Scalable Parallel RRT Method for Motion Planning

Nick Stradford[1], Sam Ade Jacobs[2], Nancy M. Amato[2]
nickstradford@gmail.com, sjacobs@cse.tamu.edu, amato@cse.tamu.edu

***Abstract*— Motion planning is not only an important component of robotics, but also bio-informatics. As these fields progress and problem sizes increase, faster approaches need to be created. Parallel computation can be used to significantly speed up the computation. This paper studies the effects of parallelizing the Rapidly-exploring Randomized Tree (RRT) method, one of the two types of state-of-the-art sampling-based algorithms for solving motion planning problems. RRT helps solve motion planning problems by creating a tree to locally explore in C-Space. One implementation of parallel RRT updates the global tree immediately, which has a large communication overhead. The strategy proposed in this paper allows the user to specify how much of the tree will be created locally before updating the global tree shared amongst the processes. This paper experiments with various values of this k-parameter that is used to control the granularity of the local growth.**

I. Introduction

Motion Planning can be defined as the problem of trying to get a start configuration to a goal configuration while avoiding obstacles [6]. Besides robotics, it is also used in computer animation [2], [11] and bio-informatics [14], [15], [3]. In robotics it can be used to create a way for a robot to navigate from configuration A to configuration B. In bio-informatics, it can be used to study protein folding problems. Common solutions to the Motion Planning problem utilize a popular sampling based technique known as Probabilistic Roadmaps (PRM) [10]. PRMs are able to effectively create maps for multiple-query path planning. For single-query path planning, the Rapidly-exploring Randomized Tree (RRT) [12] method has been the most popular choice. RRTs quickly create a tree-like structure throughout C-Space by learning and exploring at the same time.

PRMs implement two phases in their application. These phases are the learning phase and the query phase. The learning phase is the part of the algorithm in which samples are created in C-Space and stored. Upon completion of the sampling part of the learning phase, the nodes that were sampled are then connected to each other to form a roadmap of C-Space. Now the roadmap can be used to answer queries. This is done by placing a start configuration and a goal configuration onto the roadmap and calculating the path needed to get from the start to the finish. Because of their learning phase, PRMs can reuse their roadmap for multiple queries.

RRTs take a different approach in handling the motion planning problem. Essentially, a tree starts growing from the start configuration node towards the goal configuration node. At the beginning of a single tree RRT implementation, the root node is the start configuration and the tree grows incrementally node by node towards the goal configuration. RRTs merge learning about the environment and creating a path into the same step so they are better suited for single-query planning.

As efficient as the PRM and RRT methods are, they can still be improved by parallelizing them. PRMs were shown to be efficiently parallel in [1], [9]. This helped prove parallelization was the next step in improving results in motion planning. Yet, parallelizing code brings in factors unique to parallel programming. One of these factors is inter-process communication. If an excessive amount of communication is done in a parallel algorithm, it can hinder the performance of that algorithm greatly. This paper discusses a parallel RRT method we will call Distributed k-RRT to study how we may be able to increase the efficiency of RRT methods in a way that varies the amount of inter-process communication.

In this paper we will discuss a previously implemented Distributed RRT, and it's design. Secondly, we will describe our Distributed k-RRT method and how it can vary the amount of communication between processes. Thirdly, we will show the experiments we ran and show the results of those experiments. Lastly, we will conclude and talk about future work.

II. Related Work

A. Parallelizing RRT

With technology now being able to execute parallel code and parallel code garnering improved results, motion planning has been attempted to be parallelized in all aspects. In [9], a method is proposed that decomposes environments and partitions these chunks as regions on each processor. Each processor then handles constructing its each region, and then these regions are connected to form one or as few as possible connected components. This was a unique way of parallelizing motion planning.

This research supported in part by NSF awards CRI-0551685, CCF-0833199, CCF-0830753, IIS-096053, IIS-0917266 by THECB NHARP award 000512-0097-2009, by Chevron, IBM, Intel, Oracle/Sun and by Award KUS-C1-016-04, made by King Abdullah University of Science and Technology (KAUST).
The work of Stradford was supported in part by DREU, at Texas A&M University.
[1]Computer Science, University of North Texas, Denton, TX 76201, USA.
[2]Parasol Lab Department of Computer Science and Engineering, Texas A&M University, College Station, TX 77843, USA.

As for directly parallelizing RRT, it was parallelized in [13], making the Sampling-Based Roadmap of Trees (SRT) method. Their method combined the aspects of PRM sampling, with RRT exploration to do local planning. The random samples made throughout C-Space serve as the root nodes of RRTs (called milestones) that will be created in that area. Next, after the milestones are computed, each milestone is compared to other milestones to select the best pairs of milestones that may connect to each other to form an edge in their edge selection phase. Attempting to connect these milestones is done in a couple of ways in their edge computation phase. First, two milestones attempt connection through a straightline planner. Secondly, if that fails, a bi-directional RRT method can be used to connect them and create the egde that is sought after. Their parallel implementation worked by having each processor make a certain number of milestones until k milestones has been created in parallel. Their edge selection process was not parallelized because it took less thatn 3% of the computation time, but they parallelized their edge computation section. The parameters used to set up the SRT method can be tweaked to create replicas of PRM, RRT, or EST [8].

Parallelization is not limited to only CPU based-implementations. The methods presented by [4] were able to implement RRT in parallel using general GPU's. This paper parallelized the RRT method and the RRT* method. A big part of their focus was parallelizing the collision detection step needed to expand a tree. The difference between RRT and RRT* lies in how they expand their trees. Their RRT method uses a straight-line planner to see if a potential path between a node and its nearest node is collision free. If so, it adds the edge created by these two nodes to the tree. The RRT* method was more complex in how it added new nodes to the tree. It analyzed paths created by other neighbors in addition to the path found for the nearest neighbor, so it can try and find a better edge to create.

Another parallelization of RRT was discussed by [7]. This paper was able to implement three parallel versions of RRT. They used the OR paradigm, a manager/worker RRT method, and a Distributed RRT method for the third one. Their OR RRT was the most general. It parallelized RRT by having each processor run its own seperate sequential RRT method. Whenever one of the processors found a solution it stopped the others and returned its result. Their manager/worker RRT classified processors into either managers or workers. Managers delegated work to workers and were the only processors with access to the global tree. The workers in their algorithm waited for work to be given to them. The work given to them was extending the tree by creating new nodes and reporting the information about the new nodes back to the manager.

The third version of RRT [7] experimented with was their Distributed RRT. In their paper, their Distributed RRT made each processor keep a local copy of the overall tree. Whenever a new processor successfully adds a new node to its local copy of the tree, it broadcasts that creation to the other processors so that they can update their copy of the overall tree. Their Distributed RRT alerts every other processor to a new node every time one is created and we aim to study ways of improving or varying this amount of communication. This version of a Distributed RRT is the closest parallel RRT to our implementation.

III. Distributed k-RRT

Our paper proposes a different approach to creating a Distributed RRT than Devaurs et. al [7]. Our method uses the STAPL framework [5] to implement our method. STAPL allows us to create a globally shared graph, so our tree is distributed among the processors being used. Because of this, we do not store a local copy of the overall tree on each processor like the previously implemented version. Also, we introduce a parameter, k, that can be adjusted to vary how often the global graph is updated by one individual processor. The algorithm we used to create our tree is given below:

Algorithm 1 Distributed k-RRT

Input: Environment E, k-Variable k, Total # of Nodes N, # of Processes P.
Output: A tree T.

```
1:  do_once{
2:    T.initialize()
3:    T.root ← GetValidRandomNode(Env)
4:  }
5:  Barrier()
6:  for all proc p ∈ P par do
7:    i ← 0
8:    while i < N/P do
9:      NodeContainer N_c
10:     for j = 1 → k do
11:       Node rand ← GetValidRandomNode(E)
12:       Node nn ← NearNeighbor(T, rand)
13:       Node new ← Extend(nn, rand)
14:       N_c.add(new)
15:     end for
16:     for all Node n ∈ N_c do
17:       if IsValid(n) then
18:         T.AddToTree(n)
19:         i ← i + 1
20:       end if
21:     end for
22:   end while
23: end for
24: Return T.
```

Intitially, we create a root node from which all processors will grow. This is done in Algorithm 1 lines 1-4. Since this only creates one node, we only need one processor to

handle this. This is the reason we labeled this block of the alogrithm ”do_once”. Because we are using one processor to handle the creation of this root node, we need to make sure the other processors wait for this root node to be created. That is what the ”Barrier()” method does.

At this point, we now have the root node in our global tree that every processor will grow from. So, in parallel all processors will create N/P nodes; where N is the total number of nodes that are to be created, and P is the number of processors that will be used. In Algorithm 1 line 9, we instantiate a container that will hold the new local nodes we are about to create. Lines 11-14 of our algorithm are used to do three steps needed to extend our tree. First, we sample a random node that will be the direction we extend our tree. Secondly, we find out which node is the nearest node to the random node we just created. In our case, we have called this nearest node ”nn”. Lastly, we extend our tree from our ”nn” node towards our random node, and then add this new node to our temporary node container. We continue to repeat Algorithm 1 lines 11-14 until we have created k nodes locally on that particular processor. At that point we iterate over our node container, and add each new node to our global tree. These steps are done in lines 16-20. Also, note that in Algorithm 1 line 19, whenever we add a new node to our global graph, we increment our ”i” counter by 1 to keep track of the number of nodes added to the global graph by this processor. The ”i” variable is the stopping condition to determine whenever a processor has successfully made N/P nodes.

A. The k Parameter

The k parameter presented in Algorithm 1 can help limit communication to the overall graph by increasing its size. If k was set to equal the value of N/P, then each individual processor would not have to interact with the global graph until it successfully made one batch of k nodes. This would be effective in reducing communication with the global graph or other processors, but this would cause your root node to be the nearest neighbor node every time you sampled for the nearest neighbor.

This presents the factor you must consider when choosing a value for k. A lower k value has more communication/interaction with the global graph, but it has a more accurate or spread out tree created from it. A higher k value will not communicate with the global graph as much, but it may be forced to use nearest neighbors that wouldn’t be selected if every node created by a processor was already in the global tree. A user will have to think about this while experimenting with k values. One could also note that when using k set to 1, you are using a general Distributed RRT method similar to the Distributed RRT made by Devaurs et. al [7] in which you update your overall tree every time a new node is created.

IV. Experiments

A. Setup

We ran three different experiments for our results. For each experiment, we varied our values for k to be 1, 8, 16, 32, and 64, respectively. Also, we varied the number of processors to be 1, 2, 4, 8, 16, 32, and 64, respectively. The object was to create a tree of 4096 nodes using our Distributed k-RRT algorithm.

B. Environments and Robots

The environment we experimented in was a homogenous cluttered environment. The dimensions of this environment were set to 512x512x512 units with 216 obstacles as seen in Figure 1. Each obstacle had the dimensions of 2x64x64 units. As for the three robots experimented with, the first one was a cube-like rigid body with 6 degrees of freedom. The second was a 8 degree of freedom articulated linkage robot. Lastly, the third was a 16 degree of freedom articulated linkage robot.

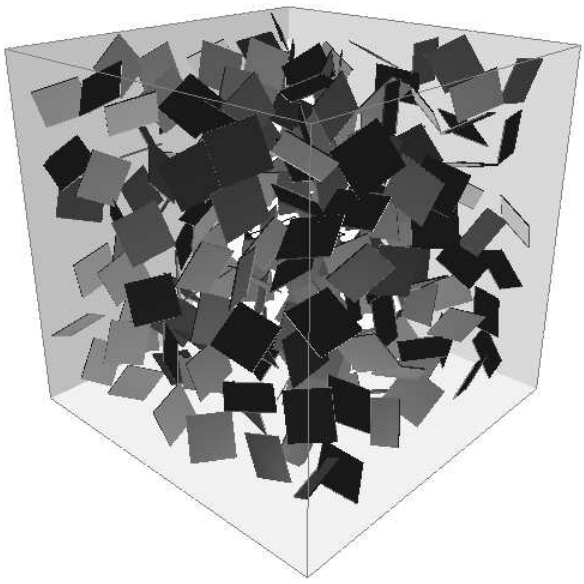

Fig. 1. The scattered environment used in our experiments.

C. Results

For our experiments, the graphs follow the trend of having a faster computation time as the value of k increases, as seen Figures 2, 3, 4. This occurs because of the fact that as k increases, processors have to interact with the global graph less. In our experiments, the higest value of k we used was 64, our highest number of processors used was 64, and we created a total of 4096 nodes. A point to note is that 64 is the square root of 4096. So, in our tests, when k and the number of processors were set to 64, we executed the fastest computation time. This occured

because each processor always creates N/P nodes, and in those cases N/P was equal to 64. So each processor had to make its N/P amount to satisfy its k value, which caused each processor to only update the global graph one time.

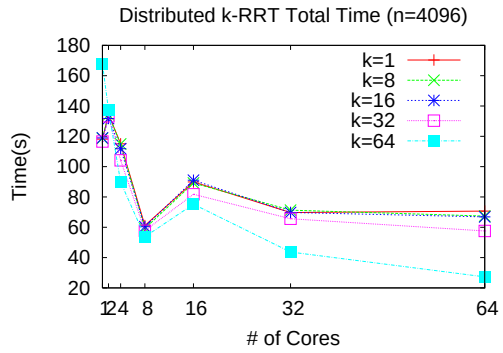

Fig. 2. Test results for a rigid cube shaped robot.

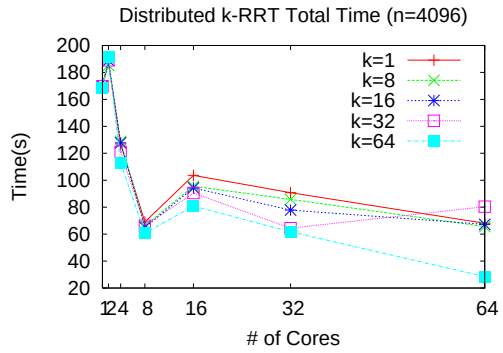

Fig. 3. Test results for a 8 degree of freedom articulated linkage robot.

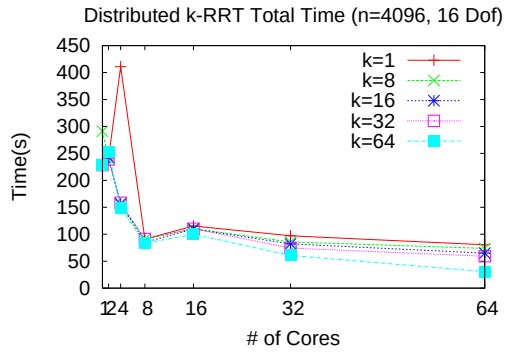

Fig. 4. Test results for a 16 degree of freedom articulated linkage robot.

V. Conclusion

We have discussed a way to vary the communication cost of distributed RRT methods. Instead of updating the global tree every time a new node is created, we create nodes and store the information about these newly created nodes locally. We continue to store information about these new nodes until our threshold of how many local nodes we will create is hit. This threshold is set with our parameter, k.

A. Future work

In regards to our implementation, finding the closest neighbor to expand our tree from was the most time expensive part of our method. In the future, we would like to parallelize this act of finding the closest neighbor. Hopefully, this will reduce the time spent finding a neighbor and give more scalable results. We believe fixing this problem would allow us to use a larger number of processors to further test on.

References

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

[2] O. B. Bayazit, J.-M. Lien, and N. M. Amato. Better group behaviors using rule-based roadmaps, 2002.

[3] O. B. Bayazit, G. Song, and N. M. Amato. Ligand binding with obprm and haptic user input: Enhancing automatic motion planning with virtual touch, 2000.

[4] J. Bialkowski, S. Karaman, and E. Frazzoli. Massively parallelizing the rrt and the rrt. In *IROS'11*, pages 3513–3518, 2011.

[5] 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 *Proceedings of the 3rd Annual Haifa Experimental Systems Conference*, SYSTOR '10, pages 14:1–14:10, New York, NY, USA, 2010. ACM.

[6] 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.

[7] D. Devaurs, T. Simon, and J. Corts. Parallelizing rrt on distributed-memory architectures. pages 2261–2266, 2011.

[8] D. Hsu, J. claude Latombe, and R. Motwani. Path planning in expansive configuration spaces. In *International Journal of Computational Geometry and Applications*, pages 2719–2726, 1997.

[9] 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. *Proc. IEEE Int. Conf. Robot. Autom*, 2012.

[10] L. E. Kavraki, P. Svestka, L. E. K. P. Vestka, J. claude Latombe, and M. H. Overmars. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *IEEE Transactions on Robotics and Automation*, 12:566–580, 1996.

[11] Y. Koga, K. Kondo, J. Kuffner, and J.-C. Latombe. Planning motions with intentions, 1994.

[12] S. M. LaValle, J. J. Kuffner, and Jr. Randomized kinodynamic planning, 1999.

[13] 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 Transactions on Robotics*, 21:597–608, 2005.

[14] A. P. Singh, J. claude Latombe, and D. L. Brutlag. A motion planning approach to flexible ligand binding, 1999.

[15] G. Song. Using motion planning to study protein folding pathways. In *Journal of Computational Biology*, pages 287–296, 2001.