A Scalable Distributed RRT for Motion Planning

Sam Ade Jacobs, Nicholas Stradford, Cesar Rodriguez, Shawna Thomas and Nancy M. Amato

Abstract— **Rapidly-exploring Random Tree (RRT), like other sampling-based motion planning methods, has been very successful in solving motion planning problems. Even so, sampling-based planners cannot solve all problems of interest efficiently, so attention is increasingly turning to parallelizing them. However, one challenge in parallelizing RRT is the global computation and communication overhead of nearest neighbor search, a key operation in RRTs. This is a critical issue as it limits the scalability of previous algorithms. We present two parallel algorithms to address this problem. The first algorithm extends existing work by introducing a parameter that adjusts how much local computation is done before a global update. The second algorithm radially subdivides the configuration space into regions, constructs a portion of the tree in each region in parallel, and connects the subtrees, removing cycles if they exist. By subdividing the space, we increase computation locality enabling a scalable result. We show that our approaches are scalable. We present results demonstrating almost linear scaling to hundreds of processors on a Linux cluster and a Cray XE6 machine.**

I. Introduction

Research in robotic motion planning spans over three decades, resulting in the development of different types of sequential and parallel algorithms for motion planning [10], [12]. The recent renewed interest in parallel motion planning algorithms 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 such as those arising in manipulation and reconfigurable robotics [1], computational biology and drug design [3], [24], as well as virtual prototyping and computer-aided design [2], [9]. These new application areas test the limit and capability of existing sequential motion planners [21]. Thus, scalable parallelism has a key role to play, both in supporting existing work and in exploring new algorithms needed to solve complex, high dimensional motion planning problems.

This research supported in part by NSF awards CNS-0551685, CCF-0833199, CCF-0830753, IIS-0917266, IIS-0916053, EFRI-1240483, RI-1217991, by NSF/DNDO award 2008-DN-077-ARI018-02, by NIH NCI R25 CA090301-11, by DOE awards DE-FC52-08NA28616, DE-AC02-06CH11357, B575363, B575366, by THECB NHARP award 000512-0097-2009, by Samsung, Chevron, IBM, Intel, Oracle/Sun and by Award KUS-C1-016-04, made by King Abdullah University of Science and Technology (KAUST). This research used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

The authors are with the Parasol Lab., Dept. of Computer Science and Engineering, Texas A&M Univ., College Station, Texas, 77843-3112, USA. {sjacobs,nds0057,cesar094,sthomas,amato}@cse.tamu.edu

There are two main sampling-based motion planning approaches: RRT [16] and Probabilistic Roadmap Method (PRM) [14]. RRT, PRM, and their variants are widely considered as state-of-the-art methods for solving motion planning problems. They are efficient and have been highly successful at solving many previously unsolved problems. RRT in particular is well suited for non-holonomic and kinodynamic motion planning problems [7], [17].

In this work, we present scalable parallel algorithms for computing Rapidly-exploring Random Trees (RRTs). In particular, we present two parallel algorithms: (i) an algorithm that extends [11] by introducing a parameter that controls how much local and concurrent computation is done before a global update and inter-processor communication, and (ii) a novel algorithm that radially subdivides $\mathbb{C}_{space}$ into regions and then concurrently builds subtrees in each region which are later connected to form a single tree. By controlling how the subtrees explore the space, we minimize the communication overhead — a major bottleneck in parallel processing. While these parallel approaches employ the standard RRT expansion techniques, we note that they both result in trees that are structurally different than would be constructed sequentially.

Key contributions of this work include:

- We extend previous work by introducing a new parameter to control the RRT's local expansion and minimize global communication, yielding a more scalable algorithm.
- A novel radial subdivision of $\mathbb{C}_{space}$ so that RRT computation can be distributed efficiently.
- A generic and efficient implementation of nearest neighbor search based on a nested map reduce parallel computation pattern.

We present results demonstating almost linear scalability to hundreds of processors on a Linux cluster and a Cray XE6 machine.

II. Preliminaries and Related Work

Sampling-based Motion Planning. The motion planning problem is to find a valid path (e.g., collision-free and satisfying any joint limit and/or loop closure constraints) for a movable object starting from a specified initial configuration to a goal configuration in an environment [10]. 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)

is configuration space ($\mathbb{C}_{space}$). $\mathbb{C}_{space}$ is partitioned into two sets: $\mathbb{C}_{free}$ (feasible) and $\mathbb{C}_{obstacle}$ (infeasible). Motion planning then becomes finding a continuous sequence of points in $\mathbb{C}_{free}$ that connects the start and the goal.

A complete solution of the motion planning problem is considered computationally intractable and has been shown to be PSPACE-hard with an upper bound that is doubly exponential in movable object's DOF [22]. As an alternative, approximate solutions have been shown to be efficient and practical. Sampling-based methods [10] are the state-of-art practical approach to solving motion planning problems. 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) [14] and tree-based methods such as Rapidly-exploring Random Tree (RRT) [16].

RRT. The basic sequential RRT (shown in Algorithm 1) grows a tree rooted at the start configuration that expands outward into unexplored areas of $\mathbb{C}_{space}$. RRT first generates a uniform random sample q_{rand}, valid or not, and identifies the closest node q_{near} in the tree to q_{rand}. q_{near} is extended toward q_{rand} a stepsize Δq. If the extension is successful, q_{new} is added to the tree as node and the pair of q_{near} and q_{new} is added as an edge. To solve a particular query, RRT repeats this process until the goal configuration is also connected to the tree. RRT-connect [15] is a variant that grows two trees towards each other: one rooted at the start configuration and the other at the goal configuration. These two trees explore $\mathbb{C}_{space}$ until they are both connected.

Algorithm 1 Sequential RRT

Input: An environment env, a root q_{root}, the number of nodes N, a stepsize Δq
Output: A tree T containing N nodes rooted at q_{root}

```
1: T.AddNode(q_root)
2: i ← 0
3: while i < N do
4:    q_rand ← GetRandomNode(env)
5:    q_near ← FindNeighbor(T, q_rand, 1)
6:    q_new ← Extend(q_near, q_rand, Δq)
7:    if !TooSimilar(q_near, q_new) ∧ IsValid(q_new) then
8:       T.AddNode(q_new)
9:       T.AddEdge(q_near, q_new)
10:      i ← i + 1
11:   end if
12: end while
13: return T
```

Parallel RRT. Early parallel motion planning methods were based on the discretization of $\mathbb{C}_{space}$ [8], [18]. The discretization as presented limits the algorithm to solving relatively low dimensional problems. However, these methods laid the foundation for subsequent work in parallelizing RRTs.

The OR paradigm [8] was applied to parallelizing RRT computations on shared-memory machines where the computation is replicated on each process [7]. Processes concurrently explore $\mathbb{C}_{space}$ and the first process to find a solution sends a termination message to other processes. Their work also explored concurrently and cooperatively building a single tree under a shared-memory model. Each process executes their own program and communicates to other processes by exchanging data through the shared memory in a concurrent read exclusive write ($CREW$) fashion. In addition, they study a hybrid algorithm combining the OR paradigm and the $CREW$ model. The processes are divided into groups and each group cooperatively build 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 [4]. They identified nearest neighbor search as the key bottleneck for scalability. Their implementation was done in CUDA on a GPU. Other work has turned to multicore architectures [11]. They present three algorithms for distributed RRT. The first is a message passing implementation of the OR 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). In this paper, we extend the second algorithm by introducing a user-defined parameter to minimize the communication overhead associated with global update.

C-space Subdivision. $\mathbb{C}_{space}$ subdivision has been very useful in solving sequential motion planning problems. Early work in $\mathbb{C}_{space}$ subdivision computes the exact representation of $\mathbb{C}_{space}$ by uniformly dividing it into cells [5]. Each cell is then classified as $empty$, $full$, or $mixed$ depending on the obstacle position in the cell. An A* search algorithm is then used to compute a path through the purely $empty$ or $mixed$ cells.

Some sampling-based motion planning approaches also employ $\mathbb{C}_{space}$ subdivision. In [19], [20], $\mathbb{C}_{space}$ is subdivided by randomly selecting splitting points from randomly selected positional DOF. These splitting points define an orthogonal boundary in the selected DOF. This splitting process is recursively repeated until homogeneous but overlapping sets of regions are obtained. Homogeneity is defined accord-

ing to a set of features measured for each region. A similar algorithm of interest is Region-Sensitive Adaptive Motion Planning (RESAMPL) [23]. RESAMPL subdivides $\mathbb{C}_{space}$ into local regions using an initial set of samples. Some of these initial samples are randomly selected as representative samples for the local regions. The distance of the representative sample to its k-closest neighbors determines the region's size. Approximate Cell Decomposition (ACD) subdivides $\mathbb{C}_{space}$ into rectangular cells [26]. Similar to [5], each cell is labeled as $empty$, $full$, or $mixed$. 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.

In our previous work [13], we present $\mathbb{C}_{space}$ subdivision-based parallel methods for graph-based randomized motion planning algorithms, particularly PRMs. We demonstrate that by subdividing the space and restricting the locality of connection attempts, scalable performance can be achieved. However, the regular subdivision method as presented is not well suited for RRT. Here, we design a novel radial subdivision technique for parallelizing RRT.

III. Parallelizing RRT

In this section, we present two different parallel algorithms for RRT computation. The first is a bulk synchronous version of the distributed RRT algorithm given in [11]. We extend the algorithm by introducing a user-defined parameter that controls how much local expansion is made before a global broadcast. The second algorithm subdivides $\mathbb{C}_{space}$ radially into regions and distributes RRT computation in each region to available processes. Regional subtrees in adjacent regions are later connected to form one single tree. In both cases, although the algorithms use the familiar RRT expansion operations, the trees that are constructed are structurally different from trees that would be constructed using a sequential method. We describe each approach in detail below.

A. Bulk Synchronous Distributed RRT

The distributed RRT algorithm in [11] incurs global broadcasts each time a new node and edge are added to the tree. However, this is not scalable. We extend this work in two key 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. 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 [11].

Algorithm 2 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 1 shows a simple illustration of bulk synchronous distributed RRT computation in which p=2, m=2 and N=8.

Algorithm 2 Bulk Synchronous Distributed RRT

Input: An environment env, a root q_{root}, the number of nodes N, a stepsize Δq, the number of processes p, the number of local expansion steps 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

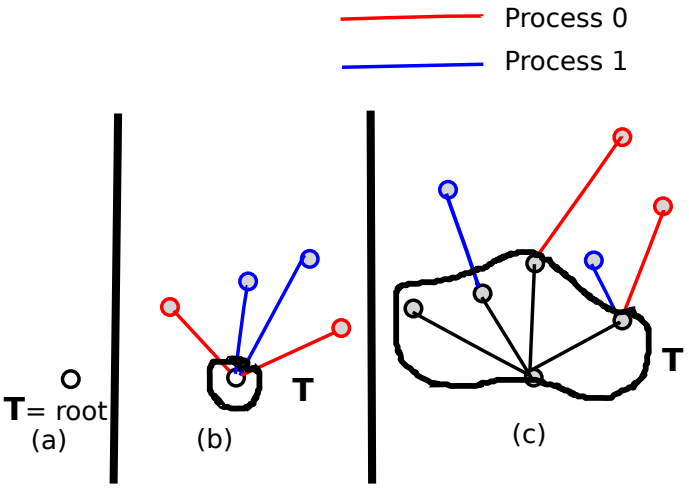


Fig. 1. Bulk Synchronous Distributed RRT. (a) T is initialized to root. (b) The first iteration with m=2. (c) The second iteration where globally communicated data is shown in black.

B. Radial Subdivision Distributed RRT

We also present a novel *radial* $\mathbb{C}_{space}$ subdivision for parallelization 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 2 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 $\mathbb{C}_{space}$.

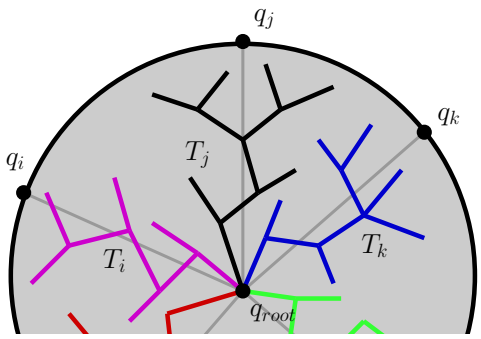


Fig. 2. Example of radial subdivision for a 2D $\mathbb{C}_{space}$. Each process concurrently builds a subtree (using sequential RRT) rooted at q_r and biased toward a target q_i (e.g., q_n for the black process).

Algorithm 3 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 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.

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.

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 3 shows a simple pictorial illustration for tree pruning.

Algorithm 3 Radial Subdivision Distributed RRT

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

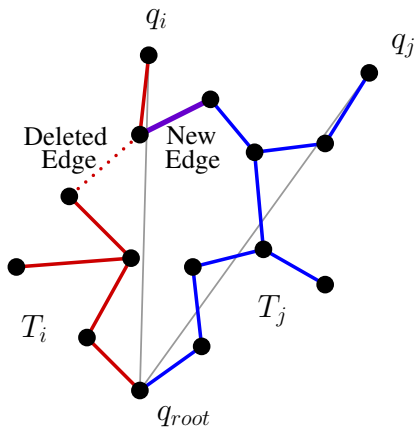


Fig. 3. Tree pruning example. The new edge (purple) between the red and blue branches causes a cycle in the red branch. The dashed edge is identified for removal.

IV. Implementation Details

A. STAPL Framework

Our code was written in C++ using the Standard Template Library (STL) and the Standard Template Adaptive Parallel Library (STAPL) [6], [25] as supporting libraries. STAPL is a platform independent superset of STL that provides a collection of building blocks for writing parallel programs.

These building blocks include a collection of parallel algorithms (pAlgorithms), parallel and distributed containers (pContainers), a general mechanism to access the data of a pContainer similar to STL *iterators* called pViews, an abstraction of the computation task graph (PARAGRAPH), and an Adaptive Runtime System (ARMI) that includes a communication library, scheduler, and performance monitor.

In this work, we made use of the STAPL Parallel Graph Library, one of the STAPL pContainers, as the parallel data structure for representing both the region graph and the RRT tree. We implemented bulk synchronous distributed RRT and radial subdivision distributed RRT as STAPL pAlgorithms. The tree pruning process described in Section III-B was implemented using the STAPL parallel breadth first search (BFS) algorithm.

B. Parallelizing Nearest Neighbor Search

There is a clear need for fine-grained parallelism in sampling-based motion planning [4], [11]. The nearest neighbor search is considered a key bottleneck to scalable performance. In this work, we implement and incorporate a nested and fine-grained parallel computation of nearest neighbor search within the two parallel RRT algorithms described in Section III. Our implementation has a *map reduce* parallel computation pattern.

Algorithm 4 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 5) 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 6) 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.

Algorithm 4 Parallel NNS Distributed RRT

Input: An environment env, a root q_{root}, the number of nodes N, a stepsize 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 ∈ P par do
3:    i ← 0
4:    while i < N/p do
5:       subtree T_p ∈ T
6:       q_rand ← GetRandomNode(env)
7:       q_near ← MapReduce(Map(T_p, q_rand),
         Reduce(q_near, q_near))
8:       q_new ← Extend(q_near, q_rand, Δq)
9:       if !TooSimilar(q_near, q_new) ∧ IsValid(q_new) then
10:         T.AddNodeToTree(q_new)
11:         T.AddEdgeToTree(q_near, q_new)
12:      end if
13:      i ← i + 1
14:   end while
15: end for
16: return T
```

Algorithm 5 Map

Input: A set of points S, a query q
Output: A map of closest point to q and its distance M

```
1: M ← FindNeighbors(S, q, 1)
2: return M
```

V. Experimental Setup and Results

We investigate the scalability of the two algorithms presented in Section III: bulk synchronous distributed RRT and radial subdivision distributed RRT. We also examine the effect of robot complexity and machine architecture.

A. Experimental Setup

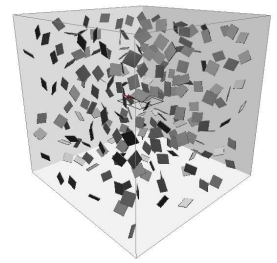
1) Machine Architecture: Experiments were conducted on two massively parallel computers. The first machine is a major Linux computing cluster. It has a total of 300 nodes, 172 of which are made of two quad core Intel Xeon and AMD Opteron processors running at 2.5GHz with 16 to 32GB memory per node. The 300 nodes have 8TB of memory and a peak performance of 24 Tflops. Each node of the Linux cluster is made of 8 processor cores, thus, for this machine we present results for processor counts in multiples of 8. The second machine is a Cray XE6 petascale machine. It has 6384 nodes, 217 TB of memory, and a peak performance of 1.288 peta-flops. Each node consists 12 processor cores. This architectural layout 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.

2) Motion Planning Problems: We studied three different kinds of environments: a $512 \times 512 \times 512$ uniformly cluttered environment (shown in Figure 4(a)) and a 7x7x7 grid environments (shown in Figure 4(b)) and another clutter

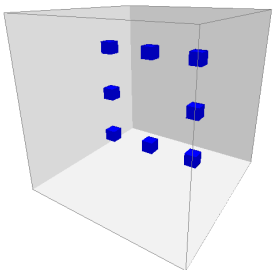
Algorithm 6 Reduce

Input: Two maps $M1$ and $M2$ of points and their distances to a query q
Output: The closest point $p \in M1 \cup M2$

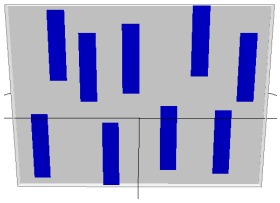
```
1: if M1.distance ≤ M2.distance then
2:    p ← M1.point
3: else
4:    p ← M2.point
5: end if
6: return p
```



(a) Clutter

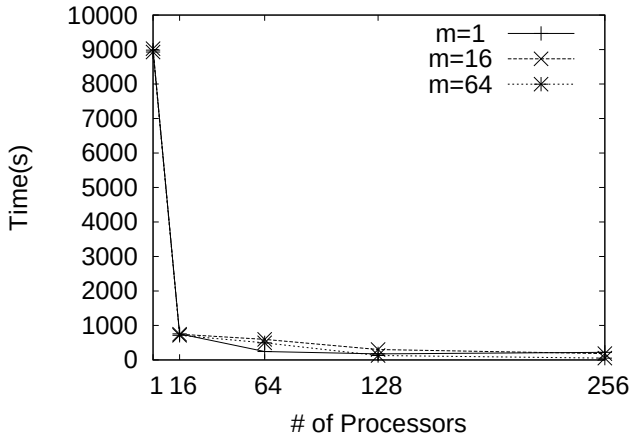


(b) Grid

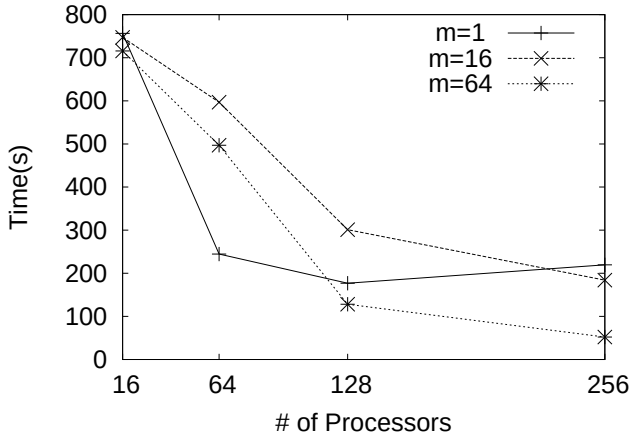


(b) Stripline

Fig. 4. Environments studied



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



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

Fig. 5. Effect of varying m in the bulk synchronous distributed RRT.

environment with strip-like obstacles (shown in Figure 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.

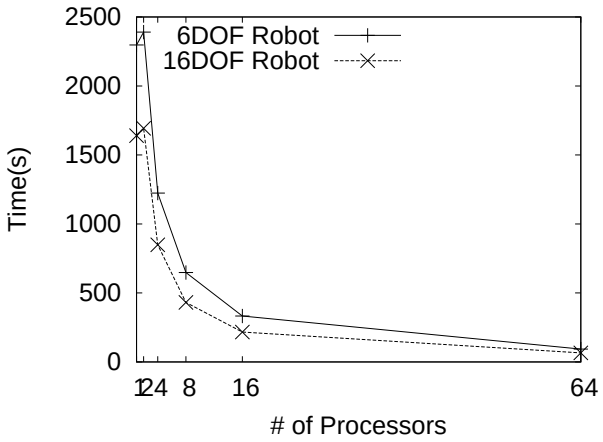
B. Experimental Results

1) Bulk Synchronous Effect: We first study the effect of the m parameter introduced in Algorithm 2 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 [11]. Figure 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.

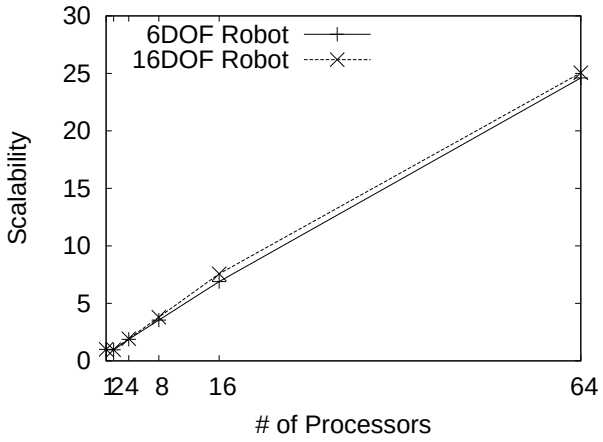
Localizing the computation and thus minimizing frequent inter-processor communication by varying m does impact performance of distributed RRT, but this effect is not obvious until higher processor counts, see Figure 5(b). In fact, $m = 1$ seems to outperform the others until around $p = 16$.

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 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.

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 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



(a) Time



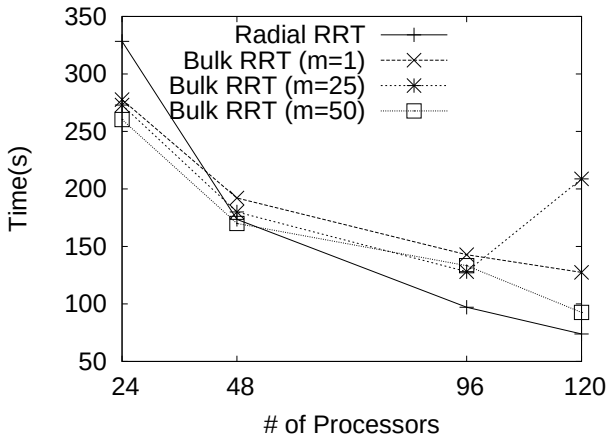
(b) Scalability

Fig. 6. Radial subdivision distributed RRT performance on Linux cluster.

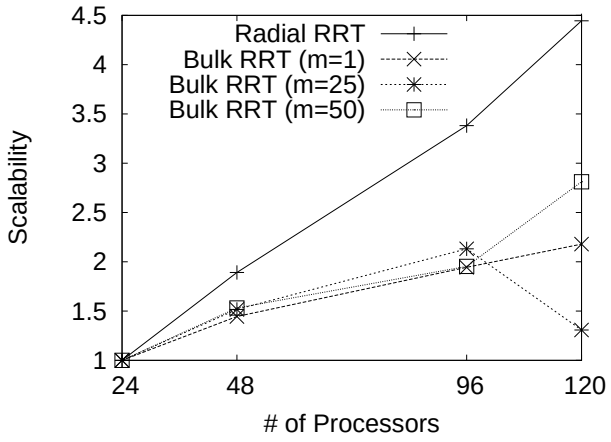
Linux cluster. Scalability of the bulk synchronous distributed RRT depends on the value of m and the number of processors. As in the previous experiments (Figure 5), the impact of increasing m is much felt at higher processor counts at which inter-processor communication become significant.

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 constant at 480 across all processor count and varied the sample size per region. The results from the evaluation are shown in Figure 8. Given different input sizes, we saw decrease in execution time as the number of processors increases.

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 9. We observed almost linear scalability in all cases.



(a) Time



(b) Scalability

Fig. 7. Distributed RRT performance on Cray XE6 machine.

VI. Conclusion

In this paper, we present two parallel algorithms for RRT computation. The first algorithm extends existing work by introducing a parameter m that controls how much local computation is done before a global update across processors. The second algorithm radially subdivides $\mathbb{C}_{space}$ into regions and lets each processor build part of the tree in each region. By controlling local computation and subdividing $\mathbb{C}_{space}$, we minimize the overhead associated with inter-processor communication in parallel processing. We present results for both a rigid body robot and an articulated linkages on two different parallel machine architectures.

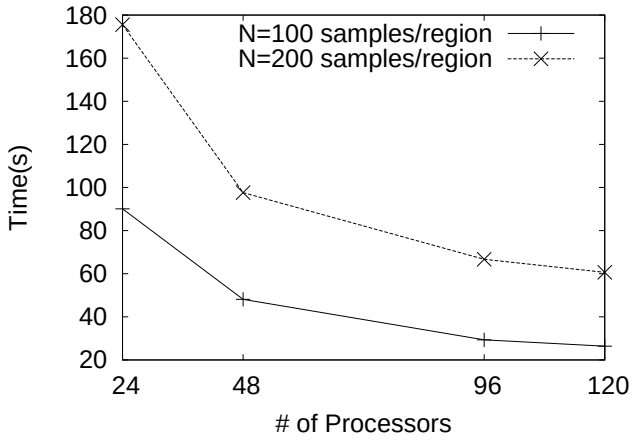


Fig. 8. Radial RRT performance results for grid environment on Cray XE6 machine

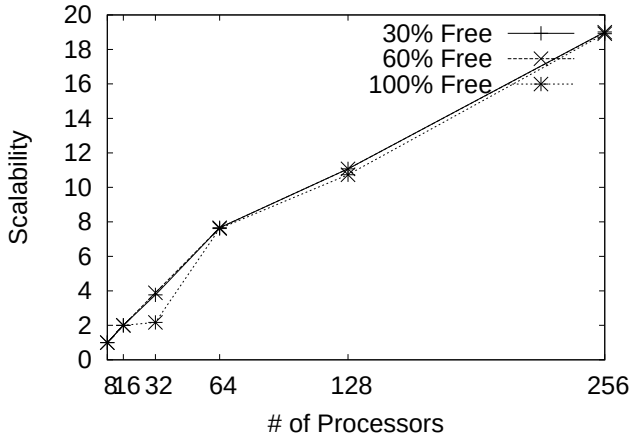


Fig. 9. Radial RRT performance results for stripline environment on Linux cluster

REFERENCES

[1] F. Aghili and K. Parsa. Configuration control and recalibration of a new reconfiburable robot. In *Proc. IEEE Int. Conf. Robot. Autom. (ICRA)*, pages 4077–4083, 2007.

[2] 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)*, pages 529–536, 2000.

[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. In *Proc. IEEE Int. Conf. Robot. Autom. (ICRA)*, pages 954–959, 2001. This work was also presented as a poster at *RECOMB 2001*.

[4] J. Bialkowski, S. Karaman, and E. Frazzoli. Massively parallelizing the rrt and the rrt*. In *Proc. IEEE Int. Conf. Intel. Rob. Syst. (IROS)*, 2011.

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

[6] 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)*, pages 1–10, New York, NY, USA, 2010. ACM.

[7] S. Carpin and E. Pagello. On parallel rrts for multi-robot systems. In *Proc. Italian Assoc. AI*, pages 834–841, 2002.

[8] D. J. Challou, M. Gini, and V. Kumar. Parallel search algorithms for robot motion planning. In *Proc. IEEE Int. Conf. Robot. Autom. (ICRA)*, volume 2, pages 46–51, 1993.

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

[10] 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.

[11] D. Devaurs, T. Simeon, and J. Cortes. Parallellizing rrt on distributed-memory architectures. In *Proc. IEEE Int. Conf. Robot. Autom. (ICRA)*, 2011.

[12] D. Henrich. Fast motion planning by parallel processing - a review. *Journal of Intelligent and Robotic Systems*, 20(1):45–69, 1997.

[13] 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.

[14] 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.*, 12(4):566–580, August 1996.

[15] 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)*, pages 995–1001, 2000.

[16] S. M. LaValle and J. J. Kuffner. Randomized kinodynamic planning. In *Proc. IEEE Int. Conf. Robot. Autom. (ICRA)*, pages 473–479, 1999.

[17] S. M. LaValle and J. J. Kuffner. Randomized kinodynamic planning. *Int. J. Robot. Res.*, 20(5):378–400, May 2001.

[18] T. Lozano-Pérez and P. O'Donnell. Parallel robot motion planning. In *Proc. IEEE Int. Conf. Robot. Autom. (ICRA)*, pages 1000–1007, 1991.

[19] 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*, pages 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.

[20] 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)*, pages 3114–3119, April 2005.

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

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

[23] S. Rodriguez, S. Thomas, R. Pearce, and N. M. Amato. (RESAMPL): A region-sensitive adaptive motion planner. In *Algorithmic Foundation of Robotics VII*, pages 285–300. Springer, Berlin/Heidelberg, 2008. book contains the proceedings of the International Workshop on the Algorithmic Foundations of Robotics (WAFR), New York City, 2006.

[24] 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)*, pages 252–261, 1999.

[25] G. Tanase, A. Buss, A. Fidel, Harshvardhan, I. Papadopoulos, O. Pearce, T. Smith, N. Thomas, X. Xu, N. Mourad, J. Vu, M. Bianco, N. M. Amato, and L. Rauchwerger. The STAPL Parallel Container Framework. In *Proc. ACM SIGPLAN Symp. Prin. Prac. Par. Prog. (PPoPP)*, pages 235–246, San Antonio, Texas, USA, 2011.

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