Path Planning for Altruistically Negotiating Processes

Deepthi Devalarazu
Department of Computer Science
Utah State University, Logan, Utah 84322-4205
deepthi@cc.usu.edu

Daniel W. Watson
Department of Computer Science
Utah State University, Logan, Utah 84322-4205
dan.watson@usu.edu

ABSTRACT

Autonomous Negotiating Systems are composed of logically (even geographically) separated software agents that control logical or physical resources that altruistically seek to perform useful work in a cooperative manner. These systems are Multi-Agent systems that consist of a population of autonomous agents collaborating to work for a common goal while simultaneously performing their individual tasks (i.e., computational resources are distributed amongst interconnected agents). With the increasing capabilities of the collaborative agents, the need for faster and more efficient methods of utilizing the distributed resources has also increased. This paper focuses on improving the performance of one such multi-agent system that deals with the path planning for autonomous robots. This is achieved by exploiting parallelism among processing resources embedded in the autonomous vehicles, using a distributed memory, message-passing execution model.

KEYWORDS: Cluster-Based Collaborative & Distributed Systems, Coordination, Cooperation, Collaboration, Coordination and Cooperation Mechanisms, Multi-Agent Systems, Rapidly Exploring Random Trees (RRTs).

1. INTRODUCTION

In recent years there has been extensive research done in the area of control and coordination of cooperating agents [1, 2, 3, 6, 7, 8], particularly those agents that supervise Unmanned Autonomous Vehicles (UAVs). In order to increase the performance of the multiple agents working together, they are often divided into teams of agents and group goals or 'missions' are allocated to different teams. Tasks that achieve these missions are assigned to various team members. Finally, a path is designed for each team member that achieves the tasks while not violating constraints related to time, space and the capabilities of the agents. UAVs may be responsible for performing their own path planning tasks using on-board dedicated processors, or they may be able to share path-planning activities among all UAVs, taking advantage of computational resources that are otherwise idle on UAVs that either have already determined their paths, or are currently idle. Thus, they altruistically seek to perform useful work in a cooperative manner. The capabilities of cooperating agents are evolving, leading to the need of faster and better strategies for coordination and control.

One of the most persistent problems in the area of coordination and control has been that of the path planning for UAVs. Several approaches such as use of potential functions [4], or more recent ones, such as dynamic programming [2] and Mixed Integer Linear Programming [1], have been proposed over the years to solve the problem of path planning for autonomous vehicles. As calculating collision-free paths in the presence of complicated constraints is considered to be a class of NP-Hard problems [11], research is often diverted to several randomized approaches [6, 7] for finding paths in high-dimensional spaces. The problem with these approaches is that, although powerful for standard path planning, they do not naturally extend to the general non-holonomic problems [16] or kinodynamic problems [10] (i.e., those problems that include both first-order differential constraints and obstacle-based global constraints, such as considering velocity and position).

One of the first randomized techniques proposed to solve the problems that have non-holonomic or kinodynamic constraints was Rapidly Exploring Random trees [9].

Rapidly exploring Random Trees (RRTs) [8, 9] have shown good results and have been applied to several real world problems. An RRT is a data structure and algorithm that incrementally explores the state space taking into consideration both global as well as differential constraints. The underlying concept is to explore new portions of the space by randomly sampling points in the state space and incrementally pulling the search tree towards them. RRTs are constructed incrementally in a way that quickly reduces the expected distance of a randomly chosen point to the tree. A path from the source to destination is found in this way for a single robot.

Although this process of coordinated multiple robots is a time-consuming task, the performance can be improved by introducing parallelism. One such strategy is to implement RRTs in a shared-memory multiprocessor environment [14] by applying the classic OR approach (running several identical problems in parallel), or embarrassingly parallel approach (where multiple processors simultaneously modify a single tree), or a hybrid approach using both these processors.

For this paper, the execution environment is assumed to be one of local memory message-passing processing elements that are geographically distributed such as that encountered by UAVs, and the assumption of shared-memory, tightly coupled processors capability is removed. Because of its distributed environment in which communication is relatively expensive, a distributed RRT algorithm is proposed that significantly reduces the overall time needed to determine a path that addresses the kinodynamic constraints of the problem. The distributed algorithm seeks to exploit processors on UAVs that are not otherwise engaged in path planning of their own. Numerical results have been obtained to show a marked speed-up when compared to the sequential algorithm and the classic parallel approaches.

The rest of the paper is organized as follows: Section 2 introduces current related works by other investigators in this field, most specifically those examining randomized techniques. Section 3 examines the implementation of the distributed RRT algorithm for Altruistically Negotiating Agents. This section also illustrates the results for the various test cases and compares the base-line algorithm with the sequential as well as the parallel versions of the algorithm. Summarizing and concluding remarks are provided in Section 4.

2. BACKGROUND

Over the years one focus in motion planning is to compute collision-free paths where the configuration space consists of stationary as well as movable objects in the presence of complicated constraints similar to the problems of path planning observed for the above mentioned scenarios. This class of problems being NP-hard [11, 12], various randomized approaches were applied that solve the general problems of path planning in high-dimensional configuration spaces but at the cost of completeness.

Among those approaches, one of the early and most popular ones is the Probabilistic Road Maps [12] method, more commonly known as the Monte Carlo method. In this approach, random configurations are generated and a local planner is used to connect pairs of configurations. The A* algorithm is then used to find the path in the road map and control inputs (e.g., steering angle) are taken from the path. Although this method works quite well for systems with holonomic or non-holonomic constraints where the robots are freely steerable, it fails in scenarios with complicated non-holonomic and dynamic constraints. The solution may not be impossible, but this approach requires connection of thousands of configurations which may seem impractical. A different strategy is to use a randomized data structure called Rapidly exploring Random Trees (RRT) [13], which is aimed at solving problems arising from non-holonomic and dynamic systems. RRTs incorporate some desirable features of the probabilistic road maps. It gains advantage from the fact that it does not require connections between the pairs of configurations. As explained shortly, RRTs induce dynamic nature in them by growing a tree which is biased towards the goal, not by deciding the path statically but expanding the tree at every node giving it the robustness to change its path according to the various available constraints.

2.1 Construction of RRT

The construction of the rapidly exploring random trees begins with generating a random point in the configuration space (C_{free}) where C_{free} can be defined as the space not including the obstacle space C_{obs}. A new point is found which is at a distance less than or equal to the RRT step size on the line joining the random point *q_rand* and the nearest point *q_near* to *q_rand* in the already existing tree. This process of adding a new point (node) to the already existing RRT is called the *expansion process*.
After expansion, the newly found point is tested for obstacle avoidance. If it collides with existing obstacles, a new point closest to this point but at a safe distance is found and added to the tree. A new edge is then added

from *q_near* to the newly added point *q_new*. The RRT is thus expanded in this manner until *q_target* is reached or is close enough to the new point.

2.2 Static Obstacle Avoidance

This process works in conjunction with the RRT expansion where in every time a new point is generated, it is checked if it is colliding with the obstacles. If it does then a point on the line joining *q_near* and *q_new* but at the closest safe position is calculated. Consider an already existing tree an obstacle that lies in the path of *q_new*. This condition is tested by checking if the new point generated lies within the polygon. If it does then the intersection point of the polygon and the line joining *q_near* and *q_new* is found and, based on the size of the robot, a safe distance is left between the new point and the obstacle. Figure 1 shows the algorithm for the basic RRT Construction.

One way to improve the performance of the RRT is to parallelize the construction process. In [14], parallelism is introduced in two main ways. In the first approach, all the processors run the same instance of the problem; the process is terminated when the first processor comes up with a solution. This is called the OR paradigm approach. In the second approach, called the embarrassingly parallel approach, every processor contributes to the same problem using a shared memory model. A hybrid approach is also employed where the same problem is solved by different groups using the embarrassingly parallel approach. The process is terminated when the first group finds the solution.

3. DISTRIBUTED RRT

In this paper, local memory message-passing Distributed RRT instead of a shared memory model. This model is more typical in the environments where UAVs are responsible for their own path planning tasks using embedded processors. In such an environment it is desirable to take advantage of the fact that not all UAVs are calculating paths at all times; thus it is possible to harness the computational resources on UAVs whose path planning resources are idle. In this study this is achieved by distributing the task of constructing the tree to available processors at regular intervals.

The construction of RRT is a step-wise process where at each step a new node is added to an already existing tree. Thus at regular intervals (i.e., after a set of nodes have been added to the tree), the problem of growing the tree from that particular point to the destination is distributed to the available processors. This process is continued until one processor arrives at a solution.

```
1. RRT_EXTEND(t, q_rand)
2.      INPUT Tree constructed so far and random
   point
3.      RETURN Trapped or True or Continue
4.      q_near = q_init
5.      q_new = Get_New_Config ( )
6.      colliding = CHECK_COLLISION (q_new)
7.      if colliding = true
8.         q_new = Get_Safe_Point( )
9.      End if
10.     t.add_vertex ( q_new)
11.     t.add_edge ( q_near, q_new)
12.     if q_new = q_target
13.        RETURN true
14.     else if node_count > Limit
15.        RETURN Trapped
16.     else
17.        RETURN  Continue
1.  CALC_RRT (q_init, q_target)
2.  INPUT starting and goal points'
    q_init and q_target
3.  Get_Obstacles ( );
4.  for n = 1 to N do
5.  q_rand <- RANDOM_POINT
6.  reached = RRT_EXTEND ( );
7.  if NOT reached = Trapped then
8.           if reached = true
9.                    GET_PATH ( )
10.          End if
11. End if
12. End for
13. RETURN Trapped
```

Figure 1. Basic RRT

Communication is required among processors to send the tree found so far and the destination. When a solution is found by one processor, communication is required to inform all the other processors to stop executing. Once they receive a message with the termination tag, the execution is stopped. The algorithm for the distributed RRT Construction is given in Figure 2. The Distributed_RRT takes three inputs. The original tree constructed so far, and the initial and target positions of the robot. This function is run initially on processor 0, which starts growing the tree at each step by calling the extend function. The Distributed_RRT_Extend function, shown in Figure 3, takes the tree generated so far and grows the tree by adding a new vertex and an edge to the tree. This function checks to see if the target is reached and returns reached, trapped or continue accordingly. The

tree being grown on processor 0 is distributed to other processors after every ten nodes added to the tree. The following code shows this:

if (node_count % 10 = 0)
DISTRIBUTED_RRT (t, q_new, q_target)

This process is continued until a solution tree is found or there are no more available processors.

```
1.  DISTRIBUTED_RRT (t, q_init, q_target)
2.  INPUT tree constructed so far, starting and goal
    points q_init, q_target
3.  if Processor_Rank = 0
4.  While (termination message not yet received) do
5.   q_rand <- RANDOM_POINT
6.    reached = DISTRIBUTED_RRT_EXTEND (t,
    q_rand )
7.   if NOT reached = Trapped then
8.     if reached = true
9.       for n = 1 to num_of_processors
10.        SEND Termination Message
11.      End for
12.      GET_FULL_PATH ( )
13.    End if
14.   End if
15.  End While
16. End if
```

Figure 2. Distributed RRT

3.1 Simulation Results

All the numerical results shown in this section are achieved by running parallel algorithms on a 17 node, Linux Beowulf cluster. This cluster consists of 17 Pentium-III nodes where each of the PCs has four 3Com 3c905C 10/100mbs fast Ethernet cards wired in hypercube architecture [17], 9 Gigabytes of memory and has 128 MB RAM. The communication among the processors is achieved using the Message Passing Interface (MPI). The algorithm was developed in C++. For all experiments an environment of 100x100 points is assumed for the configuration space (C_{free}). The obstacle space C_{obs} can have obstacles of varying size and number which can be defined at the beginning of the program. For the experiments below the obstacle space is assumed to have three rectangular obstacles, scattered over the configuration space at the following locations.

Test Case
Obstacle 1: (20, 20) (50, 20) (50, 40) (20, 40)
Obstacle 2: (0, 80) (40, 80) (40, 100) (0,100)
Obstacle 3: (70, 0) (100, 0) (100, 30) (70, 30)

For each test case, the distributed RRT algorithm was compared to the performance of a baseline 'OR Paradigm' algorithm, where the same problem is executed by several processors and the process is terminated when one processor finds the solution has an improved speed when compared to the sequential approach. Figure 4 illustrates the algorithm for the OR Paradigm approach used in this study.

```
1.  DISTRIBUTED_RRT_EXTEND (t, q_rand)
2.  INPUT Tree t, constructed so far and random
    point q_rand
3.  RETURN Trapped, True or Continue
4.  t.add_vertex ( q_init)
5.  q_near = q_init
6.  node_count = 1
7.  number_of_used_Processors = 0
8.  q_new = Get_New_Config ( )
9.  colliding = CHECK_COLLISION (q_new)
10. if colliding = true
11.       q_new = Get_Safe_Point ( )
12. End if
13. t.Add vertex (q_new)
14. t.Add_edge (q_near, q_new)
15. INCREMENT node_count
16. if Processor_Rank = 0
17.  if (node_count % 10 = 0)
18.   if num_of_used_processors < num_of
    processors
19.   DISTRIBUTED_RRT (t, q_new, q_target)
20.     INCREMENT num_of_used_processors
21.    End if
22.   End if
23. End if
24. if q_new = q_target
25.   RETURN true
26. else if node_count > LIMIT
27.   RETURN Trapped
28. else
29.   RETURN Continue
30. End if
```

Figure 3. Distributed RRT Extend

The OR Paradigm approach was executed for a series of runs using 5, 10 and 15 processors and establishes a baseline for the Distributed RRT examples. The performance of this parallel approach was measured as follows. The raw clock time is recorded for each scenario (run) of the simulation. The same scenario is used for a 5-processor, 10-processor, and 15-processor run, and the three times are recorded for that same run number in the graph. The same data when reordered with respect to clock time within each implementation size, the graph shown in Figure 5 is the result. In this graph, termed the rank ordered run time of the OR Paradigm approach, it is

easier to see that, in general, adding more processors to the RRT problem results in shorter execution times, although there is certainly an aspect of diminished returns as more processors are added. Note that, although the execution time for a particular run may actually take longer for a 15 versus 10 versus 5 processor implementation, in general a larger number of processors used will decrease the overall execution time.

```
1.  OR_PARADIGM (q_init, q_target)
2.  INPUT starting and goal points q_init and
    q_target
3.  While (termination message not yet
    received) do
4.   q_rand <- RANDOM_POINT
5.   reached = RRT_EXTEND (t, q_rand );
6.   if NOT reached = Trapped then
7.    if reached = true
8.     for n = 1 to num_of _processors
9.       SEND Termination message
10.    End for
11.    GET_PATH ( )
12.   End if
13.  End if
14. End While
```

Figure 4. OR Paradigm RRT

The speed-up of the OR Paradigm algorithm can be calculated using the formula *Speed-Up = Sequential_Time/ Parallel_Time*. The performance of the OR Paradigm algorithm can be measured by the following figure showing the speed-up calculated when the algorithm was run on 5, 10 and 15 processors for fifty runs each. Figure 6 thus shows the considerable improvement in time when compared to the sequential approach.

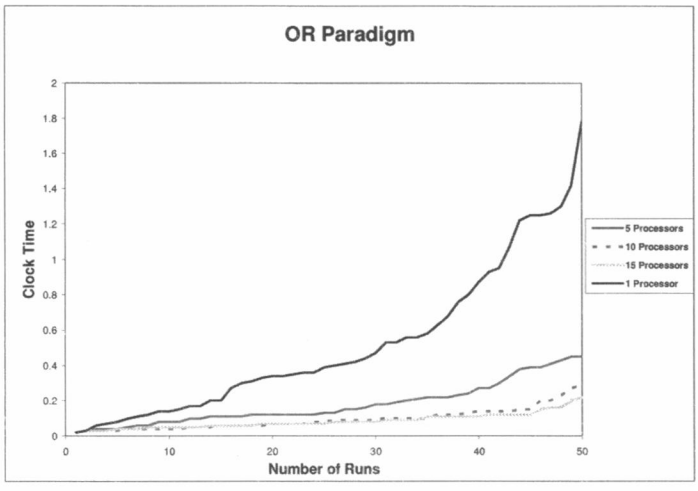

Figure 5. Rank-Ordered Run-time of OR Paradigm Approach

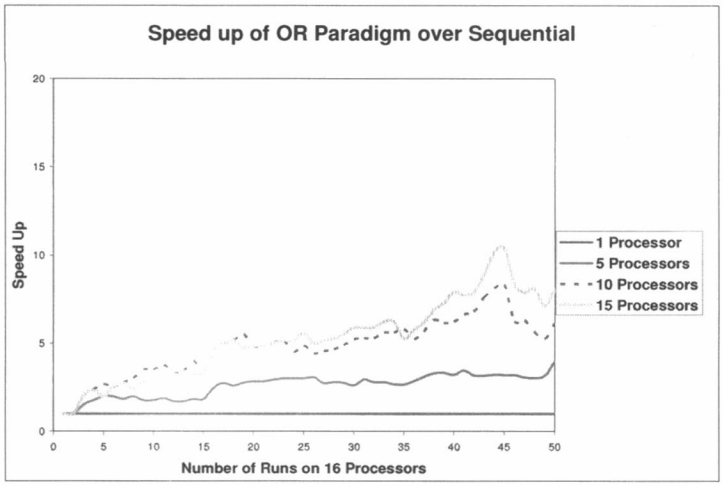

Figure 6. Speed-up of OR Paradigm Approach Over Sequential Approach

After the baseline study, experiments were conducted to observe the performance of the distributed RRT algorithm, compared to the sequential version. Again, the rank-ordered execution times for Distributed RRT simulations are calculated as shown in Figure 7. Although there is less of a distinction between the 5, 10, and 15-porocessor runs, it still holds true that, in general, adding more parallelism can improve the overall time needed to find a solution.

Finally the Distributed RRT algorithm was used to solve the same problem as the baseline case. The results listed in Figure 8 are typical.

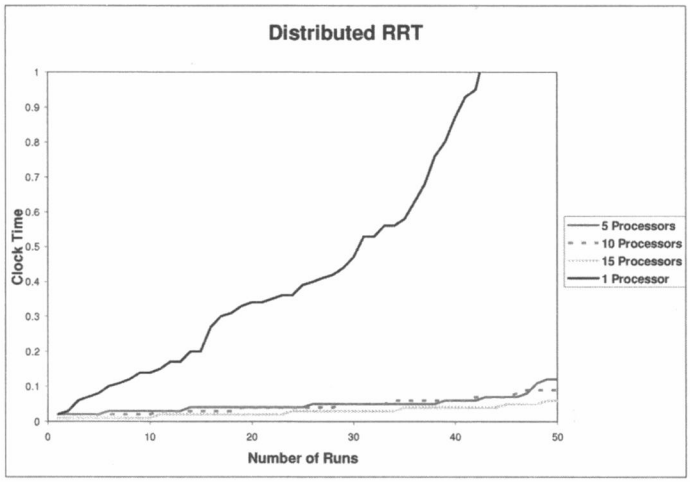

Figure 7. Run-Time Ordering Distributed RRT

The speed-up of the distributed RRT over the sequential algorithm was calculated using the formula *Speed-up = Sequential_Time/ Distributed_RRT* time. This graph indicates that there is a considerable amount of improvement in the performance of the RRT algorithm.

Similar to the OR Paradigm approach the speed-up was calculated over 5, 10 and 15 processors for about 50 runs each. In Figure 9, the rank-ordered execution time results are compared to the OR Paradigm approach. For the results shown, sixteen processors were used for both OR Paradigm and the distributed RRT approach.

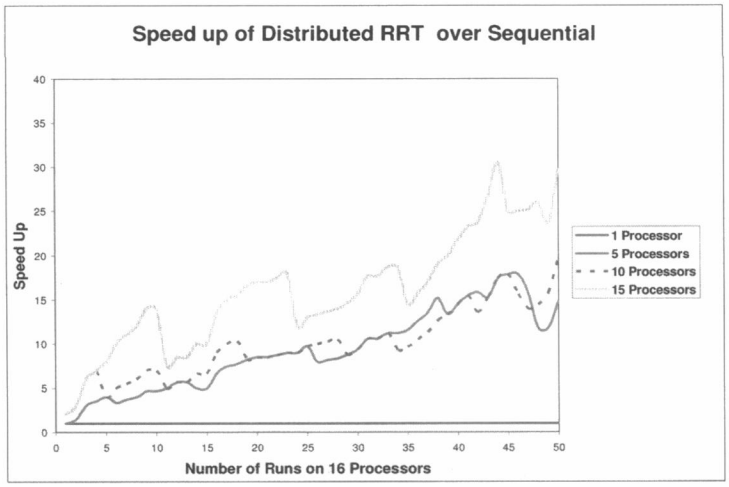

Figure 8. Speed-up of Distributed RRT Approach Over Sequential Approach

As indicated in the Figure 9 there is a visible improvement in the distributed RRT algorithm when compared to the OR Paradigm approach and the sequential approach.

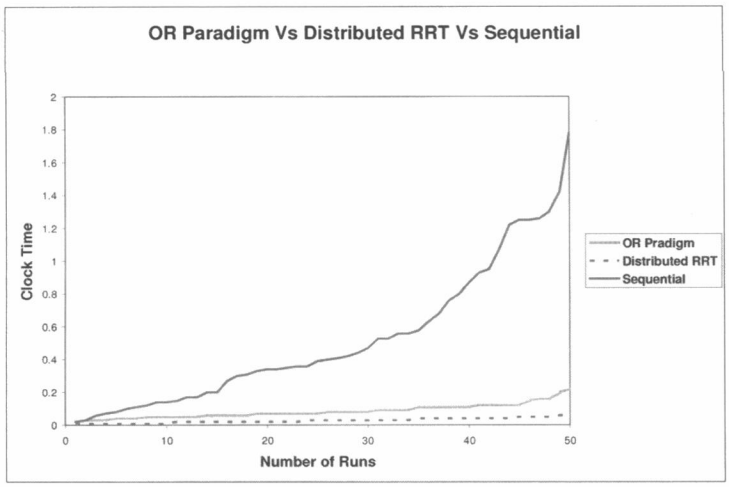

Figure 9. Run-Time Ordering OR Paradigm vs. Distributed RRT vs. Sequential

4. CONCLUSION

In this paper, a distributed processor parallel version of the RRT algorithm was introduced that can significantly improve the performance of the algorithm. This is achieved by distributing the problem to available processors at regular intervals of time. A comparison study indicates that this Distributed RRT algorithm can achieve superior results over the replicated problem OR Paradigm RRT approach. The significance of this is that it is possible to independent multiple processing elements to solve path planning problems more quickly, taking advantage of otherwise idle processing resources.

REFERENCES

[1] Schouwenaars, T., B. De Moor, E. Feron and J. P. How, "Mixed Integer Programming for Multi-Vehicle Path Planning," Proc. European Controls Conference, Seminário de Vilar, Porto, Portugal, September 2001, pp. 1-6.

[2] Flint, M., M. Polycarpou, and E. Fernández-Gaucherand, "Cooperative Path-Planning for Autonomous Vehicles Using Dynamic Programming," Proc. International Federation of Automatic Control 15th IFAC World Congress, Barcelona, Spain, July 2002, pp. 1-6.

[3] Richards, A., J. Bellingham, M. Tillerson, and J. P. How, "Coordination and Control of multiple UAVs," Proc. AIAA Guidance, Navigation and Control Conference, Monterey, CA, USA, August 2002, pp. 1-19.

[4] Latombe, J. C., Robot Motion Planning, Kluwer Academic Publishers, 1991.

[5] Goodrich, M. T., and R. Tamassia, Algorithm Design, John Wiley & Sons, Inc., 2002.

[6] Amatao, N. M., and Y. Wu. "A randomized roadmap method for path and manipulation planning," Proc. IEEE Int. Conf. Robot. & Autom., Minneapolis, MN, USA, April 1996, pp. 76-82.

[7] Kavraki, L. E., P. Svestka, Jean-Claude Latombe, and M. H. Overmars, "Probabilistic roadmaps for path planning in high-dimensional configuration spaces," IEEE Transactions on Robotics & Automation, Vol. 12, No. 4, August 1996, pp. 1-15.

[8] LaValle, S. M., "Rapidly exploring random trees: A new tool for path planning", *TR 98-11*, Computer Science Dept., Iowa State University, October 1998.

[9] LaValle, S. M. and J. J. Kuffner, "Randomized kinodynamic planning," Proc. IEEE Int'l Conf. on Robotics and Automation, Detroit, Michigan, USA, May 1999, pp. 473-479.

[10] Donald, B. R., P. Xavier, J. Canny, and J. Reif, "Kinodynamic Motion Planning", *ACM*, Vol. 40, No. 5, Nov., 1993. pp. 1048-1066.

[11] Reif, J. H., "Complexity of the mover's problem and generalizations," Proc. IEEE. Symp. on Foundat. of Comp. Sci., 1979, pp. 421–427.

[12] Cheng, p. and S. M. LaValle. "Resolution complete rapidly-exploring random trees." Proc. IEEE Int'l Conf. on

Robotics and Automation, Arlington, VA, USA, May 2002, pp. 1-8.

[13] Cheng, P., Z. Shen, and S. M. LaValle. "RRT-based trajectory design for autonomous automobiles and spacecraft," *Archives of Control Sciences*, Vol. 11(XLVII), No. 3-4, 2001, pp. 167-194.

[14] Carpin, S. and E. Pagello. "On parallel RRTs for multi-robot systems," Proc. 8th Conference of the Italian Association for Artificial Intelligence, Siena, Italy, Sept 2002, pp. 1-8.

[15] LaValle, S.M., Planning Algorithms, University of Illinois, 2004.

[16] Barraquand, J., and J. C. Latombe, "Nonholonomic Multibody Mobile Robots: Controllability and Motion Planning in the Presence of Obstacles," Algorithmica, Vol. 10, 1993, pp. 121 – 155.

[17] Valiant, L. G., "General Purpose Parallel Architectures," J. van Leeuwen ed., *Handbook of Theoretical Computer Science*, Vol. A, The MIT Press/Elsevier, New York, 1990.