A SCALABLE FRAMEWORK FOR PARALLELIZING SAMPLING-BASED MOTION PLANNING ALGORITHMS

A Dissertation

by

SAMSON ADE JACOBS

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

May 2014

Major Subject: Computer Science

ABSTRACT

Motion planning is defined as the problem of finding a valid path taking a robot (or any movable object) from a given start configuration to a goal configuration in an environment. While motion planning has its roots in robotics, it now finds application in many other areas of scientic computing such as protein folding, drug design, virtual prototyping, computer-aided design (CAD), and computer animation. These new areas test the limits of the best sequential planners available, motivating the need for methods that can exploit parallel processing.

This dissertation focuses on the design and implementation of a generic and scalable framework for parallelizing motion planning algorithms. In particular, we focus on sampling-based motion planning algorithms which are considered to be the state-of-the-art. Our work covers the two broad classes of sampling-based motion planning algorithms — the graph-based and the tree-based methods. Central to our approach is the subdivision of the planning space into regions. These regions represent sub-problems that can be processed in parallel. Solutions to the sub-problems are later combined to form a solution to the entire problem. By subdividing the planning space and restricting the locality of connection attempts to adjacent regions, we reduce the work and inter-processor communication associated with nearest neighbor calculation, a critical bottleneck for scalability in existing parallel motion planning methods. We also describe how load balancing strategies can be applied in complex environments. We present experimental results that scale to thousands of processors on different massively parallel machines for a range of motion planning problems.

DEDICATION

To my beloved wife, Wuraola; love is our greatest asset, the future is nothing to fear.

To our two adorable daughters, Molayo and Moyo; with all thy getting, get wisdom.

ACKNOWLEDGEMENTS

I am indebted to a number of people for their support in the course of my PhD program.

First and foremost, my advisor and mentor, Prof. Nancy M. Amato, for her continual support and sometime demanding request which has made me both a better researcher and a leader. Her passion for excellence is one thing I will cherish for years to come.

I would like to thank my committee members, Prof. Lawrence Rauchwerger, Prof. Valerie Taylor, and Prof. Jim Morel, for their useful suggestions and guidance.

I would like to thank faculty, staff, postdocs, alumni, and current students of Parasol Lab who in one way or other have made contributions to the work presented in this dissertation. Worth mentioning here are the following individuals: Prof. Jennifer Welch, Kay Jones, David Ramirez, Dr. Shawna Thomas, Dr. Roger Pearce, Dr. Gabriel Tanase, Dr. Timmie Smith, Dr. Nathan Thomas, Harshvardan, Adam Fidel, Antal Buss, Ioannis Papadopoulos, Shishir Sharma, Bryan Boyd, Jory Denny, Kasra Manavi, Troy McMahon, Chinwe Ekenna, Cindy Yeh, Andy Giese, and Daniel Tomkins.

I acknowledge the work of undergraduate students whom I have had the privilege to mentor, in particular, Juan Burgos, Cesar Rodriguez both from Texas A&M University, Dezshaun Meeks from Praire View A&M University, and Nicholas Stradford from University of North Texas. I am grateful to them for their contributions to this work.

Lastly, I would like to thank my family and friends for their love, patience, and support.

TABLE OF CONTENTS

LIST OF FIGURES

FIGURE Page

1. INTRODUCTION

This dissertation presents a scalable framework for parallelizing sampling-based motion planning algorithms. Motion planning is defined as the problem of finding a valid path taking a robot (or any movable object) from a given start configuration to a goal configuration in an environment. While motion planning has its roots in robotics, it now finds applications in other areas of scientific computing including protein folding [1, 2, 3], minimally-invasive surgical planning [4], and drug design [5, 6, 7, 8], and computer-aided design [9, 10, 11, 12]. These new application areas are known to test the limit and capability of existing sequential motion planners [13].

Due to the infeasibility of exact motion planning [14, 15], sampling-based methods [15] are now the state-of-the-art for solving motion planning problems. Sampling-based approaches are efficient and can be applied to problems with many degrees of freedom (e.g., robotic manipulators with many links or proteins with many amino acids). While not guaranteed to find a solution, sampling-based methods are known to be probabilistically complete, meaning that the probability of finding a solution, given one exists, increases with the number of samples generated [16]. Sampling-based motion planning algorithms have been highly successful at solving previously unsolved problems [4, 15], and much research has focused on developing more sophisticated variants of them [4, 15].

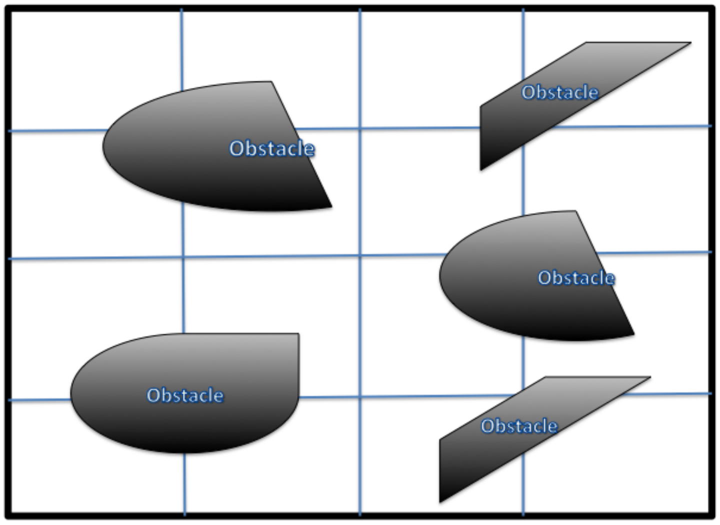
Sequential sampling-based motion planning algorithms still require substantial resources in time and hardware to solve computationally intensive applications. For example, modeling the motion of a small protein using sequential sampling-based motion planning techniques can take days on a typical desktop machine [17]. This time increases to several weeks if more accurate energy calculations are used or if larger proteins are studied. Hence, it is practically infeasible to study larger proteins or to significantly increase the detail and accuracy at which their motions are modeled. To address this problem, researchers have turned to parallel processing as an alternative option to explore. For many application areas, parallel processing offers the advantage of not only reducing computation time, but also improving the solution quality and enabling larger problems to be solved than were feasible before. Although there has been some research in parallel motion planning [18, 19, 20, 21, 17, 22, 23, 24, 25, 26, 13], no scalable solution has been proposed.

This research proposes a new framework for parallelizing sampling-based motion planning algorithms. Central to our proposed framework [27, 28, 29] is the novel subdivision of the planning space into regions and an abstraction that represents the spatial relationship between the regions called a *region graph* $R(V, E)$. The vertices, V, of the *region graph* represent the regions and the edges, E, represent the adjacencies between regions. The regions represent subproblems that can be processed independently (and in parallel). The task or subproblem associated with each region is to build a roadmap (graph or tree) that encodes representative paths approximating the topology of the planning space of the associated region. The regional roadmaps are later combined to obtain a roadmap for the entire space. This merging of regional roadmaps is facilitated using the *region graph.* In particular, the *region graph* is the enabling infrastructure facilitating the process of connecting the regional roadmaps as its edges identify adjacent regions between which connections are attempted.

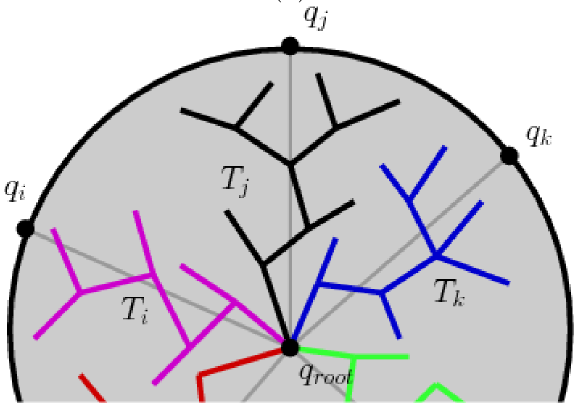
By subdividing the planning space and restricting the locality of connection attempts to adjacent regions, we reduce the work and inter-processor communication associated with nearest neighbor calculation, a critical bottleneck in the scalability of existing parallel motion planning methods [30, 31, 23, 24]. While our framework employs the standard sequential planners (e.g., the probabilistic roadmap method (PRM)[16] or rapidly-exploring random tree (RRT)[32]) as underlying motion planning algorithms, the resulting roadmap may be structurally different than would result if one of them were applied to the problem as a whole. Hence, we carried out an experimental evaluation of our algorithms to study both the structural difference and its impact on the solution of the motion planning problems.

In addition, we address the problem of load balancing [33] in complex planning spaces. For most complex planning spaces, as the granularity of the subdivision increases, the heterogeneity of the regions will increase, leading to an increase in load imbalance because the cost of planning depends on the complexity of the region. To address the load imbalance, we apply standard load balancing techniques based on data-structure redistribution and work stealing and show the effectiveness of the two techniques at combating load balancing issues that arise at scale.

Unlike other previous and related work, our work covers the two broad classes of sampling-based motion planners: graph-based (e.g., the probabilistic roadmap method (PRM)[16]) and tree-based (e.g., rapidly-exploring random tree (RRT)[32]) methods. We explored different planning space subdivision approaches suitable for the two sampling-based motion planning broad classes: a uniform mesh-like subdivision for graph-based (see Figure 1.1(a)) and radial subdivision (see Figure 1.1(b)) for tree-based. We provide both theoretical and empirical proof of scalable and superior performance compared to previous methods. We present experimental results obtained from our studies of a wide range of motion planning problems utilizing different parallel architectures; ranging from small-scale linux clusters to an IBM Power5+ machine to a Cray XE6 petascale machine.



(a)



(b)

Figure 1.1: Planning space subdivision strategies: (a) uniform subdivision and (b) radial subdivision

1.1 Research Contributions

The key contributions of this dissertation can be summarized as follows:

- The first reported work in parallel sampling-based motion planning based on spatial subdivision of the configuration space ($\mathbb{C}_{space}$). Our proposed framework is compatible with any sampling-based algorithm, including both graph-based methods, e.g., the Probablistic Roadmap (PRM) and the tree-based methods, e.g., Rapidly-exploring Random Trees (RRT).

- A novel radial subdivision of $\mathbb{C}_{space}$ suitable for tree-based planners that allows the computation to be distributed efficiently.

- A novel motion planning algorithm, Blind RRT capable of exploring the free space ($\mathbb{C}_{free}$) regardless of the obstacle space ($\mathbb{C}_{obstacle}$) to $\mathbb{C}_{free}$ ratio. Blind RRT provides both scalability and probablistic completeness for motion planning.

- Experimental results demonstrate we achieve better and more scalable performance on thousands of processors than previous parallel sampling-based planners. Application of load balancing techniques based on data-structure redistribution and work-stealing to achieve scalability across different motion planning problems.

Much of this research has been published [34, 27, 28, 29, 35, 33]. A poster [27] and a paper [28] describing the parallelization of graph-based motion planning algorithms were presented at the 2011 *ACM/Microsoft Research Student Research Poster Competition* at *Supercomputing Conference (SC)* and the 2012 *IEEE International Conference on Robotics and Automation (ICRA)*, respectively. Radial subdivision for RRT [29] and blind RRT [35] were published at *ICRA* 2013 and the *IEEE/RSJ International Conference on Robotics and Systems (IROS)* 2013, respectively. Our work on using load balancing techniques for complex motion planning problems [33] will be presented at *IEEE International Parallel and Distributed Processing Symposium (IPDPS)* in 2014.

1.2 Outline

The rest of this dissertation is organized as follows. We provide an overview of sampling-based motion planning and a survey of related work on parallel motion planning in Chapter 2. In Chapter 3, we discuss the overview of the scalable framework for parallelizing sampling-based motion planning algorithms. In Chapter 4 and Chapter 5, we focus on the specifics of our framework for parallelizing the graph-based and the tree-based motion planning algorithms, respectively. In Chapter 6, we extend our discusion on parallelizing tree-based motion planning algorithms, by presenting a novel probablistically complete and distributed RRT algorithm called Radial Blind RRT. Chapter 7 describes load balancing techniques for enabling scalable parallelization of sampling-based motion planning algorithms. In Chapter 8, we evaluate the quality and structure of the roadmaps constructed using our proposed framework. Finally, we conclude the dissertation in Chapter 9.

2. PRELIMINARIES AND RELATED WORK

2.1 Preliminaries

2.1.1 Motion Planning

The motion planning problem is to find a valid path (e.g., one that is collision-free and satisfies any joint limit and/or loop closure constraints) for a movable object starting from a specified start configuration to a goal configuration in an environment [15]. A single configuration is specified in terms of the movable object's d independent parameters or degrees of freedom (DOF). The set of all possible configurations (both feasible and infeasible) defines a configuration space ($\mathbb{C}_{space}$) [14, 15]. $\mathbb{C}_{space}$ is partitioned into two sets: $\mathbb{C}_{free}$ (the set of all feasible configurations) and $\mathbb{C}_{obstacle}$ (the set of all infeasible configurations). Motion planning then becomes the problem of finding a continuous sequence of points in $\mathbb{C}_{free}$ that connects the start and the goal configuration.

A complete solution to the motion planning problem is computationally intensive and has been proved to be PSPACE-hard with an upper bound that is exponential in the movable object's DOFs [14, 15]. In other words, for any complete planner to guarantee that a solution to a motion planning problem exists or not, exponential time in the number of DOFs is required. As an alternative, there are efficient heuristic and approximate algorithms that trade completeness for efficiency. Sampling-based motion planning is one such approach.

2.1.2 Sampling-Based Motion Planning

Sampling-based methods [15] are a state-of-the-art approach to solving motion planning problems in practice. While not guaranteed to find a solution if one exists, sampling-based methods are known to be probabilistically complete, i.e., the probability of finding a solution given one exists increases with the number of samples generated. Sampling-based methods are broadly classified into two main classes: roadmap or graph-based methods such as the Probabilistic Roadmap Method (PRM) [16] and tree-based methods such as Rapidly-exploring Random Tree (RRT) [32].

2.1.2.1 Graph-Based Methods

The Probablistic Roadmap Method (PRM) is a well known sampling-based motion planning approach. In solving motion planning problems, PRM constructs a graph $G = (V, E)$, called a roadmap, to capture the connectivity of $\mathbb{C}_{free}$ (Figure 2.1 [36]). A node in the graph G represents a valid placement of the movable object, and an edge is added between two nodes if a simple path can be defined and validated by the so-called *local planner* — an important primitives of all sampling-based planners [37, 38, 39, 40, 41]

In the original method [16] (shown in Algorithm 1), nodes are generated using uniform random sampling and connections are attempted between a node and its k-nearest neighbors as computed using some distance metric (e.g., Euclidean, Geodesic or Root-Mean-Square distance [40]). Once the roadmap is constructed, query processing is done by connecting the start and goal configurations to the roadmap and extracting a path from the roadmap that connects them. Many variants of PRMs have been proposed that bias node generation or connection or query processing in various ways [42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53].

2.1.2.2 Tree-Based Methods

The Rapidly-exploring Random Tree (RRT) is another sampling-based motion planning method used in practice. RRT is particularly well suited for non-holonomic and kinodynamic motion planning problems [54, 55]. The basic sequential RRT

Algorithm 1 Sequential PRM

```
Input: An environment $env$, the number of nodes $N$
Output: A roadmap graph $G$ containing $N$ nodes
 1: $i \leftarrow 0$
 2: while $i < N$ do
 3:     $q \leftarrow$ GetValidRandomNode($env$)
 4:     $G$.AddNode($q$)
 5:     $i \leftarrow i + 1$
 6: end while
 7: for all $q \in G$ do
 8:     $Q \leftarrow$ FindNeighbor($G, q, k$)
 9:     for all $q_{near} \in Q$ do
10:         if local planner can connect $q$ and $q_{near}$ then
11:             $G$.AddEdge($q, q_{near}$)
12:         end if
13:     end for
14: end for
15: return $G$
```

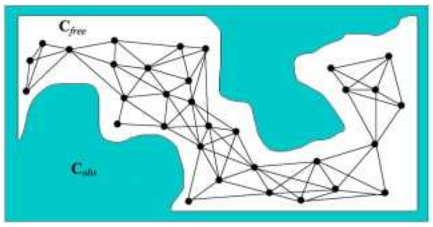


Figure 2.1: An illustration of PRM[1]

(shown in Algorithm 2 and illustrated pictorially in Figure 2.2 [36]) grows a tree rooted at the start configuration that expands outward into unexplored areas of the $\mathbb{C}_{space}$. RRT first generates a uniform random sample q_{rand}, and identifies the closest node q_{near} in the tree to q_{rand}, and then q_{near} is "extended" toward q_{rand} for

[1]Reprinted from Computer Science Review, Volume 6, I. Al-Bluwi, T. Simon, J. Cortes, Motion planning algorithms for molecular simulations: A survey, Pages 125-143., Copyright (2012), with permission from Elsevier. [36]

a stepsize of at most Δq. If the extension is successful, q_{new} is added to the tree as a node and the pair q_{near} and q_{new} is added as an edge. To solve a particular query, RRT repeats this process until the goal configuration is connected to the tree. RRT-connect is a variant that grows two trees towards each other; one rooted at the start configuration and the other at the goal configuration [56]. These two trees explore $\mathbb{C}_{space}$ until they are connected. Many variants of RRT have been proposed and discussed [15, 57, 23, 53].

Algorithm 2 Sequential RRT

```
Input: An environment $env$, a root $q_{root}$, the number of nodes $N$
Output: A tree $T$ containing $N$ nodes rooted at $q_{root}$
 1: $T$.AddNode($q_{root}$)
 2: $i \leftarrow 0$
 3: while $i < N$ do
 4:     $q_{rand} \leftarrow$ GetRandomNode($env$)
 5:     $q_{near} \leftarrow$ FindNeighbor($T, q_{rand}, 1$)
 6:     $q_{new} \leftarrow$ Extend($q_{near}, q_{rand}$)
 7:     if !TooSimilar($q_{near}, q_{new}$) $\wedge$ IsValid($q_{new}$) then
 8:         $T$.AddNode($q_{new}$)
 9:         $T$.AddEdge($q_{near}, q_{new}$)
10:         $i \leftarrow i + 1$
11:     end if
12: end while
13: return $T$
```

2.2 Related Work

2.2.1 Parallel Sampling-Based Motion Planning

Research in robotic motion planning spans over three decades, resulting in the development of different types of sequential and parallel algorithms for motion planning [15, 58]. The recent renewed interest in parallel motion planning algorithms

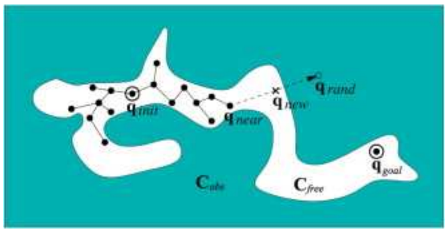


Figure 2.2: An illustration of RRT[2]

is due to the progress made in sequential algorithms, the ubiquity of parallel and distributed machines, and the demand for more efficiency in solving complex, high dimensional problems. In this section, we discuss related work in parallelizing motion planning algorithms.

One of the earliest parallel motion planning algorithm is the parallel randomized search algorithm proposed by Gini in 1999 [18]. Using the algorithm, the $\mathbb{C}_{space}$ was discretized, represented with bitmap arrays, and then broadcast to all processors. The desired goal location was also broadcast to all processors and each processor explored the entire search space randomly. The first processor to find a path from the start location to the goal location sends a termination signal to the remaining processors, and then reports its solution.

The search algorithm is as shown below:

```
(Each processor does the following in parallel)
1.  repeat until goal found or global time-out
2.      Gradient Descent until local minimum
3.      while no improvement or time-out
```

[2]Reprinted from Computer Science Review, Volume 6, I. Al-Bluwi, T. Simon, J. Cortes, Motion planning algorithms for molecular simulations: A survey, Pages 125-143., Copyright (2012), with permission from Elsevier. [36]

```
4.       repeat K times or until improvement found
5.          Random Walk to escape local minimum
6.          Gradient Descent until a local minimum
6.       if no improvement
7.          then Randomly Backtrack
8.       if improvement found
9.          then append new path to previous path
7. if goal found
8.    then broadcast termination message
```

As described in the algorithm above, two heuristic measures — the Gradient Descent and Random Walk — were used to find a better node and guide the search path at each step. The random walks and randomized backtracking also help find a place in a different region of the search space where the heuristic is more reliable. At every point in the search path, successors of a node are generated in a random manner until a successor is found with a better heuristic value that will eventually lead to the goal configuration.

Isto [19] describes a two level algorithm to solving motion planning problems of average degrees of freedom. The parallel implementation of the algorithm in [19] was reported in [20]. The basic idea of the two level algorithm is to deal with the exponential cost of the complete discretization of the $\mathbb{C}_{space}$ and the susceptibility to local minimal of local plannners. Unlike the classic grid-based approach, this approach does not explicitly compute or build the $\mathbb{C}_{space}$. Rather, landmarks or subgoals are generated in the space and attempts are made to connect them. Thus, the path from the start to the goal is found via a number of randomly generated subgoal configurations.

In its parallel implementation [19], an implicit grid representation of the $\mathbb{C}_{space}$ was made. Local planners were distributed as tasks across slave processors. Each slave process generates a minimal number of subgoals and landmarks and attempts to connect them using the local planners. The local planners are adaptive and are coordinated by the global planner on the master processor using some heuristic measures. This heuristic measure decision is based on how many subgoals are generated in each cell. As more subgoals are needed and generated for solving the problem, the global planner increases the capability of the local planner. The author exprimented with the 5 DOF benchmark of Hwang and Ahuja [59] to solve the problem with a $296 \times 171 \times 42 \times 191 \times 105$ grid representation of $\mathbb{C}_{space}$ in seconds. A further resolution of the $\mathbb{C}_{space}$ into $2960 \times 1710 \times 420 \times 1910 \times 1050$ grid was also solved in minutes on a Linux PC cluster with 11 processors.

PRM was the underlying sequential algorithm for the work reported in [21, 17]. The parallel algorithm is as shown below. The algorithm proceeds in two stages. First, node generation which was reported to be 2-3% of the total execution time. At the node generation stage, each of the p processors samples the $\mathbb{C}_{space}$ in parallel to generate N/p configurations. The second stage was that of connecting nodes generated in the first stage to form roadmap. At the second stage, attempts are made by each processor to connect each sample to its k nearest neighbors. The original parallel algorithm [21] was implemented in a shared-memory machine focusing on robotics applications. The parallel approach was later extended to a protein folding application [17] and was implemented on distributed memory machines.

```
PRM NodeGeneration
(Each processor does the following)
1. for $1 \leq i \leq n/p$
2.     generate a random cfg, $c$
```

```
3.    if $c$ is free
4.       save $c$
5.    endif
6. endfor
```

<u>PRM NodeConnection</u>

(Each processor does the following; each has a unique *pid*)

```
1. for each cfg $c$, indexed $p * (pid - 1)$ to $p * (pid)$
2.    $N := k$ closest neighboring from *all* cfg's to $c$
3.    for each $n \in N$
4.       if local planner can connect $n$ and $c$
5.          save edge($n$,$c$)
6.       endif
7.    endfor
8. endfor
```

In [22], the authors adopt the OR parallel paradigm to parallelize RRT computations on shared-memory machines. The RRT computation is replicated on each process and processes concurrently explore the entire $\mathbb{C}_{space}$. The first process to find a solution sends a termination message to other processes. In the same work, the authors present a parallel algorithm in which processes concurrently and cooperatively build a single tree under a shared-memory model. Each process executes its own program and communicates to the other processes by exchanging data through the shared memory in a concurrent read exclusive write ($CREW$) fashion. The authors also study a hybrid algorithm combining the OR parallel paradigm and the $CREW$ model. The processes are divided into groups and each group cooperatively builds its own tree. The first group to find a solution sends a termination message to the others.

Bialkowski et al. parallelize RRT and RRT* by focusing on parallelizing the collision detection phase [23]. The implementation was done in CUDA and GPU. A more recent work focuses on multicore architectures [24]. The authors present three algorithms for distributed RRT. The first algorithm is a message passing implementation of the OR parallel paradigm. In the second algorithm, each process builds part of tree and globally communicates with the other processes each time a new node and edge is added. The third algorithm adopts a manager-worker approach. Instead of having multiple copies of the tree, only the manager initializes and maintains the tree while the expansion computation is delegated to the worker processes. The drawback with the manager-worker approach is that it does not scale well as it is prone to load imbalance with more workload on master process(es).

Another algorithm of interest is the Parallel Sampling-based Roadmap of Trees (PSRT) [25, 26, 13]. PSRT combines the multiple query sampling characteristics of PRMs with the efficient local planning capabilities of single query of RRTs. In the PSRT roadmap graph, the nodes represent trees and not individual configurations as in regular PRM. The collections of these trees form the roadmap (Figure 2.3). Connections between trees are attempted between closest pairs of configurations between the two trees. Similar to the third algorithm of [24], the authors adopted the manager-worker architecture. Each worker process computes a predefined number of trees in the entire $\mathbb{C}_{space}$. The manager is responsible for arbitration of tree ownership, nearest neighbor computations, and determination of which pairs of trees to attempt for connection. Edge validation is distributed to the worker processes.

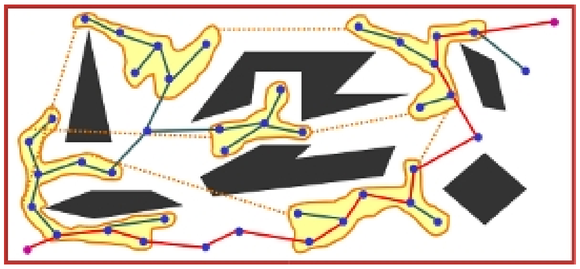


Figure 2.3: An illustration of SRT

2.2.2 Space Subdivision

The concept of $\mathbb{C}_{space}$ subdivision has been proposed and used in many existing sequential motion planning algorithms. One of the earliest complete (or exact) motion planning algorithms computes an exact representation of C-space by uniformly dividing it along the robot's degrees of freedom into cells [60]. However, this approach is not practical for high dimensional problems because of its exponential computation complexity.

Another space subdivision approach is the Approximate Cell Decomposition (ACD) method [61]. ACD subdivides the C-space into rectangular cells. Each generated cell is labelled as *empty* if it lies completely in free space, *full* if it lies completely in obstacle space, or *mixed* otherwise. PRM is combined with ACD to compute localized roadmaps by generating samples within these cells. The connectivity graph for adjacent cells in ACD is augmented with pseudo-free edges that are computed based on localized roadmaps.

Feature sensitive motion planning [62, 63] proposes a supervised method of recursively breaking up an environment into regions and classifying these regions as free, clutter, narrow, or blocked by comparing region features to a database of known region types. Roadmaps are constructed in each region and recombined to form a final roadmap. Partitioning was first done by randomly choosing one of the robot's degrees of freedom and dividing along a random value for that parameter [62]. This partitioning process was repeated recursively until homogeneous but overlapping sets of regions are obtained where homogeneity is defined according to a set of features measured for each region. The partitioning approach in [63] is based on knowledge of the environment gained by building a small roadmap and using configurations from the roadmap to determine the best degrees of freedom to subdivide and the best splitting point within those degrees of freedom.

RESAMPL [64] subdivides the C-space into local regions based on an initial sampling of the entire space. As a partitioning strategy, RESAMPL first generates a small set of samples, both valid and invalid, in the entire space. Some of these samples, selected from the set randomly, become representative samples for the local regions. Region sizes are determined by the distance of the representative sample to its k-nearest neighbors in the initial set.

2.2.3 Load Balancing Techniques

Load balancing is the practice of distributing computation or workload among parallel processing elements in an approximately equal manner. It is a well-studied problem in parallel and distributed computing. Load balancing is critical to the overall performance of a parallel algorithm. The overall performance is affected because the slowest process(or) (possibly with more work than other process(or)) determines the overall performance and scalability of the parallel algorithm. There are a number of load balancing techniques in the literatures, but work-stealing (active attempts to "steal" work from possibly overloaded process(or)) has become the *de facto* dynamic scheduling technique for various parallel programming environments and runtimes, including Cilk [65], TBB [66], UPC [67] and many others. Blumofe and

Leiserson [68] show that work-stealing is provably optimal within a constant factor for scheduling multithreaded computations with dependences. These approaches prove successful in shared-memory architectures, but have their limitations when applied to distributed-memory. For shared-memory implementations, the issue of locality is generally not stressed, due to the relatively uniform level of memory access compared with distributed memory. Recently, locality-aware work stealing implementations began placing more emphasis on the notion of affinity [69] and have shown to perform well in practice.

The issue of locality in work stealing scheduling becomes more important in distributed-memory, as assigning a task to a non-affine core could result in a severe degradation in performance due to remote memory accesses. In some PGAS programming environments such as UPC [70], it is suggested that coarse-grained tasks be preferred over fine-grained tasks, as a large number of small remote accesses will have a higher impact on performance.

The X10 programming language [71] and runtime system offers work-stealing in distributed-memory architectures. Of particular interest, X10's lifeline work-stealing approach has shown success in balancing load for various applications, including the popular UTS [72] benchmark. Chapel [73] is a programming language for parallel computations that runs in distributed-memory. It provides work-stealing scheduling, but is currently limited to only computations which run on shared-memory.

Charm++ [74] is a parallel programming language and runtime environment that supports a large suite of load balancing mechanisms. In the Charm programming environment, computations are expressed as objects that represent both the work and associated data. In such a model, the work and data are inherently coupled, making it difficult to reason about a data structure or describe a computation in parametric and data- independent fashion.

In addition to work stealing, other popular approaches for load imbalance include using partitioning tools for meshes, arbitrary graphs and other data structures. Zoltan [75], ParMetis [76] and Jostle [77] are just a few such redistribution frameworks that provide various repartitioning algorithms and data management tools. These approaches are suited for algorithms that follow a pattern of partitioning followed by computation separated by global barriers, but do not allow for asynchronous migration of elements during a computation.

3. STRATEGY FOR PARALLELIZING SAMPLING-BASED MOTION PLANNING ALGORITHMS*

In this chapter, we discuss our approach for parallelizing sampling-based motion planning algorithms, starting with general overview that is common to both graph-based and tree-based motion planning algorithms. We then present an overview of the Standard Template Adaptive Parallel Library (STAPL), the parallel $C++$ library from which all parallel algorithms presented in this dissertation are built.

3.1 Strategy Overview

We present a four step strategy for parallelizing sampling-based motion algorithms. These steps are the high-level description of our approach and are shown in Algorithm 3.

```
Algorithm 3 Parallel Sampling-based Motion Planning
Input: An environment $E$, A set of motion planners $S$, number of regions $N_R$
Output: A roadmap graph $G$ or tree $T$
1: Decompose $E$ into $N$ regions
2: Make a region graph $R = (V_R, E_R)$ with $V_R$ and $E_R$ representing each region and
   adjacency information between regions, respectively
3: Independently and in parallel, construct roadmaps or trees in each region using
   any desired planner $s \epsilon S$
4: Connect regional roadmaps or trees in adjacent regions to form a roadmap $G$ or
   tree $T$ for the entire problem
```

In step 1, we subdivide a given environment describing the obstacles and movable

*Part of the data reported in this chapter is reprinted with permission from "A Scalable Method for Parallelizing Sampling-Based Motion Planning Algorithms" in *Proc. IEEE Int. Conf. Robot. Autom. (ICRA)* by S. A. Jacobs, K. Manavi, J. Burgos, J. Denny, S. Thomas, and N. M. Amato, 2012. Copyright ©2012, IEEE. [28]

object into regions. These regions represent sub-problems that can be processed in parallel. The subdivision procedure is generic so as to support different planning space decomposition strategies depending on the nature of the problem. For example, a uniform workspace subdivision may be sufficient for a motion planning problem of average degrees of freedom or a uniformly cluttered environment (see Figure 3.1(a)). In another case, the subdivision could be radial that is tailored to a particular motion planner (e.g., RRT) (see Figure 3.1(b)). In some other cases, adaptive subdivision that is tailored to the heterogeinity of the environment is needed so as to adapt suitable motion planners different part of the environment (see Figure 3.1(c)). A combination of both uniform and adaptive subdivision can also be applied if need be.

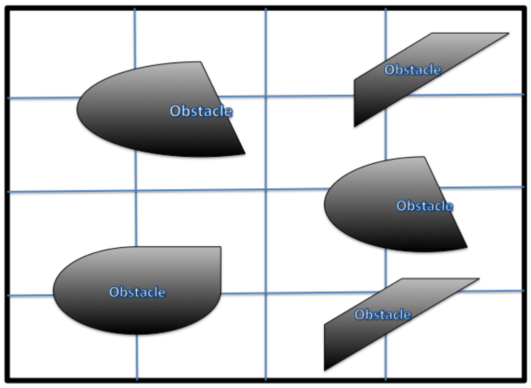
In step 2, we make a *region graph* of the regions resulting from the planning space subdivision in step 1. The *region graph* is an abstraction that represents the spatial relationship between regions. In particular, the vertices of the *region graph* represent the regions and the edges represent the adjacencies between regions. As a relational concept, no assumption is made about the nature of the *region graph*; it could be a mesh graph, a graph of fixed degree where the number of neighbors is the same or fixed a prior, or a graph of graphs representing the hierarchical nature of the regions. As an example, a hierarchical region graph would have the outer graph as super-vertices representing outer regions and an inner graph representing the inner regions. The region graph facilitates the inter-regional roadmap or tree connection at a later stage. The flexibility of constructing such a graph lends itself to graph algorithms that can be easily parallelized and redistributed to resolve load imbalance, a common occurrence in complex non-homogeneous motion planning problems.

In Step 3, having subdivided the planning space into regions, we independently and in parallel, construct a roadmap or tree in each region.The roadmap or tree construction does not depend on the underlying sampling-based motion planning algorithm or strategy and can handle a variety of planning schemes. In other words, an appropriate sequential planner (e.g., PRM or RRT or their variants) can be used in constructing regional roadmap (subgraph) or regional trees (subtrees). This phase of the computation is embarrassingly parallel. The task or subproblem associated with each region is to build a roadmap (graph or tree) that encodes representative paths approximating the topology of the planning space of the associated region, this is done in parallel without inter-regional (or interprocess) communication.

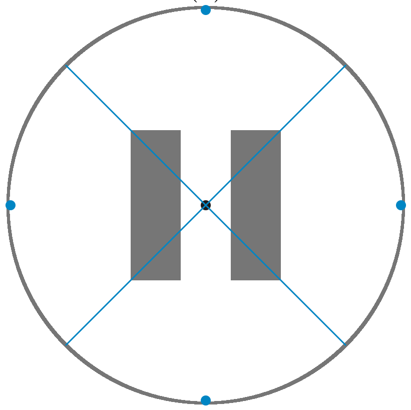
In Step 4, we connect nearby regional roadmaps or trees to form a roadmap representing the entire planning space. The region graph is the enabling infrastructure facilitating the process of connecting the region roadmaps. The region graph infrastructure aids identification of adjacent regions between which connections are attempted. In this way, communication is limited only to adjacent regions. As is shown in subsequent chapters, the implementation of the region connections is flexible and influenced by specific subdivision strategies or underlying sequential planners (e.g., possibility of a cycle is avoided when a single-rooted tree is desired).

3.2 STAPL Framework

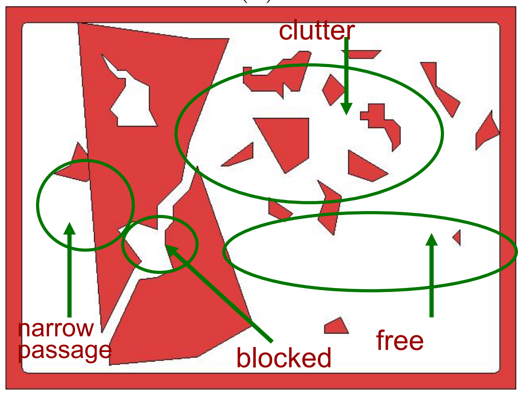
All the parallel algorithms discussed in this dissertation have been implemented using STAPL (Standard Template Adaptive Parallel Library), a research project in the Parasol Lab at Texas A&M University. STAPL [78, 79, 80, 81] is a generic, scalable framework for parallel C++ code development. STAPL is designed as a superset of ISO Standard C++ Standard Template Library (STL) [82]. STAPL is platform independent and supports both shared and distributed memory. STAPL provides a collection of building blocks (as shown in Figure 3.2) for writing parallel programs. These building blocks are commonly referred to as components and include a collec-



(a)



(b)



(c)

Figure 3.1: Types of subdivision: (a) uniform (b) radial (c) adaptive

tion of parallel algorithms (`pAlgorithm`s), parallel and distributed data structures (`pContainer`s) and views to abstract data access in `pContainer`s.

STAPL `pContainer`s are similar to the STL containers but much more enriched and support both static and dynamic parallel and distributed data structures. The `pContainer`s include pVectors, pArray, pList, pMatrices and pGraphs, which are parallel versions of vector, array, linked list, matrices and graphs respectively. The STAPL `pAlgorithm`s provide parallel versions of the STL algorithms and are written in terms of views similar to how STL algorithms are written in terms of iterators. The STAPL $PARAGRAPH$ abstracts the concept of a task graph needed for parallel execution. Each task in the task graph consists of a workfunction and a view representing the data on which the workfunction will be applied. STAPL also provides a communication infrastructure called an adaptive runtime system ($ARMI$). $ARMI$ is built on MPI and hides machine specific details and provides a uniform communication interface.

Except otherwise noted, all algorithms presented in this work were written in $C++$ and implemented within the STAPL framework as STAPL `pAlgorithm`s. These `pAlgorithm`s are implemented using the `pContainer` as data structure. In particular, we made use of the STAPL graph library [83] to represent the parallel data structures (e.g., the region graph, the roadmap graph, the rapidly-exploring random tree (RRT)) and a number of STAPL graph algorithms such as bread-first-search (BFS), pagerank, connected components, diameter, and single-source shortest path (SSSP).

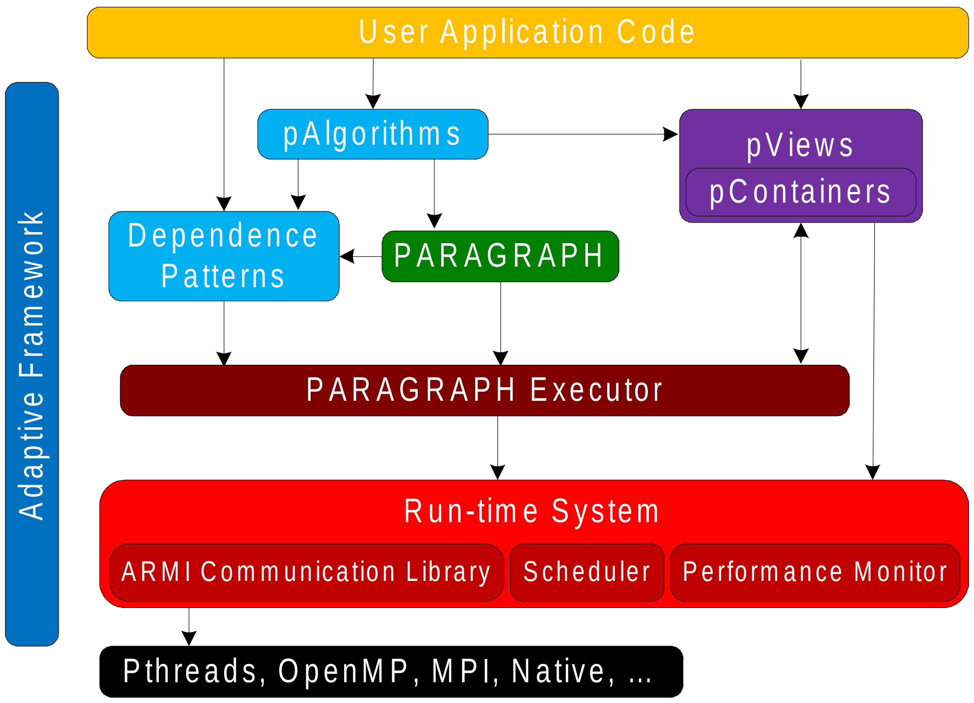


Figure 3.2: STAPL software architecture.

4. GRAPH-BASED PARALLEL MOTION PLANNING*

In this chapter, we discuss our approach for parallelizing graph-based motion planning algorithms. The overall algorithm is shown in Algorithm 4. We discuss the core subroutines of the algorithm in the following sections.

Algorithm 4 Graph-Based Algorithm

```
Input: An environment $env$, the number of nodes $N$, the number of processes $p$,
    the number of regions $N_r$
Output: A roadmap graph $G$ containing $N$
 1: Let region graph $R(V, E) = \emptyset$.
 2: Let $R_d$ = Subdivide $E$ into $N_r$ regions.
 3: Add a vertex for each region $r$ of $R_d$ to $R$.
 4: for all neighboring regions $(r_1, r_2) \epsilon R_d)$ par do
 5:     Add the edge $(r_1, r_2)$ to $R$.
 6: end for
 7: for all regions $v_i \in V$ par do
 8:     $G \leftarrow$ Construct regional roadmap using sequential planner
 9: end for
10: for all neighboring regions $(v_i, v_j) \in E$ par do
11:     $G \leftarrow$ Connect roadmap of regions $v_i$ and $v_j$
12: end for
13: return $G$
```

4.1 Space Subdivision and Region Graph Construction

In line 2 of Algorithm 4, the environment representing the movable object and the obstacles is subdivided into regions. The subdivision is based on the geometry of the planning space. The planning space may be subdivided into regions using the $\mathbb{C}_{space}$

*Part of the data reported in this chapter is reprinted with permission from "A Scalable Method for Parallelizing Sampling-Based Motion Planning Algorithms" in *Proc. IEEE Int. Conf. Robot. Autom. (ICRA)* by S. A. Jacobs, K. Manavi, J. Burgos, J. Denny, S. Thomas, and N. M. Amato, 2012. Copyright ©2012, IEEE. [28]

positional degrees of freedom, i.e., the x, y and z dimensions. A simple illustration of a 2D environment subdivided into nine regions is shown in Figure 4.1(a). We maintain some user-defined overlap between regions to allow sampling in the portions of the space that are at the boundaries that may facilitate connection between regional roadmaps at a later stage.

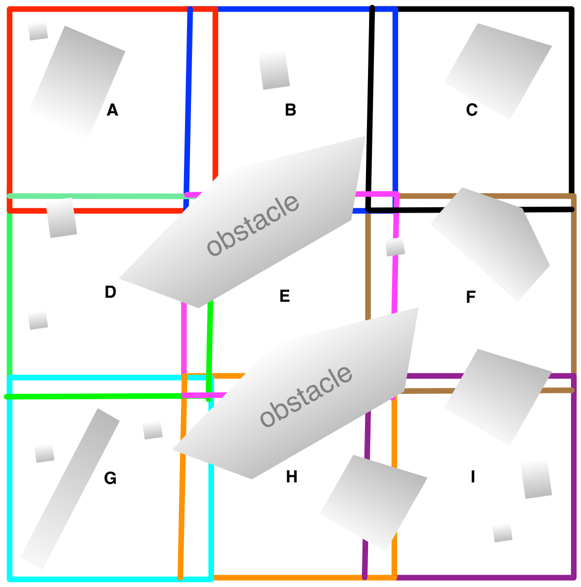
The subdivision is represented by a *region graph*, whose vertices represent regions and whose edges encode the adjacency information between regions. Figure 4.1(b) shows the region graph corresponding to the subdivision shown in Figure 4.1(a). The algorithm for the region graph construction is shown in Algorithm 5. In addition to geometric and adjacency information, the *region graph* also maintains additional information that keeps track of the connected components in each region. This additional information is used when connecting adjacent regions.

Algorithm 5 Region Graph Construction

```
Input: An environment $E$ and the number of regions $N_R$.
Output: A region graph $R$.
Let $R = \emptyset$.
Let $R_d$ = SubDivideSpace($E, N_R$).
Add a vertex for each region $r$ of $R_d$ to $R$.
for all neighboring regions $(r_1, r_2) \in R_d$) par do
    Add the edge $(r_1, r_2)$ to $R$.
end for
return $R$.
```

4.2 Constructing Regional Roadmaps

Sequel to space subdivision and region graph construction, each processor is assigned at least one region and the task of building a regional roadmap in its assigned region(s) using sequential planner. At this step, any of the existing sampling-based



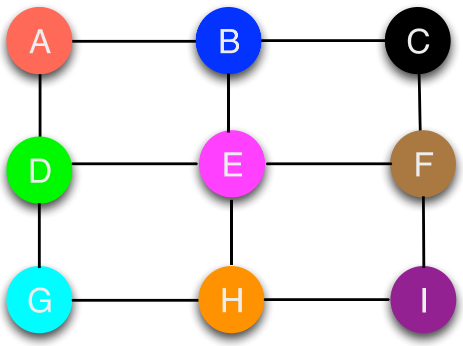


Figure 4.1: Space subdivision: (a) A 2D environment subdivided into 9 regions, (b) region graph - the 9 vertices represent each of the 9 regions with corresponding color, edges encode the adjacency information between regions.

motion planning algorithms, such as PRM (and its variants) or RRT (and its variants) can be used. This step is independent of the sampling strategy employed. In constructing the regional roadmap, each processor independently generates and connects samples in its assigned region with no communication with other regions. The nodes and edges made at this step are added to the roadmap graph. These nodes and edges represent the valid configurations of the movable object and the connections between the configurations, respectively.To facilitate and streamline the connection at the next step, we keep track of the size and a vertex representative for each connected component ($CCIDs$) in the regional roadmap. These $CCIDs$ are stored in the region graph for each region.

4.3 Connecting Regional Roadmaps

The final step in constructing the full roadmap is to connect the regional roadmaps. Prior to this step, we track the sizes and number of connected components in each region. The regional graph stores this information which is input to the region connection algorithm shown in Algorithm 6. Other inputs to the algorithm include: k, the number of connections to be attempted between adjacent regions, the type of connection method, and a local planner used to verify connections.

For every edge identifying neighboring regions in the region graph, we attempt a connection between candidate node(s) of connected components in the source region to candidate node(s) of connected components in the target region. Even though our implementation is independent of which region connection method is used, for the results presented in this dissertation, we attempt to connect regions based on the size of connected components in each region and the distance between connected components across regions [84]. For the size-based connection, we attempt connections between a user-defined k largest connected components from the source region and k

Algorithm 6 Region Roadmap Connection

```
Input: A region graph $R$, connection method, $k$ number of candidates, local planner $lp$
Output: A roadmap graph $G$.
for all edges $E \epsilon R$ par do
  if (connection method == closest) then
    sourceCC = select $k$ center of mass based closest CC to target region from $E.source$
    targetCC = select $k$ center of mass based closest CC to source region from $E.target$
  end if
  if (connection method == largest) then
    sourceCC = select $k$ largest CCs from $E.source$
    targetCC = select $k$ largest CCs from $E.target$
  end if
  for all pairs($sourceCC, targetCC$) do
    if lp.IsConnectable($sourceCC, targetCC$) then
      Add the edge($sourceCC, targetCC$) to $G$.
    end if
  end for
end for
return $G$.
```

largest connected components from the target region. For the distance-based connection, we attempt to connect the k-closest connected components between the regions based on the distance between them. This distance is computed between the centers of mass (a measure of average of all configurations in the connected component) of the two connected components.

4.4 Algorithm Analysis

The original PRM algorithm as reported in [16] requires $O(N^2)$ time and $O(N)$ space to construct a roadmap with N configurations. This serves as the basis for our analysis and is used for the complexity of constructing a regional roadmap.

The overall time complexity of our approach as described in Algorithm 3 can be given as:

$$T = T_d(Env, n_r) + \sum_{i=1}^{n_r} T_r(i) + T_c(i,j)|E_R| \tag{4.1}$$

where T is the sum of the cost of space decomposition T_d for a given environment Env subdivided into n_r regions, the cost $T_r(i)$ of roadmap construction in region r_i, for all $v_i \in V_R$, and the cost $T_c(i,j)$ of connecting regional roadmaps in regions r_i and r_j, for all $(r_i, r_j) \in E_R$. If we make a simplifying assumption that the cost of constructing regional roadmap is the same for all regions $v_i \in V_R$, then equation above can be rewritten as :

$$T = T_d(Env, n_r) + T_r(i)|V_R| + T_c(i,j)|E_R| \tag{4.2}$$

Step 1 involves space decomposition. We assume p processors/tasks and that the regions are divided equally among the p processors. In this case, space decomposition and region graph construction can be done in $O(|V_R| + |E_R|)/p$ where V_R and E_R are the vertices and edges of the region graph, respectively.

Step 2 of Algorithm 3 involves the construction of the regional roadmaps. Since we are assuming there are p regional roadmaps, each of the same size, this implies they will have N/p nodes each, and hence the cost of (sequentially) constructing the PRM roadmap for each region will be $O((N/p)^2)$. If regional roadmaps are RRTs instead, one would use the cost of constructing an RRT of size N/p here instead, and similarly for any other desired approach for constructing a regional roadmap.

Inter-processor communication occurs when connecting regional roadmaps. The region graph infrastructure helps to limit both computation and communication to adjacent regions. If we assume a naive connection attempt between every configuration in a region to every configuration in neighboring region, this worst case scenario will result in $O((N/p)^2)$ edge computations plus the cost of communication between neighboring processors.

Thus, in summary, the time, work and space complexity of this approach can be given as $O((N/p)^2)$ time, $O((N^2)/p)$ work, and $O(N)$ space respectively.

4.5 Experimental Evaluation

In this section, we analyze the performance of our strategy for parallelizing graph-based motion planning algorithms comparing the results with two similar previous methods. We evaluate the performance of our framework on two different parallel machines for two different motion planning problems. We demonstrate that our approach achieves more scalable performance than the previous parallel algorithms.

4.5.1 Algorithms

We implemented four different algorithms. The first two were based on our proposed approach but with two different strategies as the underlying sequential planner. These two implementations are referred to as pSBMP-RRT, a parallel sampling-based motion planning method with RRT as the underlying sequential planner, and pSBMP-PRM, a parallel sampling-based motion planning method with PRM as the underlying sequential planner. For evaluation and comparison, we implemented two additional parallel algorithms: the parallel PRM (pPRM) [21] and parallel sampling-based roadmap of trees (pSRT)[25, 26]. Please note that pPRM and pSRT were implemented based on our understanding of how they were described in the literature and it is possible that different implementations may perform better.

4.5.2 Machine Architectures

Our experiment was carried out on two massively parallel computers. The first is a Cray XE6 petascale machine at Lawrence Berkely National Laboratory. It has 6384 nodes and a total of 153,216 cores with 217 TB of memory and peak performance of 1.288 peta-flops. The second machine is a major computing cluster at Texas A&M University. It has a total of 300 nodes, 172 of which are made of two quad core Intel Xeon and AMD Opteron processors running at 2.5GHz with 16 to 32GB per node. The 300 nodes have 2400 cores in all with over 8TB of memory and a peak performance of 24 Tflops.

4.5.3 Motion Planning Problems

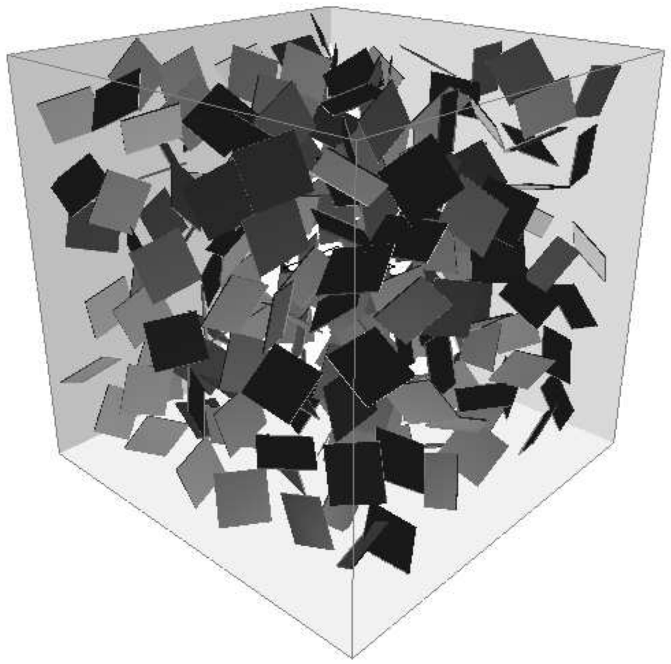
We used two different kinds of environments. The first is a homogeneous cluttered environment with dimensions of 512 x 512 x 512 units. The cluttered elements span the x-axis. The cluttered environment has a total of 216 obstacles, each of size 2 x 64 x 64 units, as shown in Figure 4.2(a). The second environment shown in Figure 4.2(b) is a non-homogeneous cluttered environment. This particular environment models the floor plan of the H.R. Bright building (HRBB), the building that houses the Departments of Computer Science and Engineering and Aerospace Engineering at Texas A&M University.

In both environments, we use two different kinds of robots: a 4 x 4 x 4 unit cube-like rigid body robot and a three-link articulated linkage robot, with each link having dimensions of 7 x 1 x 1 units.

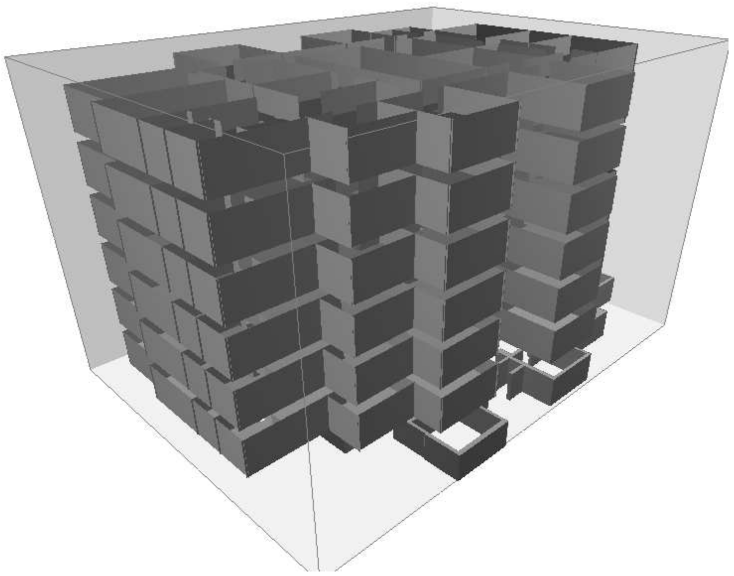
4.5.4 Experimental Results

4.5.4.1 Comparison with Previous Approaches

We tested the four algorithms (pSBMP-PRM, pSBMP-RRT, pPRM and pSRT) on the Linux cluster varying the processor count from 1 to 16. The input sample size was fixed at 9600 for each of the four algorithms. Each experiment was run five times and the average maximum time for the 5 runs was computed. Figures 4.3(a) and (b) show the running time and speedup for the four algorithms. From Figure 4.3, one

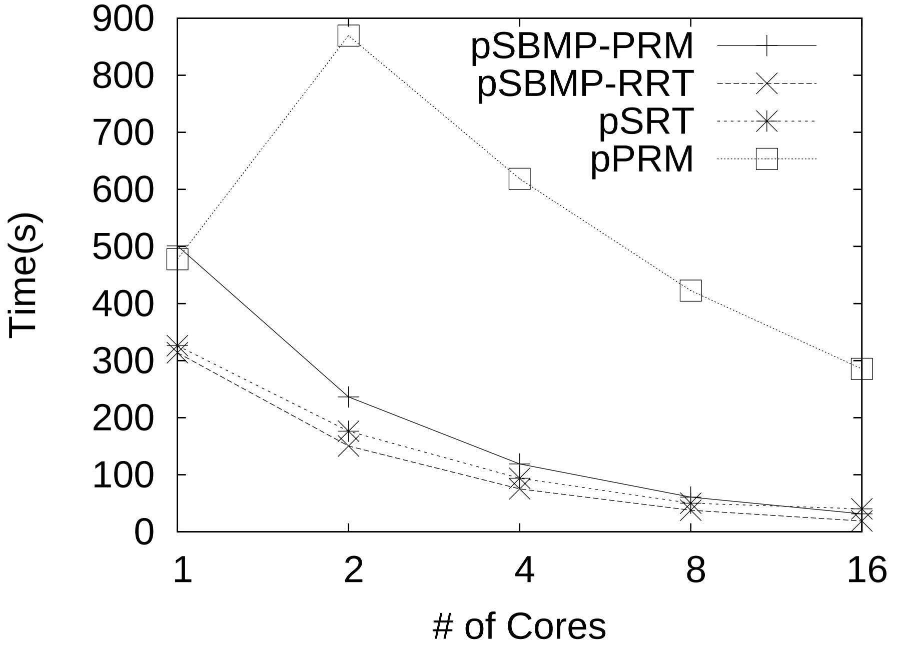


(a) Clutter

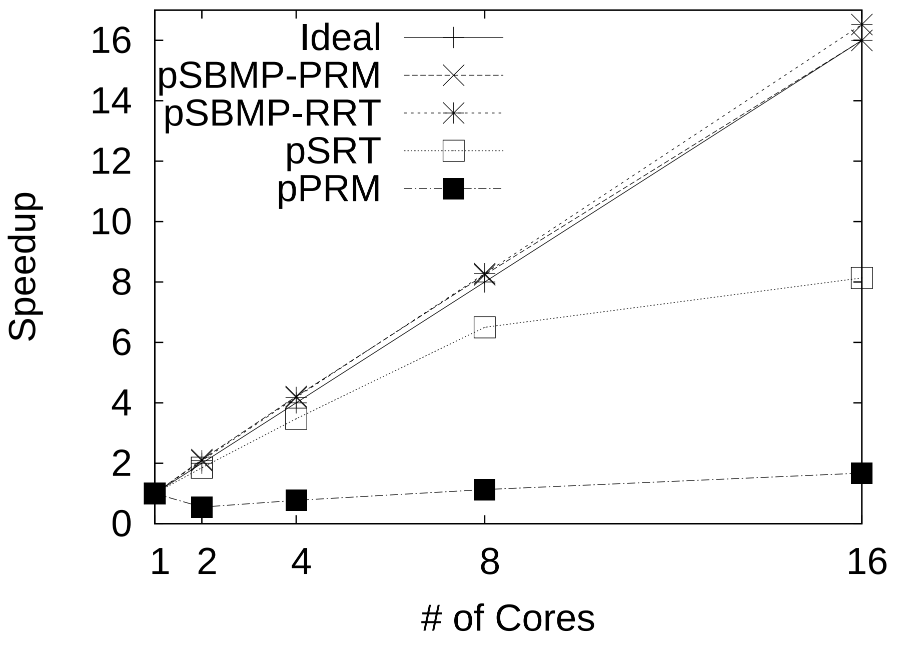


(b) Building

Figure 4.2: Environments studied for graph-based method



(a) Time



(b) Speed up

Figure 4.3: Comparison of our proposed method (pSBMP-PRM and pSBMP-RRT) to two existing approaches: pPRM and pSRT

will observe that our proposed method (pSBMP-PRM and pSBMP-RRT) achieves good scalability compared to the existing methods. For this particular experiment, we stopped at a processor count of 16 because the two existing algorithms (the pPRM in particular) could no longer scale beyond 16 processor counts. The existing algorithms are limited in scalability primarily because of the inherent interprocessor communication overhead they incurred.

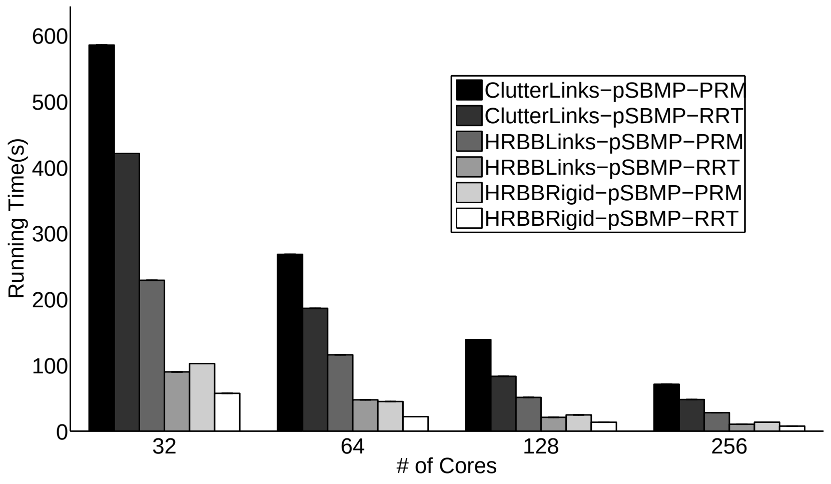
4.5.4.2 Effects of Different Environments and Machine Architectures

We conducted further experiments in order to observe how our method would perform in different environments and machine architectures. Even though these problems exhibit different levels of difficulty and homogeneity leading to differences in running time, we observe that their relative performances are still similar.

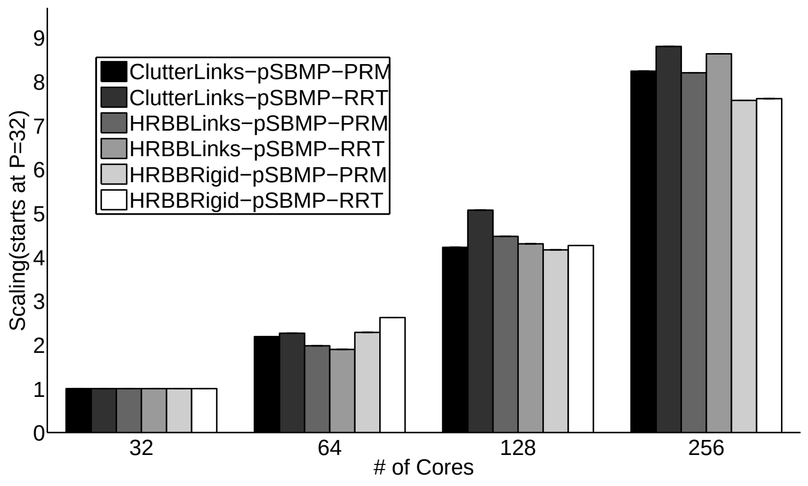
Figure 4.4 shows both the timing and scalability results for three different motion planning problems. The first problem is the cluttered environment with an articulated linkage robot (ClutterLinks), the second is the building environment with an articulated linkage robot (HRBBLinks), and the third is the building environment with a rigid body robot (HRBBRigid). We observe that the more difficult the problem, the better the scaling. The basic reason for this is that processors (cores) are fully engaged with computation which in some cases (if the algorithm and experiments are properly designed) lowered the overhead cost of idle or inter-processor communication.

We also observe that scalability improves with an increase in sample size. For the same reason as with problem difficulty, increasing sample size ensures that the processors are fully engaged with computation. Figure 4.5 shows results for varying sample size for the articulated linkage robot in a cluttered environment problem. This set of experiments was carried out on the Linux cluster with processor counts from 32 to 256.

To study scalability and test the limit of our method, we explore further experiments on a Cray XE6 petascale machine. In this experiment, we tested processor counts of 240, 480, 720, 960 and 1200. The results are shown in Figure 4.6. We observe that scalability is still possible on a massively parallel machine such as the Cray XE6. The results also suggest that, to the extent possible, our proposed method is independent of machine architecture. Thus, though there may be variance in results, we still expect to see similar performance and scalability across different platforms.

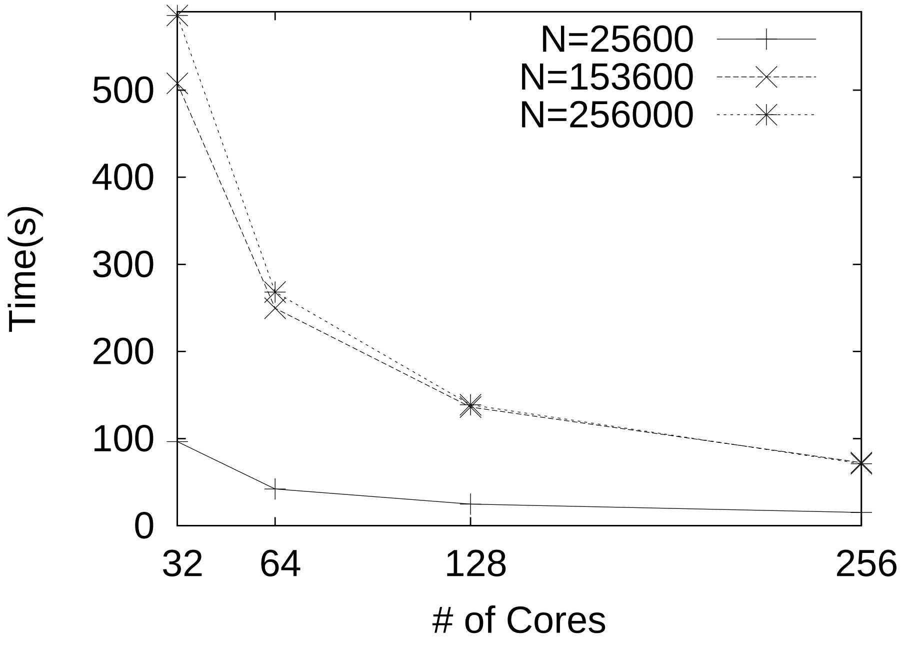


(a) Time

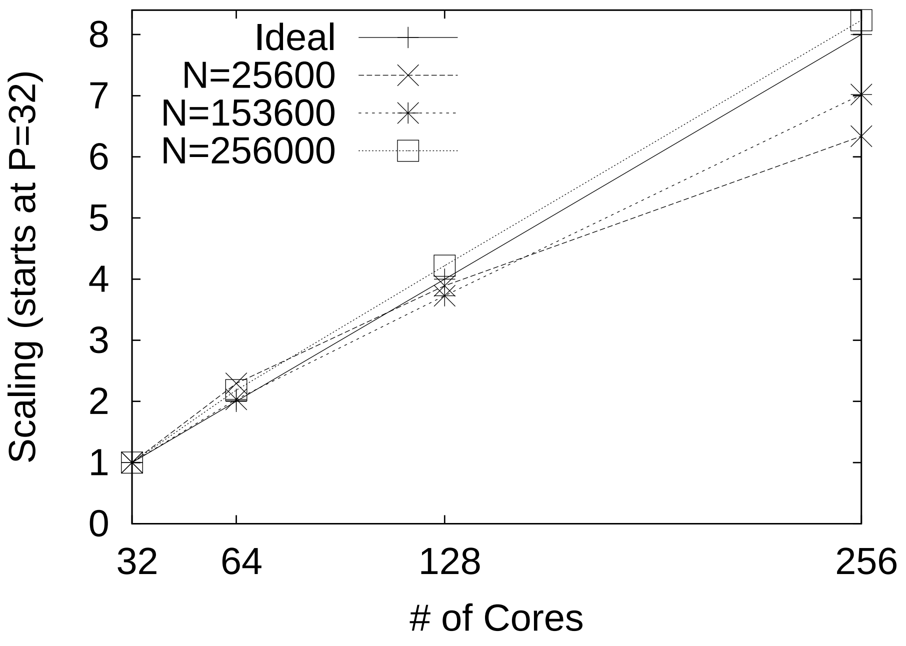


(b) Scaling (Processor counts at P= 32, 64, 128, 256)

Figure 4.4: Results from three different motion planning problems on Linux cluster using pSMBP-PRM and pSMBP-RRT methods

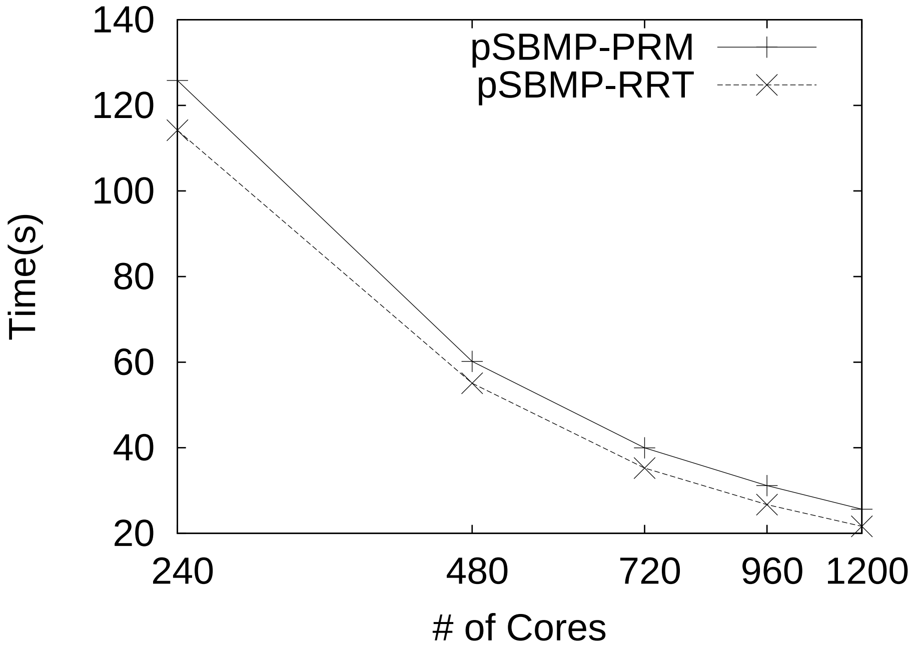


(a) Time

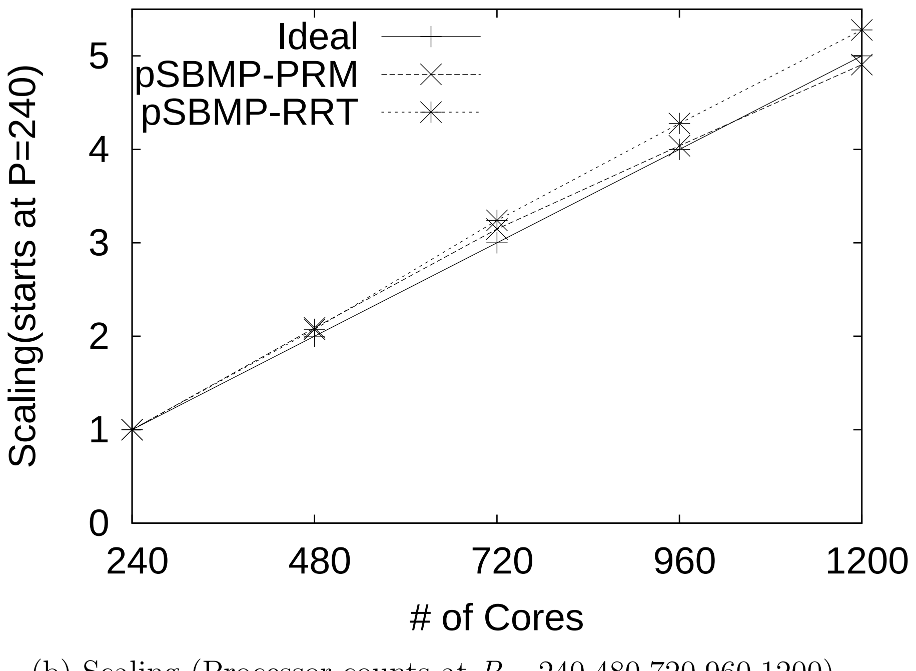


(b) Scaling (Processor counts at $P=$ 32, 64, 128, 256)

Figure 4.5: Results from varying input size for the articulated linkage robot in a cluttered environment using pSMBP-PRM method



(a) Time



(b) Scaling (Processor counts at $P=$ 240,480,720,960,1200)

Figure 4.6: Higher processor counts on Cray XE6 petascale machine

5. TREE-BASED PARALLEL MOTION PLANNING[*]

Inspired by the growth nature of RRT, in this chapter, we discuss a novel parallel and distributed RRT algorithm (Radial RRT). Radial RRTradially subdivides the $\mathbb{C}_{space}$ into regions, constructs a portion of the tree in each region in parallel, and connects the subtrees, removing cycles if they exist. Unlike the spatial subdivision discussed in Chapter 4, the radial subdivision method discuss in this chapter is well suited for tree-based motion planning algorithm that (radially) grows a tree starting from a single root whereas the previous method builds a tree of multiple roots.

We present a novel *radial* subdivision for parallelization that is especially suited for RRTs. Starting from the root q_{root}, we subdivide $\mathbb{C}_{space}$ into conical regions and build part of the tree (subtrees) in each region. These subtrees are later connected in a manner such that no cycle exists after region connection. We exploit locality by only attempting to connect branches that reside in neighboring regions. Figure 5.1 shows an example for a two dimensional $\mathbb{C}_{space}$. Each process builds a branch (shown in different colors) starting at the root that is biased toward their region of $\mathbb{C}_{space}$.

5.1 Space Subdivision and Region Graph Construction

Algorithm 7 describes the $\mathbb{C}_{space}$ subdivision-based RRT computation in detail. Region construction first creates a hypersphere S^d in d-dimensional $\mathbb{C}_{space}$ centered at $q_{root} \in R^d$ with radius r. We generate N_r random points at distance r from q_{root}. Each point q_i defines a conical region centered around the ray $\overrightarrow{q_{root}q_i}$. We construct a region graph $G(V, E)$ where each vertex v_i represents a region defined by q_i and an

[*]Part of the data reported in this chapter is reprinted with the kind permission of IEEE from "A Scalable Distributed RRT for Motion Planning " in *Proc. IEEE Int. Conf. Robot. Autom. (ICRA)* by S. A. Jacobs, N. Stradford, C. Rodriguez, S. Thomas, and N. M. Amato, 2013. Copyright ©2013, IEEE. [29]

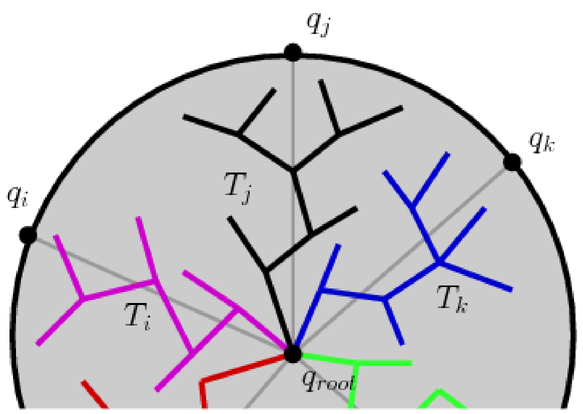


Figure 5.1: Example of radial subdivision for a 2D $\mathbb{C}_{space}$. Each process concurrently builds a branch (using sequential RRT) rooted at q_r and biased toward a target q_i (e.g., q_n for the black process).

edge (v_i, v_j) is added if q_j is one of the $k - closest$ neighbors of q_i. Thus, the edges in the region graph encode the neighborhood information between regions.

5.2 Constructing Regional Subtrees

After region graph construction, we independently (in parallel) run sequential RRT in each region. The RRT construction is done in a way that the tree is biased toward the region target q_i. Each region is centered around the random ray $\overrightarrow{q_{root}, q_i}$. Some overlap between regions is allowed so subtrees can explore part of the space in adjacent regions, enabling easier connection between subtrees in the next phase.

5.3 Connecting Regional Subtrees

Using the adjacency information provided by the region graph, we make connection attempts between each region branch and its adjacent neighbors. We check if any edge connection at this point creates a cycle. If a cycle exists, we prune the tree so as to remove any cycles. In the results presented here, tree pruning is performed by running a graph search algorithm. Figure 5.2 shows a simple pictorial illustration

Algorithm 7 Radial Subdivision Distributed RRT

```
Input: An environment $env$, a root $q_{root}$, the number of nodes $N$, a stpdfize $\Delta q$, the
    number of processes $p$, the number of regions $N_r$, a region radius $r$, the number
    of adjacent regions $k$
Output: A tree $T$ containing $N$ nodes rooted at $q_{root}$
 1: $Q_{N_r} \leftarrow$ generate $N_r$ random points of $r$ distance from $q_{root}$
 2: Initialize region graph $G(V, E)$ with $V \leftarrow Q_{N_r}$ and $E \leftarrow \emptyset$
 3: for all $q_i \in Q_{N_r}$ par do
 4:     $neighbors \leftarrow$ FindNeighbors$(G, q_i, k)$
 5:     for all $n \in neighbors$ do
 6:         G.AddEdge$(q_i, n)$
 7:     end for
 8: end for
 9: for all $v_i \in V$ par do
10:     $T \leftarrow$ ConstructBiasedRRT$(env, q_{root}, N/p, \Delta q, q_i)$
11: end for
12: for all $(v_i, v_j) \in E$ par do
13:     ConnectTree$(T, v_i, v_j)$
14:     if Cycle$(T)$ then
15:         Prune$(T)$
16:     end if
17: end for
18: return $T$
```

for tree pruning.

5.4 Algorithm Analysis

The complexity analysis of the parallel algorithms for radial subdivision RRT can be broken down into the following phases: the region construction phase, the regional radial RRT construction phase, the region connection phase, and removal of cycles phase. The overall time complexity of the algorithm can be described in terms of these phases as:

$$T = T_d(Env, n_r, d) + T_r(i)|V_R| + T_c(i, j)|E_R| + T_{cycle} \tag{5.1}$$

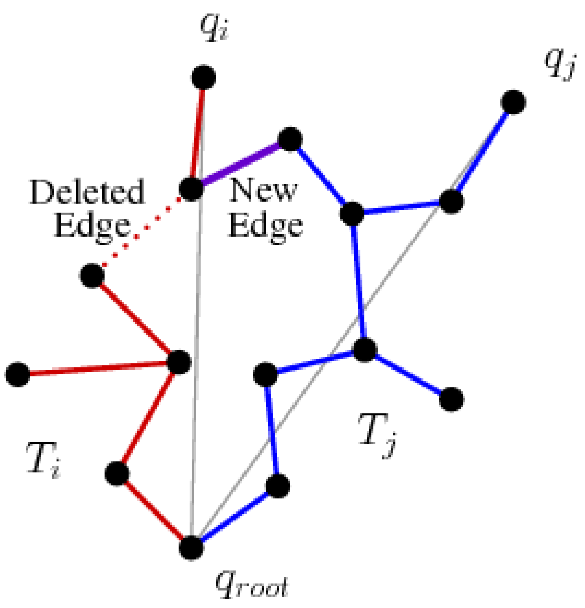


Figure 5.2: Tree pruning example, the new edge (purple) between the red and blue branches causes a cycle in the red branch, the dashed edge is identified for removal.

where the total time T is the sum of the cost T_d of region graph $G(V_R, E_R)$ construction for a given environment Env subdivided into n_r regions with each region having d neighbors, the cost T_r of constructing sequential Radial RRTs in region r_i, for all $v_i \ \epsilon \ V_R$, the cost T_c of connecting neighboring subtrees between adjacent regions r_i and r_j, for all $(r_i, r_j) \ \epsilon \ E_R$, and the cost T_{cycle} of removing cycle that may exist after region connection. In our analysis, p refers to the number of parallel processing elements (processors), we assume there as many regions as number of processors. In other words, n_r is some constant factors of p and $n_r \leq p$. Please note that our analysis assumes a uniform cost of constructing subtrees in each region; this assumption may fail in a situation where the regions are non-uniform.

In the first phase, we construct the region graph of n_r vertices and dn_r edges. The dominant factor in constructing the region graph is the d-nearest neighbor search, with $O(n_r^2 \log d)$ complexity assuming a brute force search. Each processor p will generate $\frac{n_r}{p}$ regions. So in parallel constructing the dn_r edges will take $O(\frac{n_r^2 logd}{p})$ time.

The second phase of the radial subdivision RRT parallel algorithm involves radial RRT construction in each region.Time complexity of the operation at this phase can be computed from our understanding of $O(N^2)$ complexity of sequential RRT algorithm [32]. From phase one we know there are n_r regions. If we assume a uniform distribution of work and that n_r is a constant factor of p, a subtree of size $c * N/p$ is expected from each region, this size is equivalent of N/p where N is the expected overall tree size and c is the constant factor relating n_r to p. With this assumption, the expected cost of constructing subtree in each region given p processing elements is $O((N/p)^2)$. Similar to region construction in the first phase, the expected dominant factor in constructing the regional subtree as the size of the tree grows asymptotically is the nearest neighbor search. Also note that we assume a brute force nearest neighbor search, recognizing the fact that the complexity can be reduced with approximate nearest neighbor data structure such as kd-tree.

Similar to the graph-based method presented in the previous chapter, inter-processor communication occurs when connecting regional subtrees. However, this communication is managed using the region graph. The region graph limits the communication to adjacent regions. The worst case scenario in region connection is a naive connection attempts between every node in the source regional subtree to every node in the target region subtree. If we assume this naive approach, the expected computational cost will be $O((N/p)^2)$ plus the cost of communication among the processing elements.

The last phase of the radial subdivision RRT parallel algorithm is in removing cycle that may exist as a result of region connection. To remove the cycle, we compute a bread-first-search (BFS) of the resulting roadmap/tree and then remove edges that are not in the BFS tree resulting in a computational complexity of $O((N+m)/p) + O(m/p)$ where N is as previously defined (e.g., the expected overall tree size) and m are the number of edges in the roadmap/tree prior to cycle removal.

We assume that $N >> n_r$, so, the final time, work, and space complexity of Radial RRT can be given as $O((N/p)^2) + O((N+m)/p) + O(m/p)$, $O((N^2)/p) + O(N+m) + O(m)$, and $O(N)$ respectively.

5.5 Experimental Evaluation

In this section, we evaluate the performance of radial subdivision distributed RRT (radial RRT) comparing the experimental results to an existing distributed RRT algorithm. We demonstrate that radial RRT achieves more scalable performance than the existing parallel algorithm. We present results from two different parallel machines for two different motion planning problems.

5.5.1 Bulk Synchronous Distributed RRT

For the primary purpose of evaluation and comparison with our proposed method, we implement and extend the distributed RRT algorithm presented in [24] in two ways. First, in order to optimize the use of space and memory, each process does not maintain a copy of the tree. Instead, they all have shared access to the tree which is stored in a global, distributed data structure. Requests to access an element on another process are sent and received through the global identifier (GID) assigned to the element. Second, we regulate inter-processor communication by introducing a variable m that controls how much expansion will be done before a global update and broadcast. Setting $m = 1$ gives the same computational pattern as in [24].

Algorithm 8 describes bulk synchronous distributed RRT. We first initialize the tree T with the root node q_{root}. Subsequently, each process locally (in parallel) samples m nodes and finds its nearest node q_{near} in the tree. If the expansion

q_{near} toward q_{rand} is successful, then the pair (q_{new}, q_{near}) is added to a temporary container N_m. After m steps, the global tree is updated. This process continues until the termination condition is met. Figure 5.3 shows a simple illustration of bulk synchronous distributed RRT computation in which p=2, m=2 and N=8.

Algorithm 8 Bulk Synchronous Distributed RRT

```
Input: An environment $env$, a root $q_{root}$, the number of nodes $N$, a stpdfize $\Delta q$, the
    number of processes $p$, the number of local expansion stpdf $m$
Output: A tree $T$ containing $N$ nodes rooted at $q_{root}$
 1: $T$.AddNode($q_{root}$)
 2: for all proc $p \in P$ par do
 3:   $i \leftarrow 0$
 4:   while $i < N/p$ do
 5:     localContainer $N_m$
 6:     for $j = 1 \ldots m$ do
 7:       $q_{rand} \leftarrow$ GetRandomNode($env$)
 8:       $q_{near} \leftarrow$ FindNeighbor($T, q_{rand}, 1$)
 9:       $q_{new} \leftarrow$ Extend($q_{near}, q_{rand}, \Delta q$)
10:       if !TooSimilar($q_{near}, q_{new}$) $\wedge$ IsValid($q_{new}$) then
11:         $N_m$.Insert($q_{near}$, $q_{new}$)
12:       end if
13:     end for
14:     for all node pair $n \in N_m$ do
15:       $T$.AddNode($n.q_{new}$)
16:       $T$.AddEdge($n.q_{near}, n.q_{new}$)
17:       $i \leftarrow i + 1$
18:     end for
19:   end while
20: end for
21: return $T$
```

5.5.2 Parallelizing Nearest Neighbor Search

There is a clear need for fine-grained parallelism in sampling-based motion planning [23, 24]. The nearest neighbor search is considered a key bottleneck to scalable

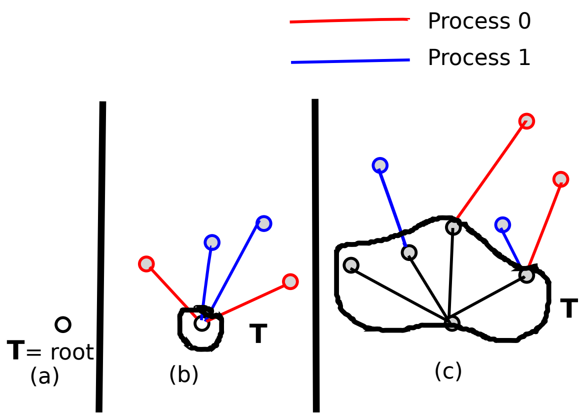


Figure 5.3: Bulk synchronous distributed RRT. (a) T is initialized to root, (b) The first iteration with m=2, (c) The second iteration where globally communicated data is shown in black.

performance. In this work, we implement and incorporate a nested and fine-grained parallel computation of nearest neighbor search within the radial subdivision distributed RRT and bulk synchronous distributed RRT algorithms described earlier. Our implementation has a *map reduce* parallel computation pattern [85].

Algorithm 9 describes the approach in the context of a distributed RRT. To compute the nearest point q_{near} to a query point q_{rand}, each processing element sends q_{rand} to the other processing elements by calling $MapReduce()$. The mapping function (Algorithm 10) receives the query point q_{rand} and locally computes its nearest neighbor in its local portion of the tree (T_p) based on a given distance metric. The reduce function (Algorithm 11) takes the two inputs returned by the mapping function and computes the nearest neighbor to q_{rand} from the two inputs based on the same distance metric.

5.5.3 Machine Architecture

The same parallel machines as presented in Chapter 4 (the Linux cluster and the Cray XE6 machines) were used for the experiments in this chapter. Each node of the

Algorithm 9 Parallel NNS Distributed RRT

Input: An environment env, a root q_{root}, the number of nodes N, a stpdfize Δq,the number of processes p

Output: A tree T containing N nodes rooted at q_{root}

```
 1: $T$.AddNode($q_{root}$)
 2: for all proc $p \in P$ par do
 3:   $i \leftarrow 0$
 4:   while $i < N/p$ do
 5:     subtree $T_p \in T$
 6:     $q_{rand} \leftarrow$ GetRandomNode($env$)
 7:     $q_{near} \leftarrow$ MapReduce($Map(T_p, q_{rand})$,
        Reduce($q_{near}, q_{near}$))
 8:     $q_{new} \leftarrow$ Extend($q_{near}, q_{rand}, \Delta q$)
 9:     if !TooSimilar($q_{near}, q_{new}$) $\wedge$ IsValid($q_{new}$) then
10:       T.AddNodeToTree($q_{new}$)
11:       T.AddEdgeToTree($q_{near}, q_{new}$)
12:     end if
13:     $i \leftarrow i + 1$
14:   end while
15: end for
16: return $T$
```

Algorithm 10 Map

Input: A set of points S, a query q

Output: A map of closest point to q and its distance M

```
1: $M \leftarrow$ FindNeighbors($S, q, 1$)
2: return $M$
```

Algorithm 11 Reduce

Input: Two maps $M1$ and $M2$ of points and their distances to a query q

Output: The closest point $p \in M1 \cup M2$

```
1: if $M1.distance \leq M2.distance$ then
2:   $p \leftarrow M1.point$
3: else
4:   $p \leftarrow M2.point$
5: end if
6: return $p$
```

Linux cluster is made of 8 processor cores, thus, for this machine we present results for processor counts in multiples of 8. Each node of the Cray XE6 machine consists 12 processor cores. This architectural layout also influenced our choice of processor counts to be in multiple of 12. Our code was written in C++ and compiled with gcc-4.5.2 on the Linux cluster and gcc-4.6.3 on the Cray XE6 machine. Using STAPL, the same C++ code was used on both architecture types.

5.5.4 Motion Planning Problems

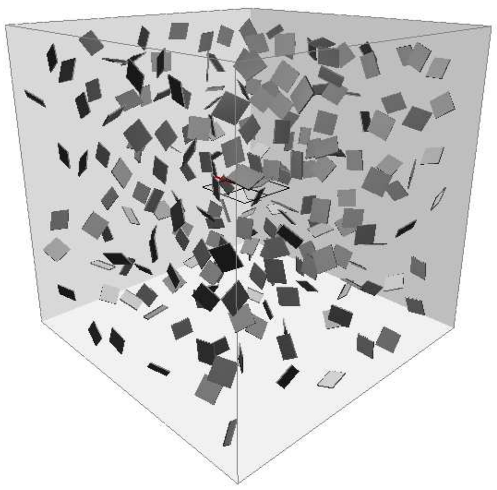
We studied three different kinds of environments: a 512×512×512 uniformly cluttered environment (shown in Figure 5.4(a)) and a 7x7x7 grid environments (shown in Figure 5.4(b)) and another clutter environment with strip-like obstacles (shown in Figure 5.4(c)). There are 216 obstacles each of size $2 \times 4 \times 4$ uniformly scattered in the clutter environment. The grid environment has eight obstacles placed in a grid form. We studied two different kinds of robot types: a $4 \times 4 \times 4$ units 6 DOF cube-like rigid body robot and an eleven-link (16 DOF) articulated linkage robot, with each link having dimensions of $7 \times 1 \times 1$ units.

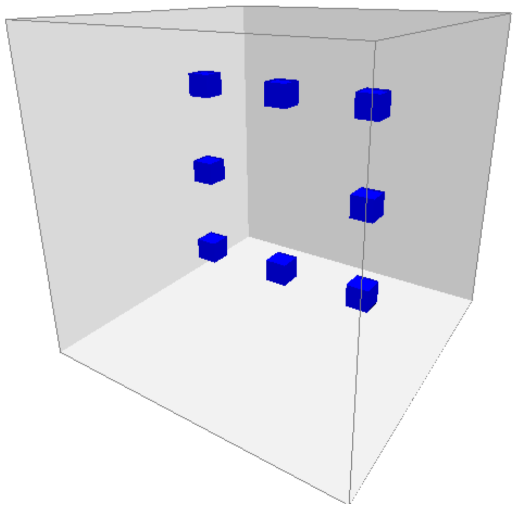
5.5.5 Experimental Results

5.5.5.1 Bulk Synchronous Effect

We first study the effect of the m parameter introduced in Algorithm 8 to tune the amount of local expansion done before a global update. We fixed the sample size at 16,384 and used $m = \{1, 16, 64\}$. Note that $m = 1$ is the same as the distributed algorithm presented in [24]. Figure 5.5 shows the running time as a function of the number of processors on the Linux cluster for the rigid body robot up to 256 processors.

Localizing the computation and thus minimizing frequent inter-processor communication by varying m does impact performance of distributed RRT, but this effect


(a) Clutter


(b) Grid

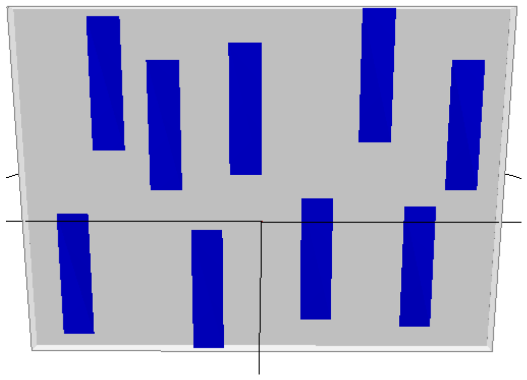

(b) Stripline

Figure 5.4: Environments studied for tree-based method

is not obvious until higher processor counts, see Figure 5.5(b). In fact, $m = 1$ seems to outperform the others until around $p = 16$.

5.5.5.2 Radial Subdivision Scalability Study

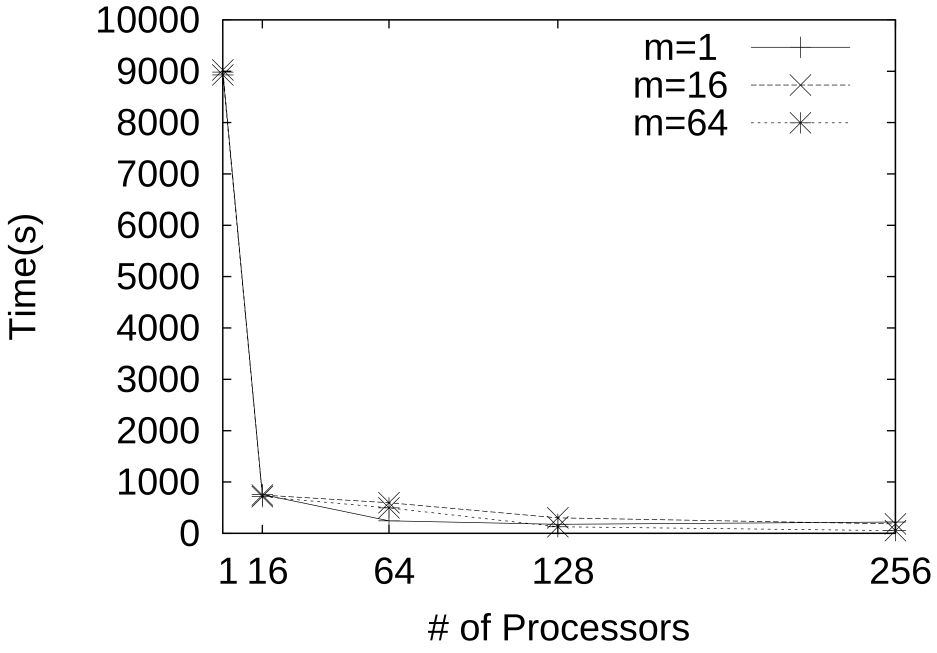
As seen with the bulk synchronous distributed RRT, localizing computation reduces communication overhead which in turn improves the overall scalability of the algorithm. We now look at the scalability of radial subdivision distributed RRT on the two different robots: the 6 DOF rigid body and the 16 DOF articulated linkage. Figure 5.6 shows performance result on the Linux cluster up to 64 processors. Radial subdivision RRT was able to achieve almost near linear speedups for both robot types.

5.5.5.3 Effect of Machine Architecture

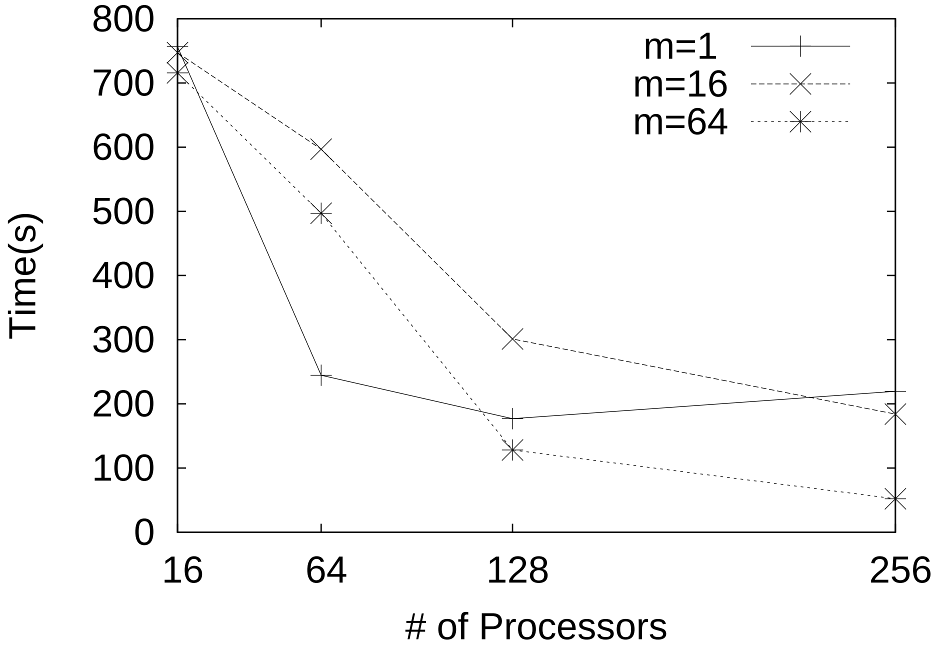
We next study how the machine architecture impacts performance for both the bulk synchronous distributed RRT and the radial subdivision distributed RRT. For the bulk synchronous distributed RRT we use $m = \{1, 25, 50\}$ while keeping the sample size constant. Figure 5.7 shows performance results for the rigid body robot on the Cray XE6 machine. Radial subdivision distributed RRT scales almost linearly, similar to what was observed on Linux cluster. Scalability of the bulk synchronous distributed RRT depends on the value of m and the number of processors. As in the previous experiments (Figure 5.5), the impact of increasing m is much felt at higher processor counts at which inter-processor communication become significant.

5.5.5.4 Grid Environment

To further understand the performance of radial subdivision in a different scenario, we evaluated the radial subdivision algorithm in a grid environment with rigid body robot on Cray XE6 machine. In this evaluation, we kept the number of regions

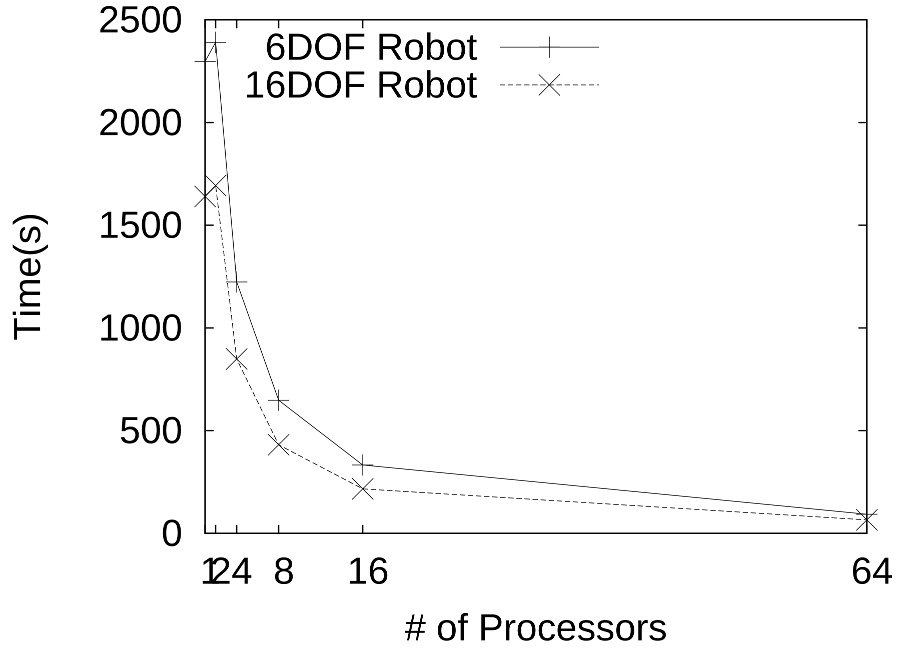


(a) X-axis starting from $p = 1$

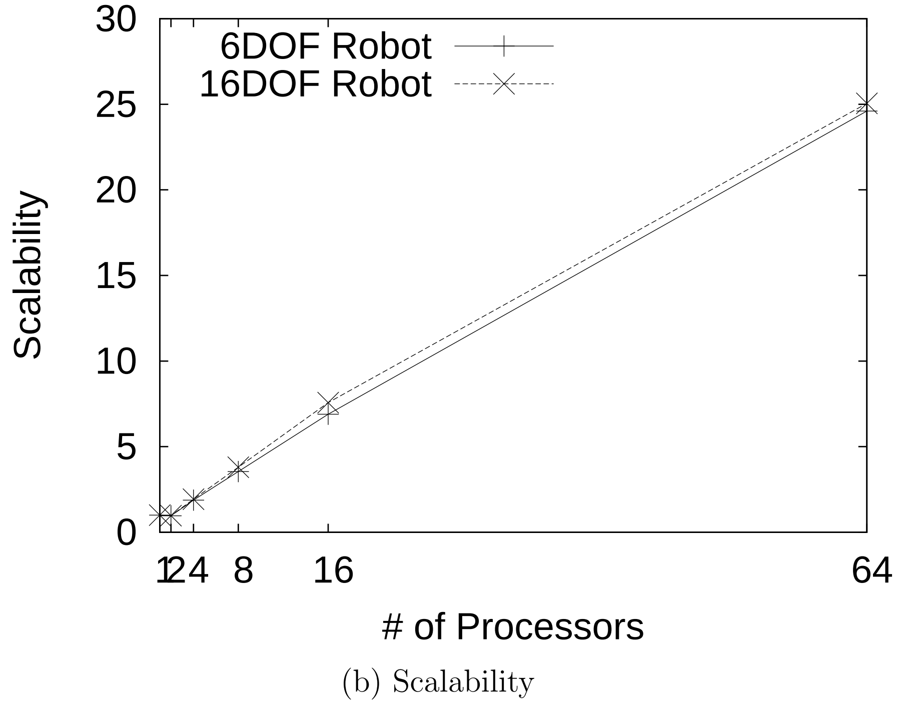


(b) X-axis starting from $p = 16$

Figure 5.5: Effect of varying m in the bulk synchronous distributed RRT.



(a) Time



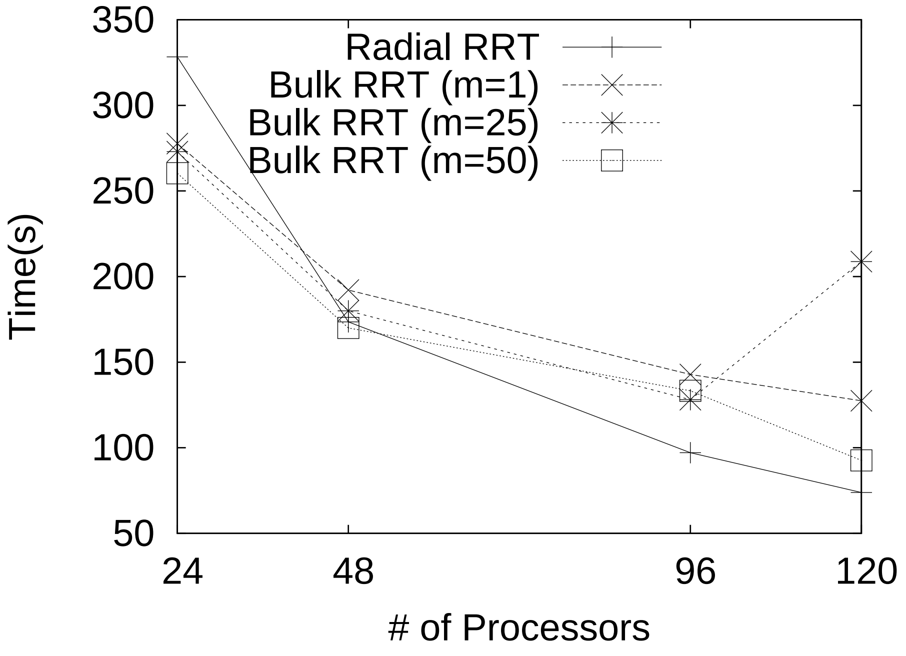
(b) Scalability

Figure 5.6: Radial subdivision distributed RRT performance on Linux cluster.

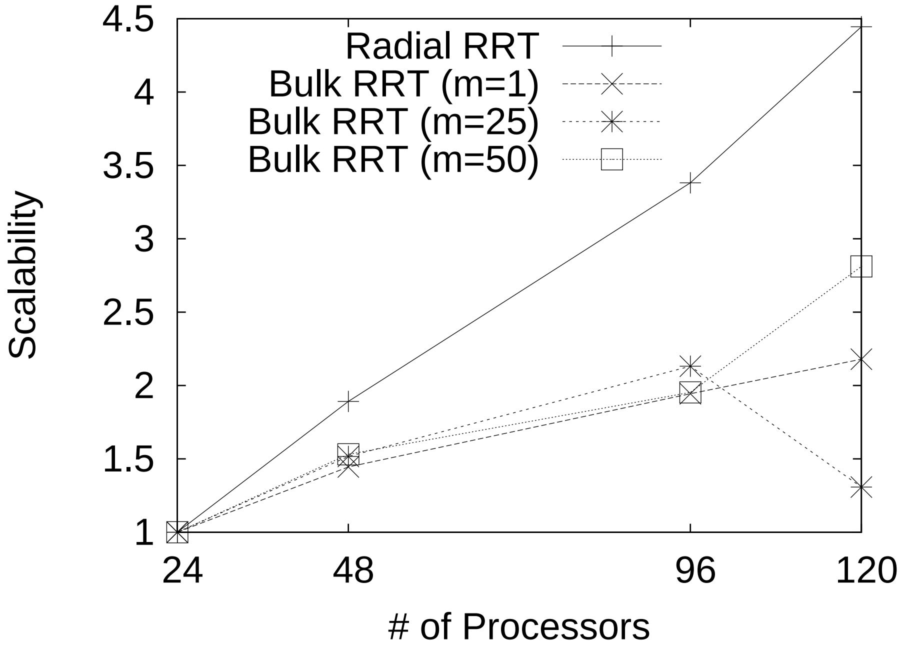
constant at 480 across all processor count and varied the sample size per region. The results from the evaluation are shown in Figure 5.8. Given different input sizes, we saw decrease in execution time as the number of processors increases.

5.5.5.5 Stripline Environment

We conduct another experiment using the stripline environment. In this environment, we varied the ammount of $\mathbb{C}_{free}$ volume by varying the obstacles sizes. We fixed the samples sizes at 4096 per region for 256 regions and varied the processor count from 8 to 256. This experiment was conducted on Linux cluster and the results are shown in Figure 5.9. We observed almost linear scalability in all cases.



(a) Time



(b) Scalability

Figure 5.7: Distributed RRT performance on Cray XE6 machine.

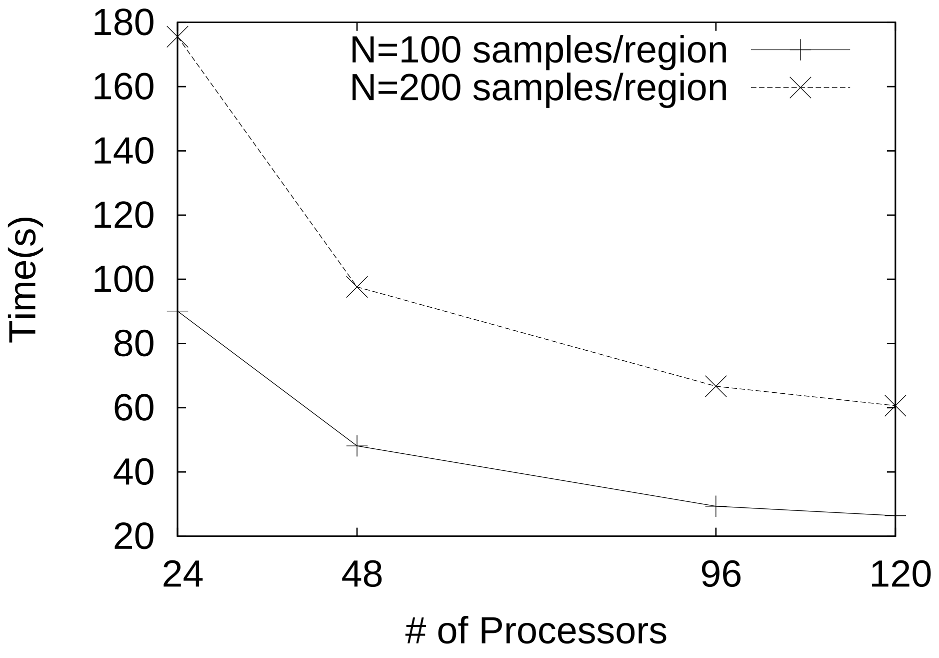


Figure 5.8: Radial RRT performance results for grid environment on Cray XE6 machine

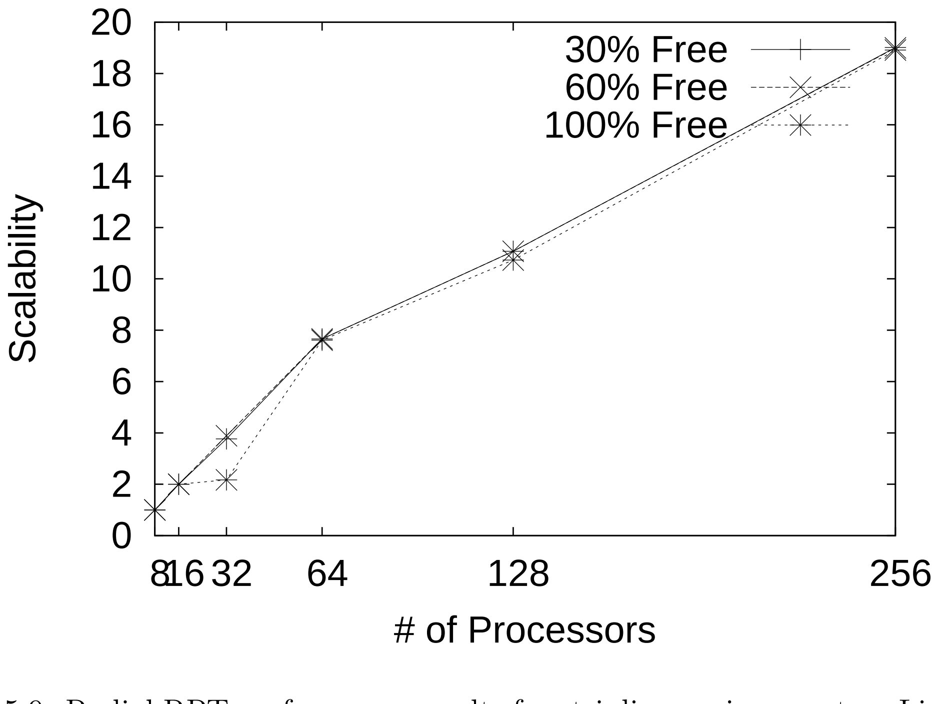


Figure 5.9: Radial RRT performance results for stripline environment on Linux cluster

6. RADIAL BLIND RRT*

The radial distributed RRT algorithm presented in Chapter 5 does not work efficiently for all problem instances. As an example, if an obstacle completely blocks RRT growth in a region, the free planning space that is beyond the blockage will not be covered and thus planning problems cannot always be solved. In this chapter, we extend the idea of radial subdivision and develop a new algorithm, Radial Blind RRT [35]. Radial Blind RRT ignores obstacles during initial growth to efficiently explore the entire space. By ignoring obstacles, Radial Blind RRT explores the space efficiently while keeping track of feasible paths. It later merges parts of the tree that may have become disconnected from the root by using RRT-Connect [56].

We start our discussions with Blind RRT — a novel sequential motion planning algorithm — the idea on which Radial Blind RRT is built. The sequential Blind RRT will serve as a subroutine for the parallel Radial Blind RRT algorithm.

6.1 Blind RRT

In this section, we describe the design, motivation and advantages of Blind RRT compared to the standard RRT. Although used in this work to improve Radial RRT, we present Blind RRT as a probabilistically complete strategy for motion planning, capable of solving problems independently of parallel computation. The motivation behind Blind RRT is the incapability of expansion for Radial RRT when an obstacle completely blocks progress in a region. Therefore, we propose to ignore obstacles, or blindly expand through them. Blind RRT takes advantage of the rapid expansion

*Part of the data reported in this chapter is reprinted with permission from "Blind RRT: A Probabilistically Complete Distributed RRT " in *Proc. IEEE/ Int. Conf. Intel. Rob. Syst. (IROS)* by C. Rodriguez, J. Denny, S. A. Jacobs, S. Thomas, and N. M. Amato, 2013. Copyright ©2013, IEEE. [35]

rate of RRTs, i.e., growing towards unexplored areas of $\mathbb{C}_{space}$.

6.1.1 Algorithm

The Blind RRT strategy, shown in Algorithm 12, starts by iteratively expanding a tree τ rooted at a configuration q_{root}, similar to RRT. We alter the standard RRT `Expand` subroutine to continue growing through obstacles recording a set of configurations Q_{new} that occur during an expansion step. These witnesses are added to τ in the `Update` function. If valid edges exist between successive nodes in Q_{new}, these edges are added as well. Note that at this point, the Blind RRT has the same nodes as an RRT in an obstacle free environment and the RRT edges that are valid considering obstacles. After performing N_{br} Blind RRT expansion iterations, Blind RRT deletes all invalid nodes in τ and performs a connection phase that attempts to connect the various connected components (CCs). Following this, all CCs other than the CC containing the root are deleted, and τ is returned.

Algorithm 12 Blind RRT

Input: A root configuration q_{root}, the initial number of nodes N_{br}, a maximum expanding distance Δq

Output: A tree τ containing N_{br} nodes rooted at q_{root}

```
1: $\tau \leftarrow \{q_{root}\}$
2: for $n = 1 \ldots N_{br}$ do
3:     $q_{rand} \leftarrow$ RandomNode()
4:     $q_{near} \leftarrow$ NearestNeighbor($\tau, q_{rand}$)
5:     $Q_{new} \leftarrow$ Expand($q_{near}, q_{rand}, \Delta q$)
6:     $\tau$.Update($q_{near}, Q_{new}$)
7: end for
8: $\tau$.DeleteInvalidNodes()
9: ConnectCCs($\tau$)
10: $\tau$.DeleteInvalidCCs()
11: return $\tau$
```

Note that one benefit of the standard RRT algorithm is early termination if coarse coverage is sufficient to solve the query. This can be achieved here by interleaving tree construction and evaluation and setting N_{br} appropriately.

6.1.1.1 Blind Tree Expansion

We describe two alternatives when performing blind expansion. Note that other RRT expansion algorithms could be modified and used appropriately. The first performs validity checking for the entire line from q_{near} to q_{new} (either at a distance Δq from q_{near} towards q_{rand} or q_{rand} itself, whichever is closer), collecting nodes that are valid along the boundary of $\mathbb{C}_{obstacle}$ (Figure 6.1(b)). It adds all the nodes collected as well as q_{new} (which itself may or may not be valid). The second stops at the first validity change, records the valid node, and directly jumps to q_{new} and adds it (Figure 6.1(c)). The latter skips part of the collision detection, while the former keeps track of more valid nodes. It is important to note that nodes contained in $\mathbb{C}_{obstacle}$ may be added to the tree if they are found Δq away from q_{near}, but only edges between valid configurations are added to the tree.

6.1.1.2 Connected Component Connection

At the end of the first step, any obstacles found along the expansion may have caused parts of τ to become disconnected from the root, yielding multiple CCs in τ. However, we would like to only have one CC in τ that contains the root. For this, we join pairs of CCs, CC_a and CC_b, using RRT-Connect [56] where $\tau_a = CC_a$ and $\tau_b = CC_b$. In the connection step, we first choose CC_a as a random CC, and then choose a target CC, CC_b, by the CC whose centroid is closest to the centroid of CC_a. A nearest neighbor query is performed from the centroid of CC_a to the centroids of the other CCs. It significantly reduces the nearest neighbor computation, as the number of CCs is much less than the number of nodes in the tree. This is used

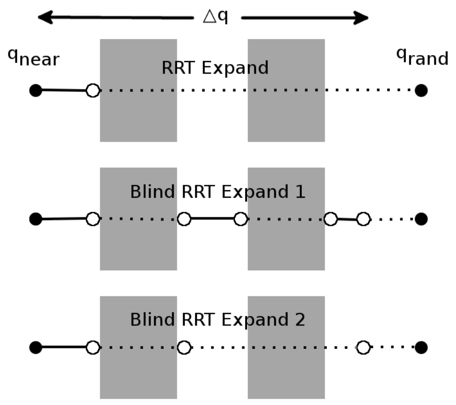


Figure 6.1: RRT expands greedily up to Δq, q_{rand}, or an obstacle is hit (a) Blind RRT Expand always expands up to Δq distance or q_{rand} while also retaining either all free witnesses (b) or only the first free witness (c) to return a set of expansion nodes Q_{new}.

as an approximation scheme in selecting the closest CC. Component connection iteratively selects CCs to connect to until either one CC is achieved or a maximum number of failures is reached. After the CC connection phase we retain only the component containing the root.

6.1.2 Probabilistic Completeness

Probabilistic completeness is a desirable property of randomized planners which describes their ability to find a solution path, assuming one exists, as the number of samples tends to infinity. In this section, we describe and prove the probabilistic completeness of Blind RRT. We assume that the $\mathbb{C}_{space}$ is ϵ-good [86] for some $\epsilon > 0$.

Theorem 1. *Blind RRT is probabilistically complete.*

Proof. Given any two configurations q_s and q_g in the same connected component of

$\mathbb{C}_{free}$, a path exists between q_s and q_g. If no obstacles are present in the environment, i.e., $\mathbb{C}_{free} \equiv \mathbb{C}_{space}$, then an RRT rooted at q_s will reach within ϵ of q_g after n_0 fixed step expansions of distance Δq, (i.e., a path exists in the tree between q_s and q_g), n_0 Blind RRT expansions are also sufficient to reach within ϵ of q_g. This is due to the fact that Blind RRT explores $\mathbb{C}_{space}$ identically to an RRT grown in the absence of obstacles because Blind RRT expansions ignore $\mathbb{C}_{obstacle}$.

After Blind RRT removes invalid nodes of the tree, q_g exists in some component of the tree CC_g. If $CC_s \equiv CC_g$, where CC_s is the component of the tree containing q_s, then a path exists in the tree between q_s and q_g. If $CC_s \not\equiv CC_g$, then Blind RRT uses RRT-Connect to merge CC_g with CC_s. It follows from the probabilistic completeness of RRT-Connect [56] that Blind RRT will connect CC_g to CC_s to yield a valid path between q_s and q_g. $\square$

6.2 An Improved Radial RRT using Blind RRT

In this section, we introduce an improved Radial RRT framework for parallelizing RRTs which uses Blind RRT as a subroutine.

6.2.1 Algorithm

Radial Blind RRT starts by radially subdividing $\mathbb{C}_{space}$ as in Radial RRT presented in Chapter 5. It makes use of a region graph, which is an abstraction of the different subdivisions of $\mathbb{C}_{space}$. To subdivide $\mathbb{C}_{space}$ and construct the region graph, the algorithm randomly samples N_r points Q_{N_r} on a d-dimensional hypersphere of radius r centered at q_{root}, where d is the dimension of $\mathbb{C}_{space}$, r is a bound on the growth of the region, and q_{root} is the root configuration of the RRT. Note that this applies to any dimension of $\mathbb{C}_{space}$. These samples that define the subdivision become the vertices of the region graph. A k-closest connection routine determines the adjacency of the regions defining the edges of the region graph. The number of neighbors a region has, and thus the communication later required, can be tuned by k.

Radial Blind RRT, shown in Algorithm 13, constructs a Blind RRT in parallel in each region. Each region constructs a tree of N_{br}/p nodes whose growth is bounded to the region, where N_{br} is input as the number of nodes for the tree and p is the number of processing elements. Most likely, each region will contain several CCs that need to be connected back to the root component. This takes place in a global region connection phase.

Algorithm 13 Radial Blind RRT

```
Input: A root configuration $q_{root}$, the number of nodes $N_{br}$, a maximum expansion
    distance $\Delta q$, the number of processors $p$, the number of regions $N_r$, a region
    radius $r$, the number of adjacent regions $k$
Output: A tree $\tau$ containing $N_{br}$ nodes rooted at $q_{root}$
1: $G_r(V, E) \leftarrow$ ConstructRegionGraph($N_r, r, k$)
2: for all $v_i \in V$ par do
3:     $\tau \leftarrow \tau \cup$ BlindRRT($q_{root}, N_{br}/p, \Delta q, v_i$)
4: end for
5: $\tau_{mst} \leftarrow$ MinimumSpanningTree($G_r(V, E)$)
6: for all $(v_i, v_j) \in \tau_{mst}$ par do
7:     ConnectRegions($v_i, v_j$)
8: end for
9: return $\tau$
```

The region connection phase, described in Algorithm 14, attempts to connect CCs from neighboring regions. The neighboring regions identified from the region graph allow for reducing the global communication between processing elements, thus improving scalability of the approach. Prior to the region connection phase, a minimum spanning tree of the region graph is computed so that no cycles are produced in the tree when connecting regions. Additionally, the minimum spanning tree provides information as to which neighbors are closest, and thus there is a higher probability of successful connection. To reduce the communication overhead in the region connection phase, we import all necessary information from the target region R_t, instead of updating the CC information every time a connection is performed. At the beginning, we know that none of the CCs in the source region R_s are connected to the CCs in R_t, so we initialize two sets: U the unconnected CCs and C the already merged CCs. Initially, the first contains all R_t CCs and the second is the empty set. P is a queue containing all the CCs in the source region, R_s. The goal is to merge U with P without creating cycles. First, we dequeue a CC, CC_{local} from P and iterate through the CCs in C, attempting connections and stopping if one is found. Then, we iterate through the CCs in U, attempting connections to all of them; if a connection is made, we update the sets C and U by adding CC_{local} to C and removing it from U. We perform this operation until P is empty. Note that connections between CCs from the same region are not explicitly attempted in this phase. They have already been attempted in the BlindRRT call for each region (see Algorithm 13, line 3). However, multiple CCs may connect to the same remote CC progressively merging the CCs into one. This procedure not only performs region connection with reduced communication overhead, but may also indirectly connect local CCs through the remote CCs. After this global CC connection step, we may or may not have connected all CCs of the overall tree back to the root component. Therefore, we remove all remaining CCs.

Figure 6.2 shows an example of the different steps of the parallel algorithm on a simple 2-D environment with $p = 4$ processors. Figure 6.2(a) shows the example environment with regions decomposed. Regions are represented by points (blue) on the outer sphere. Figure 6.2(b) shows a Radial Blind RRT expanded for $N_{br}/p = 20$ expansions. Notice how Radial Blind RRT ignores and expands through $\mathbb{C}_{obstacle}$

Algorithm 14 Connect Regions

```
Input: Two regions $R_s$ and $R_t$
1: Pending $CCs$ Queue $P \leftarrow R_s$.GetCCs()
2: Connected $CCs$ $C \leftarrow \emptyset$
3: Unconnected $CCs$ $U \leftarrow R_t$.GetCCs()
4: while $\neg P$.IsEmpty() do
5:   $CC_{local} \leftarrow P$.Dequeue()
6:   for all $CC_{remote} \in C$ do
7:     if RRT − Connect($CC_{local}, CC_{remote}$) then
8:       break
9:     end if
10:  end for
11:  for all $CC_{remote} \in U$ do
12:    if RRT − Connect($CC_{local}, CC_{remote}$) then
13:      $C = C \cup CC_{remote}$
14:      $U = U \backslash CC_{remote}$
15:    end if
16:  end for
17: end while
```

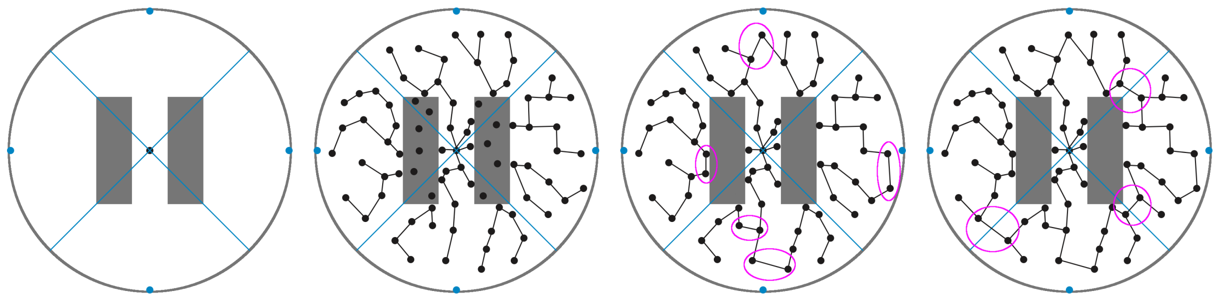
covering all of $\mathbb{C}_{space}$. Figure 6.2(c) shows the tree after local CC connection is performed. New edges are emphasized by magenta ellipses. Figure 6.2(d) shows the tree after global region connection. Again new edges are emphasized with magenta ellipses.

6.2.2 Probabilistic Completeness

In this section, we show two things: the probabilistic incompleteness of Radial RRT and the probabilistic completeness of Radial Blind RRT.

Observation 1. *Radial RRT is not probabilistically complete because an obstacle can entirely block exploration of a region, in such a way that connections between adjacent regions will not be able to cover $\mathbb{C}_{space}$, see Figure 6.3.*

Theorem 2. *Radial Blind RRT is probabilistically complete.*



(a) Region Decomposition (b) Blind RRT Expansion (c) Local CC Connection (d) Region Connection

Figure 6.2: (a) An example environment with four regions, represented by their points (blue) on the outer circle. (b) Radial Blind RRT concurrently expanding in the four regions ignoring obstacles as it goes. (c) Radial Blind RRT concurrently and locally removes invalid nodes of the tree and connects CCs within each region (new edges emphasized in magenta). (d) Radial Blind RRT connects CCs between regions yielding a final tree.

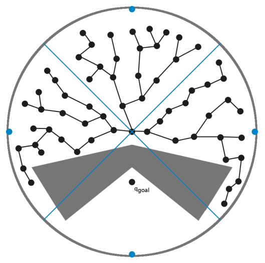


Figure 6.3: Example of Radial RRT not being able to solve an example query.

Proof. Without loss of generality assume $\mathbb{C}_{free}$ is a single connected component. Collectively the Blind RRTs built in each region will be able to expand and cover all of $\mathbb{C}_{space}$ in the initial expansion phase, for the reasons stated in the proof of Theorem 1. After the local connection phase, Radial Blind RRT recombines adjacent regions with RRT-Connect. By the probabilistic completeness of RRT-Connect [56], all regions will be merged and all components of the tree will be merged into one.

Thus, Radial Blind RRT is probabilistically complete. □

6.2.3 Algorithm Analysis

In this section, we present complexity analysis of Radial Blind RRT. Recall that the original RRT algorithm as presented in [32] requires $O(N^2)$ time (in the worst case) to construct a tree with N configurations. That analysis assumes a brute force strategy for nearest neighbor queries, as will our analysis. We note to the reader that more efficient mechanisms for nearest neighbor queries exist in the literature, e.g., KD-trees, but for simplicity of analysis we assume worst case computation.

The Radial Blind RRT given in Algorithm 13 can be broken down into four phases: region graph construction, Blind RRT construction, minimum spanning tree (MST) computation, and region connection. The overall time complexity of the algorithm can be described in terms of these four phases as:

$$T = T_{rg} + T_{BRRT} + T_{MST} + T_c$$

where the total time T is the sum of the cost T_{rg} of region graph construction for a given environment subdivided into n_r regions with each region having d neighbors, the cost T_{BRRT} of constructing n_r sequential Blind RRTs, the cost T_{MST} of computing an MST of the region graph, and the cost T_c of connecting neighboring CCs between adjacent regions given from the MST. In our analysis, p refers to the number of parallel processing elements. Please note that our formulation assumes a uniform cost of constructing subtrees in each region; this assumption may fail in a situation where the regions are non-uniform.

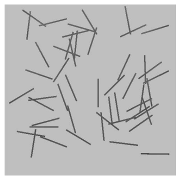
In the first phase, we construct the region graph of n_r vertices and dn_r edges. The dominant factor in constructing the region graph is the d-nearest neighbor search, with $O(n_r^2 \log d)$ complexity assuming a brute force search. Each processor p will generate $\frac{n_r}{p}$ regions. So in parallel constructing the dn_r edges will take $O(\frac{n_r^2 logd}{p})$ time.

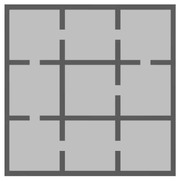
The second phase of the algorithm constructs a Blind RRT in each region of size $N_{br} = \frac{N}{n_r}$, where N is the expected number of nodes for tree construction. Since the complexity of sequential Blind RRT is equivalent to the complexity of RRT, the total work for this phase is $O((\frac{N}{n_r})^2)$. Assuming uniform distribution of work across the processing elements, the time complexity is $O((\frac{N}{n_r})^2/p)$.

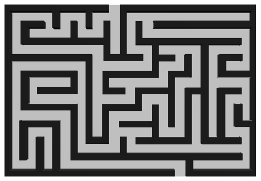
Most inter-processor communication occurs when connecting regional subtrees in phase four. However, this communication is managed by reducing the region graph to a MST in phase three, which will require a time complexity of $O(\frac{dn_r \log n_r}{p})$ [87]. Then, phase four will require $O(cn_r)$ instantiations of RRT-Connect, each requiring $O(N_{rrtc}^2)$ work, where c is the maximum number of CCs within a region and N_{rrtc} is the maximum number of nodes allotted per RRT-Connect tree. Upon parallelization, phase four requires $O(\frac{cn_r N_{rrtc}^2}{p})$ time.

We assume that $N >> n_r$ and $N_{rrtc} >> n_r$, so the final time complexity for Radial Blind RRT can be reduced to:

$$T = O((\frac{N/n_r}{p})^2) + O(\frac{cn_r N_{rrtc}^2}{p})$$


(a) 2-D Clutter


(b) 2-D Grid


(c) 2-D Maze

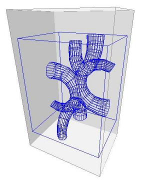

(d) 3-D Maze

Figure 6.4: Motion planning problems.

implying that the time spent per phase can vary based upon the success in covering the space with fewer CCs, i.e., lower connection time.

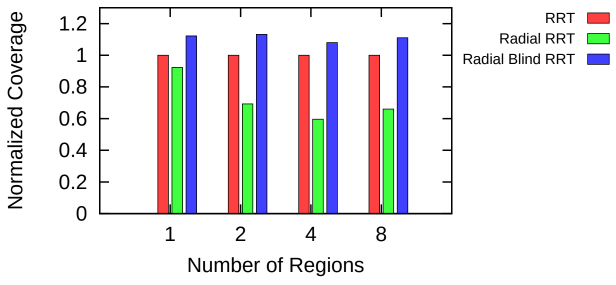
6.3 Experimental Evaluation

In this section, we analyze Radial Blind RRT under two different perspectives. We compare its effectiveness to that of sequential RRT and Radial RRT. Also, we present the scalability of the algorithm against Radial RRT. Section 6.3.2 compares the methods in a few environments showing the efficiency of Radial Blind RRT exploration, and Section 6.3.3 presents the performance of Radial Blind RRT with different processor counts. Recall, the goal of this algorithm is to have a scalable RRT useful for motion planning. Standard parallel RRT methods do not scale well, whereas Radial RRT does. However, Radial RRT is unable to cover the planning space as well as RRT. Thus, the goal of this section is to show that Radial Blind RRT allows both scalability, like Radial RRT, and good coverage, comparable to RRT.

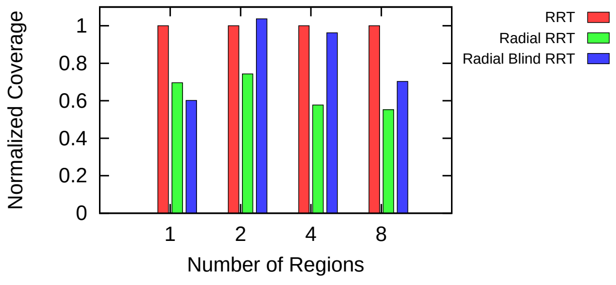
6.3.1 Experimental Setup

Experiments were conducted on a Linux computer center at Texas A&M University. The cluster has a total of 300 nodes, 172 of which are made of two quad core Intel Xeon and AMD Opteron processors running at 2.5GHz with 16 to 32GB per node. The 300 nodes have 8TB of memory and a peak performance of 24 Tflops. Each node of the cluster is made of 8 processor cores, thus, for this machine we present results for processor counts in multiples of 8.

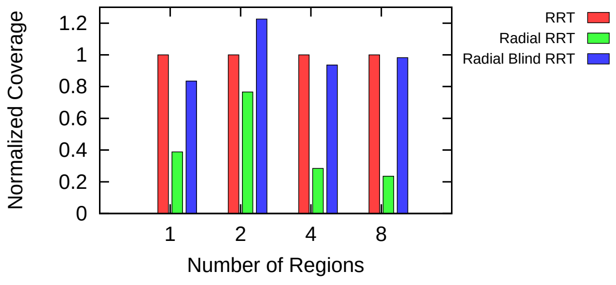
All the methods use Euclidean distance as the distance metric, straight-line local planning, brute force neighborhood finding, and collision detection tests as validity tests. Four different environments were used: 2-D Clutter (Figure 6.4(a)), 2-D Grid (Figure 6.4(b)), 2-D Maze (Figure 6.4(c)), and 3-D Maze (Figure 6.4(d)).



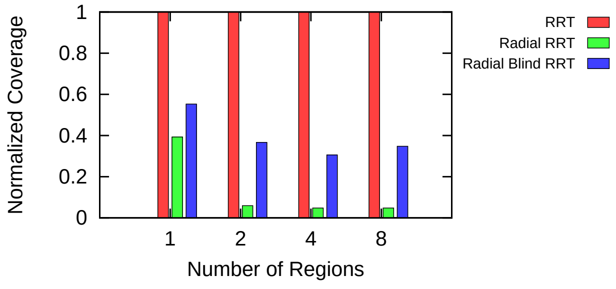
(a) 2-D Clutter



(b) 2-D Grid



(c) 2-D Maze



(d) 3-D Maze

Figure 6.5: Comparing coverage after performing RRT, Radial RRT, and Radial Blind RRT. All results are normalized to RRT.

6.3.2 Map Coverage

In this section, we compare each method's ability to map space by analyzing the coverage of the generated trees. We approximate coverage with a sample size of 250 uniformly sampled nodes. Since Radial Blind RRT deletes nodes at two points of its execution, it is not effective to use a desired final number of nodes. Instead, we fixed the parameter N_{br} to be 500. Another parameter that plays an important role is the number of CC connection attempts in the local phase. Given that for each environment the number of CCs will vary, we set the number of CC connection attempts to be five times the number of CCs after the initial expansion phase. This number was chosen according to initial testing results which demonstrated that a high number of CC connection attempts only increases the number of nodes but does not connect the tree significantly better, making the method rather slow. To have a fair comparison between methods, for each random seed, we ran Radial Blind RRT first and recorded the number of nodes, N_i. Then, we took the number of nodes to be the N_i for both RRT and Radial RRT. Radial Blind RRT and Radial RRT were tested with $N_r = [1, 2, 4, 8]$. Coverage results are averaged over 10 random seeds and normalized to RRT. Results are shown in Figure 6.5.

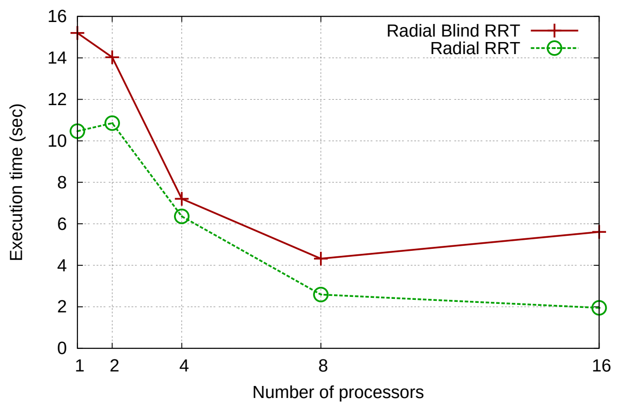
Radial Blind RRT results in better map coverage compared with Radial RRT, except in one case (2D-Grid with one region) in which Radial Blind RRT was comparable to Radial RRT. Moreover, in most of the 2D environments Radial Blind RRT has higher or comparable coverage compared to RRT. In higher dimensional cases (3D-Maze), we believe radial decomposition hampers exploration for a fixed number of nodes. However, we note that Radial Blind RRT still performs better than Radial RRT in these cases. We will look at improving these results for higher dimensional problems in the future. From these results, we can see that Radial Blind

RRT shows usefulness in planning over Radial RRT alone, and in some cases, e.g., 2D homogeneous environments, Radial Blind RRT actually outperforms RRT.

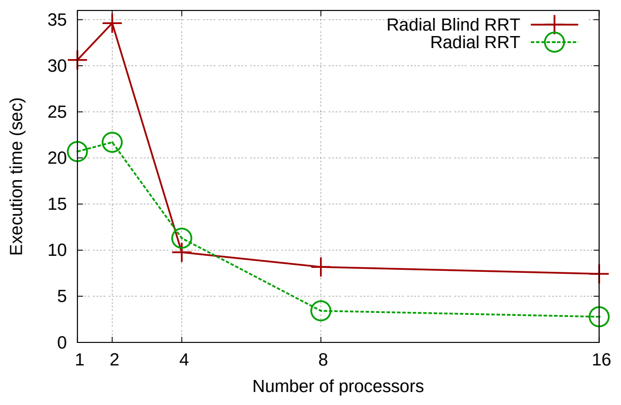
6.3.3 Parallel Performance

We evaluated Radial Blind RRT on the Linux cluster varying the processor count from 1 to 16. We compare Radial Blind RRT to Radial RRT to see differences in performance as the number of regions increases. In these experiments, the number of cores is equal to the number of regions. It is important to note that Radial RRT's and Radial Blind RRT's trees differ as the region count differs. We carried out the experiments in the 2-D Clutter, 2-D Grid, and 3-D Maze environments. The initial input sample size was fixed at 1600. Each experiment was run five times and the average runtime of the longest running processor was computed. Results are shown in Figure 6.6.

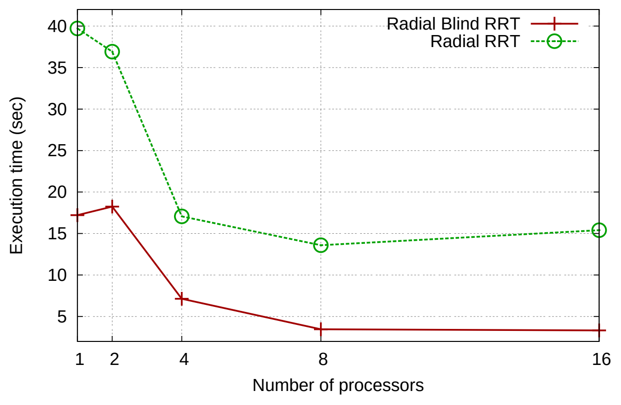
We observe that the relative performance of each algorithm depends on the environment. When Radial RRT requires more time, many failed attempts occur as regions restrict the expandability of the tree. In these cases, more computation time is spent attempting expansions as the tree attempts to grow to a specific size. When Radial Blind RRT requires more time, more work is spent in attempting to connect disconnected components. The tree size for Radial Blind RRT is larger, so we expect that Radial Blind RRT requires more work from an initial sample set. Considering runtime, we can see that even though Radial Blind RRT does more work, as it explores space better, running times are still comparable. Additionally, as the number of regions and processors increases, the running times decrease.



(a) 2-D Clutter



(b) 2-D Grid



(c) 3-D Maze

Figure 6.6: Execution times of Radial RRT and Radial Blind RRT.

7. USING LOAD BALANCING TO SCALABLY PARALLELIZE SAMPLING-BASED MOTION PLANNING ALGORITHMS

Regular spatial subdivision is limited in the types of motion planning environments it can handle. This method performs well in uniform and homogeneous environments, but not complex, non-uniform and heterogenous environments. For example, a house or factory floor is typically composed of logically separate parts; open or free space, cluttered space, doorway, narrow passages, stair, rooms etc. Regular subdivision in this scenario is limited and prone to load imbalance. As an illustration, consider the regular subdivision of the planning space in Figure 7.2; if different processors are assigned to each region, processors assigned to $region_0$ are apparently overloaded. This irregularity in planning space leads to workload imbalance, which will have an overall negative affect on scalability.

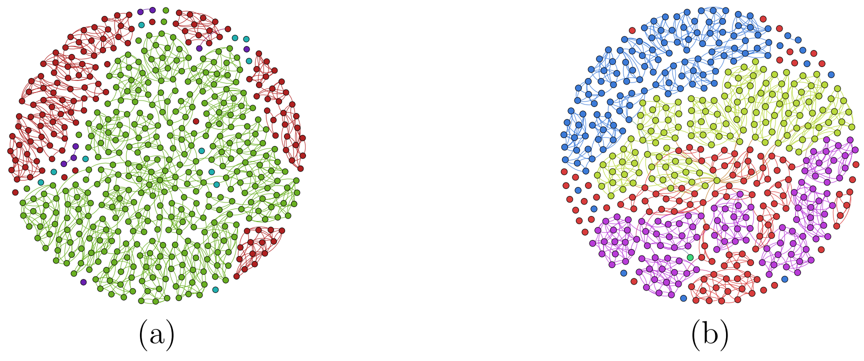


Figure 7.1: Roadmap graph node distribution (a) before rebalancing: majority of nodes are present on two processors (green and brown color) (b) after rebalancing: almost even distribution of nodes.

Figure 7.1 illustrates the distribution of the roadmap graph for an environment that suffers from a high degree of load imbalance using regular spatial subdivision. Shown is a sample run with four processors where the color of a node represents a single processor. In Figure 7.1 (a), it is clear that the majority of the roadmap nodes are only present on two processors, and the remaining two processors have only a small number of vertices. In contrast, Figure 7.1 (b) shows an even distribution of roadmap nodes after applying load balancing techniques.

One important consideration that any load balancing strategy must take into account is the granularity in which the problem is partitioned. This is because the size of the biggest quanta of work establishes a lower bound by which the problem can be balanced using a perfect load balancing strategy. In addition, a more refined problem provides more opportunity to distribute work amongst processing elements. For parallel motion planning, regions represent the quanta of work and thus for the presented load balancing strategies we consider an over partitioned region graph.

In this section we will describe two load balancing techniques that will benefit parallel sampling-based motion planning algorithms.

7.1 Basic Load Balancing Techniques

Work stealing [68, 65] is an important technique used to balance an imbalanced computation. In this method the computation is logically divided into a collection of tasks. When a processing element runs out of its local tasks it attempts to steal tasks from potential victims. This strategy is well suited for shared-memory systems but has some drawbacks in distributed-memory systems. In such systems, an important decision to make when stealing tasks is whether the data associated with those tasks should be moved to the thief processing element. This decision is usually application dependent and is influenced by the following factors:

- Access to remote data can adversely affect the performance of the application.

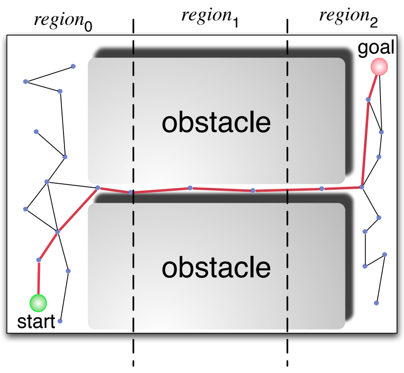


Figure 7.2: Regular subdivision method for parallel PRM.

- The same data may be required by the thief processor for subsequent phases of the computation.

There are two variations on the way data can be made available to the thief: replication and ownership transfer. In the case of replication, some sort of software coherence mechanism may be required to deal with the multiple copies of data, while in some cases the overheads associated with transferring ownership to the thief processor may be prohibitively high. In this work we have a model in which transfer of ownership is considered.

Repartitioning of data is another strategy to address load imbalance. In an owner-computes model of computation, it is well known that data distribution is fundamental to achieving acceptable levels of load balance. There exists an exhaustive amount of literature regarding partitioning [75, 76, 77] of distributed data structures. We focus on computing, and enforcing through data migration, high quality partitions of the problem across processing elements.

In general, the type of load balancing technique applied to an imbalanced computation depends on the nature of the computation itself. Repartitioning of data structures is well suited for applications in which a good estimate of the computation associated with the data can be easily computed. Furthermore, the total amount and structure of the computation is known *a priori*. In contrast, work stealing is best suited for dynamic applications in which either the execution of the algorithm defines more computation as the algorithm progresses, or the work associated with the input data cannot be easily estimated to a reasonable degree of accuracy.

For regular spatial subdivision parallel PRM, the number of sampled configurations within a region provides a reasonable estimate for the amount of computation associated with that region. For this reason, we expect repartitioning to be the load balancing strategy of choice.

7.2 Load Balancing for PRM

For the approaches to parallel motion planning discussed in previous chapters, it is difficult to compute a good partition of the data structures offline due to the input-dependence and random nature of the algorithms. For this reason, an online repartitioning strategy is the most natural answer to finding a relatively high quality partition for a given environment and processor configuration.

In parallel motion planning, the two data structures of interest are the graph representation of regions and the roadmap or RRT graph itself. Regions represent spatial subdivisions of the environment in which configurations will be sampled. Connections are attempted between configurations through the use of collision detection methods. It is well known in motion planning that the cost of connecting samples in $\mathbb{C}_{space}$ is highly representative of the amount of time the overall algorithm will take in generating a solution. This in fact is the most time consuming phase of the entire computation. As regions that have a high number of samples will generally incur a large amount of collision detection calls, a good metric for approximating the amount of work that a region will generate is the number of samples in the roadmap that lie within that region.

Using this information, we can determine that load imbalance in terms of regions corresponds to the number of roadmap samples of the region, and this metric can be used to weight regions. A high quality partition of the region graph will attempt to balance the regions based on this metric. However, as regions are also spatial entities, the spatial geometry of regions should also be preserved in an ideal partition. By partitioning the region graph using these approximations of the amount of work that a region will perform, the algorithm will see a higher level of load balance for subsequent phases of computation.

Algorithm 15 Regular Subdivision with Repartioning

Input: Regional roadmap graph with sample configurations $R(V, E)$.

Output: Connected roadmaps with in regional graph

```
1: for all $v_i \in V$ par do
2:     $W \leftarrow$ ComputeRegionWeight($v_i$)
3: end for
4: GraphRepartition($R$, $W$)
5: for all $v_i \in V$ par do
6:     $G \leftarrow$ ConstructRegionalRoadmap($v_i$)
7: end for
```

In Algorithm 15, we show how to use repartitioning to influence load balancing in parallel PRM. The main imbalanced computation, ConstructRegionalRoadmap for a given region, is performed only after attempting to redistribute the regional graph based on the weight for each region. This will ensure that this phase of computation will be balanced according the metric of the number of sampled configurations within a region.

Since different regions represent different amounts of work due to presence of obstacles and differences in generated samples, some processing elements will deplete their local work faster than others. This property of the computations also makes it amenable for work-stealing strategies. In the experimental results, we present the performance gains due to these two strategies when compared to no load balancing.

7.3 Load Balancing for RRT

Radial subdivision for RRT discussed in Chapter 5 and Chapter 6 is an inherently dynamic application, the amount of work that a region will perform is difficult to estimate beforehand, work stealing is a prime candidate for this algorithm. Naturally, some branches will have more difficulty exploring $\mathbb{C}_{space}$ than others, and processors assigned to branches that correspond to relatively simple portions of the environment will run out of local work quickly.

Algorithm 16 Work-stealing Radial Subdivision

```
Input: Regional roadmap graph, steal policy.
Output: Set of constructed RRT branches
 1: while Global termination not detected do
 2:    for all $p \in Processors$ par do
 3:       $Q \leftarrow \{$ Regions of $p$ $\}$
 4:       while $Q$ is not empty do
 5:          $R_{current} \leftarrow Q_{pop}$
 6:          ConstructBiasedRRT($R_{current}$)
 7:       end while
 8:       $V \leftarrow$ choose victim based on steal-policy
 9:       Steal regions from $V$ based on policy
10:    end for
11: end while
```

Algorithm 16 shows work stealing during the construction of the biased RRTs for radial subdivision. The main computation in which RRTs are expanded in independent branches is shown in Line 6. As each processor is assigned regions in which to explore, we model these branches in a local work queue. When this local queue is depleted, the processing element will issue steal requests to potential victims in hopes of receiving additional branches in which to explore. On a victim processor, work is stolen from the back of its local work queue. Potentially, priority could be given to regions to send to the requesting processor; however, computing such a priority is non-trivial due to the dynamic nature of the algorithm.

The choice of selecting a victim is a particularly important decision. This is because the cost of stealing from a processor on the same shared-memory node is generally less than the cost of stealing from a processor on another node. More importantly, for parallel motion planning, the choice of victims should also be related to the distribution of the region graph among the processors. After the RRT construction phase, neighboring processing elements will communicate with each other to perform region connections. This indicates that stealing from neighbors in the RRT construction phase would also benefit the region connection phase, as the regions to which to connect will be local to the same processing element.

We consider several work stealing strategies in the context of parallel motion planning. One strategy (RAND-K) is a randomized strategy in which a thief requests additional regions from k random processors. For the purpose of our experimental evaluation, we have fixed k to be 8. Another strategy we employ is a heuristic (HYBRID) wherein processors are assumed to be arranged in a 2D mesh and underloaded processors will first ask neighboring processors for work. In the event that no request could be serviced in the neighborhood, requests are sent to a random processor. In the experimental results section, we compare and contrast these two strategies.

7.4 Implementation in STAPL

Load imbalance in parallel computations is dealt with in various ways in STAPL. The repartitioning-based approach to load balancing discussed in previous sections, this is realized in STAPL through redistribution of the two `pContainer`s in the parallel motion planning algorithms.

In its most basic form, an application can be instrumented to perform repartitioning by simply providing a view of the container to migrate, and weights of the individual elements of the container (Figure 7.3). Additionally, a user-defined function can be provided that will define actions that need to be taken upon a migration, such as additional migrations of secondary data structures. Internally, the data structure will be redistributed using various techniques, including STAPL algorithms that diffusively move work to neighbors and attempt to minimize edge cuts and by extension preserve geometric features of the graph, or those that globally balance weight in blocks. Alternatively, the STAPL Load Balancing Framework can also be used interoperably with external partitioning libraries, such as Zoltan [75].

```
container.migrate(GID g, Location loc);
container.redistribute(View view);
rebalance(View vw, WeightMap w_map, ActionMap a_map);
```

Figure 7.3: The fundamental migrate primitive, redistribution of a container based on a view and rebalancing a view based on weights.

An alternative approach to help address load imbalance is to employ a work-stealing strategy during the computation itself. This is realized in STAPL by providing support for customizable schedulers. Figure 7.4 shows the call site of an algorithm that is explicitly instrumented to use a work-stealing scheduler.

The work-stealing scheduler moves tasks from from overloaded processors to underloaded processors. In addition to moving the specification of work, STAPL will also migrate the data associated with the work to the thief location. Migration of a container's elements during a computation will have the additional benefit of balancing data in the container according to a metric that is directly associated with the performed computation. Because of this, any subsequent computation that will have an affinity pattern similar to the previous computation will be balanced in terms of load. Thus, the goal of migrating the task's data would be to seed work for future computations in a balanced manner.

In order to improve programmer productivity, STAPL provides a shared-object view to an application developer. The details about the distribution and locality of data are hidden from the users. Advanced users still have access to this information, but this is not the default programming model. The advantage of this model is that the user is free to choose the most natural expression for the application without worrying about performance. In the context of parallel motion planning, this translates to a high number of fine grained accesses to the vertices of the region graph. To support such a model and achieve good performance, STAPL relies on placing the computation and data near each other and automatically aggregating any requests for remote data. By moving the data associated with a task, the work stealing scheduler reduces the number of remote accesses.

In essence, the work stealing scheduler's migration of tasks and data not only achieves balanced computation, but also a data distribution that is most suited for that distribution of computation. For many iterative applications, such changes in the initial iterations of the computation can be beneficial for achieving balanced computation in the later iterations of the application.

An important decision that the scheduler has to make is the choice of the victims. The cost of stealing from a location on the same node can be less that the cost of stealing from a location on another node. More importantly, for parallel motion planning, the choice of victims should also be related to the distribution of the region graph among the locations. After the node connection phase, neighboring locations will communicate with each other to perform region connections. This indicates that stealing from neighbors in the node connection phase would also benefit the region connection phase.

In the experimental section we describe some of the strategies for choosing victims with which we have experimented.

```
p_algorithm(view, work_stealing_scheduler(...))
```

Figure 7.4: Customizable scheduling scheme for a call to a parallel algorithm.

7.5 Experimental Evaluation

7.5.1 Setup

Experimental studies were conducted on two massively parallel machines: a 153,216 core Cray XE6 (Hopper) and a 2,400 core Opteron cluster (opteron-cluster). The environments considered in this section are a 3D narrow passage with a rigid-body robot in which 90% of the space is blocked (walls) and a cluttered narrow passage environment with 40% blocked space (narrow).

7.5.2 Parametrically Imbalanced Environment

Consider an environment with a single $\beta \times \beta$ cube obstacle. The distance from the bounding box to the cube is α.

In such an environment, we have the following:

$$V_{total} = (2\alpha + \beta)^2 \tag{7.1}$$

$$V_{obs} = \beta^2 \tag{7.2}$$

$$V_{free} = (2\alpha + \beta)^2 - \beta^2 \tag{7.3}$$

The coordinates of the obstacle are (α, α) and $(\alpha + \beta, \alpha + \beta)$.

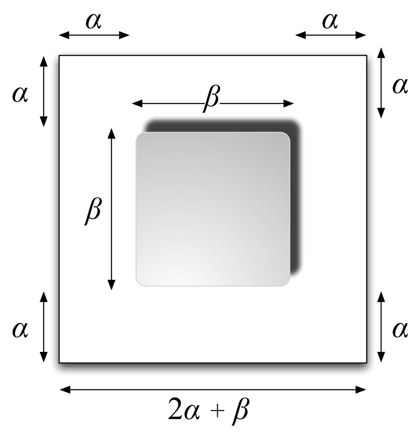


Figure 7.5: Imbalanced cube environment

Consider an arbitrary two-dimensional subdivision of this environment, where the number of cuts in the x dimension is P_x and the number of cuts in the y dimension is P_y. A region r_{ij} is formed by these cuts.

The size of any region in the x dimension is $P_x^{-1}(2\alpha+\beta)$ and $P_y^{-1}(2\alpha+\beta)$ in the y dimension.

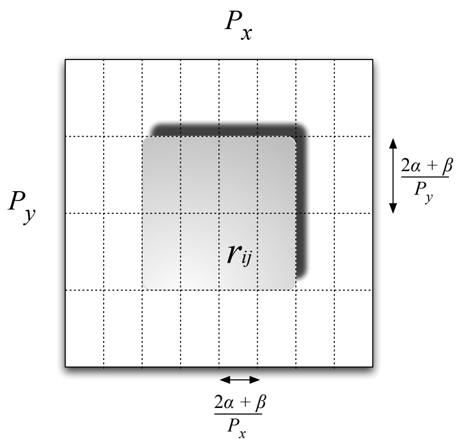


Figure 7.6: Subdivision of imbalanced cube environment

The bounding box for region r_{ij} can be computed by:

$$BB_{x_0}(r_{ij}) = i \times P_x^{-1}(2\alpha + \beta)^2 \tag{7.4}$$

$$BB_{y_0}(r_{ij}) = j \times P_y^{-1}(2\alpha + \beta)^2 \tag{7.5}$$

$$BB_{x_1}(r_{ij}) = BB_{x_0}(r_{ij}) + P_x^{-1}(2\alpha + \beta)^2 \tag{7.6}$$

$$BB_{y_1}(r_{ij}) = BB_{y_0}(r_{ij}) + P_y^{-1}(2\alpha + \beta)^2 \tag{7.7}$$

The obstacle within the region can be found by:

$$obs_{x_0}(r_{ij}) = \max\{BB_{x_0}(r_{ij}), obs_{x_0}\} \tag{7.8}$$

$$obs_{y_0}(r_{ij}) = \max\{BB_{y_0}(r_{ij}), obs_{y_0}\} \tag{7.9}$$

$$obs_{x_1}(r_{ij}) = \min\{BB_{x_1}(r_{ij}), obs_{x_1}\} \tag{7.10}$$

$$obs_{y_1}(r_{ij}) = \min\{BB_{y_1}(r_{ij}), obs_{y_1}\} \tag{7.11}$$

From the bounding box and obstacle, we can compute the volume of the free space (V_{free}) by using the total volume of a region and the volume of the obstacle within the region.

$$V_{total}(r_{ij}) = (BB_{x_1}(r_{ij}) - BB_{x_0}(r_{ij}))(BB_{y_1}(r_{ij}) - BB_{y_0}(r_{ij})) \tag{7.12}$$

$$V_{obs}(r_{ij}) = (obs_{x_1}(r_{ij}) - obs_{x_0}(r_{ij}))(obs_{y_1}(r_{ij}) - obs_{y_0}(r_{ij})) \tag{7.13}$$

$$V_{free}(r_{ij}) = V_{total}(r_{ij}) - V_{obs}(r_{ij}) \tag{7.14}$$

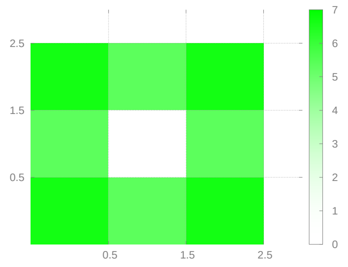
With the estimation of the free space in the environment, we can say that the total load that that region will experience is proportional to V_{free}, the amount of free space within that region.

A naive mapping of regions to processors would perform a 1D partitioning of the region mesh and assign columns of regions to processors. Given p processors and a region mesh size of $R_x \times R_y$, we linearize regions in the x dimension and assign a balanced partition of regions to each processor. The total load of a processor $\mathcal{L}(p)$ is the sum of V_{free} for each region assigned to that processor.

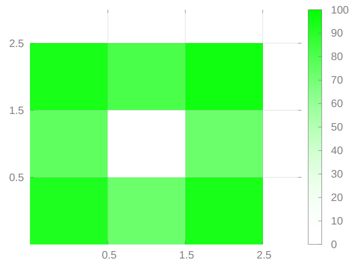
A measure of imbalance among processors is the *coefficient of variation*, defined to be the ratio of the standard deviation σ and mean μ load. The naive region mapping will have a high coefficient of variation for the model environment. We also compute the best possible partitioning of the region graph statically using a greedy global partitioning algorithm, as the exact problem is NP-complete.

7.5.2.1 Model Evaluation

The following figures show the model's calculation of the volume of the free space for the imbalanced environment ($\alpha = 2, \beta = 4$).



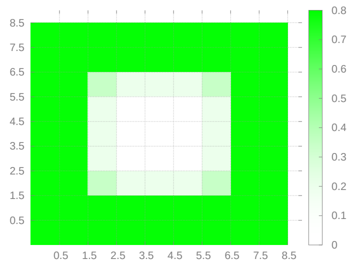
(a)



(b)

Figure 7.7: In a 3x3 spatial decomposition, the (a) model's estimation of the volume of free space and (b) the number of roadmap nodes sampled per region in a test run.

Figure 7.7 (a) shows the model's estimation of V_{free} for each region of the region graph in a 3x3 decomposition. It accurately models the size of the obstacle in the center of the environment, and a diffusive amount of free space radiating outward


(a)

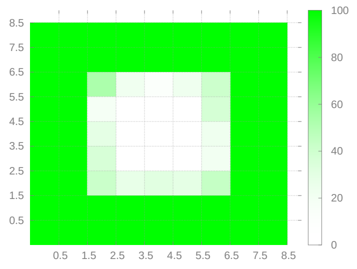

(b)

Figure 7.8: In a 9x9 spatial decomposition, the (a) model's estimation of the volume of free space and (b) the number of roadmap nodes sampled per region in a test run.

from the origin. Figure 7.7 (b) shows the number of roadmap nodes generated per region after running the algorithm on the model environment. As we can see, it closely tracks the model's estimation of the free space per region, and thus the amount of load per region. Similarly, Figure 7.8 shows close tracking between the model's estimation and experimental evaluation for a 9x9 decomposition.

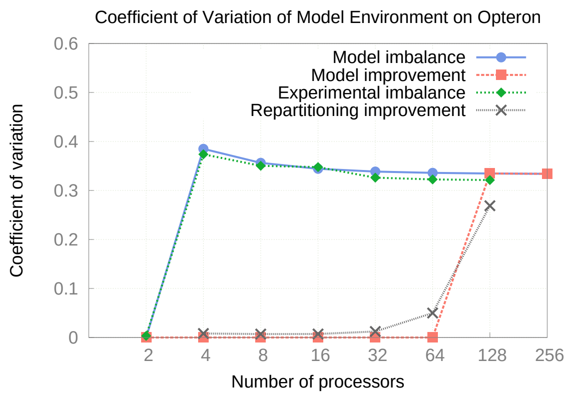


Figure 7.9: Experimental validation of measure of load imbalance in model environment. ($\alpha = 2, \beta = 4$ and $R_x = 256, R_y = 1$)

Figure 7.9 shows the model's prediction of the imbalance with the naive partitioning strategy and the best possible load balance possible. In addition, we plot the measure of load imbalance experienced during a trial run of the application and show that we closely track the model. As shown, the best possible distribution of regions to processors for higher core counts shows less benefit, as each processor has an increasingly smaller granularity of work as the number of processors increases.

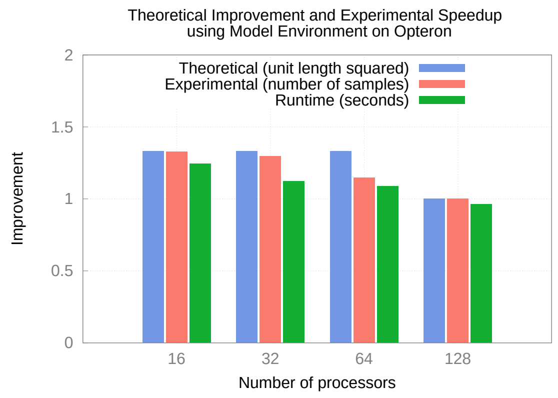


Figure 7.10: Experimental validation of potential improvement in model environment. ($\alpha = 2, \beta = 4$ and $R_x = 256, R_y = 1$)

Figure 7.10 shows the total improvement for various metrics according to the model and an experimental evaluation. We study the potential improvement according to the model, which measures the total reduction in V_{free} for the processor with the highest amount of V_{free}, the reduction in the number of roadmap nodes on the highest loaded processor and the overall improvement in execution time for the node connection phase with using repartitioning. In general, we track the model's theoretical estimate of the best load distribution in terms of roadmap nodes, which in turn closely tracks the improvement in execution time. The discrepancies between best distribution of V_{free} and roadmap node distribution can be explained by both the probabilistic nature of the computation and by the geometric restrictions enforced by the repartitioning. The gap between the improvement in roadmap distribution and total time reduction is a result of the number of roadmap nodes per region being an imperfect indicator of the total amount of work generated by that region.

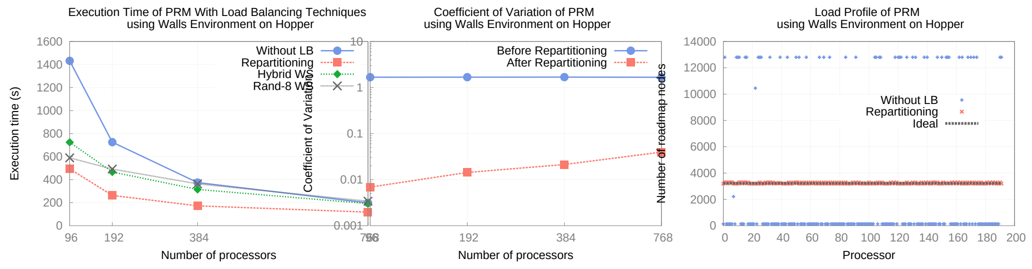
7.5.3 Experimental Results

7.5.3.1 PRM

For this experiment, we evaluated our load balancing techniques on a highly load imbalanced environment (walls). Figure 7.11 (a) shows raw execution time of computing the final roadmap on the Hopper platform for this strong scaling experiment. We can see that using repartitioning, we are able to achieve a 2.9 times improvement over the baseline on 96 cores and a 1.68 times improvement on 768 cores. Because of the strong scaling nature of our experiment, there are significantly fewer regions per processor at 768 cores, which allows for less opportunity for moving load across processors. One metric to quantify the degree of load imbalance is the coefficient of variation, defined to be the ratio of the standard deviation σ to the mean μ. From Figure 7.11 (b), we can see that although the coefficient of variation is substantially lower for all processor counts after repartitioning, the difference is not as much for higher processors counts simply because of less opportunity to rebalance. Figure 7.11 (c) shows the distribution of load across processors on a 192-core run on Hopper. We see that without load balancing, there is a wide spread in work and after applying repartitioning, a distribution closer to the ideal is achieved.

In addition to repartioning, Figure 7.11 also illustrates the difference between the two work stealing strategies. For lower core counts, rand-k performs better, due to the higher probability of finding work faster than hybrid.

For the same experiment, we show the breakdown of the various phases of parallel



(a) (b) (c)

Figure 7.11: Evaluation of (a) execution time and (b) coefficient of variation and (c) load distribution for PRM on Hopper.

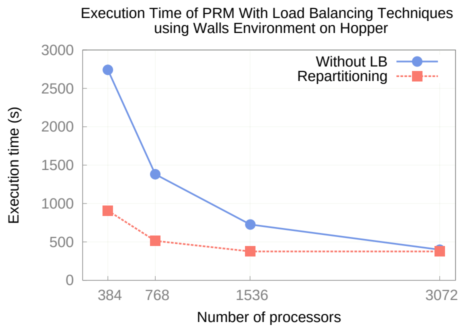


Figure 7.12: Evaluation of computing roadmap in the walls environment for a rigid body robot on Hopper

PRM in Figure 7.15 (a). As suspected, the portion of the computation connecting roadmap nodes in a region dominates most of the computation at 90% of the total execution time. After load balancing for both methods, the total time decreases, mainly because of the decrease in node connection time. For repartitioning, there is an increase in region connection time, which can be partially attributed to an increase in remote accesses in the region connection phase, as shown in Figure 7.15 (b). This is due to an increase in edge cuts, which was induced by repartitioning. The work-stealing method performs better than the non-load-balanced run, but not as well as repartitioning. We can see that the node connection phase does not improve to the

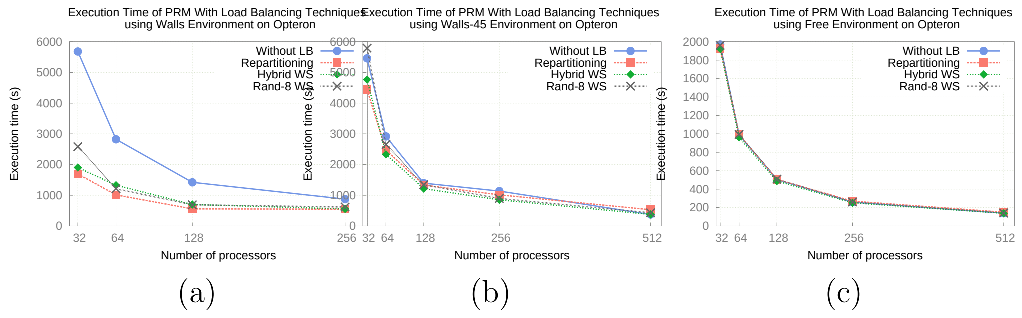
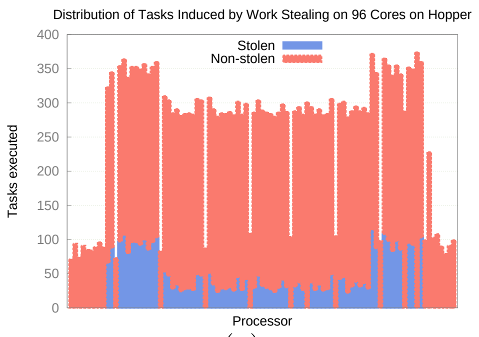


Figure 7.13: Execution time for PRM with various load balancing strategies in (a) walls (b) walls-45 (c) and free environment.

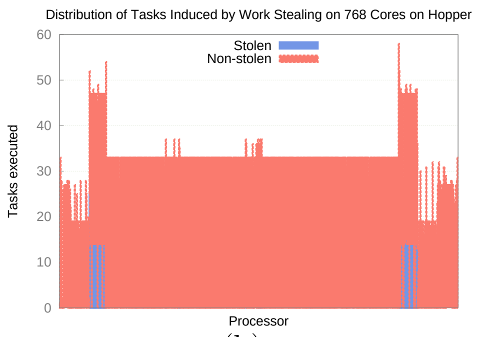
extent of repartitioning, due to the random and non-exact nature of work-stealing and various overheads involved. However, region connection was not affected to the degree as shown with repartitioning because the method ultimately did not move a large amount of regions and thus the number of edge cuts were not affected as severely.

Figure 7.12 shows that the general trend shown in the previous analysis holds for higher processor counts on HOPPER. Similarly, Figure 7.13 demonstrates the portability of these techniques to OPTERON-CLUSTER.

Figure 7.14 provides a detailed breakdown for HYBRID work-stealing illustrating the number of tasks that were executed locally and the number of stolen tasks for each processor. In Figure 7.14 (a), we see that a substantial number of underloaded processors find work to be stolen and execute a large amount of stolen tasks. In contrast, we find that at higher processor counts, such as those shown in Figure 7.14 (b), it becomes difficult for underloaded processors to find work to be stolen, as the work per processor decreases and the pool of potential processors from which to request increases. The figure shows that few processors are able to find work once they have exhausted their local regions. Moreover, the amount of work available



(a)



(b)

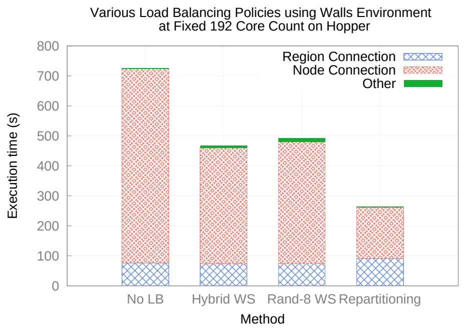
Figure 7.14: Breakdown of the amount of tasks stolen vs. executed locally for PRM on (a) 96 and (b) 768 cores on Hopper.

for stealing also decreases. Both these behaviors are expected for strong scaling experiments.

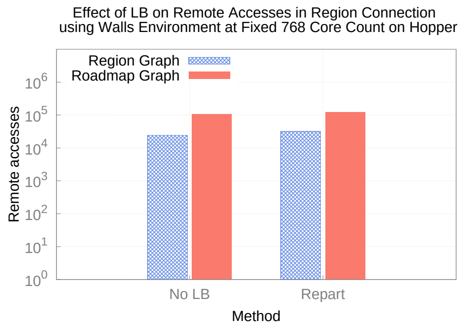
7.5.3.2 Radial RRT

We also evaluated our load balancing techniques on the radial RRT parallel motion planning algorithm. As discussed in Section 7, it is difficult to estimate the amount of work that a radial branch will compute due to the probabilistic and dynamic nature of the algorithm. Thus, computing an effective partition for load balancing is difficult and therefore, this study will focus on work stealing strategies for radial subdivision.

Figure 7.16 shows the total execution time for computing the final RRT for a rigid body robot in the narrow environment on OPTERON-CLUSTER. Using the HYBRID work-stealing strategy allowed the algorithm to achieve a speedup of 1.65 times on 32 cores and a slight decrease in performance at 256 cores. A similar pattern of decreasing marginal benefit of work-stealing from regular subdivision is exhibited in this experiment. As with regular subdivision, the amount of available work to steal per processor decreases as the number of processors increase, while the number of

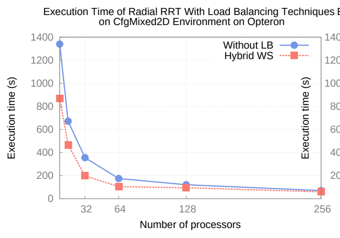


(a)

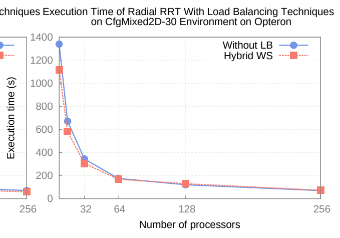


(b)

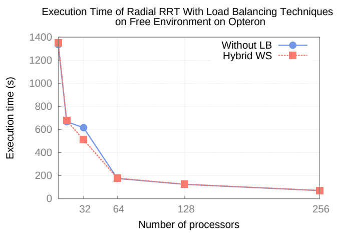
Figure 7.15: Breakdown of (a) the various phases of PRM (b) and the effect of load balancing on remote accesses.



(a)



(b)



(c)

Figure 7.16: Execution time for RRT with various load balancing strategies in (a) mixed (b) mixed-30 (c) and free environment.

potential victims from which to steal also increases. For these reasons, work stealing does not improve the total execution time to the same degree on higher core counts as much it could when work was plentiful at lower processor counts.

8. ROADMAP QUALITY ANALYSIS

In this chapter, we evaluate and compare the quality and structure of roadmaps constructed using our framework for parallelizing sampling-based motion planning algorithms with roadmaps constructed using traditional sequential planners. Also, we provide experimental results that show that motion planning problems involving heterogenous environments are a natural fit for spatial subdivision based parallel processing.

8.1 Evaluation Metrics

We start by presenting the evaluation metrics to be used in our experimental study.

8.1.1 Edge Metrics

8.1.1.1 Number of Edges

We consider the number of edges generated for a roadmap graph to be an important measure of the quality of the roadmap graph. In general, the number of edges is indicative of the connectivity and structure of the roadmap graph. Roadmap graphs with more edges tend to be better connected and are likely to have smaller diameters. Therefore, it is important that we consider the number of edges as a primary roadmap quality metric.

Given a roadmap graph $G(V, E)$ of V vertices and a connection method that attempts to connect the k closest or random neighbors to each vertex $v \in V$, an upper bound on the number of edges E using a sequential planner is:

$$E = O(V * k) \tag{8.1}$$

Our proposed parallel algorithm will yield a roadmap graph with an upper bound on the number of edges as:

$$E = O(V * k + E_R * k') \tag{8.2}$$

where E_R is the number of edges in the region graph and k' is the number of closest pair connections made between adjacent regions.

If the number of edges is a measure of roadmap quality, then we expect that the roadmap generated using parallelism and spatial subdivision to be of higher quality because:

$$O(V * k + E_R * k') > O(V * k) \forall E_R > 1 \tag{8.3}$$

In the above, we assume that each k or k' identified neighbor or resulting edge connection is unique. This may not be the case if the edges are not unique, a condition which may occur if V is low and k or k' is high. Typically, k is a constant and is normally very low compared to V [15, 88], so our assumption is emperically valid. Also, selecting unique random neighbors increases the possibility that the resulting edge connection is unique. While selecting closest neighbors is the defacto standard in motion planning algorithms, it is not uncommon to explore random neighbor selection [25, 26, 13]. In fact, some studies have shown that random neighbor selection does improve roadmap quality [88].

8.1.1.2 Edge Length

Edge length is a useful metric as it has the potential to affect the diameter of the roadmap graph as well as the path length of a resulting query. Typically, shorter edges are preferred but this is not always the case; it largely depends on the motion planning problem. If we adopt the so called k nearest neighbor selection and edge connection method, the intuition is that edges made by the subdivision approach are likely to be longer compared to edges without subdivision. This is because of the possibility that there would be closer neighbor(s) in another region. This issue can be addressed by increasing the overlap distance between regions. Also, depending on the neighbor selection policy chosen and the problem, the average edge length of the region-based (parallel) roadmap graph should be shorter than the average edge length of the roadmap generated using a sequential planner. This is because of the closer proximity between nodes in the region-based roadmap graph.

8.1.2 Coverage and Connectivity Metrics

8.1.2.1 Coverage

Coverage is a measure of node distribution and reachability [89] of a roadmap graph. Given a configuration c in $\mathbb{C}_{free}$ F, we define coverage of c as the subset of F that is visible from c:

$$Coverage(c) = \forall c' \in F | visible(c, c') = true \tag{8.4}$$

The coverage of a set $S = c_1, c_2, ..., c_n$ can be defined as the union of the coverage of the set elements:

$$Coverage(S) = \bigcup_{i=1}^{n} Coverage(c_i) \tag{8.5}$$

The roadmap graph $G(V, E)$ is said to cover the $\mathbb{C}_{free}$ when each configuration $c \in \mathbb{C}_{free}$ can be connected using the local planner to at least one node $v \in V$ [89]. The higher the coverage, the better the roadmap should be.

8.1.2.2 Connectivity

Connectivity is a measure of how well a roadmap graph is connected or how close it is in representing the connectivity of the free space $\mathbb{C}_{free}$. One common way to compute the connectivity of a graph is to define connections between a pair of nodes (v, v') in the graph. A pair of nodes is said to be connectible if a local planner could find a path between them. The roadmap graph $G(V, E)$ is said to be *maximally connected* if for all pairs of nodes $(v, v') \in V$, if there exists a path in $\mathbb{C}_{free}$ between v and v', then there exists a path in G between v and v' [89].

8.1.2.3 Number and Size of Connected Components

The number and size of the connected components (CC) is another useful metric, particularly in our work in which the planning space is subdivided into regions and regions are assigned to processors with the task of constructing roadmap in each region. In this scenario, at best, without regional roadmap connection, we end up with a number of connected components that equals the number of regions. Even so, connecting the regional roadmaps must take place for complete solution. The number and size of the resulting CCs after region connection is a good way to measure how effective the region connection phase is as well as a measure of the overall quality of the final roadmap. The number of connected components of the roadmap graph should represent the topology of the underlying $\mathbb{C}_{space}$ as much as possible. Typically, the fewer the connected components the better the quality of the generated roadmap graph.

8.1.3 Query Processing and Path Length

8.1.3.1 Witness Query Processing

While not a sufficient metric for evaluating the quality of roadmap, witness query processing is still a common way to do such evaluation. The motion planning problem is considered solved when a movable object starting at an initial configurations reaches its final or goal configuration. A practical way to validate this is to use the witness query processing metric.

8.1.3.2 Witness Query Path Length

With every successful processing of a witness query, we could extract and compute the path that takes a movable object to a specified goal position from a start position. While different subdivision and processor counts may produce different path lengths and results may be biased for every pair of witness queries, a shorter path length is typically considered as resulting from a better roadmap.

8.1.4 Structural Metrics

8.1.4.1 Diameter

The diameter is an important metric to understanding the structure of the roadmap graph. Commonly defined as the length of the shortest path between two extreme nodes in a graph, the diameter of a graph gives us more insight into the underlying changes in the roadmap graph construction. It is an indication of how long paths in the roadmap graph would be. While it may be sufficient to keep track of the diameter of the largest connected component in the graph sometimes it is useful to track other connected components as well.

8.1.4.2 Average Shortest Path

The diameter tells us about the longest shortest path in a graph. It will be useful to also know about the average of the shortest paths as well as their standard deviation. This knowledge gives a form of balance in understanding the structural differences between both the sequential and parallel roadmap graph. The diameter tells us about the worst case scenario, the average shortest path tells us about the average or expected behavior in querying the roadmap graph.

8.1.4.3 Page (Node) Rank

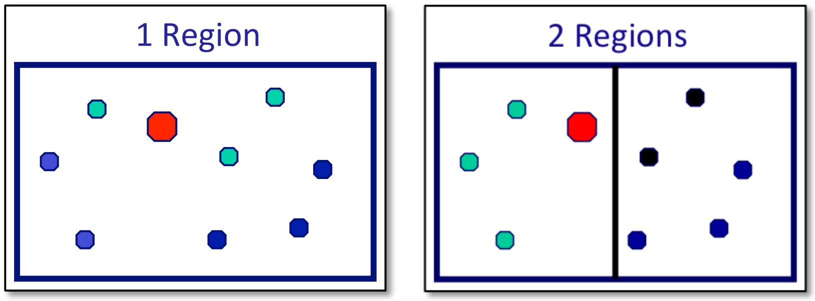
PageRank [90] is a popular algorithm for computing the relative importance of nodes in a graph. It was made popular by web graphs but could be used with any graph. PageRank (PR) or NodeRank is used here to understand how node or vertices with higher ranks could impact the shortest path, diameter or the overall structural properties of roadmap graph. A larger diameter could mean that there is a node of higher PR on the critical path. This node then becomes a "$must-pass-through$" in the graph and could become a bottleneck if a shorter path is desired.

8.2 Roadmap Graph Properties

Our framework for parallelizing sampling-based motion planning made no assumption about the underlying sampling scheme, nearest neighbor search, local planning, connection method, or the sequential planner in general. In other words, our proposed method inherits the probablistic completeness properties of the underlying sequential planner. It is trivial to show that for two connected components in two regions a RRT-Connect will find a path if one exists. The probabilistic completeness of our framework then follows from the probabilistic completeness of RRT-Connect [56].

However, it is expected that there will be structural differences in the roadmap graph generated by the sequential planner and our spatial subdivision-based parallel planner. The resulting roadmap graph structures will not be identical. The structure is impacted by both node (configuration) generation as well as the connection between each configuration and its neighbors. The difference in node generation results from the fact that different processes will pick a random configuration in $\mathbb{C}_{space}$ differently, but this is not critical, since, for probabilistic completeness, how sampling is done is not as important as the denseness of the sampling sequence [86]. The other more important issue that impacts the structure of the graph is edge connection or distribution. A critical component of the node connection phase of sampling-based motion planning is the nearest neighbor search. The spatial subdivision affects how neighbors are selected which then impacts the resulting edges and eventually the structure of the graph. As an illustration, consider the picture shown in Figure 8.1a, for the query point shown in red, the three nearest neighbors are shown in green. In Figure 8.1b, the $\mathbb{C}_{space}$ is now subdivided into two regions. Because of this subdivision, the three nearest neighbors points (in green) to the query point (in red) have now changed. This change will impact the structure of the graph. Depending on the environment and ratio of the sample set to the neighbor set, the resulting edges and diameter could be longer or shorter.

For the two reasons highlighted above, therefore, the roadmap graph built by concurrent processes exploring separate regions of the planning space cannot be identical to the one built by a sequential planner with or without spatial subdivision.



(a) 1 Region (b) 2 Regions

Figure 8.1: Impact of space subdivision on graph structure: For a given query point (red), 3-nearest neighbors are shown in green. Selected neighbors differ as a result of space subdivision.

8.3 Experimental Evaluation

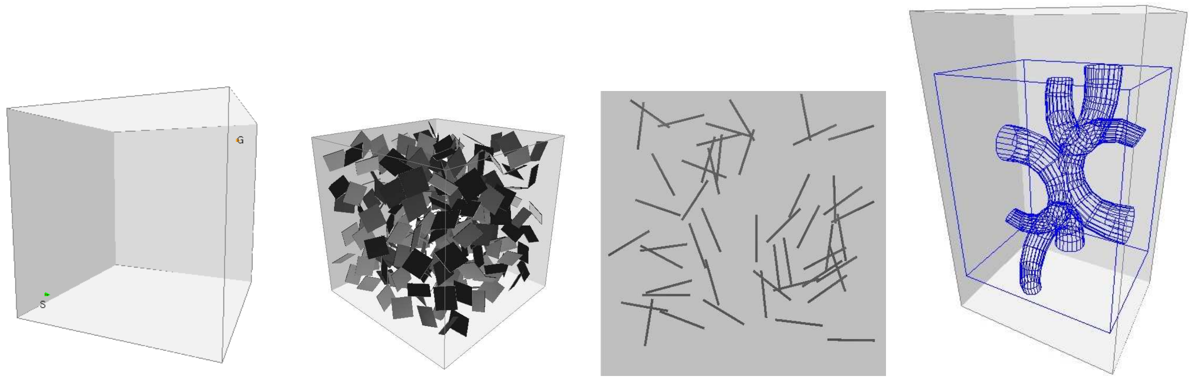
8.3.1 Setup

Our experimental studies were carried out on a wide range of motion planning problems as depicted in the environments shown in Figure 8.2. These environments, represent the class of enviroments that are commonly used as motion planning benchmark. Unless otherwise stated, each experiment was averaged over five runs.

8.3.2 Experimental Results

8.3.2.1 Free Environment

Our first experiment was conducted in a free environment modelled as a $50 \times 30 \times 50$ unit box without obstacles as shown in Figure 8.2(a). The movable object was a rigid cube robot of $2 \times 1 \times 2$ units. For this experiment, we keep the number of samples generated fixed at 800 nodes and the number of regions at 2 per processor while varying the processor counts from 1 to 4. We also ran the same experiment using the sequential planner. The result of this experiment is shown in Figure 8.3. Each quality metric is normalized to its equivalent result using sequential planner.



(a) Free (b) 3D Clutter (c) 2D Clutter (d) Maze

Figure 8.2: Environments

For most of the metrics, we observed that the parallel roadmap graph is at par with the sequential roadmap graph. The parallel graph has more edges than the sequential which can be explained from the previous section that the roadmap graph generated using our framework should theoretically have a higher upper bound in terms of number of edges. The claim for more edges in the parallel graph is feasible in a free environment where the probability of generating unique edges is higher. Also, we observed that the average edge length in the parallel graph is lower compared to the sequential graph. This is expected because the ratio of the sample size N to the k nearest neighbor is reduced as compared to the sequential planner. A shorter average edge length could also be a reason for a shorter path length, if the shorter edges are in the path for the witness queries. We also observed that the sequential planner generated a graph of smaller diameter compared to the parallel planner. This observation led us to assert that even though both the sequential and the parallel roadmap are closely related using different metrics, they are somewhat structurally different.

To better understand the reason and the nature of these structural differences, we compute the average and standard deviation of all the shortest paths. We expect that the average shortest path will give a clearer picture than just computing the approximate diameter (the longest of all the shortest paths). We also compute the page rank for all the vertices in the graph to identify which are the most prominent of all the vertices as we further subdivide the space. The results for both shortest paths and page rank are shown in Figure 8.4 and Figure 8.5, respectively. Figure 8.4 shows that the overall average shortest paths reduces with an increase in regions, similar, as to what was observed with the diameter evaluation. Figure 8.5 is a plot of the page rank of each node (vertex) in the graph against the node (vertex) descriptor for different numbers of regions in the subdvision. We observed that nodes in our graph are ranked differently as we recursively subdivide the $\mathbb{C}_{space}$. This difference is fundamental to the difference we observed in other metrics and among other things impacts the structural difference in the graph. For instance, the diameter will be different as the shortest path algorithms traverse the graph, i.e., the diameter could be shorter or longer depending on the ranks of the nodes on the longest shortest path. Figure 8.6 shows a frequency distribution of node (page) ranks for one and four regions. We observed that the frequency distributions are not uniform. This non-uniform distribution has a potential impact on the diameter of the graph and the graph's structural properties in general.

8.3.2.2 3D Clutter Environment

Our second experiment was conducted in a large uniformly cluttered enviroment with dimensions 512 x 512 x 512 units. The environment has a total of 216 obstacles, each of size 2 x 64 x 64 units as shown in Figure 8.2(b). In this experiment, we are interested in understanding the quality of the parallel roadmap graph when obstacles are present in an environment. Each of the evaluation metrics previously

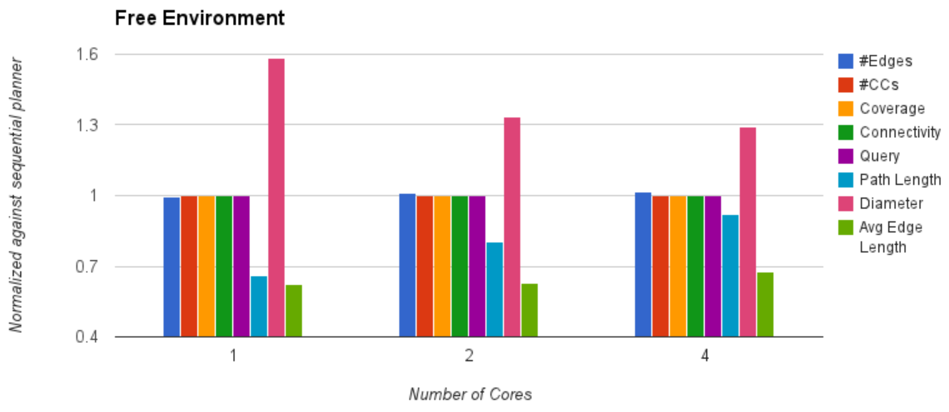


Figure 8.3: Quality evaluation in free environment

discussed were evaluated against the roadmap graph generated using sequential planner. Similar to the previous experiment, we fixed the sample size at 800 nodes, 2 regions per processor and varied the number of processor counts from 1 to 4. We observed that most quality evaluation metrics for both sequential and parallel (almost) matched. The parallel planner mapped the space with a single connected component that equals the graph size. The sequential computed a roadmap that has two connected components one with 799 nodes and the other a singleton. The parallel graphs also have more edges compared to the sequential graphs. We observed that the diameter of the parallel graphs are much longer than the sequential graphs. One way to explain this could be that there are longer edges in the longest shortest path of the parallel roadmap graphs compared to sequential roadmap graphs. This could be a fair assumption as our experiment shows that while the average edge lengths of both graphs are almost the same, the maximum edge length of parallel graph ranges from 2.6 times (2.6x) to 4.6 times (4.6x) the sequential graph.

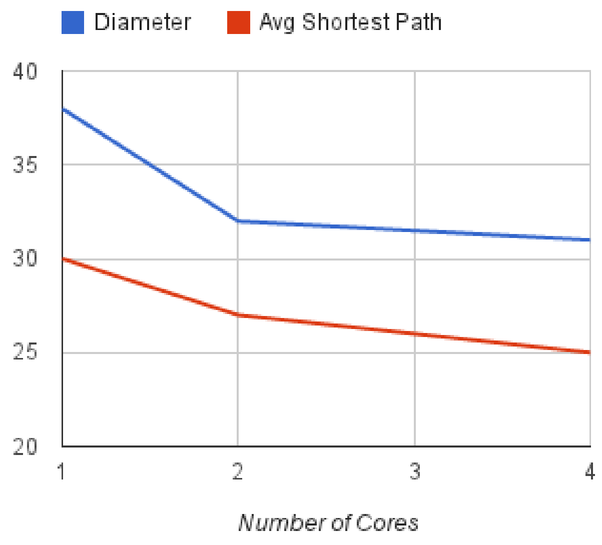


Figure 8.4: Relationship between diameter and average shortest paths

8.3.2.3 2D Clutter Environment

We conducted another experiment in a 2D clutter environment (shown in Figure 8.2(b)). We observed similarity in most of the evaluation metrics comparing the parallel and sequential graphs. Results from our experiment are shown in Figure 8.8. The noticeable exceptions are in the diameter, and the number and size of the connected components. While there is a close similarity in the diameter of the largest connected components, we observed that the sum of the diameters of the parallel roadmap graph is at most 1.9 times (1.9x) that of the sequential roadmap graph. Even though the average edge lengths for the parallel and the sequential graph are closely matched, we observed that the maximum edge length of the parallel graph is about 3 times (3x) that of the sequential graph, this observation could possibly explain the difference in the sum of the diameters for both graphs. Both sequential and parallel planners were successful in solving the witness queries. However, we

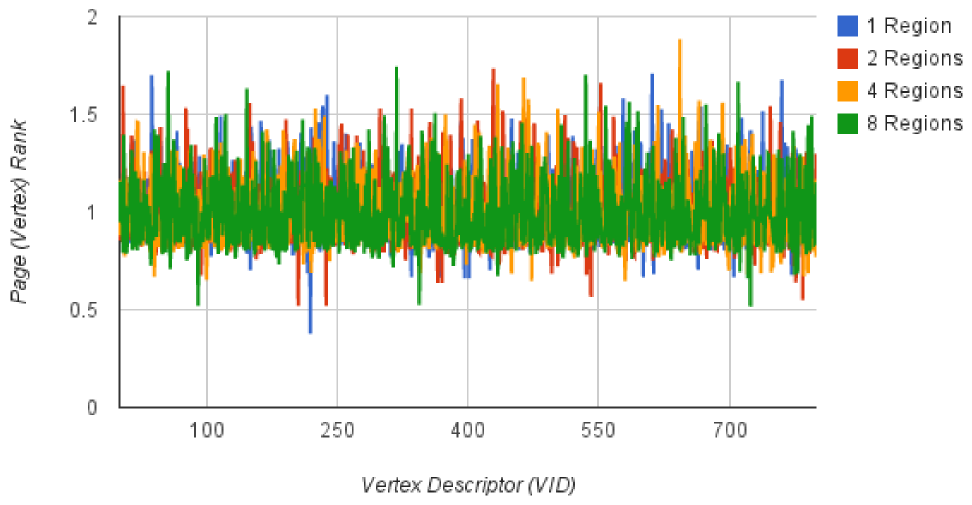


Figure 8.5: Page (vertex) rank for different region subdivision

observed that in most cases, the parallel planner found better or shorter paths than the sequential planner. Pictures of paths produced by both planners are shown in Figure 8.9 and Figure 8.10.

8.3.2.4 Maze Environment

The next environment we studied was a 3D maze environment shown in Figure 8.2(d). Similar to the previous experiments, we kept the number of samples generated fixed at 800 nodes, the number of regions at 2 per processor and varied the processors counts from 1 to 4. The results for each metric were normalized against results from sequential planner as shown in Figure 8.11. We observed that for many of the evaluation metrics, the results are similar to or at par with roadmaps generated by the sequential planner. Both parallel and sequential planners built graphs with similar coverage, connectivity, and with the ability to solve witness queries. The

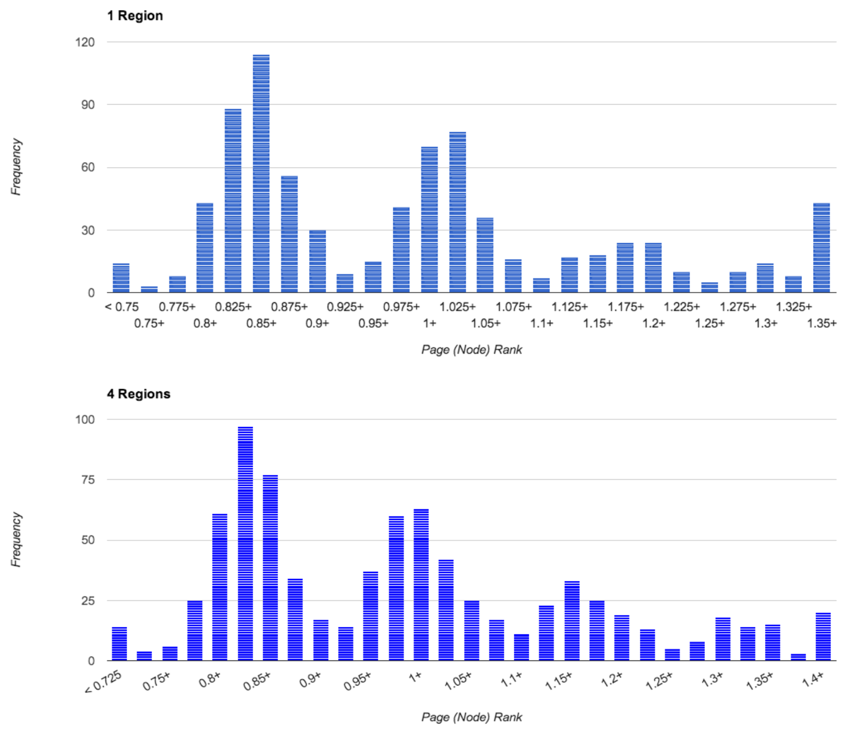


Figure 8.6: Page (vertex) rank distributions for 1 and 4 regions

observable differences are in the number and size of the connected components as well as the diameter of the graphs. The parallel graph made 1.5 times (1.5x) to 2.5 times (2.5x) the number of connected components (CCs) compared to the sequential planner. However, the size of the largest CC varies a little more on 4 processors and a little less on 2 processors. Given that there is more than one CC for both sequential and parallel graphs, we evaluate the diameter of the graph for both the largest connected component (max diameter) and the sum of the diameter for all non-singleton connected components. We observed that the roadmap graphs generated using our subdivision based parallel processing framework have larger diameters. In a difficult environment such as the 3-D maze we could reduce the number of connected components using other region connection methods that are known to work well in

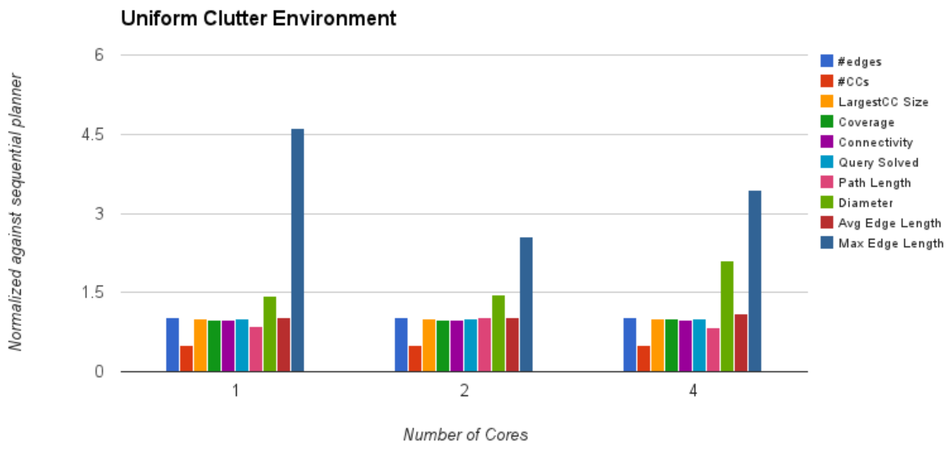


Figure 8.7: Quality evaluation in 3D clutter environment

expanding connected components in difficult environment.

8.4 Heterogeneous Environment: A Natural Fit for Spatial Subdivision and Parallelism

The heterogeneous environment underscores the importance of motion planning in a real-world scenario. Most realistic environments for motion planning problems are not homogenous. Rather, they are composed of subproblems that may be homogeneous. These types of heteregenous environments are a natural fit for our proposed framework in which we subdivide the planning space into regions, assign the regions to processors to work on as subproblems and then combine solutions to each subproblems, to form a solution for the entire problem. As discussed in the related work section in Chapter 2, there have been many different motion planning algorithms that extend the original basic planning algorithms [42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53]. These algorithms focus on dealing with particular instances of motion planning problems using different heuristics. For

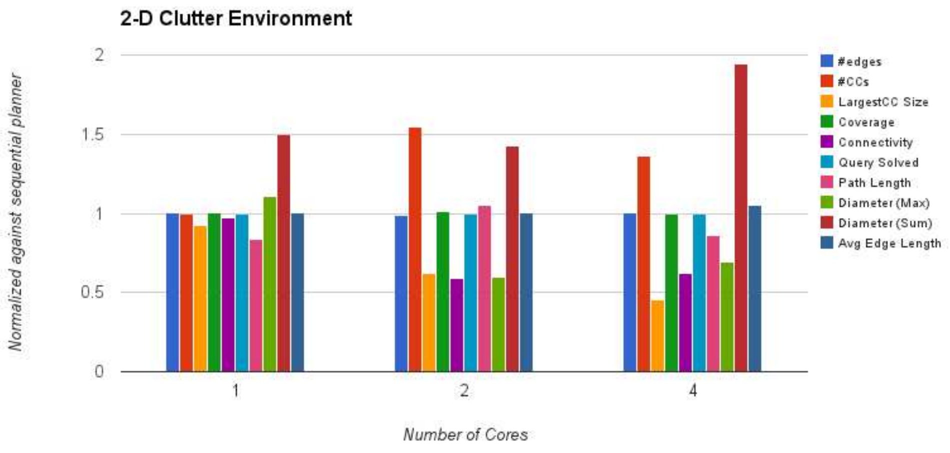
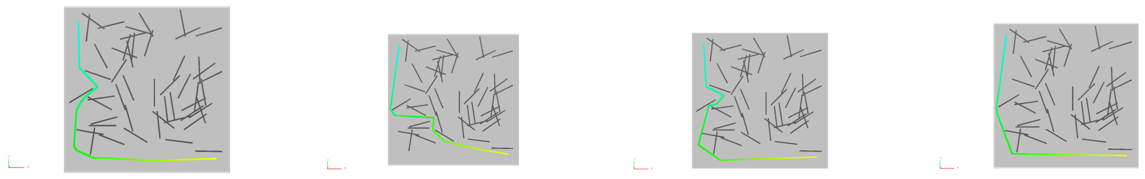


Figure 8.8: Quality evaluation in 2D clutter environment

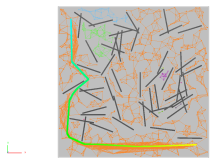


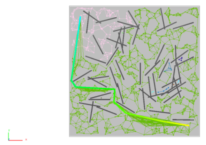
(a) Sequential (b) P1 (c) P2 (d) P4

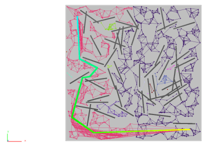
Figure 8.9: Paths for (a) sequential planner, and (b-d) parallel planner at different processor counts

instance, obstacle-based PRM (OBPRM) [42, 46] uses information about the obstacle space to deal with narrow passage problems or medial-axis PRM (MAPRM) [43] is effective when clearance from obstacles is needed. Likewise, many studies have considered region identification and adaptive planning [64]. Adaptive planning identifies and maps appropriate sampling techniques to a region of an heterogenous enviroment. Other approaches have also explored hybrid planning [91] in such a way that existing planners are combined when needed to provide a better solution.

We leverage the idea of region classification and adaptive planning to underline


(a) Sequential


(b) P1


(c) P2

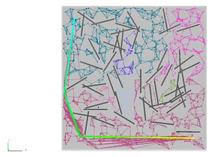

(d) P4

Figure 8.10: Roadmap and paths for (a) sequential planner, and (b-d) parallel planner at different processor counts

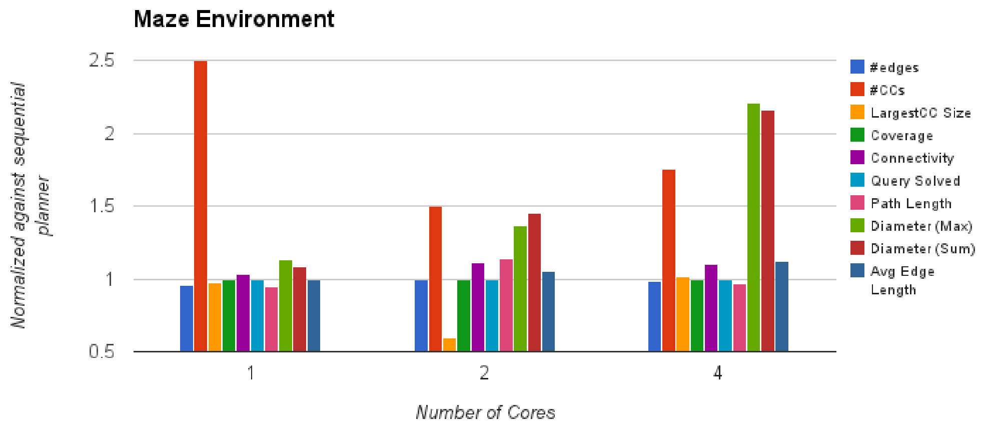


Figure 8.11: Quality evaluation in maze environment

the significance of our framework. The idea of adaptive sampling which researchers have proposed over the years will find usefulness in our spatial subdivision parallel framework. Problems involving heterogeneous environments are a natural fit for our proposed framework; such problems are large-scale and are suitable for any spatial subdivision method such as proposed in this dissertation. Moreover, using our approach on a large-scale heterogeneous problems benefits from the scalability that is possible parallel processing.

8.4.1 Adaptive Sampling and Connection

We apply the work in [64, 34] to identify regions so as to map appropriate samplers to a region and to also adaptively select appropriate neighbor selection method for node connection in a region. In Algorithm 17, we show a modified version of the original algorithm presented in Chapter 4. The overall approach is still essentially the same but differs in two ways. First, a region classification process is inserted between region construction and roadmap construction. Second, the roadmap construction now uses an adaptive node connection (ANC) algorithm [34] for node connection. Sequel to region graph construction and prior to roadmap construction, we classify each region as either free, blocked, narrow, surface, transition or unknown. As part of region classification, we map an appropriate sampling method to the region based on the region type. This sampling method will be used in constructing a regional roadmap in the regional roadmap construction phase.

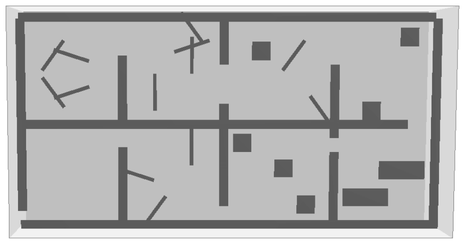
Algorithm 17 Parallel Sampling-Based Motion Planning with Region Classification

```
Input: Region graph $R(V, E)$.
Output: Roadmap graph $G$
1: for all $v \in V$ par do
2:     $sampler \leftarrow$ ClassifyRegion($v$)
3: end for
4: for all $v \in V$ par do
5:     $G \leftarrow$ ConstructRegionalRoadmap($v$, $v_{sampler}$)
6: end for
7: for all $e \in E$ par do
8:     $G \leftarrow$ ConnectRegionalRoadmap($e_{source}$, $e_{target}$)
9: end for
```

To identify a given region, we applied the entropy-based region classification model proposed in [64]. The entropy-based model classifies a region based on a measure of disorder of the training sample (configuration) in the region. This measure is based on a validity test of the training samples. Each region could be potentially classified to have high or low entropy. For instance, regions containing samples that are completely free (valid) or completely blocked (invalid) are considered to have low entropies. Regions with a mixture of both free and blocked samples are considered to have high entropies. Regions with high entropies are also likely to be classified as narrow, transition or surface. Further refinement may be required in most cases and this could involve further sampling beyond initial coarse sampling before a decision on type or class of a region is made. This refinement improves the fidelity of the classifier.

8.4.2 Experimental Results

In this section, we present experimental results that show the significance of the parallel spatial subdivision planning for heterogenous environments. The environments used in this experiment are shown in Figure 8.12. Each of the environments shown is a combination or mixture of different homogeneous environments ranging from free, clutter, narrow passage and blocked. In the 2D environment (Figure 8.12(a)), a rod-like robot must traverse a series of free, narrow and cluttered


(a) 2D

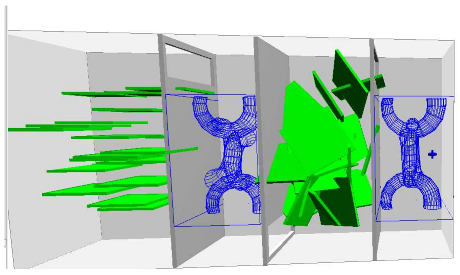

(b) 3D

Figure 8.12: Heterogeneous environments

environments starting from the bottom left to the top left. In the 3D environment (Figure 8.12(b)), a large spinning (spherical) robot would also have to traverse a series of narrow passages and cluttered environments starting from the bottom left to the top right of the environment.

For the experiments reported here, we subdivide the environments into 16 regions and apply the region classification algorithm to classify each region and map a suitable sampler to each region. Figure 8.13 shows a plot of the number of regions that uses a particular sampler for both the 2D environment and 3D environment. Information about the sampler to use is stored as part of the region properties in the region graph. This information is later used in the regional roadmap construction phase. From Figure 8.13, we observe that the region classifier does in fact return more than one sampling strategy for the heterogenous environment.

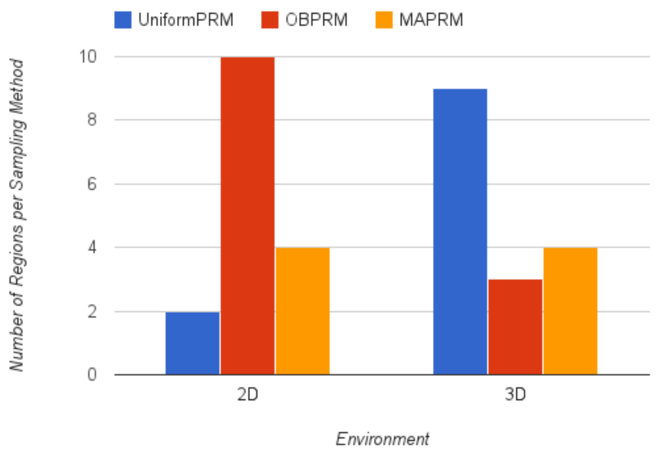


Figure 8.13: Region classification : number of regions per sampler for both 2D and 3D heterogenous environments

We evaluate the quality of the roadmap constructed using our framework with roadmaps constructed by the sequential algorithms. In comparing with the sequential planner, we used each of the three sampling strategies that is contained in set of samplers for parallel planners (e.g., Uniform PRM (UniformPRM), Obstacle-based PRM (OBPRM), and Medial-axis PRM (MAPRM)). Our approach (PSBMP) used all of the sampling strategies in an adaptive manner (i.e., it adaptively selects which sampler is appropriate for each region). In the node connection phase, the adaptive node connection strategy (ANC) was used. The adaptive node connection strategy adaptively selects an appropriate connector for each region after it has "learnt" which connector is suitable for the region.

In the 2D environment, we observed a superior performance of our approach (PSBMP) across almost all metrics in comparison to the sequential versions. The reason for this superior performance can be explained in two ways. First, mapping an appropriate node generation method to each region improves the quality of samples generated because of the inherent advantage of applying an appropriate node generation method for the region. The second issue is the benefit of using an adaptive node connection (ANC) method. This benefit comes from the fact that the heterogeneous environment is already subdivided into almost homogeneous regions. Therefore, it was easier for ANC to quickly learn an appropriate connector for the region leading to an increase in the number of edges and better connectivity. This is not the case with the sequential planner without spatial subdivision, thus the learning process using the sequential planner without subdivision incurs more penalty than reward leading to possibly lower edge counts and relatively poor connectivity. The result from this experiment is shown in Figure 8.14. From the figure, we observe that our approach (PSBMP) made more edges than Uniform PRM and OBPRM, and fewer connected components than OBPRM and MAPRM. Our approach has better connectivity because the connected component with largest size is about 1/8 less than the graph size of 1600 nodes. This is not the case with other methods. The size of the largest connected component in the roadmap constructed using PSBMP is 1.8 times (1.8x), 6 times (6x), and 4 times (4x) that of roadmaps constructed using Uniform PRM, OBPRM and MAPRM, respectively. PSBMP produces roadmap graphs with better connectivity and shorter diameter than the other methods. We observed the same trend in the 3D environment (results shown in Figure 8.15), except that MAPRM is now more competitive compared to what we saw in the 2D environment. Still, our results indicate that PSBMP has superior performance across all metrics of interest in both 2D and 3D heterogeneous environments.

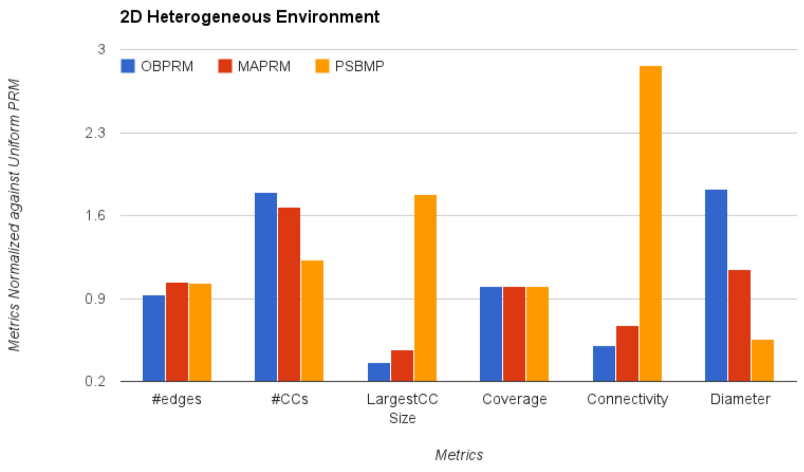


Figure 8.14: Quality evaluation in 2D heterogenous environment

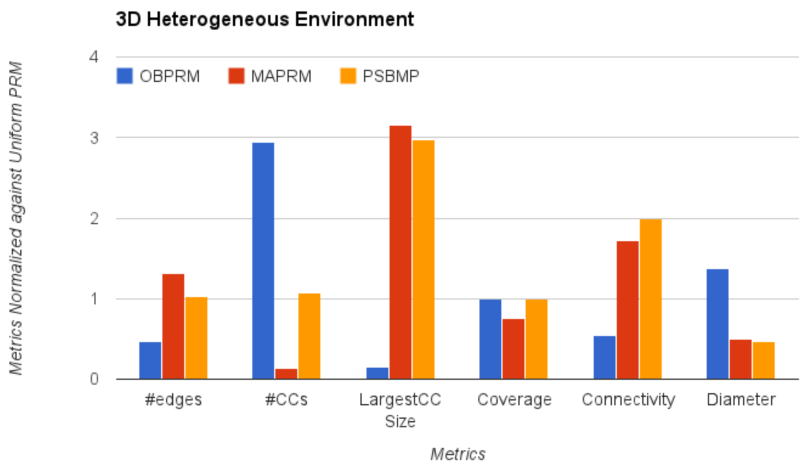


Figure 8.15: Quality evaluation in 3D heterogenous environment

9. CONCLUSION

The need for solving large problems within an acceptable time frame is at the center of demand for parallel computing. Numerous areas of computing and various applications now require such solutions. One such area is motion or path planning. While motion planning has its roots in robotics, it now finds applications in other areas of scientific computing including protein folding, minimally-invasive surgical planning and drug design and virtual prototyping and computer-aided design. These application areas test the limit and capability of existing sequential motion planners, motivating the need for methods that can exploit parallel processing.

In this dissertation, we present a scalable framework for parallelizing sampling-based motion planning algorithms. Central to our method is the novel subdivision of the planning space into regions and an abstraction of the relationship between regions called a region graph $R(V, E)$. The vertices, V, of the region graph represent the regions and the edges, E, represent adjacencies between regions. Having subdivided the planning space into regions, each region is assigned to a processor to work on independently (and in parallel) as a subproblem. The task or subproblem for each processor is to build a roadmap (graph) or tree approximating the topology of the planning space. Solutions to the subproblems are later combined to form a solution to the entire problem. This combination is facilitated using the region graph. The region graph is the enabling infrastructure facilitating the process of connecting the region roadmaps as it aids identification of adjacent regions between which connections are attempted.

By subdividing the planning space and restricting the locality of connection attempts, we reduce the work and inter-processor communication associated with nearest neighbor calculation, thus enabling scalable results and better performance compared to previous methods. In addition, we address the problem of load balancing in complex planning spaces. In addressing the load balancing problems, we applied standard load balancing techniques based on data-structure redistribution and work stealing and show the effectiveness of the two techniques at alleviating load balancing problems that arise at scale. Furthermore, we carried out an experimental evaluation of our framework to study the structural differences of the resulting roadmaps in comparison to those produced by sequential planners, and the impact of these differences on the solutions to motion planning problems.

Unlike previous work, the work presented in this dissertation covers the two broad classes of sampling-based motion planning: the graph-based (e.g., probabilistic roadmap method (PRM)) and the tree-based (e.g., rapidly-exploring random tree (RRT)) methods. Although we used a general framework, we explored different planning space subdivision approaches suitable for the two classes of sampling-based motion planning. We provide both theoretical and empirical proof of scalable and superior performance compared to previous methods. We present experimental results obtained from our studies of a wide range of motion planning problems utilizing different parallel architectures; ranging from small-scale linux clusters to an IBM Power5+ machine to Cray XE6 petascale machine. In particular, we show that our proposed method results in a more scalable and load-balanced computation on a single-node with 8 cores up to a distributed shared-memory of 3000+ cores.

Future work will extend our current approach such that we can more efficiently handle more complex environments and attempt higher dimensional problems. In the future, we would also like to explore parallel algorithms for dealing with motion planning under uncertainty, motion planning in dynamic environments, and parallel algorithms to handle both motion and task planning in a single framework. While these areas are currently being explored in the sequential domain, it will be worthwhile to explore parallel algorithms that could deal with these problems in an efficient manner.

REFERENCES

[1] M. Apaydin, A. Singh, D. Brutlag, and J.-C. Latombe, "Capturing molecular energy landscapes with probabilistic conformational roadmaps," in *Proc. IEEE Int. Conf. Robot. Autom. (ICRA)*, 2001, pp. 932–939.

[2] N. M. Amato and G. Song, "Using motion planning to study protein folding pathways," *J. Comput. Biol.*, vol. 9, no. 2, pp. 149–168, 2002, Special issue of Int. Conf. Comput. Molecular Biology (RECOMB) 2001.

[3] X. Tang, B. Kirkpatrick, S. Thomas, G. Song, and N. M. Amato, "Using motion planning to study RNA folding kinetics," in *Proc. Int. Conf. Comput. Molecular Biology (RECOMB)*, 2004, pp. 252–261.

[4] J. C. Latombe, "Motion planning: A journey of robots, molecules, digital actors, and other artifacts," *Int. Journal of Robotics Research*, vol. 18, no. 11, pp. 1119–1128, 1999.

[5] A. P. Singh, J.-C. Latombe, and D. L. Brutlag, "A motion planning approach to flexible ligand binding," in *Int. Conf. on Intelligent Systems for Molecular Biology (ISMB)*, 1999, pp. 252–261.

[6] O. B. Bayazit, G. Song, and N. M. Amato, "Ligand binding with OBPRM and haptic user input: Enhancing automatic motion planning with virtual touch," in *Proc. IEEE Int. Conf. Robot. Autom. (ICRA)*, 2001, pp. 954–959, This work was also presented as a poster at *RECOMB 2001*.

[7] G. Song and N. M. Amato, "Using motion planning to study protein folding pathways," in *Proc. Int. Conf. Comput. Molecular Biology (RECOMB)*, 2001, pp. 287–296.

[8] L. Kavraki and J. C. Latombe, "Randomized preprocessing of configuration space for fast path planning," in *Proc. IEEE Int. Conf. Robot. Autom. (ICRA)*, 1994, pp. 2138–2145.

[9] Y. Koga, K. Kondo, J. Kuffner, and J. Latombe, "Planning motions with intentions," in *Proc. ACM SIGGRAPH*, 1995, pp. 395–408.

[10] O. B. Bayazit, J.-M. Lien, and N. M. Amato, "Better flocking behaviors using rule-based roadmaps," in *Proc. Int. Workshop on Algorithmic Foundations of Robotics (WAFR)*, Dec 2002, pp. 95–111.

[11] H. Chang and T. Y. Li, "Assembly maintainability study with motion planning," in *Proc. IEEE Int. Conf. Robot. Autom. (ICRA)*, 1995, pp. 1012–1019.

[12] O. B. Bayazit, G. Song, and N. M. Amato, "Enhancing randomized motion planners: Exploring with haptic hints," in *Proc. IEEE Int. Conf. Robot. Autom. (ICRA)*, 2000, pp. 529–536.

[13] E. Plaku and L. E. Kavraki, "Distributed sampling-based roadmap of trees for large-scale motion planning," *IEEE Transactions on Robotics and Automation*, vol. 38, pp. 793–884, 2005.

[14] J. H. Reif, "Complexity of the mover's problem and generalizations," in *Proc. IEEE Symp. Foundations of Computer Science (FOCS)*, San Juan, Puerto Rico, October 1979, pp. 421–427.

[15] H. Choset, K. M. Lynch, S. Hutchinson, G. A. Kantor, W. Burgard, L. E. Kavraki, and S. Thrun, *Principles of Robot Motion: Theory, Algorithms, and Implementations*, MIT Press, Cambridge, MA, June 2005.

[16] L. E. Kavraki, P. Švestka, J. C. Latombe, and M. H. Overmars, "Probabilistic roadmaps for path planning in high-dimensional configuration spaces," *IEEE*

Trans. Robot. Automat., vol. 12, no. 4, pp. 566–580, August 1996.

[17] S. Thomas, G. Tanase, L. K. Dale, J. M. Moreira, L. Rauchwerger, and N. M. Amato, "Parallel protein folding with STAPL," *Concurrency and Computation: Practice and Experience*, vol. 17, no. 14, pp. 1643–1656, 2005.

[18] M. Gini, "Parallel search algorithms for robot motion planning," *Workshop on Practical Motion Planning in Robotics: Current Approaches and Future Directions, IEEE Conference on Robotics and Automationificial Intelligence*, 1996.

[19] P.Isto, "A two level search algorithm for motion planning," *in Proceedings International Conference on Advanced Robotics*, pp. 2025–2031, 1997.

[20] P.Isto, "A parallel motion planner for systems with many degrees of freedom," *in Proceedings International Conference on Advanced Robotics*, pp. 339–344, 2001.

[21] N. M. Amato and L. K. Dale, "Probabilistic roadmap methods are embarrassingly parallel," in *Proc. IEEE Int. Conf. Robot. Autom. (ICRA)*, 1999, pp. 688–694.

[22] S. Carpin and E. Pagello, "On parallel RRTs for multi-robot systems," in *Proc. Italian Assoc. AI*, 2002, pp. 834–841.

[23] J. Bialkowski, S. Karaman, and E. Frazzoli, "Massively parallelizing the RRT and the RRT*," in *Proc. IEEE Int. Conf. Intel. Rob. Syst. (IROS)*, 2011.

[24] D. Devaurs, T. Simeon, and J. Cortes, "Parallelizing RRT on distributed-memory architectures," in *Proc. IEEE Int. Conf. Robot. Autom. (ICRA)*, 2011.

[25] M. Akinc, K. E. Bekris, B. Y. Chen, A. M. Ladd, E. Plaku, and L. E. Kavraki, "Probabilistic roadmaps of trees for parallel computation of multiple query

roadmaps," in *Proceedings of International Symposium on Robotics Research*, Sienna, Italy, October 2003.

[26] E. Plaku, K. E. Bekris, B. Y. Chen, A. M. Ladd, and L. E. Kavraki, "Sampling-based roadmap of trees for parallel motion planning," *IEEE Trans. Robot. Automat.*, 2005.

[27] S. A. Jacobs and N. M. Amato, "From days to second: Scalable parallel algorithms for motion planning," in *ACM Student Research Compet, Conf. on High Performance Computing, Networking, Storage and Analysis Companion Proceedings*, Seattle, WA, USA, 2011, SC '11.

[28] 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 *Proc. IEEE Int. Conf. Robot. Autom. (ICRA)*, 2012.

[29] S. A. Jacobs, N. Stradford, C. Rodriguez, S. Thomas, and N. M. Amato, "A scalable distributed RRT for motion planning.," in *Proc. IEEE Int. Conf. Robot. Autom. (ICRA)*, 2013.

[30] E. Plaku and L. Kavraki, "Distributed computation of the knn graph for large high-dimensional point sets," *Journal of Parallel and Distributed Computing*, vol. 67, no. 3, 2007.

[31] V. Garcia, E. Debreuve, and M. Barlaud, "Fast k nearest neighbor search using gpu," in *CVPR Workshop on Computer Vision on GPU, Anchorage, Alaska, USA*, 2008.

[32] S. M. LaValle and J. J. Kuffner, "Randomized kinodynamic planning," in *Proc. IEEE Int. Conf. Robot. Autom. (ICRA)*, 1999, pp. 473–479.

[33] A. Fidel, S. A. Jacobs, S. Sharma, N. M. Amato, and L. Rauchwerger, "Using load balancing to scalably parallelize sampling-based motion planning algorithms," in *Proc. International Parallel and Distributed Processing Symposium (IPDPS)*, Phoenix, Arizona, USA, May 2014.

[34] C. Ekenna, S. A. Jacobs, S. Thomas, and N. M. Amato, "Adaptive neighbor connection for prms: A natural fit for heterogeneous environments and parallelism," in *Proc. IEEE Int. Conf. Intel. Rob. Syst. (IROS)*, November 2013, pp. 1–8.

[35] C. Rodriguez, J. Denny, S. A. Jacobs, S. Thomas, and N. M. Amato, "Blind RRT: A probabilistically complete distributed RRT," in *Proc. IEEE Int. Conf. Intel. Rob. Syst. (IROS)*, November 2013.

[36] I. Al-Bluwi, T. Simeon, and J. Cortes, "Motion planning for molecular simulations; a survey," *Computer Science Review*, vol. 6, no. 4, pp. 125–143, 2012.

[37] T. Horsch, F. Schwarz, and H. Tolle, "Motion planning for many degrees of freedom – random reflections at c-space obstacles," in *Proc. IEEE Int. Conf. Robot. Autom. (ICRA)*, 1994, pp. 3318–3323.

[38] K. K. Gupta and Z. Guo, "Motion planning for many degrees of freedom: Sequential search with backtracking," *IEEE Trans. Robot. Automat.*, vol. 11, no. 6, pp. 897–906, 1995.

[39] D. Hsu, J.-C. Latombe, and R. Motwani, "Path planning in expansive configuration spaces," *Int. J. Comput. Geom. & Appl.*, pp. 495–517, 1999.

[40] N. M. Amato, O. B. Bayazit, L. K. Dale, C. V. Jones, and D. Vallejo, "Choosing good distance metrics and local planners for probabilistic roadmap methods," *IEEE Trans. Robot. Automat.*, vol. 16, no. 4, pp. 442–447, August 2000.

[41] J. Denny and N. M. Amato, "The toggle local planner for sampling-based motion planning," in *Proc. IEEE Int. Conf. Robot. Autom. (ICRA)*, St. Paul, Minnesota, USA, May 2012, pp. 1779–1786.

[42] Y. Wu, "An obstacle-based probabilistic roadmap method for path planning," M.S. thesis, Department of Computer Science, Texas A&M University, 1996.

[43] S. A. Wilmarth, N. M. Amato, and P. F. Stiller, "MAPRM: A probabilistic roadmap planner with sampling on the medial axis of the free space," Tech. Rep. TR98-022, Dept. of Computer Science, Texas A&M University, College Station, TX, Nov 1998.

[44] V. Boor, M. H. Overmars, and A. F. van der Stappen, "The Gaussian sampling strategy for probabilistic roadmap planners," in *Proc. IEEE Int. Conf. Robot. Autom. (ICRA)*, May 1999, vol. 2, pp. 1018–1023.

[45] J. Denny and N. M. Amato, "Toggle PRM: Simultaneous mapping of C-free and C-obstacle - a study in 2D -," in *Proc. IEEE Int. Conf. Intel. Rob. Syst. (IROS)*, San Francisco, California, USA, Septempber 2011, pp. 2632–2639.

[46] H.-Y. C. Yeh, S. Thomas, D. Eppstein, and N. M. Amato, "UOBPRM: A uniformly distributed obstacle-based PRM," in *Proc. IEEE Int. Conf. Intel. Rob. Syst. (IROS)*, Vilamoura, Algarve, Portugal, 2012, pp. 2655–2662.

[47] D. Hsu, J.-C. Latombe, and R. Motwani, "Path planning in expansive configuration spaces," in *Proc. IEEE Int. Conf. Robot. Autom. (ICRA)*, 1997, pp. 2719–2726.

[48] N. M. Amato and Y. Wu, "A randomized roadmap method for path and manipulation planning," in *Proc. IEEE Int. Conf. Robot. Autom. (ICRA)*, 1996, pp. 113–120.

[49] R. Bohlin and L. E. Kavraki, "Path planning using Lazy PRM," in *Proc. IEEE Int. Conf. Robot. Autom. (ICRA)*, 2000, pp. 521–528.

[50] L. Guibas, C. Holleman, and L. Kavraki, "A probabilistic roadmap planner for flexible objects with a workspace medial-axis-based sampling approach," in *Proc. IEEE Int. Conf. Intel. Rob. Syst. (IROS)*, 1999, vol. 1, pp. 254–259.

[51] M. H. Overmars and P. Švestka, "A probabilistic learning approach to motion planning," in *Algorithmic Foundations of Robotics*. A. K. Peters, Wellesley, MA, 1995.

[52] J. Denny, K. Shi, and N. M. Amato, "Lazy toggle PRM: A single-query approach to motion planning," in *Proc. IEEE Int. Conf. Robot. Autom. (ICRA)*, Karlsruhe, Germany, May 2013, pp. 2407–2414.

[53] S. Karaman and E. Frazzoli, "Sampling-based algorithms for optimal motion planning," *International Journal of Robotics Research (IJRR)*, vol. 30, pp. 846–894, 2011.

[54] S. M. LaValle and J. J. Kuffner, "Randomized kinodynamic planning," *Int. J. Robot. Res.*, vol. 20, no. 5, pp. 378–400, May 2001.

[55] S. M. LaValle and J. J. Kuffner, "Rapidly-exploring random trees: Progress and prospects," in *New Directions in Algorithmic and Computational Robotics*, pp. 293–308. A. K. Peters, 2001, book contains the proceedings of the International Workshop on the Algorithmic Foundations of Robotics (WAFR), Hanover, NH, 2000.

[56] J. J. Kuffner and S. M. LaValle, "RRT-connect: An efficient approach to single-query path planning," in *Proc. IEEE Int. Conf. Robot. Autom. (ICRA)*, 2000, pp. 995–1001.

[57] S. Rodriguez, X. Tang, J.-M. Lien, and N. M. Amato, “An obstacle-based rapidly-exploring random tree,” in *Proc. IEEE Int. Conf. Robot. Autom. (ICRA)*, 2006.

[58] D. Henrich, “Fast motion planning by parallel processing - a review,” *Journal of Intelligent and Robotic Systems*, vol. 20, no. 1, pp. 45–69, 1997.

[59] Y. K. Hwang and N. Ahuja, “Gross motion planning – a survey,” *ACM Computing Surveys*, vol. 24, no. 3, pp. 219–291, 1992.

[60] R. A. Brooks and T. Lozano-Pérez, “A subdivision algorithm in configuration space for findpath with rotation,” in *Proc. Int. Conf. Artif. Intel.*, 1983, pp. 799–806.

[61] L. Zhang, Y. Kim, and D. Manocha, “A hybrid approach for complete motion planning,” in *IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2007, pp. 7–14.

[62] M. Morales, L. Tapia, R. Pearce, S. Rodriguez, and N. M. Amato, “A machine learning approach for feature-sensitive motion planning,” in *Algorithmic Foundations of Robotics VI*, pp. 361–376. Springer, Berlin/Heidelberg, 2005, book contains the proceedings of the International Workshop on the Algorithmic Foundations of Robotics (WAFR), Utrecht/Zeist, The Netherlands, 2004.

[63] M. A. Morales A., L. Tapia, R. Pearce, S. Rodriguez, and N. M. Amato, “C-space subdivision and integration in feature-sensitive motion planning,” in *Proc. IEEE Int. Conf. Robot. Autom. (ICRA)*, April 2005, pp. 3114–3119.

[64] S. Rodriguez, S. Thomas, R. Pearce, and N. M. Amato, “(RESAMPL): A region-sensitive adaptive motion planner,” in *Algorithmic Foundation of Robotics VII*, pp. 285–300. Springer, Berlin/Heidelberg, 2008, book contains the proceed-

ings of the International Workshop on the Algorithmic Foundations of Robotics (WAFR), New York City, 2006.

[65] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, “Cilk: an efficient multithreaded runtime system,” in *Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming*, New York, NY, USA, 1995, PPOPP ’95, pp. 207–216, ACM.

[66] Intel, *Reference Manual for Intel Threading Building Blocks, version 1.13*, 2009.

[67] T. El-Ghazawi and L. Smith, “Upc: unified parallel c,” in *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, New York, NY, USA, 2006, SC ’06, ACM.

[68] R. D. Blumofe and C. E. Leiserson, “Scheduling multithreaded computations by work stealing,” *J. ACM*, vol. 46, no. 5, pp. 720–748, Sept. 1999.

[69] Y. Guo, J. Zhao, V. Cave, and V. Sarkar, “Slaw: a scalable locality-aware adaptive work-stealing scheduler for multi-core systems,” in *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, New York, NY, USA, 2010, PPoPP ’10, pp. 341–342, ACM.

[70] S. jai Min, C. Iancu, and K. Yelick, “Hierarchical work stealing on manycore clusters,” in *In Fifth Conference on Partitioned Global Address Space Programming Models*, 2011.

[71] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, “X10: an Object-Oriented Approach to Non-Uniform Cluster Computing,” in *Annual ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, New York, NY, USA, 2005, pp. 519–538, ACM Press.

[72] S. Olivier, J. Huan, J. Liu, J. Prins, J. Dinan, P. Sadayappan, and C.-W. Tseng, "Uts: an unbalanced tree search benchmark," in *Proceedings of the 19th international conference on Languages and compilers for parallel computing*, Berlin, Heidelberg, 2007, LCPC'06, pp. 235–250, Springer-Verlag.

[73] D. Callahan, B. L. Chamberlain, and H. P. Zima, "The Cascade High Productivity Language," in *The Ninth Int. Workshop on High-Level Parallel Programming Models and Supportive Environments*, Los Alamitos, CA, USA, 2004, vol. 26, pp. 52–60.

[74] L. V. Kale and S. Krishnan, "CHARM++: A portable concurrent object oriented system based on C++," *SIGPLAN Not.*, vol. 28, no. 10, pp. 91–108, 1993.

[75] K. Devine, E. Boman, R. Heaphy, B. Hendrickson, and C. Vaughan, "Zoltan data management services for parallel dynamic applications," *Computing in Science and Engineering*, vol. 4, no. 2, pp. 90–97, 2002.

[76] G. Karypis and V. Kumar, "Parallel multilevel k-way partitioning scheme for irregular graphs," in *Proceedings of the 1996 ACM/IEEE conference on Supercomputing (CDROM)*, Washington, DC, USA, 1996, Supercomputing '96, IEEE Computer Society.

[77] C. Walshaw and M. Cross, "JOSTLE: Parallel Multilevel Graph-Partitioning Software – An Overview," in *Mesh Partitioning Techniques and Domain Decomposition Techniques*, F. Magoules, Ed., pp. 27–58. Civil-Comp Ltd., 2007, (Invited chapter).

[78] 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 *Int. Workshop on Languages and Compilers for Parallel Computing (LCPC),*

in Lecture Notes in Computer Science (LNCS), Houston, TX, USA, September 2010.

[79] A. Buss, Harshvardhan, I. Papadopoulos, O. Pearce, T. Smith, G. Tanase, N. Thomas, X. Xu, M. Bianco, N. M. Amato, and L. Rauchwerger, "STAPL: Standard template adaptive parallel library," in *Proc. Annual Haifa Experimental Systems Conference (SYSTOR)*, New York, NY, USA, 2010, pp. 1–10, ACM.

[80] N. Thomas, G. Tanase, O. Tkachyshyn, J. Perdue, N. M. Amato, and L. Rauchwerger, "A framework for adaptive algorithm selection in STAPL," in *Proc. ACM SIGPLAN Symp. Prin. Prac. Par. Prog. (PPoPP)*, Chicago, IL, USA, 2005, pp. 277–288, ACM.

[81] S. Saunders and L. Rauchwerger, "ARMI: an adaptive, platform independent communication library," in *Proc. ACM SIGPLAN Symp. Prin. Prac. Par. Prog. (PPoPP)*, San Diego, California, USA, 2003, pp. 230–241, ACM.

[82] D. Musser, G. Derge, and A. Saini, *STL Tutorial and Reference Guide, Second Edition*, Addison-Wesley, 2001.

[83] Harshvardhan, A. Fidel, N. M. Amato, and L. Rauchwerger, "The stapl parallel graph library," in *Languages and Compilers for Parallel Computing*, Lecture Notes in Computer Science, pp. 46–60. Springer Berlin Heidelberg, 2012.

[84] M. Morales, S. Rodriguez, and N. M. Amato, "Improving the connectivity of PRM roadmaps," in *Proc. IEEE Int. Conf. Robot. Autom. (ICRA)*, September 2003, vol. 3, pp. 4427–4432.

[85] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008.

[86] D. Hsu, J.-C. Latombe, and H. Kurniawati, "On the probabilistic foundations of probabilistic roadmap planning," *Int. J. Robot. Res.*, vol. 25, pp. 627–643, July 2006.

[87] S. Chung and A. Condon, "Parallel implementation of boruvka's minimum spanning tree algorithm," *Parallel Processing Symposium, International*, vol. 0, pp. 302, 1996.

[88] T. McMahon, S. Jacobs, B. Boyd, L. Tapia, and N. M. Amato, "Local randomization in neighbor selection improves prm roadmap quality," in *Proc. IEEE Int. Conf. Intel. Rob. Syst. (IROS)*, 2012.

[89] R. Geraerts, "Sampling-based motion planning: Analysis and path quality," Ph.D. thesis, Utrecht University, 2006.

[90] S. Brin and L. Page, "The anatomy of a large-scale hypertextual web search engine," *Comput. Netw. ISDN Syst.*, vol. 30, no. 1-7, pp. 107–117, 1998.

[91] D. Hsu, G. Sánchez-Ante, and Z. Sun, "Hybrid PRM sampling with a cost-sensitive adaptive strategy," in *Proc. IEEE Int. Conf. Robot. Autom. (ICRA)*, 2005, pp. 3885–3891.