a thorough analysis of the space, work, and time complexity of the algorithm.

3.1.1 Subdivision Parallel PRM

Our adapted version of the Subdivision Parallel PRM algorithm from [2] serves as the primary tool for evaluating our scalable and portable motion planning framework. This version of Subdivision Parallel PRM, described in Algorithm 3.1, differs primarily from the original algorithm in [2] in the removal of the region graph, instead using basic arithmetic to subdivide the workspace into regions and determine adjacent regions as shown in Algorithm 3.2. Our implementation also employs different methods for determining candidate vertices to connect from adjacent region, as well as a different communication schema for sending candidate vertices to other STAPL locations.

Initially, the environment is subdivided into p regions, where p is the number of STAPL locations. The subdivision method prioritizes generating equally-sized regions with as close to equally sized lengths along each dimension as possible. This is done according to Algorithm 3.2 by first initializing the region to the entire workspace, as shown in Figure 3.1(a). Next, the prime factorization of p is computed as a list sorted from largest to smallest. The workspace is then greedily subdivided along the longest length by the next largest prime

factor until the space has been subdivided into p equally sized areas, as shown in Figure 3.1(b)—3.1(d). Lastly, the region is translated to a unique position in the grid as shown in Figure 3.1(e), according to its STAPL location. Then, within each region and in parallel, a probabilistic roadmap is constructed.



(a) Initial region (entire workspace)

(b) Region after being divided in thirds

(c) Region after being divided in half

(d) Region after being divided in half a second time

(e) Region after being translated to correct location

Figure 3.1: The construction of a unique region in a 2D workspace for $p = 12$. The prime factorization of 12 is $3 * 2 * 2$. Thus, the region is divided into thirds, then in half twice, each time along the longest dimension, before being translated to its correct position in the grid.

Next, each STAPL location determines candidate vertices to connect with adjacent regions by calculating the distance of each vertex in its region to the boundaries which are adjacent to neighbors. Any vertex that is within a parameterized threshold distance from a boundary shared by a neighbor will be considered a candidate to connect with vertices from that neighbor. This is just one method to determine candidate vertices to connect between regions, but is interchangeable with any other strategy, such as attempting to connect distinct component components in each region [2], or even attempting to connect all vertices in adjacent regions to each other.

As shown in Figure 3.2(b), every STAPL location is assigned a unique pair or triple representing its 2D or 3D position in the region grid. For a region (i, j) in a 2D grid, it's 4 adjacent neighbors would be represented by $(i-1, j), (i, j-1), (i+1, j)$, and $(i, j+1)$, assuming those indices do not lie outside the bounds of the grid. Thus, in step 3 of Algorithm 3.1, each region (i, j) will, in parallel and for each valid neighbor, find candidate vertices near the boundary adjacent to that neighbor. Then, in step 4, each region (i, j) will, in parallel,

Algorithm 3.1: Subdivision Parallel PRM

Data: An environment E, the total number of STAPL locations across all nodes p, the ID of the current STAPL location $loc \in [0, 1, \ldots, p-1]$, motion planner S

Result: A roadmap graph G

1 Independently and in parallel, subdivide the workspace among p locations as shown in Algorithm 3.2.
2 Independently and in parallel, construct roadmaps in each region using motion planner S.
3 Independently and in parallel, find candidate vertices near boundaries adjacent to neighbors.
4 In parallel, request candidate vertices from neighbors.
5 In parallel, attempt to connect candidate vertices from current region to candidate vertices from neighboring regions.

Algorithm 3.2: Workspace Subdivision

Data: An environment E, the total number of STAPL locations across all nodes p, the ID of the current STAPL location $i \in [0, 1, \ldots, p-1]$

Result: A region $R_i \subseteq E$ assigned to location i

1 $R_i = E$; `/* Init region to whole environment */`
2 $PF =$ GetPrimeFactorization(p) ; `/* Prime factors, sorted descending */`
3 **for** $factor \in PF$ **do**
4 $\quad R_i =$ DivideLongestDimension$(R_i, factor)$; `/* Subdivide region */`
5 **end**
6 $R_i =$ TranslateRegion(R_i, p, i) ; `/* To a unique location in grid */`
7 **return** R_i

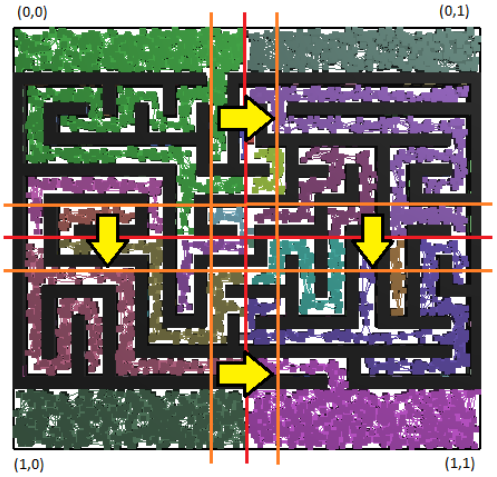
request candidate vertices to connect with from regions $(i-1, j)$ and $(i, j-1)$, if those neighbors exist, and send any candidate vertices requested by neighbors with higher indices, as shown in Figure 3.2(b). More generally, each location will request candidate vertices from adjacent neighbors with indices directly *below* theirs. Each region will then, in parallel and for each neighbor below it, try to connect each of the vertices it had already determined were adjacent to that neighbor, to the k nearest vertices from the set of vertices sent by that lower neighbor, thus stitching together the regions into a unified roadmap graph, as shown in Figure 3.2(c).

3.1.2 Algorithm Complexity Analysis

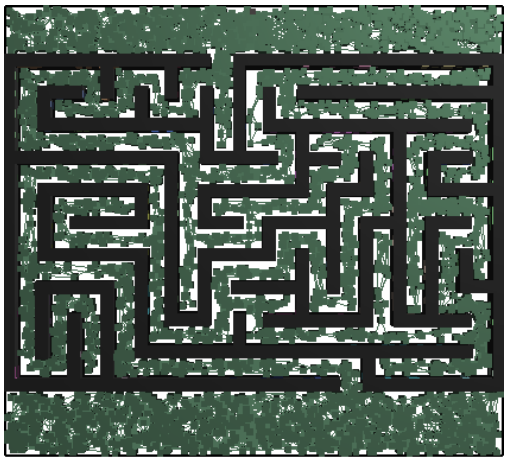
The time complexity of roadmap construction as described in Algorithm 3.1 is given by:

$$T = T_d(E, p) + T_r(R_i)p + T_c(i, j)|Adj(p)| \tag{3.1}$$

(a) 2D Maze before region connect.



(b) Each region sends all candidate vertices within a threshold distance of its boundaries to its neighbors with higher indices.



(c) 2D Maze after region connect.

Figure 3.2: A 2D Maze environment subdivided into 4 regions before, during, and after the region connect step. Differently colored segments represent distinct connected components.

where T_d is the time it takes to decompose the environment E into p regions, T_r is the time it takes to construct a roadmap R_i in region i, and T_c is the time it takes to connect regions i, j, for all adjacent cells i, j in $[0, 1, ..., p-1]$.

The total space complexity is given by $O(N)$, where N is the number of configurations in the final roadmap graph.

In step 1, the time it takes to subdivide the environment is given by the time to compute the prime factors of p, which is given by $O(\sqrt{p})$, then divide our region pf times, where pf is the size of the set of prime factors of p, which is trivially bounded by p.

In step 2, as in the original algorithm [2], we also assume the cost of constructing roadmaps of size N/p in each region is uniform. As the time it takes to construct a roadmap of size N is given by $O(N^2)$ [10], the time it takes to construct p roadmaps of size N/p sequentially is given by $O(N^2/p)$, while in parallel the time is $O((N/p)^2)$.

In steps 3, 4, 5, for a 2D mesh with p elements, the number of (non-diagonal) adjacent cells is bounded by $4p$, as every cell can have no more than 4 neighbors. Similarly, for a 3D mesh, the number of adjacent cells is bounded by $6p$, as no cell can have more than 6 neighbors. In the worst case, each pair of cells may try to connect all N/p vertices in their respective roadmaps with all N/p vertices from their neighbor's roadmap, leading to $O((N/p)^2)$ attempted connections. In parallel, this takes $O((N/p)^2)$ time plus the overhead associated with sending vertex data between STAPL locations, while sequentially this takes $O(N^2/p)$ time.

The overall time, work, and space complexity of our algorithm, then, is given by $O((N/p)^2)$, $O((N^2)/p)$, and $O(N)$ respectively, which does not differ from the original algorithm presented in [2].

3.2 CONTAINERIZATION PLATFORM OVERVIEW

In order to form the basis of this work, we had to create a Docker container capable of supporting our motion planning library, the Parasol Planning Library (PPL), as well as the library we use to achieve scalable parallelism, the Standard Template Adaptive Parallel Library (STAPL). Docker is a containerization platform with a proven capability to facilitate reproducability and portability for software design [80]. Containers built from our Docker image allow us to solve motion planning problems with MPI on a single machine, utilizing the host's computational resources with minimal overhead. We can then employ Docker Swarm mode to generalize the problem to allow for use of MPI with multiple hosts by deploying many containers distributed across multiple hosts.

3.2.1 Docker Image Structure

Our Docker container is built using a nested structure of containers to separate the logical layers of the platform, as shown in Figure 3.3.

Our base layer inherits from the prebuilt Docker image with the distribution for Ubuntu 16.04. This layer additionally installs all the dependencies and sets all environmental variables necessary for PPL to compile with support for STAPL, including specific versions of GCC, Boost, Eigen, MPICH, and Qt, among other libraries. It also creates a default user profile for security purposes, to avoid needing to run MPI programs as the root user. This layer serves as a self-contained environment ready to compile and run PPL with STAPL,

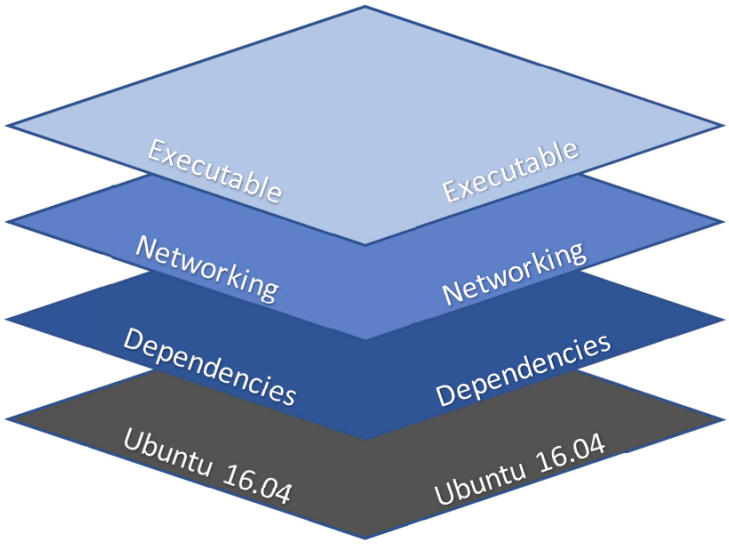


Figure 3.3: The layered structure of our Docker image.

or any other distributed motion planning library that relies on MPI for its communication interface, and dependencies can be freely added or removed as needed.

The next layer inherits from the base layer and encapsulates the networking logic. It installs dependencies required for communication between containers using the ssh protocol as well as various networking tools such as dnsutils and net-tools. Finally, it contains a bootstrapping script to run on creation that will allow the container to find other containers in an overlay network specified in a configuration file. This allows us to construct a cluster of containers leveraging multiple hosts with heterogeneous architectures and operating systems using Docker Swarm.

The last layer inherits from the layer below and is used to copy PPL from a local file system into the image, compile the library, and give ownership and execute privileges to the user defined in the base layer.

A container that is created from the final image is then able to run PPL programs with MPI in concert with other Docker containers orchestrated by Docker Swarm.

By using this final Docker image with an updated version of the build system described in [43] to support Docker version 2, we are able to dynamically spin up and scale any number of containers among the available nodes in the Swarm. We additionally created a script to automatically initialize the Swarm with all the nodes listed in a given hostfile, further automating the procedure of initializing the containerized computing cluster.

Thus, the creation of a heterogeneous computing swarm capable of leveraging any number of computers with different computing architectures in a simple overlay network is simplified to creating a hostfile with a list of all the machines we'd like to be available for the swarm

and running a simple connection script. Scaling the number of nodes in the swarm can be performed in a single line, and takes only seconds to accomplish. This script could also be integrated into any number of controlling applications that can leverage information about the environment to dynamically scale up or down the size of the swarm. As demonstrated in [43], this swarm is also capable of operating in the cloud.

Enabling Robots as Edge Devices for Motion Planning By employing this containerized platform using Docker Swarm mode, we are then able to distribute work among heterogeneous computing devices simultaneously. Docker can run natively on almost all Linux platforms with support for x86-64, ARM and many other CPU architectures [81]. This platform thus allows for the engagement of both servers, personal computers, and onboard robot CPU's to join a heterogeneous computing swarm capable of working in tandem to solve the motion planning problem. By combining this platform with our motion planning library, PPL, with support for massive parallelism using STAPL, we can take full advantage of all the computing resources available in an efficient and straightforward manner.

3.2.2 Singularity

In order to make this system compatible with large-scale shared HPC clusters, we employ Singularity [82] to convert our Docker image into a flat and lightweight Singularity image that can be safely run on any shared HPC cluster that supports Singularity. To use our container with MPI, we utilize Singularity's *hybrid approach* [83]. In this method, the MPI launcher (mpirun, mpiexec) directly launches the Singularity containers across the available nodes, which then internally perform the required computations.

CHAPTER 4: EXPERIMENTAL DESIGN

4.1 EXPERIMENT OVERVIEW

In order to evaluate the efficacy of our platform with respect to parallel sampling-based motion planning methods, we designed a set of experiments to compare the performance and parallel scalability of containerization vs bare-metal performance for our algorithm using the following distributed computing platforms:

1. A single PC with 18 cores.
2. Three PC's with identical architectures and operating systems in a private cluster.
3. A single node in the Illinois Campus Cluster, a large-scale shared HPC cluster.
4. Multiple nodes on the Illinois Campus Cluster

For each of these platforms, we ran our version of the Subdivision Parallel PRM algorithm using both bare-metal implementations and containerization technologies in order to observe the overhead associated with containerization and distributed inter-container communication, if any exists.

Additionally, we further examine the portability and scalability of our containerization method by employing multiple PC's with heterogeneous operating systems and computer architectures working in tandem in a small cluster using Docker Swarm:

5. Two PC's with heterogeneous computing architectures and operating systems in a private cluster.

In each experiment, we perform 32 iterations of constructing a roadmap graph that models the connectivity of the configuration space using the Subdivision Parallel PRM method described in Section 3.1.1. The parameters to each experiment are as follows:

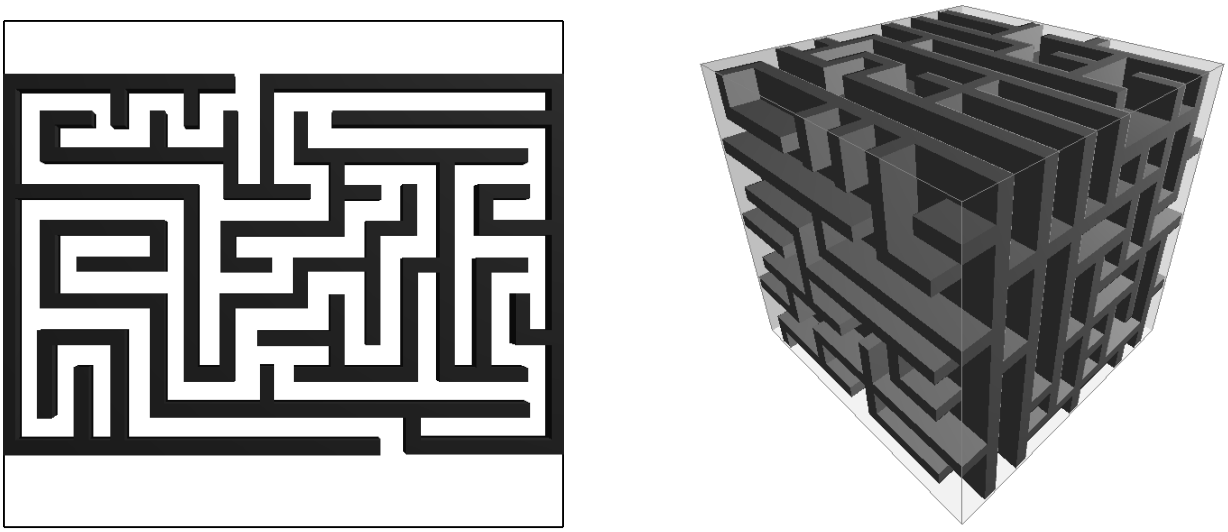
- N, the minimum size of the final roadmap graph to construct
- k, the number of nearest neighbors we use when attempting connections between vertices
- p, the total number of computing cores enlisted and the number of STAPL locations
- E, the environment (and robot) used to generate the roadmap

- *platform*, the platform of the experiment, (i.e. one machine, local cluster, Illinois Campus Cluster, *etc.*)
- *containerized*, whether the experiment employs containers or not

For each experiment, we utilize a sampling strategy that samples the configuration space with uniform randomness.

4.2 ROBOT ENVIRONMENTS

We used two environments to compare the scalability of our method for bare-metal and container implementations. The first environment is a 2D maze environment, as shown in Figure 4.1(a) in which our agent is a small, rigid-body square robot. The second environment is a large-scale 3D maze environment, as shown in Figure 4.1(b), in which our agent is a rigid-body cubic robot.



(a) 2D Maze (b) 3D Maze

Figure 4.1: Environments studied

4.3 HARDWARE PLATFORMS

The different computing architectures employed during experimentation are as follows:

- Three laboratory workstation machines known as Parasol Work 01, Parasol Work 02, and Parasol Work 03. Each of these machines is equipped with 18 Intel(R) Core(TM) i9-10980XE CPU @ 3.00GHz cores, and utilizes the x86_64 architecture.
- One laboratory PC known as Baretta. This machine is equipped with 4 Intel(R) Core(TM) i7-3770 CPU @ 3.40GHz cores, and utilizes the x86_64 architecture.

- The Illinois Campus Cluster, a large-scale shared computing cluster with roughly 750 nodes. Up to date information about the specifications of the cluster can be found on its website [84].

4.4 SOFTWARE PLATFORMS

For all experiments aside from those on the Illinois Campus Cluster and the experiment with heterogeneous computers, we use the Parasol Work Station machines which natively run CentOS Linux release 7.9.2009 (Core), and use MPICH version 3.0.2 for the bare-metal implementation and Ubuntu 16.04 with MPICH version 3.2 for the Docker implementation, with GCC version 4.8.5. For the experiment with heterogeneous computers in a Docker Swarm, we also employ Baretta, which runs Ubuntu 18.04. Every Docker container utilizes Docker version 23.0.3, with native support for Docker Swarm mode.

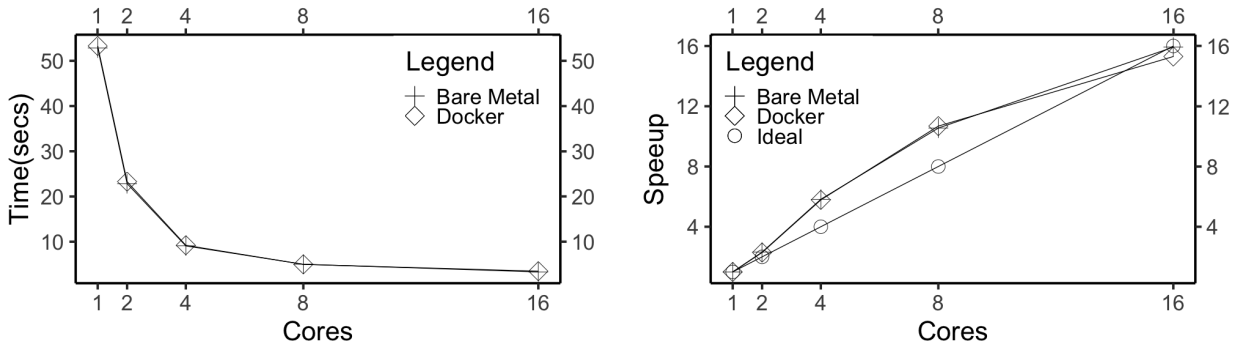
For the experiments on the Illinois Campus Cluster, we are using the same base Docker container from above converted to a Singularity container with Singularity version 3.8.5, with the same MPICH version internally (version 3.2) and GCC version 4.8.5 running Ubuntu 16.04. However, for the Hybrid Mode execution described in Section 3.2.2, and for the bare-metal implementation, we are using MVAPICH2 version 2.3.5. This is necessary to make use of the InfiniBand networking fabric used by the nodes in our cluster experiments. Additionally, for the bare metal experiments on the cluster, we are using GCC version 7.2.0.

CHAPTER 5: EXPERIMENTAL EVALUATION

For Experiments 1 and 2, we employ the Parasol Work Station machines described in Section 4.3.

5.1 EXPERIMENT 1 - DOCKER FOR PLANNING ON ONE MACHINE

To establish a baseline of scalability on a single machine, we chose to evaluate the performance of our algorithm running natively on Parasol Work 01. We use the 2D Maze environment and evaluate runtime performance and parallel scalability for a single Docker container vs bare-metal implementations for $p = 1, 2, 4, 8, 16$ cores. We attempt to generate 4000 valid configurations, which we previously determined was sufficient to construct a roadmap that models the connectivity of the configuration space, using $k = 8$ nearest neighbors for each connect attempt. A visualization of one of these such trials can be found in Figure 3.2.



(a) Time

(b) Speed up

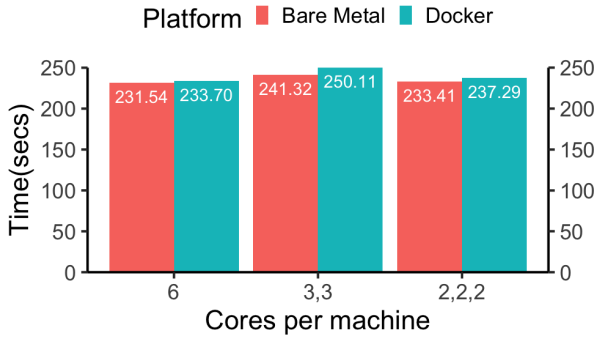
Figure 5.1: Comparison of bare-metal vs Docker runtime and scalability on Parasol Work 01

As seen in Figure 5.1, we achieve nearly identical results for both the bare-metal and containerized methods. Both methods achieve nearly linear speed up with Docker losing scalability slightly faster than bare-metal. This demonstrates the ability of our Docker container to nearly fully utilize a given host machine's computational resources.

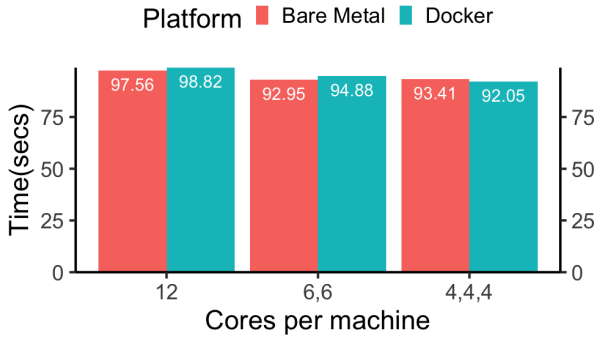
5.2 EXPERIMENT 2 - DOCKER SWARM FOR PLANNING WITH HOMOGENEOUS CLUSTER

To test scalability across multiple machines, and visualize any overhead associated with distributing work across multiple containers on different hosts with MPI, we designed a set of three experiments that utilize the three Parasol Work Station machines. In each experiment,

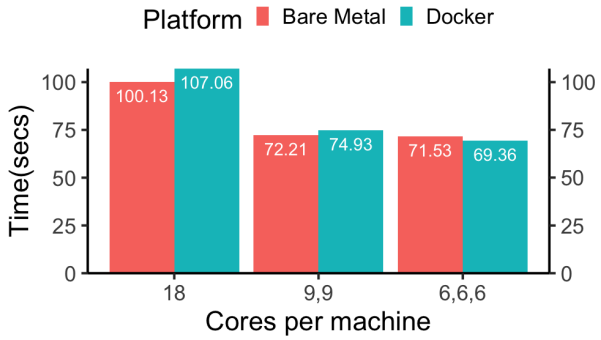
we construct a roadmap of size $N = 32000$ in the 3D Maze environment, using $k = 4$ nearest neighbors, using both bare-metal and Docker Swarm mode implementations. We fixed the total number of cores available to $p = 6, 12, 18$ for each of the three experiments respectively. We then distributed that workload, and number of cores p evenly across 1, 2, and 3 machines to view any overhead associated with distributing the workload across multiple machines. For example, for the experiment with a fixed $p = 18$, we evaluated the runtime and scalability of the algorithm running on a single machine using all 18 cores, 2 machines using 9 cores each, and 3 machines using 6 cores each. The same logic follows for $p = 12$ and $p = 6$.



(a) Time Comparison (Bare Metal vs Docker Swarm) for $P = 6$



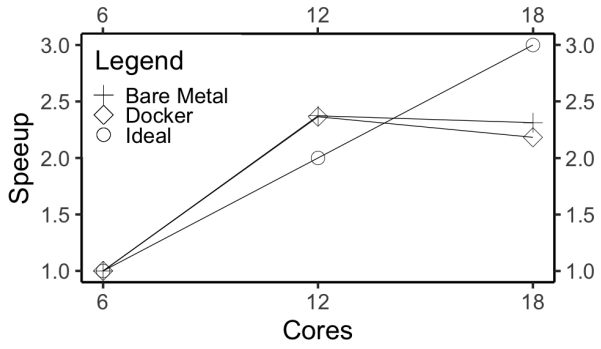
(b) Time Comparison (Bare Metal vs Docker Swarm) for $P = 12$



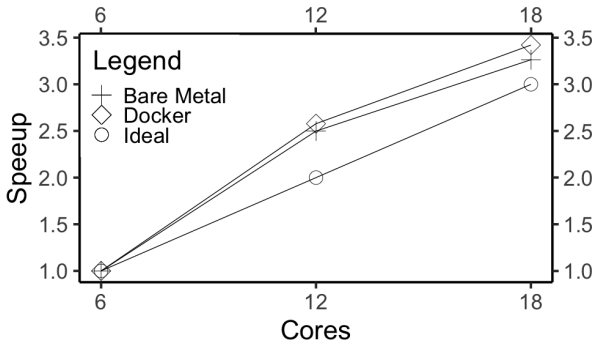
(c) Time Comparison (Bare Metal vs Docker Swarm) for $P = 18$

Figure 5.2: Comparison of 1, 2, and 3 Parasol Work Machines enlisted to perform Subdivision Parallel PRM with 6, 12, and 18 total cores.

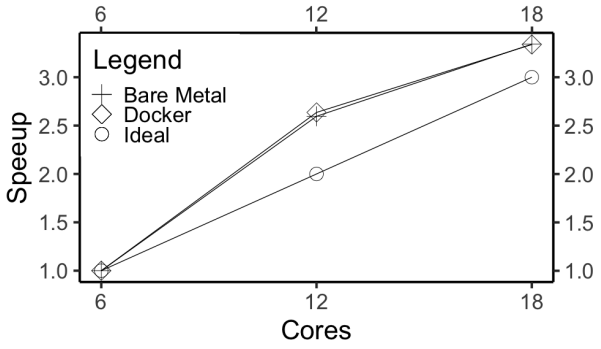
As seen in Figure 5.2, distributing work between more machines does not seem to introduce significant additional overhead, and can in fact improve results by ensuring no one machine is fully subscribed, as shown in Figure 5.2(c) when all 18 cores were being used on one machine. It is also worth noting that even when a single machine is fully subscribed, as



(a) Speed up on one Parasol Work Station (starts at $p = 6$)



(b) Speed up on two Parasol Work Stations (starts at $p = 6$)



(c) Speed up on three Parasol Work Stations (starts at $p = 6$)

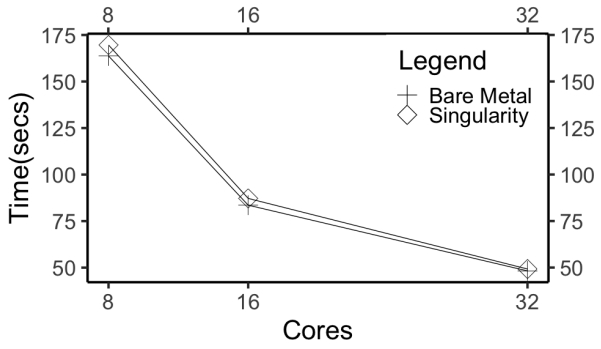
Figure 5.3: Comparison of speedups as work is distributed among 1, 2, and 3 machines respectively.

in Figure 5.3(a), the impact on scalability shows a similar trend for both the Docker and bare-metal implementations.

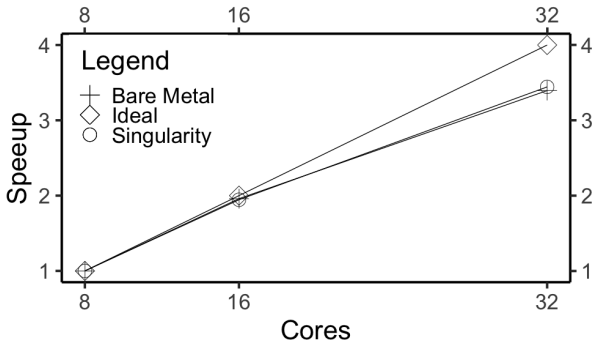
Illinois Campus Cluster To test scalability with Singularity versus bare-metal on the Illinois Campus Cluster, we performed two experiments:

5.3 EXPERIMENT 3 - SINGULARITY FOR PLANNING ON A SINGLE HPC NODE

Similar to Experiment 1, when analyzing the performance of our method on the Illinois Campus Cluster (ICC), we first establish a baseline of our method on the new computing architecture by utilizing only a single node. We enlisted one node to construct a roadmap of size $N = 32000$, using $k = 4$ nearest neighbors, in the 3D Maze environment using $p = 8, 16, 32$ cores.



(a) Time (Single Cluster Node)

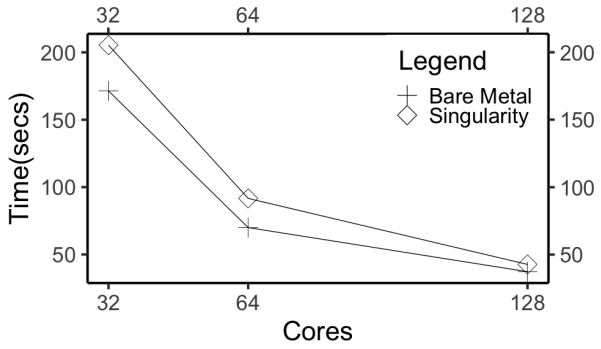


(b) Speed up (Single Cluster Node, starts at $p = 8$)

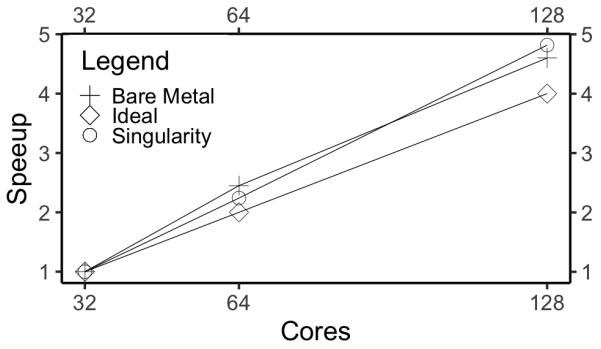
Figure 5.4: Comparison of bare-metal vs Singularity on a single node in the ICC.

As can be seen in figures 5.4(a) and 5.4(b), Singularity is capable of taking advantage of the host machine's computing resources nearly perfectly when only a single node is employed, and similar scalability is attained for bare-metal and containerized performance.

5.4 EXPERIMENT 4 - SINGULARITY FOR PLANNING ON SHARED HPC'S



(a) Time (Multiple Cluster Nodes, starts at $p = 32$)



(b) Speed up (Multiple Cluster Nodes, starts at $p = 32$)

Figure 5.5: Comparison of bare-metal vs Singularity across multiple nodes in the ICC.

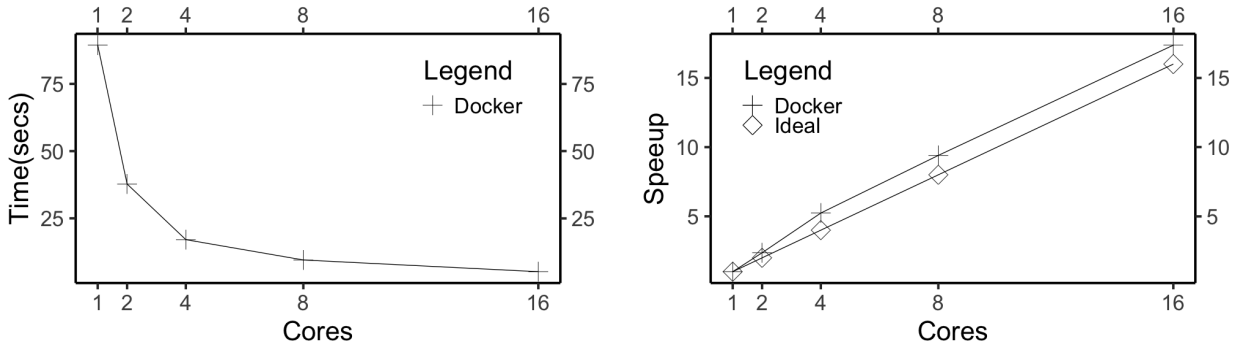
For the second cluster experiment, we scaled the problem up to enlist 11 nodes, with 196 total cores combined, to construct a roadmap of size $N = 64000$, using $k = 4$ nearest neighbors, in the 3D Maze environment using $p = 32, 64, 128$ cores respectively.

As evident in figure 5.5(a) and 5.5(b), the use of Singularity did appear to introduce some overhead when the problem was scaled across many shared nodes, however, both single and multi-node platforms still appear to achieve near linear scalability. Additionally, when the Singularity experiment was run, the Illinois Campus Cluster was busier than when the bare-

metal experiment was run, and this likely caused additional contention along inter-node communication lines. However, this kind of erratic contention occurs with shared clusters, and does not appear to affect the general linearity of the scaling.

5.5 EXPERIMENT 5 - DOCKER SWARM FOR PLANNING WITH HETEROGENEOUS CLUSTER

To establish a baseline for the runtime and scalability of the Docker Swarm method on machines with heterogeneous operating systems and computing power, we employed Parasol Work 01, running CentOS, as well as Baretta, running Ubuntu. We construct a roadmap of size $n = 4000$ nodes with $k = 8$ nearest neighbors, in the 2D Maze environment across $p = 1, 2, 4, 8, 16$ cores. We ensure both machines are engaged in each iteration by enforcing a policy for MPI to assign slots first to Baretta, the smaller machine, and then any remaining work to Parasol Work 01. We make sure to avoid oversubscribing either machine by allowing Baretta to employ at most 2 of its 4 cores, and Parasol Work 01 to employ at most 14 of its 18 cores.



(a) Time

(b) Speed up

Figure 5.6: Runtime and speed up for Docker Swarm implemented on platform consisting of two machines with heterogeneous architectures and operating systems.

As seen in Figure 5.6, scalability for heterogeneous private clusters follows the same trends we observe for homogeneous clusters. This also successfully demonstrates the ability for container swarms to continue leveraging even significantly less powerful machines to aid in computation, which could theoretically include even onboard computers on robots, allowing them to serve as mobile edge devices.

CHAPTER 6: CONCLUSION

In this work we analyze the necessary components required to enlist varied distributed computing platforms such as large-scale computing clusters, cloud resources, and edge devices, to solve the motion planning problem. One of these such components is a platform that can solve distributed motion planning algorithms in a portable fashion.

To this end, we present a method capable of enlisting multiple private or shared nodes to host containers capable of working in tandem to solve large-scale, motion planning problems using parallel algorithms. We test our distributed container platform on a variety of infrastructures, including the Illinois Campus Cluster, and local clusters comprised of both homogeneous and heterogeneous in-lab work stations. We show that the overhead associated with containerization does not inhibit the method from achieving the same near linear scalability of its corresponding bare-metal implementation on any of these platforms.

Future work will extend this approach to a wider array of motion planning algorithms, including those that target the multi-agent path-finding problem. It will also test this method using actual onboard computers on robots as mobile edge devices as part of the swarm, as well as on cloud-based infrastructures, as suggested in [43]. Future work will also explore how all these various platforms can be integrated together into a single ecosystem to simultaneously work together to solve the motion planning problem.

This method has far reaching implications, as it allows for the enlisting of a large variety of distributed computing platforms to solve the motion planning problem. This work then serves as a key step in moving the field of robotics toward dynamically scalable open-source motion planning on the mobile edge and in the cloud.

REFERENCES

[1] J. Bialkowski, S. Karaman, and E. Frazzoli, "Massively parallelizing the rrt and the rrt*," 09 2011, pp. 3513–3518.

[2] S. A. Jacobs, K. Manavi, J. Burgos, J. Denny, S. Thomas, and N. M. Amato, "A scalable method for parallelizing sampling-based motion planning algorithms," in *2012 IEEE International Conference on Robotics and Automation*, 2012, pp. 2529–2536.

[3] H. Lee, J. Motes, M. Morales, and N. M. Amato, "Parallel hierarchical composition conflict-based search for optimal multi-agent pathfinding," *IEEE Robotics and Automation Letters*, vol. 6, no. 4, pp. 7001–7008, 2021.

[4] I. A. Sucan, M. Moll, and L. E. Kavraki, "The open motion planning library," *IEEE Robotics & Automation Magazine*, vol. 19, no. 4, pp. 72–82, 2012.

[5] H. Fan, F. Zhu, C. Liu, L. Zhang, L. Zhuang, D. Li, W. Zhu, J. Hu, H. Li, and Q. Kong, "Baidu apollo EM motion planner," *CoRR*, vol. abs/1807.08048, 2018. [Online]. Available: http://arxiv.org/abs/1807.08048

[6] J.-C. Latombe, "Motion planning: A journey of robots, molecules, digital actors, and other artifacts," *The International Journal of Robotics Research*, vol. 18, no. 11, pp. 1119–1128, 1999. [Online]. Available: https://doi.org/10.1177/02783649922067753

[7] O. Burchan, B. Guang, S. Nancy, and N. Amato, "Ligand binding with obprm and haptic user input: Enhancing automatic motion planning with virtual touch," 11 2000.

[8] D. Uwacu, A. Ren, S. Thomas, and N. Amato, "Using guided motion planning to study binding site accessibility," 09 2020, pp. 1–10.

[9] J. Pan and D. Manocha, "Efficient configuration space construction and optimization for motion planning," *Engineering*, vol. 1, 03 2015.

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

[11] S. M. LaValle, "Rapidly-exploring random trees : a new tool for path planning," *The annual research report, Computer Science Department, Iowa State University*, 1998.

[12] J. Kuffner and S. LaValle, "Rrt-connect: An efficient approach to single-query path planning," in *Proceedings 2000 ICRA. Millennium Conference. IEEE International Conference on Robotics and Automation. Symposia Proceedings (Cat. No.00CH37065)*, vol. 2, 2000, pp. 995–1001 vol.2.

[13] S. Karaman and E. Frazzoli, "Sampling-based algorithms for optimal motion planning," *CoRR*, vol. abs/1105.1186, 2011. [Online]. Available: http://arxiv.org/abs/1105.1186

[14] P. Fiorini and Z. Shiller, "Motion planning in dynamic environments using velocity obstacles," *The international journal of robotics research*, vol. 17, no. 7, pp. 760–772, 1998.

[15] J.-P. Laumond, S. Sekhavat, and F. Lamiraux, "Guidelines in nonholonomic motion planning for mobile robots," *Robot motion planning and control*, pp. 1–53, 2005.

[16] H. Ma, "Intelligent planning for large-scale multi-agent systems," *AI Magazine*, vol. 43, no. 4, pp. 376–382, 2022. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/aaai.12069

[17] R. Stern, N. R. Sturtevant, A. Felner, S. Koenig, H. Ma, T. T. Walker, J. Li, D. Atzmon, L. Cohen, T. K. S. Kumar, E. Boyarski, and R. Barták, "Multi-agent pathfinding: Definitions, variants, and benchmarks," *CoRR*, vol. abs/1906.08291, 2019. [Online]. Available: http://arxiv.org/abs/1906.08291

[18] R. Cui, B. Gao, and J. Guo, "Pareto-optimal coordination of multiple robots with safety guarantees," *Auton. Robots*, vol. 32, no. 3, p. 189–205, apr 2012. [Online]. Available: https://doi.org/10.1007/s10514-011-9265-9

[19] J. Motes, T. Chen, T. Bretl, M. Morales, and N. M. Amato, "Hypergraph-based multi-robot task and motion planning," *arXiv preprint arXiv:2210.04333*, 2022.

[20] G. Sharon, R. Stern, A. Felner, and N. R. Sturtevant, "Conflict-based search for optimal multi-agent pathfinding," *Artificial Intelligence*, vol. 219, pp. 40–66, 2015. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0004370214001386

[21] G. Blake, R. G. Dreslinski, T. Mudge, and K. Flautner, "Evolution of thread-level parallelism in desktop applications," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ser. ISCA '10. New York, NY, USA: Association for Computing Machinery, 2010. [Online]. Available: https://doi.org/10.1145/1815961.1816000 p. 302–313.

[22] A. R. Brodtkorb, T. R. Hagen, and M. L. Sætra, "Graphics processing unit (gpu) programming strategies and trends in gpu computing," *Journal of Parallel and Distributed Computing*, vol. 73, no. 1, pp. 4–13, 2013.

[23] L. Dagum and R. Menon, "Openmp: an industry standard api for shared-memory programming," *IEEE computational science and engineering*, vol. 5, no. 1, pp. 46–55, 1998.

[24] L. Clarke, I. Glendinning, and R. Hempel, "The mpi message passing interface standard," in *Programming Environments for Massively Parallel Distributed Systems: Working Conference of the IFIP WG 10.3, April 25–29, 1994*. Springer, 1994, pp. 213–218.

[25] J. H. Reif, "Depth-first search is inherently sequential," *Information Processing Letters*, vol. 20, no. 5, pp. 229–234, 1985.

[26] Y. Lee, R. Avizienis, A. Bishara, R. Xia, D. Lockhart, C. Batten, and K. Asanović, "Exploring the tradeoffs between programmability and efficiency in data-parallel accelerators," *SIGARCH Comput. Archit. News*, vol. 39, no. 3, p. 129–140, jun 2011. [Online]. Available: https://doi.org/10.1145/2024723.2000080

[27] J. Legaux, F. Loulergue, and S. Jubertie, "Development effort and performance trade-off in high-level parallel programming," in *2014 International Conference on High Performance Computing Simulation (HPCS)*, 2014, pp. 162–169.

[28] S. Li, M. A. Maddah-Ali, Q. Yu, and A. S. Avestimehr, "A fundamental tradeoff between computation and communication in distributed computing," *IEEE Transactions on Information Theory*, vol. 64, no. 1, pp. 109–128, 2018.

[29] D. Callahan, K. Kennedy, and J. Subhlok, "Analysis of event synchronization in a parallel programming tool," in *Proceedings of the second ACM SIGPLAN symposium on Principles & practice of parallel programming*, 1990, pp. 21–30.

[30] B. Acun, A. Gupta, N. Jain, A. Langer, H. Menon, E. Mikida, X. Ni, M. Robson, Y. Sun, E. Totoni et al., "Parallel programming with migratable objects: Charm++ in practice," in *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2014, pp. 647–658.

[31] B. Nichols, D. Buttlar, J. Farrell, and J. Farrell, *Pthreads programming: A POSIX standard for better multiprocessing*. " O'Reilly Media, Inc.", 1996.

[32] J. Sanders and E. Kandrot, *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional, 2010.

[33] A. Munshi, "The opencl specification," in *2009 IEEE Hot Chips 21 Symposium (HCS)*. IEEE, 2009, pp. 1–314.

[34] L. V. Kale and S. Krishnan, "Charm++ a portable concurrent object oriented system based on c++," in *Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*, 1993, pp. 91–108.

[35] A. Buss, A. Fidel, Harshvardhan, T. Smith, G. Tanase, N. Thomas, X. Xu, M. Bianco, N. M. Amato, and L. Rauchwerger, "The stapl pview," in *Languages and Compilers for Parallel Computing*, K. Cooper, J. Mellor-Crummey, and V. Sarkar, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 261–275.

[36] G. Tanase, A. Buss, A. Fidel, Harshvardhan, I. Papadopoulos, O. Pearce, T. Smith, N. Thomas, X. Xu, N. Mourad, J. Vu, M. Bianco, N. M. Amato, and L. Rauchwerger, "The stapl parallel container framework," in *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP 1́1. New York, NY, USA: Association for Computing Machinery, 2011. [Online]. Available: https://doi.org/10.1145/1941553.1941586 p. 235–246.

[37] M. Zandifar, N. Thomas, N. M. Amato, and L. Rauchwerger, "The stapl skeleton framework," in *Languages and Compilers for Parallel Computing*, J. Brodman and P. Tu, Eds. Cham: Springer International Publishing, 2015, pp. 176–190.

[38] I. Papadopoulos, N. Thomas, A. Fidel, N. M. Amato, and L. Rauchwerger, "Stapl-rts: An application driven runtime system," in *Proceedings of the 29th ACM on International Conference on Supercomputing*, ser. ICS '15. New York, NY, USA: Association for Computing Machinery, 2015. [Online]. Available: https://doi.org/10.1145/2751205.2751233 p. 425–434.

[39] N. Thomas, S. Saunders, T. Smith, G. Tanase, and L. Rauchwerger, "Armi: A high level communication library for stapl," *Parallel Processing Letters*, vol. 16, no. 2, pp. 261–280, 2006.

[40] P. Mell and T. Grance, "Draft nist working definition of cloud computing-v15," *21. Aug 2009*, vol. 2, pp. 123–135, 2009.

[41] M. N. Sadiku, S. M. Musa, and O. D. Momoh, "Cloud computing: Opportunities and challenges," *IEEE Potentials*, vol. 33, no. 1, pp. 34–36, 2014.

[42] L. Benedicic, F. A. Cruz, A. Madonna, and K. Mariotti, "Sarus: Highly scalable docker containers for hpc systems," in *High Performance Computing*, M. Weiland, G. Juckeland, S. Alam, and H. Jagode, Eds. Cham: Springer International Publishing, 2019, pp. 46–60.

[43] N. Nguyen and D. Bein, "Distributed mpi cluster with docker swarm mode," in *2017 IEEE 7th Annual Computing and Communication Workshop and Conference (CCWC)*, 2017, pp. 1–7.

[44] K. Cao, Y. Liu, G. Meng, and Q. Sun, "An overview on edge computing research," *IEEE Access*, vol. 8, pp. 85 714–85 728, 2020.

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

[46] R. Nissim and R. I. Brafman, "Multi-agent a* for parallel and distributed systems." in *AAMAS*, 2012, pp. 1265–1266.

[47] S. Mukherjee, S. Aine, and M. Likhachev, "Mplp: Massively parallelized lazy planning," *IEEE Robotics and Automation Letters*, vol. 7, no. 3, pp. 6067–6074, 2022.

[48] C. Park, F. Rabe, S. Sharma, C. Scheurer, U. E. Zimmermann, and D. Manocha, "Parallel cartesian planning in dynamic environments using constrained trajectory planning," in *2015 IEEE-RAS 15th International Conference on Humanoid Robots (Humanoids)*, 2015, pp. 983–990.

[49] J. Pan, C. Lauterbach, and D. Manocha, "G-planner: Real-time motion planning and global navigation using gpus," in *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence*, ser. AAAI'10. AAAI Press, 2010, p. 1245–1251.

[50] N. Atay and B. Bayazit, "A motion planning processor on reconfigurable hardware," in *Proceedings 2006 IEEE International Conference on Robotics and Automation, 2006. ICRA 2006.*, 2006, pp. 125–132.

[51] R. Li, X. Huang, S. Tian, R. Hu, D. He, and Q. Gu, "Fpga-based design and implementation of real-time robot motion planning," in *2019 9th International Conference on Information Science and Technology (ICIST)*, 2019, pp. 216–221.

[52] A. Kosuge and T. Oshima, "A 1200×1200 8-edges/vertex fpga-based motion-planning accelerator for dual-arm-robot manipulation systems," in *2020 IEEE Symposium on VLSI Circuits*, 2020, pp. 1–2.

[53] J. Ichnowski, J. Prins, and R. Alterovitz, *Cloud-based Motion Plan Computation for Power-Constrained Robots*. Cham: Springer International Publishing, 2020, pp. 96–111. [Online]. Available: https://doi.org/10.1007/978-3-030-43089-4_7

[54] A. Vick, V. Vonásek, R. Pěnička, and J. Krüger, "Robot control as a service — towards cloud-based motion planning and control for industrial robots," in *2015 10th International Workshop on Robot Motion and Control (RoMoCo)*, 2015, pp. 33–39.

[55] S. Rosa, L. O. Russo, and B. Bona, "Towards a ros-based autonomous cloud robotics platform for data center monitoring," in *Proceedings of the 2014 IEEE Emerging Technology and Factory Automation (ETFA)*, 2014, pp. 1–8.

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

[57] J. Ichnowski, W. Lee, V. Murta, S. Paradis, R. Alterovitz, J. E. Gonzalez, I. Stoica, and K. Goldberg, "Fog robotics algorithms for distributed motion planning using lambda serverless computing," in *2020 IEEE International Conference on Robotics and Automation (ICRA)*, May 2020, pp. 4232–4238.

[58] R. Anand, J. Ichnowski, C. Wu, J. M. Hellerstein, J. E. Gonzalez, and K. Goldberg, "Serverless multi-query motion planning for fog robotics," in *2021 IEEE International Conference on Robotics and Automation (ICRA)*, 2021, pp. 7457–7463.

[59] B. Addad, S. Amari, and J.-J. Lesage, "Analytic calculus of response time in networked automation systems," *Automation Science and Engineering, IEEE Transactions on*, vol. 7, pp. 858 – 869, 11 2010.

[60] B. Addad, S. Amari, and J.-J. Lesage, "Client-server networked automation systems reactivity: Deterministic and probabilistic analysis," *IEEE Transactions on Automation Science and Engineering*, vol. 8, no. 3, pp. 540–548, 2011.

[61] A. Ambike, W. jong Kim, and K. Ji, "Real-time operating environment for networked control systems," in *Proceedings of the 2005, American Control Conference, 2005.*, 2005, pp. 2353–2358 vol. 4.

[62] B. Liu, Y. Chen, E. Blasch, K. Pham, D. Shen, and G. Chen, "A holistic cloud-enabled robotics system for real-time video tracking application," in *Future Information Technology*, J. J. J. H. Park, I. Stojmenovic, M. Choi, and F. Xhafa, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 455–468.

[63] Y. Nan, W. Li, W. Bao, F. C. Delicato, P. F. Pires, and A. Y. Zomaya, "A dynamic tradeoff data processing framework for delay-sensitive applications in cloud of things systems," *Journal of Parallel and Distributed Computing*, vol. 112, pp. 53–66, 2018.

[64] A. Yousefpour, G. Ishigaki, and J. P. Jue, "Fog computing: Towards minimizing delay in the internet of things," in *2017 IEEE international conference on edge computing (EDGE)*. IEEE, 2017, pp. 17–24.

[65] J. Ichnowski, J. Prins, and R. Alterovitz, "The economic case for cloud-based computation for robot motion planning," in *Robotics Research*, N. M. Amato, G. Hager, S. Thomas, and M. Torres-Torriti, Eds. Cham: Springer International Publishing, 2020, pp. 59–65.

[66] K. Kaur, T. Dhand, N. Kumar, and S. Zeadally, "Container-as-a-service at the edge: Trade-off between energy efficiency and service availability at fog nano data centers," *IEEE wireless communications*, vol. 24, no. 3, pp. 48–56, 2017.

[67] R. Hadidi, J. Cao, M. Woodward, M. S. Ryoo, and H. Kim, "Distributed perception by collaborative robots," *IEEE Robotics and Automation Letters*, vol. 3, no. 4, pp. 3709–3716, 2018.

[68] A. Chowdhury, "Iot and robotics: a synergy," 01 2017.

[69] G. Von Krogh and S. Spaeth, "The open source software phenomenon: Characteristics that promote research," *The Journal of Strategic Information Systems*, vol. 16, no. 3, pp. 236–253, 2007.

[70] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, A. Y. Ng et al., "Ros: an open-source robot operating system," in *ICRA workshop on open source software*, vol. 3, no. 3.2. Kobe, Japan, 2009, p. 5.

[71] A. Koubaa, M. Alajlan, and B. Qureshi, *ROSLink: Bridging ROS with the Internet-of-Things for Cloud Robotics*. Cham: Springer International Publishing, 2017, pp. 265–283. [Online]. Available: https://doi.org/10.1007/978-3-319-54927-9_8

[72] R. Doriya, P. Chakraborty, and G. C. Nandi, "'robot-cloud': A framework to assist heterogeneous low cost robots," in *2012 International Conference on Communication, Information Computing Technology (ICCICT)*, 2012, pp. 1–5.

[73] K. E. Chen, Y. Liang, N. Jha, J. Ichnowski, M. Danielczuk, J. Gonzalez, J. Kubiatowicz, and K. Goldberg, "Fogros: An adaptive framework for automating fog robotics deployment," in *2021 IEEE 17th International Conference on Automation Science and Engineering (CASE)*, 2021, pp. 2035–2042.

[74] S. Chitta, I. Sucan, and S. Cousins, "Moveit![ros topics]," *IEEE Robotics & Automation Magazine*, vol. 19, no. 1, pp. 18–19, 2012.

[75] R. Diankov and J. Kuffner, "Openrave: A planning architecture for autonomous robotics," *Robotics Institute, Pittsburgh, PA, Tech. Rep. CMU-RI-TR-08-34*, vol. 79, 2008.

[76] G. Echeverria, N. Lassabe, A. Degroote, and S. Lemaignan, "Modular open robots simulation engine: Morse," *2011 IEEE International Conference on Robotics and Automation*, pp. 46–51, 2011.

[77] Y. Zhang, H. Sun, J. Zhou, J. Pan, J. Hu, and J. Miao, "Optimal vehicle path planning using quadratic optimization for baidu apollo open platform," in *2020 IEEE Intelligent Vehicles Symposium (IV)*. IEEE, 2020, pp. 978–984.

[78] X. Wang, A.-K. Rettinger, M. T. B. Waez, and M. Althoff, "Coupling apollo with the commonroad motion planning framework," in *Proceedings of the FISITA Web Congress*, vol. 2020, 2020, p. 24.

[79] M. Jamal and A. Panov, "Adaptive maneuver planning for autonomous vehicles using behavior tree on apollo platform," in *Artificial Intelligence XXXVIII: 41st SGAI International Conference on Artificial Intelligence, AI 2021, Cambridge, UK, December 14–16, 2021, Proceedings 41*. Springer, 2021, pp. 327–340.

[80] C. Boettiger, "An introduction to docker for reproducible research," *SIGOPS Oper. Syst. Rev.*, vol. 49, no. 1, p. 71–79, jan 2015. [Online]. Available: https://doi.org/10.1145/2723872.2723882

[81] D. Merkel et al., "Docker: lightweight linux containers for consistent development and deployment," *Linux j*, vol. 239, no. 2, p. 2, 2014.

[82] G. M. Kurtzer, V. Sochat, and M. W. Bauer, "Singularity: Scientific containers for mobility of compute," *PLOS ONE*, vol. 12, no. 5, pp. 1–20, 05 2017. [Online]. Available: https://doi.org/10.1371/journal.pone.0177459

[83] Y.-T. S. Chang, S. Heistand, R. Hood, and H. Jin, "Feasibility of running singularity containers with hybrid mpi on nasa high-end computing resources," in *2021 3rd International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC)*, 2021, pp. 17–28.

[84] "Illinois campus cluster program," 2021. [Online]. Available: https://campuscluster.illinois.edu/about/system-info/hardware/