Parallel Sampling-Based Motion Planning with Superlinear Speedup

Jeffrey Ichnowski and Ron Alterovitz

***Abstract*— We present PRRT (Parallel RRT) and PRRT* (Parallel RRT*), sampling-based methods for feasible and optimal motion planning that are tailored to execute on modern multi-core CPUs. Our algorithmic improvements enable PRRT and PRRT* to achieve a superlinear speedup: when p processor cores are used instead of 1 processor core, computation time is sped up by a factor greater than p. To achieve this superlinear speedup, our algorithms utilize three key features: (1) lock-free parallelism using atomic operations to eliminate slowdowns caused by lock overhead and contention, (2) partition-based sampling to reduce the size of each processor core's working data set to improve cache efficiency, and (3) parallel backtracking to reduce the number of rewiring steps performed in PRRT*. Our parallel algorithms retain the ability to integrate with existing CPU-based libraries and algorithms. We demonstrate fast performance and superlinear speedups in two scenarios: (1) a holonomic disc-shaped robot moving in a planar environment and (2) an Aldebaran Nao small humanoid robot performing a 2-handed manipulation task using 10 DOF.**

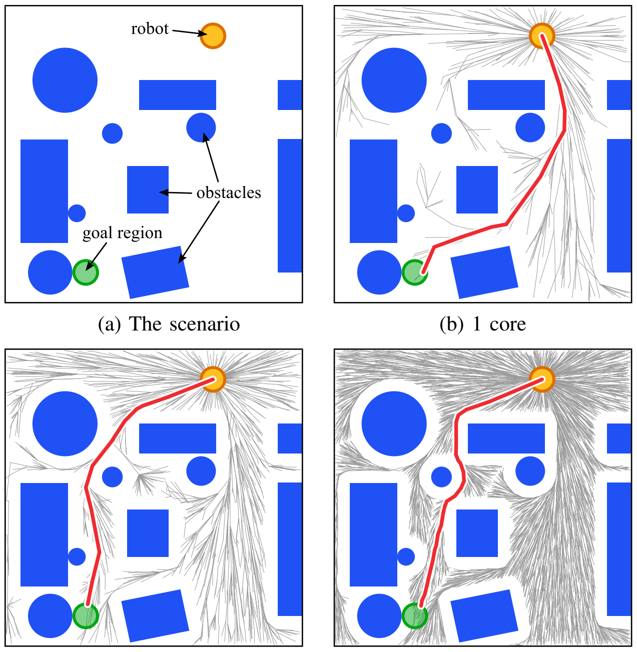
I. Introduction

Incremental sampling-based motion planners, such as the Rapidly-exploring Random Tree (RRT) and RRT*, are used in a variety of robotics applications including manipulation, computational biology, and autonomous vehicles to find paths through the robot's free configuration space from a start location to a goal location [1], [2]. We introduce PRRT (Parallel RRT) and PRRT* (Parallel RRT*), parallelized versions of RRT and RRT* that are tailored to execute on modern multi-core CPUs.

Most modern PCs and mobile devices have between 2 and 32 processing cores, and this number is increasing. We leverage these parallel architectures to introduce substantial speedups in motion planning. Our algorithmic improvements enable PRRT and PRRT* to achieve a *superlinear speedup*: when p processor cores are used instead of 1 processor core, computation time is sped up by a factor greater than p.

Our focus is on challenging motion planning scenarios for which a large number (tens or hundreds of thousands) of configuration samples is necessary to find a path or to achieve the desired closeness to optimality. In RRT and RRT*, the time spent computing nearest neighbors grows logarithmically with each iteration as the number of samples rises, whereas the time spent on collision detection decreases as the expected distance between samples shrinks. Collision detection typically dominates computation time in the early iterations. But as the number of iterations rises and the number of samples increases, nearest neighbor search will dominate the overall computation.

J. Ichnowski and R. Alterovitz are with the Department of Computer Science at the University of North Carolina at Chapel Hill, USA {jeffi,ron}@cs.unc.edu



(a) The scenario (b) 1 core (c) 4 cores (d) 32 cores

Fig. 1. We ran PRRT* for a 2D holonomic motion planning problem for a disc-shaped robot for 10 ms on 1, 4, and 32 processor cores. The red line shows the optimal path found. With the same wall clock time, adding more processor cores increases the size of the tree, enabling fast computation of higher quality motion plans on modern multi-core computers.

To enable speedup regardless of the computational bottleneck (e.g. collision detection or nearest neighbor searching), we parallelize the outer loop of RRT(*): we create multiple threads that each generate samples and incrementally extend the graph data structure based on those samples. To parallelize at this level, independently working threads must share access to a common nearest neighbor search and motion planning graph.

Traditionally, shared access might be controlled using locks. In the lock-based approach, when a thread must access a shared data structure, it first locks the data structure, then accesses it, and finally unlocks it. When another thread attempts to access a locked data structure it waits (i.e. is blocked) until the data structure is unlocked. If the lock covers a large data structure, then one thread may unnecessarily block other threads. If instead many locks are used to cover smaller data structures, then threads will repeatedly lock data structures unnecessarily, leading to high overhead. As the number of processor cores increases and as the number of samples grows to handle more complex motion planning problems, more computation time must be spent on nearest neighbor checking and lock contention rises, causing sublinear speedup.

To achieve superlinear speedup, PRRT and PRRT* introduce three key algorithmic features:

1) *Lock-free parallelism using atomic operations*: To eliminate slowdowns caused by lock overhead and contention, PRRT(*) uses lock-free shared data structures that are updated using an atomic compare-and-swap (CAS) operation, a universal primitive [3]. A CAS operation compares a single variable to a given value and, only if they are the same, modifies the variable to the given value. If the comparison fails due to an update made by another thread, the update is reformulated with the new information, and tried again until it succeeds or is no longer necessary. In PRRT(*) we observe that as the number of samples n increases, the probability that any of the p threads are updating the same part of the motion planning tree decreases ($\lim_{n\to\infty} O(p/n) = 0$). As a consequence, CAS operations rarely fail and we avoid the unnecessary blocking and high overhead associated with locks, giving us the expectation of at least linear speedup.
2) *Partition-based sampling*: To reduce the size of each thread's working data set, we partition the configuration space into non-overlapping regions and assign a partition to each thread. Partitioning has two benefits. First, it reduces the likelihood that two threads will simultaneously attempt to modify the same part of the shared data structures, reducing CAS failures. Second, as each processor core is expected to work in a smaller subset of the nearest neighbor data structure, more of the relevant structure can reside in each core's cache, thus creating an opportunity for superlinear speedup.
3) *Parallel backtracking*: During the rewiring phase of RRT*, the algorithm evaluates the costs of paths to nearby nodes, rewires them through the new node if such routing would produce a shorter path, and percolates updates up the tree. To reduce the number of rewiring operations in RRT*, we ensure that when multiple threads attempt to rewire the same portion of the tree, only the one with the better update continues. This frees the other threads to continue expanding the RRT*, effectively reducing computation effort relative to single-threaded RRT* for percolating rewiring up the tree.

A key advantage of creating a parallel motion planner for a standard shared-memory CPU-based computing platform (rather than, for example, a GPU) is that it enables direct integration with existing libraries for collision detection, robot kinematics, and physics-based simulation. We evaluate our method in 2 scenarios: (1) a 2-DOF disc-shaped robot in a planar environment, and (2) an Aldebaran Nao small humanoid robot performing a 2-handed manipulation task using 10 DOF. The latter scenario requires a large number of samples (tens to hundreds of thousands) to obtain a high quality solution. Our results confirm the superlinear speedup: computation time for the Nao robot is reduced by a factor of 36x when using 32 processor cores instead of 1 core.

II. Related Work

Sampling-based motion planners include several components that can naturally be parallelized, and prior work has taken multiple avenues to exploit this parallelism using multiple CPUs and GPUs.

Amato et al. [4] observed that sampling-based motion planning using probabilistic roadmaps (PRMs) is "embarrassingly parallel." With their CPU-based approach, they generate millions of samples and connect 1500. Computation time is ostensibly dominated by computing samples and collisions. Their results however exhibit sublinear speedup, with lock contention possibly slowing it down.

Several approaches to parallelizing motion planning across multiple cores/processors have utilized multiple tree-like data structures. Carpin et al. [5] introduced two approaches for shared-memory CPU-based platforms: (1) running several RRT computations in parallel and choosing the best, and (2) utilizing locked structures to synchronize updates to the shared data structures between parallel computations. They use a linear nearest neighbor search, and their results taper off at 3500 samples. Plaku et al. [6] introduced the Sampling-based Roadmap of Trees (SRT) algorithm, which distributes computation on a cluster to solve high-dimensional planning problems. SRT subdivides the motion planning problem into "milestones" solved by another planner, e.g. RRT, and then connects the milestones together using a bi-directional algorithm. SRT achieves near-linear speedup that slightly tapers at higher processor counts. Otte et al. [7] also distribute the generation of independent random path-planning trees among several threads and share information between threads about the best known path, allowing threads to reduce computation (and thus gain a speedup) by not introducing samples that are worse than the best known.

GPU-based parallel computation is increasingly being used to parallelize and accelerate different components of sampling-based motion planners. G-Planner [8] uses GPUs to accelerate all components of a PRM motion planner. Lengel et al. [9] use GPU hardware to accelerate path planning for low DOF robots using rasterization techniques. Hoff et al. [10] and Kenneth et al. [11] use GPUs to accelerate sampling via Voronoi diagrams to bias probabilistic roadmaps. Foskey et al. also used GPUs to accelerate hybrid planners based on sampling and Voronoi diagrams [12]. Kider et al. [13] created a GPU-based planner for R*, a randomized version of A*, that retains the theoretical properties of R* but with improved experimental results.

Although GPUs are a powerful tool for some applications, their single-instruction-multiple-data (SIMD) execution model constrains algorithm design. When each thread needs to do something different (inherently divergent), such as traversing a space partitioning tree, the SIMD model loses nearly all ability to parallelize [14]. Another challenge with GPU approaches is that, while they typically gain the benefit of higher computational throughput associated with GPUs, they sacrifice some interoperability with standard systems and libraries based upon CPUs.

Bialkowski et al. [15] parallelize RRT* and related methods by improving the rate of collision detection. This approach results in substantial speedups for environments where collision detection dominates processing time. But due to Amdahl's law [16], parallel performance will taper as the number of samples increases and nearest neighbor checks begin to dominate as discussed in Sec. I.

III. Problem Formulation

A. Parallel Computing Environment

Our target computing environment is the one available in almost every modern computer: a multi-core/multi-processor concurrent-read-exclusive-write (CREW) shared-memory system with atomic operations that synchronize views of memory between threads running on different cores. This is the model in current generation of x86-64 multi-core processors as well as many other CPU architectures.

In this environment we use *core* to mean a single hardware based execution unit. A *processor* may have one or more cores, and a computer may have one or more processors. Each core is capable of executing an independent *thread* simultaneous to all other cores. The total number of cores p is maximum number of simultaneously executing threads. *Speedup* $S_p = T_1/T_p$ is the ratio of the sequential (single-threaded) execution time T_1 to parallel execution time T_p with p cores. Linear speedup means $S_p = p$, superlinear means $S_p > p$.

Multi-core processors typically have a cache memory hierarchy that includes one or more small but fast caches local to each core (L1 and L2) and a larger and slower cache shared among cores (L3). When the data set in use by a core is smaller, the core uses the faster local caches more often and gains a proportional speed benefit. CPU caches can be leveraged to gain superlinear speedups by distributing the working dataset into smaller chunks across multiple cores.

B. Problem Definition

Let $\mathcal{C} \in \mathbb{R}^d$ be the configuration space of the robot and $\mathcal{C}_{\text{free}} \subseteq \mathcal{C}$ denote the subspace of the configuration space for which the robot is not in collision with an obstacle. Let $q \in \mathcal{C}$ denote a configuration of the robot. The PRRT(*) planner requires as input the start configuration q_{init} of the robot and a set of goal configurations $Q_{\text{goal}} \subseteq \mathcal{C}_{\text{free}}$.

The objective of PRRT (feasible motion planning) is to find a path $\Pi : (q_0, q_1, q_2, \ldots, q_{\text{end}})$ such that $q_0 = q_{\text{init}}$ and $q_{\text{end}} \in Q_{\text{goal}}$ and Π lies in $\mathcal{C}_{\text{free}}$. The objective of PRRT* (optimal motion planning) is to find a feasible path such that the cost of the path is minimized, where the cost of a path is defined to be the summation of the costs of sequential pairs of configurations along the path Π.

C. Algorithm Inputs

Similar to their sequential motion planning counterparts RRT and RRT*, PRRT(*) requires as input the definition of the problem-specific functions. Given a set of configurations V and any two configurations $q_1, q_2 \in V$, $\texttt{FEASIBLE}(q_1, q_2)$ returns `false` if the path (computed by a local planner [1]) from q_1 to q_2 collides with an obstacle or violates some motion constraint, and `true` otherwise. The function $\texttt{STEER}(q_1, q_2)$ returns a new configuration that would be reached if taking a trajectory from q_1 heading toward q_2 up to some maximum user-specified distance. For PRRT*, the function $\texttt{COST}(q_1, q_2)$ specifies the cost associated with moving between two configurations q_1 and q_2, which can equal control effort, Euclidean distance, or any problem-specific user-specified metric. PRRT* has the same additional constraints on the cost function as RRT*. We also require a function $\texttt{GOAL}(q)$ that returns `true` if $q \in Q_{\text{goal}}$ and `false` otherwise.

The above functions are standard in RRT and RRT*, allowing existing systems based on these implementations to remain largely unchanged yet benefit from parallelism. For the algorithm we present here, the only additional requirement we add is that the problem-specific functions must be either (1) thread-safe and non-blocking or (2) capable of having multiple non-shared instances in the same program.

IV. The PRRT and PRRT* Algorithms

The PRRT and PRRT* algorithms each maintain shared data structures for nearest neighbor checks, the RRT or RRT* graph τ, the approximate iteration number, and the best path to goal found by any of the threads. PRRT and PRRT*, shown in Algorithm 1, begin by partitioning the configuration space into non-overlapping regions and launching an independent thread for each partition. For peak performance, each thread runs on a dedicated core.

Algorithm 1 PRRT and PRRT*

```
1: initialize $\tau$
2: for $i = 1 \ldots$ thread_count do
3:     $s \leftarrow$ partition($i$, thread_count)
4:     $w_i \leftarrow$ start new thread PRRT(*)_Thread($\tau, s$)
```

A. Partition-based Sampling

PRRT(*) partitions the configuration space into non-overlapping regions for random sampling in order to localize each thread's operations to a smaller portion of configuration space, and thus a smaller portion of the kd-tree and RRT(*) graph. The result is that each core's working data set on average shrinks allowing for more effective use of its caches and enabling superlinear performance. In our results presented in Sec. V, we partition by evenly subdividing along one dimension of the configuration space. The best choice of partitioning scheme may vary based on the robot's environment.

B. PRRT

Lock-free parallel RRT, shown in Algorithm 2, is nearly identical to standard RRT except that each thread only samples in a partition, and it uses a lock-free nearest-neighbor data structure (introduced in Sec. IV-E). Additionally before adding newly initialized RRT path nodes to the nearest-neighbor data structure (and thus becoming "visible" to other threads), a suitable memory fence operation is issued to prevent other threads from seeing a partially initialized view of the newly added path node.

Algorithm 2 PRRT_Thread(τ, s)

```
1:  while not done do
2:      $q_{rand} \leftarrow$ random sample from $s$
3:      $q_{near} \leftarrow$ nearest($\tau, q_{rand}$)
4:      $q_{new} \leftarrow$ STEER($q_{near}, q_{rand}$)
5:      if FEASIBLE($q_{near}, q_{new}$) then
6:          PRRT_Insert($\tau, q_{near}, q_{new}$)
7:          if GOAL($q_{new}$) then
8:              done $\leftarrow$ true
```

C. PRRT* Overview

The parallel lock-free version of RRT* expands the tree much like RRT with the additional step of "rewiring" a small neighborhood of the tree to more optimal paths. PRRT*_Thread, shown in Algorithm 3, is the main loop of a thread running PRRT*. It works much like standard RRT* with four notable differences: (1) sampling is done within a partition of the configuration space (line 2), (2) nearest-neighbor search is performed using a lock-free kd-tree (lines 3 and 6), (3) new configurations are added to the RRT(*) tree in a manner that accounts for parallelism (lines 15–18)—specifically they are fully initialized before being added to the nearest-neighbor structure, and (4) rewiring is accomplished entirely via lock-free operations.

Algorithm 3 PRRT*_Thread(τ, s)

```
1:  while not done do
2:      $q_{rand} \leftarrow$ random sample from $s$
3:      $n_{nearest} \leftarrow$ nearest($\tau, q_{rand}$)
4:      $q_{new} \leftarrow$ STEER($q_{nearest}, q_{rand}$)
5:      if FEASIBLE($n_{nearest}$.config, $q_{new}$) then
6:          $N_{near} \leftarrow$ NEAR($\tau, q_{new}, |\tau|$)
7:          $c_{min} \leftarrow \infty$
8:          for all $n_{near} \in N_{near}$ do
9:              if FEASIBLE($n_{near}$.config, $q_{new}$) then
10:                 $c_{link} \leftarrow$ COST($n_{near}$.config, $q_{new}$)
11:                 $c_{path} \leftarrow n_{near}$.edge.cost $+ c_{link}$
12:                 if $c_{path} < c_{min}$ then
13:                     $n_{min} \leftarrow n_{near}$
14:                     $c_{min} \leftarrow c_{path}$
15:         $n_{new}$.config $\leftarrow q_{new}$
16:         $e_{new} \leftarrow (n_{new}, c_{min}, n_{min})$
17:         $n_{new}$.edge $\leftarrow e_{new}$
18:         LockFreeKDInsert($n_{new}$)
19:         if $e_{new}$ is expired then
20:             PRRT*_Update($n_{new}$.edge, $e_{new}$)
21:         if GOAL($e_{new}$) then
22:             record goal
23:         for all $n_{near} \in N_{near} \setminus \{n_{min}\}$ do
24:             RRT*_Rewire($\tau, n_{near}, n_{new}$)
```

D. PRRT* Rewiring

During the rewiring phase of the algorithm, RRT* evaluates the costs of paths to nearby nodes and rewires them through the new sample if such routing is both FEASIBLE and results in a shorter path. One approach caches the path cost with the node, and pushes updates down the tree when a node is rewired.

To rewire in parallel, PRRT* formulates rewiring (Algorithm 4) into a CAS operation that guarantees the above conditions hold true, even if another thread is updating the same node. If the CAS update fails, it means that the assertion about the new trajectory being shorter may now be incorrect, and the update is re-evaluated and tried again if it would still be shorter.

Algorithm 4 PRRT*_Rewire(τ, n_{near}, n_{new}): conditionally rewires a near node through a newly created node, if doing so creates a short path

```
1:  $e_{new} \leftarrow n_{new}$.edge
2:  $e_{near} \leftarrow n_{near}$.edge
3:  $c_{link} \leftarrow$ COST($n_{new}$.config, $n_{near}$.config)
4:  $c'_{near} \leftarrow e_{new}$.cost $+ c_{link}$
5:  if $c'_{near} \geq e_{near}$.cost or
        not FEASIBLE($n_{new}$.config, $n_{near}$.config) then
6:      return
7:  repeat
8:      $e'_{near} \leftarrow (n_{near}, c'_{near}, n_{new})$
9:      if CAS($n_{near}$.edge, $e_{near}$, $e'_{near}$) then
10:         add $e'_{near}$ to $e_{new}$.children
11:         PRRT*_Update($e'_{near}$, $e_{near}$)
12:         if $e_{new}$ is expired then
13:             PRRT*_Update($n_{new}$.edge, $e_{new}$)
14:         remove $e_{near}$ from $e_{near}$.parent.children
15:         return
16:     $e_{near} \leftarrow n_{near}$.edge
17: until $c'_{near} \geq e_{near}$.cost
```

CAS operations only work on single memory operands. The rewiring assertion however is made about two pieces of information: the trajectory and the cost of that trajectory. We thus modify the data structures to encapsulate both trajectory and cost into a single unit making it suitable for a CAS. The data structures we define are *nodes*, representing reachable valid configurations, and *edges*, representing trajectories from one node to another. The edges form a linked tree structure that represents known trajectories to any nodes. To get from the initial configuration to any node's configuration, the edge structure is followed (in reverse) from the node back to the root of the tree where the initial configuration is stored. An edge's path to root never changes, and thus its computed trajectory cost never changes. When a PRRT* finds a shorter path to a node, the node's *edge* is CAS with the better edge. The old edge will still essentially be present in the edge tree, but is no longer referenced from the node. We call an edge in this state "expired", and detect it when $edge.node.edge \neq edge$. (Expired edges can be "garbage collected" and reused, but care must be taken to avoid the ABA problem [3].)

By computing CAS operations around an edge, PRRT* guarantees that any update it makes results in an equal or

Algorithm 5 PRRT*_Update(e_{new}, e_{old}): Moves all the active children from a now expired parent edge to the new parent edge.

```
1: $n_{parent} \leftarrow e_{new}$.parent
2: done ← false
3: repeat
4:     $e_{child} \leftarrow$ remove_first $e_{old}$.children
5:     if $e_{child} = \varnothing$ then
6:         if $e_{new}$ is expired then
7:             PRRT*_Update($e_{new}$.node.edge, $e_{new}$)
8:         done ← true
9:     else if $e_{child}$ is not expired then
10:        $n_{child} \leftarrow e_{child}$.node
11:        $c_{child} \leftarrow e_{new}$.cost $+\texttt{COST}(n_{child}, n_{parent})$
12:        $e'_{child} \leftarrow (n_{child}, c'_{child}, e_{new})$
13:        if CAS($n_{child}$.link, $e_{child}$, $e'_{child}$) then
14:            add $e'_{child}$ to $e_{new}$.children
15:            PRRT*_Update($e'_{child}$, $e_{child}$)
16: until done
```

Algorithm 6 LockFreeKDInsert(n_{new})

```
1: $q_{new} \leftarrow n_{new}$.config
2: $k_{new}$.rrt_node $\leftarrow n_{new}$
3: $c_{min} \leftarrow C_{min}$
4: $c_{max} \leftarrow C_{max}$
5: $k \leftarrow$ kd_root
6: for $a = 0 \to \infty$ do
7:     $p \leftarrow (c_{min}[a \bmod \kappa] + c_{max}[a \bmod \kappa]) \div 2$
8:     if $q_a < p$ then
9:         if ($k$.lt $= \varnothing$) and CAS($k$.lt, $\varnothing$, $k_{new}$) then
10:            return
11:        $c_{max}[a \bmod \kappa] \leftarrow p$
12:        $k \leftarrow k$.lt
13:    else
14:        if ($k$.gt $= \varnothing$) and CAS($k$.gt, $\varnothing$, $k_{new}$) then
15:            return
16:        $c_{min}[a \bmod \kappa] \leftarrow p$
17:        $k \leftarrow k$.gt
```

better path, and thus the solution converges towards optimality. After rewiring a node through a better path, the new shorter path is recursively percolated to the nodes that link in to the rewired node. This update process (Algorithm 5) atomically replaces edges to the expired parent with shorter ones.

In the case of single-threaded execution, PRRT* runs exactly like sequential RRT*. When multiple threads are executing simultaneously, however, some rewiring decisions will be made without the information available from a simultaneous update of another thread. In such a case, a rewiring that would occur under sequential execution would be missed. The probability of such a missed update increases with the number of processors p but decreases with the size of the PRRT* graph n. As time progresses, a previously missed update will be fixed with a future sample with increasing probability. When viewed from the perspective of one of the running threads, the thread is executing RRT* with additional information from other threads occasionally being added, which can only improve the solution. Thus the asymptotic optimality of RRT* holds for PRRT*.

E. Lock-Free KD-Tree

PRRT(*) performs nearest-neighbor searches using a kd-tree [17] split on mid-points [18] and is adapted to to allow concurrent inserts using CAS. A kd-tree successively divides space with splitting hyperplanes at each level of the tree.

LockFreeKDInsert (Algorithm 6) repeatedly bisects the region for insertion by a different dimension (line 7, 8) until if finds a region that does not contain a kd-node. Once found, it performs a CAS (lines 9, 14) to change the pointer from *null* to the new kd-tree node. If the CAS succeeds, the node is inserted and the algorithm returns. If instead another thread already updated the pointer, the CAS will fail, and the algorithm will continue to walk down the tree until it can attempt another insert.

The implementations of NEAREST and NEAR based upon this kd-tree implementation do not change from their standard version. When they encounter a *null* branch, they terminate their traversal as with the sequential version. The parallel nature of the inserts and queries changes the semantics to mean that they return a snapshot in time of the nearest node and near nodes respectively.

The kd-tree can be used for any number of dimensions, but may become inefficient in very high dimensional spaces [19]. Even in such cases, kd-trees distribute random updates throughout the tree, leading to low contention over insertion points. In brute-force approaches based upon arrays or lists, inserts at a single insertion point (e.g. the tail of the list/array) may result in contention.

In practice PRRT(*) could be used with other nearest-neighbor search approaches that allow for non-blocking searches, allow low-contention updates, and provide partitioned locality properties.

V. Results

We evaluate our method with two scenarios: (1) a holonomic disc-shaped robot moving in a planar environment, and (2) an Aldebaran Nao small humanoid robot performing a 2-handed manipulation task using 10 DOF. Results are computed on a system with four Intel x7550 2.0GHz 8-core Nehalem-EX processors for a total of 32 cores. Each processor has an 18MB shared L3 cache and each core has a private 256KB L2 cache as well as 32KB L1 data and instruction caches. Although the x7550 supports Hyper-Threading (Intel's simultaneous multithreading technology), this facility is turned off in the BIOS.

A. 2D Holonomic Disc-shaped Robot

We executed our PRRT* implementation for a 2D holonomic disc-shaped robot that must move to the goal in the environment shown in Fig. 1(a). To demonstrate PRRT*'s ability to compute higher quality paths per given time than

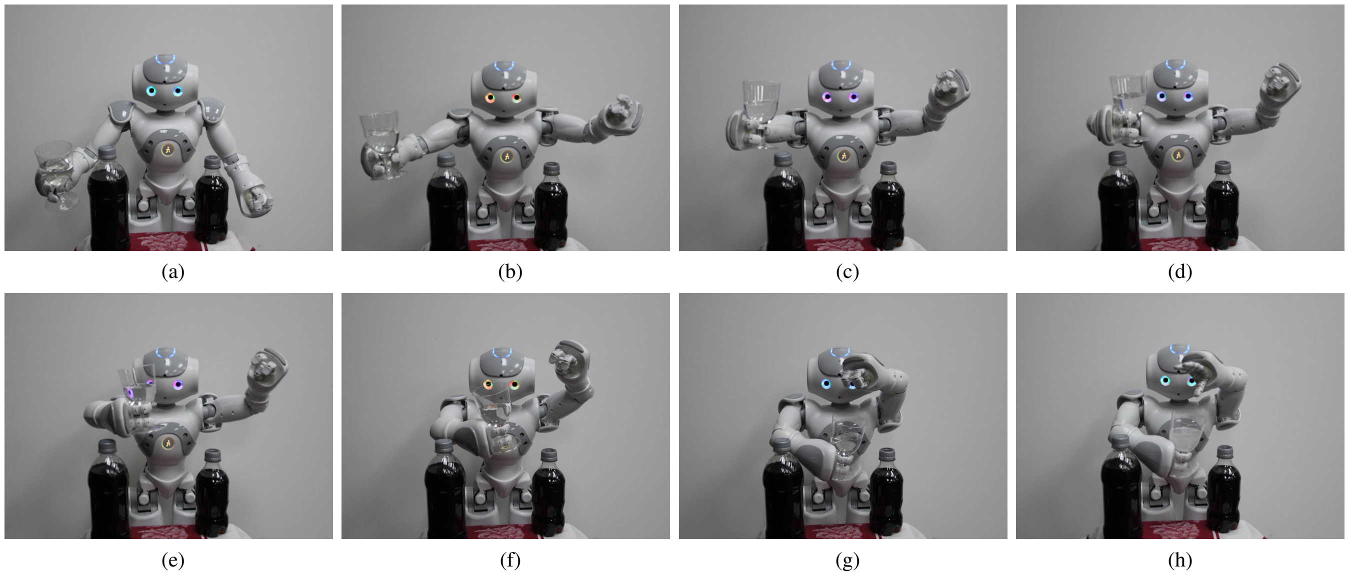


Fig. 2. An example motion plan created for the Nao robot. The robot carries an effervescent antacid in one hand and places it over a glass of water held in the other hand, all while avoiding the bottles on the table and not spilling the water (i.e. `FEASIBLE` is constrained to keep the glass mostly level). In the last frame, after the robot reaches the goal configuration, it drops the antacid into the water.

RRT* on a multi-core PC, we executed the algorithms on 1, 4, and 32 cores for 10 ms of wall clock time. As shown in Fig. 1, with more cores the size of the constructed tree in the 10 ms increases substantially, visibly improving the quality of the computed motion plan. More space is explored and more narrow passages are discovered. The best path is improved from 6.92% above optimal with 1 core to 1.62% above optimal with 4 cores and to 0.98% above optimal with 32 cores.

B. 2-handed Aldebaran Nao task (10 DOF)

We executed our PRRT* implementation on an Aldebaran Nao robot with the task of dropping an object held in one hand into a cup held in the other while avoiding obstacles. Each arm has 5 degrees of freedom—shoulder pitch/roll, elbow yaw/roll and wrist yaw. All joints are bounded revolute joints, and we define `COST` as a Euclidian distance. The robot must avoid obstacles on the table in front of it while keeping the cup upright throughout its motion. We define `GOAL` to return true for configurations that satisfy the following constraints (subject to a tolerance): (1) the (x, y) coordinate for the left hand and the right hand is the same, (2) the left hand's z coordinate is higher than the right hand, (3) the object in the left hand is pointing down, and (4) the cup in the right hand is held upright. The function `FEASIBLE` tests not only for collisions of the robot and objects in the environment but also requires that the cup be upright.

To demonstrate PRRT*'s ability to compute high quality solutions faster on multiple cores, we executed the Nao scenario for $n = 100,000$ samples with varying core counts and averaging over 10 runs. As shown in Fig. 3, we observe superlinear speedup with PRRT*. Executing PRRT* on 1 core (thus making it equivalent to standard RRT*) requires 420 seconds. On 32 cores, PRRT* required only 11.6 seconds for the same number of samples. PRRT* was 36x faster with

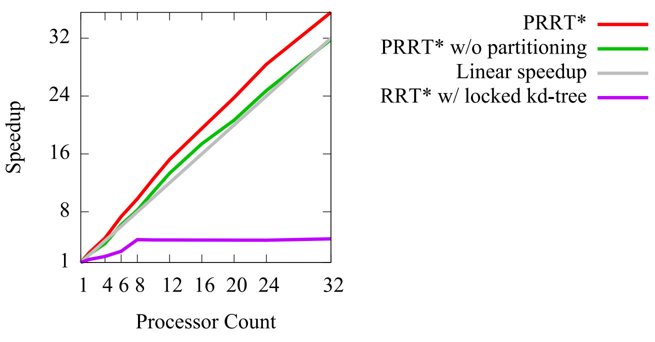


Fig. 3. PRRT* run on the Nao 10 DOF task for 100,000 samples at varying core counts. At 100,000 samples and high core counts, PRRT* experiences superlinear speedup, whereas a lock-based approach cannot exceed 4x speedup.

no significant difference in the quality of the computed paths.

We also executed the Nao scenario using other parallelization approaches. When we use the lock-free approach of PRRT* but without partition-based sampling, we see linear speedup. We also executed RRT* parallelized by locking the kd-tree: at 100,000 samples, nearest neighbor searches dominate the computation time and threads spend most of their time waiting for access to the kd-tree. However, we note that when the size of the tree τ is smaller, collision-detection dominates computation time and the lock-based approach achieves a more reasonable speedup. At 2,000 samples on 32 cores, we observe a 16.4x speedup with locked kd-trees, although PRRT* still outperforms with a 28.9x speedup. The locked version's speedup diminishes as more samples are added. In contrast, the lock-free PRRT* overcomes thread startup overhead and reaches 32x speedup by the 20,000th sample before increasing to 36x speedup by 100,000 samples.

To demonstrate how PRRT* can be used to produce better

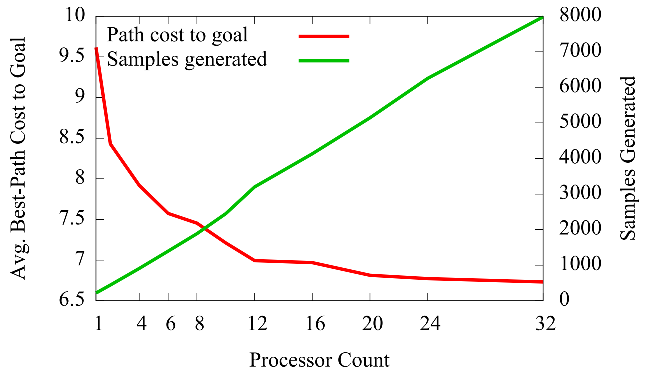


Fig. 4. PRRT* run for 3 seconds on the Nao scenario. The graph shows increasing processor counts generating more samples, resulting in better solutions.

results per unit time, we also ran the Nao scenario 50 times for 3 seconds at various processor core counts. As shown in Fig. 4, increasing the number of processor cores enables us to build trees with more samples per second and find better solutions. The path cost from the initial configuration to the goal shows convergence to an optimal solution as the number of samples increases, as expected with RRT*. We also observed that RRT* would find paths to the goal in only 80% of the 3-second runs on 1 core. With 2 cores, PRRT* found solutions in 98% of the runs. At higher core counts, PRRT* found solutions in all runs.

VI. Conclusion

We presented PRRT (Parallel RRT) and PRRT* (Parallel RRT*), sampling-based methods for feasible and optimal motion planning that are tailored to execute on modern multi-core CPUs. Using atomic updates and lock-free data structures, PRRT and PRRT* remove barriers to scaling to higher processor core counts, enabling linear speedup. We further show how using a non-overlapping partition-based sampling strategy effectively localizes a thread's computation to a region of memory, increasing the overall cache efficiency and enabling superlinear speedup.

Our method is best suited for challenging motion planning problems in which a large number of samples is required to find a feasible or near optimal solution. As the number of samples increases, computation time gradually changes from being dominated by collision detection to being dominated by nearest neighbor search. PRRT and PRRT* parallelize the entire computation of the motion planning tree and thus maintain speedup ratios regardless of which portion of the computation is dominating. We demonstrated fast performance and superlinear speedups in 2 scenarios: (1) a holonomic disc-shaped robot moving in a planar environment, and (2) the Aldebaran Nao small humanoid robot performing a 2-handed, 10-DOF manipulation task.

In future work, we will investigate adaptive approaches for distributing sampling partitions between threads, and additional nearest neighbor approaches that may allow for even more effective use of cache locality. We will also leverage the CPU-based nature of PRRT(*) to integrate with OMPL [20] and ROS [21].

VII. Acknowledgment

The authors thank Allan Porterfield at the North Carolina Renaissance Computing Institute (RENCI) for providing access to computation hardware and Jan Prins and Stephen Olivier for their input on parallel algorithms and platforms. This research was supported in part by the National Science Foundation (NSF) through awards #IIS-0905344, #IIS-1117127, and #IIS-1149965 and by the National Institutes of Health (NIH) under award #R21EB011628.

References

[1] S. M. LaValle, *Planning Algorithms*. Cambridge, U.K.: Cambridge University Press, 2006.

[2] S. Karaman and E. Frazzoli, "Sampling-based algorithms for optimal motion planning," *Int. J. Robotics Research*, vol. 30, no. 7, pp. 846–894, June 2011.

[3] J. D. Valois, "Lock-free linked lists using compare-and-swap," in *Proc. ACM Symp. Principles of Distributed Computing*, 1995, pp. 214–222.

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

[5] S. Carpin and E. Pagello, "On parallel RRTs for multi-robot systems," in *Proc. 8th Conf. Italian Association for Artificial Intelligence*, 2002, pp. 834–841.

[6] E. Plaku and L. E. Kavraki, "Distributed sampling-based roadmap of trees for large-scale motion planning," in *Proc. IEEE Int. Conf. Robotics and Automation (ICRA)*, April 2005, pp. 3879–3884.

[7] M. Otte and N. Correll, "Path planning with forests of random trees: Parallelization with super linear speedup," Department of Computer Science University of Colorado at Boulder, Tech. Rep. CU-CS 1079-11, Apr. 2011.

[8] J. Pan, C. Lauterbach, and D. Manocha, "g-Planner: Real-time motion planning and global navigation using GPUs," in *AAAI Conference on Artificial Intelligence (AAAI)*, July 2010, pp. 1245–1251.

[9] J. Lengyel, M. Reichert, B. R. Donald, and D. P. Greenberg, "Real-time robot motion planning using rasterizing computer graphics hardware," in *Proc. ACM SIGGRAPH*, 1990, pp. 327–335.

[10] K. Hoff III, T. Culver, J. Keyser, M. C. Lin, and D. Manocha, "Interactive motion planning using hardware-accelerated computation of generalized Voronoi diagrams," in *Proc. IEEE Int. Conf. Robotics and Automation (ICRA)*, Apr. 2000, pp. 2931–2937.

[11] C. P. Kenneth, K. H. Iii, M. C. Lin, and D. Manocha, "Randomized path planning for a rigid body based on hardware accelerated voronoi sampling," in *Proc. Workshop on the Algorithmic Foundations of Robotics (WAFR)*, 2000.

[12] M. Foskey, M. Garber, M. C. Lin, and D. Manocha, "A Voronoi-based hybrid motion planner," in *Proc. IEEE/RSJ Int. Conf. Intelligent Robots and Systems (IROS)*, Oct. 2001, pp. 55–60.

[13] J. T. Kider Jr., M. Henderson, M. Likhachev, and A. Safonova, "High-dimensional planning on the GPU," in *Proc. IEEE Int. Conf. Robotics and Automation (ICRA)*, 2010, pp. 2515–2522.

[14] W. Hwu, *GPU Computing Gems Jade Edition*, ser. Applications of GPU Computing Series. Elsevier Science & Technology, 2011.

[15] J. J. Bialkowski, S. Karaman, and E. Frazzoli, "Massively parallelizing the RRT and the RRT*," in *Proc. IEEE/RSJ Int. Conf. Intelligent Robots and Systems (IROS)*, Sept. 2011, pp. 3513–3518.

[16] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 3rd ed. Morgan Kaufmann, 2003.

[17] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Commun. ACM*, vol. 18, no. 9, pp. 509–517, Sept. 1975.

[18] S. Maneewongvatana and D. M. Mount, "It's okay to be skinny, if your friends are fat," in *Center for Geometric Computing 4th Annual Workshop on Computational Geometry*, 1999.

[19] A. Yershova and S. M. LaValle, "Improving motion-planning algorithms by efficient nearest-neighbor searching," *IEEE Trans. Robotics*, vol. 23, no. 1, pp. 151–157, 2007.

[20] The Open Motion Planning Library (OMPL), "The Open Motion Planning Library (OMPL)," http://ompl.kavrakilab.org, 2010.

[21] Willow Garage, "Willow Garage Robot Operating System (ROS)," http://ros.org, 2010.