Path Planning with Forests of Random Trees: Parallelization with Super Linear Speedup

Michael Otte and Nikolaus Correll

michael.otte@colorado.edu, nikolaus.correll@colorado.edu

Department of Computer Science
University of Colorado at Boulder

Technical Report CU-CS 1079-11

April 2011

Path Planning with Forests of Random Trees: Parallelization with Super Linear Speedup

Michael Otte and Nikolaus Correll

***Abstract*—We propose a new parallelized high-dimensional single-query path planning technique that uses a coupled forest of random trees (i.e., instead of a single tree). We present both theoretical and experimental results that show using forests of random trees can lead to expected super linear speedup, with respect to the number of trees in the forest. In other words, with T trees running in parallel, we expect to get a particular quality result in less than $1/T$ the time required by a single tree (this is also known as having efficiency greater than 1). Our algorithm works by linking the random sampling and pruning mechanisms of all trees in the forest to the length of the current best path found by any tree. This enables all trees to avoid sampling from large portions of the configuration space that cannot possibly lead to better solutions, and increases speed by enabling trees to prune obsolete nodes. The current best solution is also passed between trees, so that it may be improved by any tree in the forest. Given the potential of super linear speedup, we additionally propose a sequential version of the forest algorithm that works by dividing computation time between each of T trees. We perform a series of experiments and find that both the parallel and sequential versions of the forest algorithm perform well in practice (e.g., vs. a single tree or smaller forests). To demonstrate that our algorithm is generally applicable, experiments are performed using two different state-of-the-art random tree algorithms for the underlying random trees. Theoretical analysis suggest that these results can be duplicated for any random tree path planning algorithm that meets a few requirements stated in the paper.**

I. Introduction

We present a new technique for single-query high-dimensional path planning. The basic idea is to use a forest of random trees instead of a single tree. The forest is special in the sense that both the random sampling and pruning mechanisms of all trees are coupled. That is, they are defined in terms of the length of the current best solution known to any tree in the forest. This allows all trees to prune themselves based on the current best path, and also to avoid sampling new points from portions of the configuration space that cannot possibly produce better solutions. This reduces the time required to insert new nodes into the tree, and increases the chances new points produce better solutions, respectively. Additionally, the current best solution is shared throughout the forest so that all trees have a chance to improve it directly. In section III we show that using a forest of random trees can lead to super linear efficiency vs. number of trees in the forest. That is, with T trees, we can expect solutions of a particular quality to be found in less than $1/T$ the time required by a single tree. This is also known as having efficiency > 1.

In Sections II and IV we propose two similar but different forest algorithms: a parallel version where each computational unit grows a single tree, and a sequential version where computation time is divided between each tree in the forest, respectively. Either algorithm is able to utilize any underlying random tree algorithm and configuration space, as long as the following three conditions are met: (1) the configuration space obeys the triangle inequality. (2) a heuristic estimate of the cost between two points can be obtained that is exactly what the actual cost would be if no obstacles were present. (3) The underlying tree is expected to converge to an optimal solution, given infinite time.

The rest of this paper is organized as follows: the remainder of this section is devoted to background information on path planning and related work. In Section V we perform a series of experiments comparing the performance of various sized forests to single trees. Both the parallel and sequential version of the forest algorithm are evaluated using two different state-of-the-art random trees. Discussion and conclusions are presented in Sections VI and VII, respectively.

A. Background

The path planning problem is that of finding a sequence of actions that cause a system to transition from an initial state to a goal state. While efficient complete grid-based methods exist for the 2D and 3D cases (Dijkstra [4], Hart et al. [9], Stentz [21], Koenig and Likhachev [12], Ferguson and Stentz [5]), the PSPACE-hardness of complete planning causes complete algorithms to be impractical in higher dimensions (Reif [18], Hopcroft et al. [10]). Current algorithms of choice for planning in higher-dimensions work by randomly sampling the environment to create a random graph that is then searched using standard graph techniques. In general, these algorithms are probabilistically complete—the chance that they find a solution (if one exists) approaches one as time approaches infinity. Unlike low-dimensional grid methods, most of the computational effort is spent on graph creation (vs. graph search for a path). Depending on the application, high-dimensional graph based planners tend to come in one of two flavors: multi-query and single-query.

Multi-query planners are used when many searches are expected to be performed in the same environment. A detailed graph through the configuration space is created, stored, and possibly improved over time. Paths are calculated by connecting start and goal states to the graph and then searching for a path between them (Overmars and Svestka [17], Koga and Latombe [13], Sanchez and Latombe [19, 20], Clark et al. [3]).

Single-query planners are used when a planner is expected to encounter a different environment every time it plans. A detailed graph is not saved, since each graph will only be used

once. Instead, the planner builds the best graph it can within the allotted planning time. Single-query planners usually take the form of random tree algorithms that fuse the graph creation and search operations. Newly sampled points are inserted into the tree as soon as they can be connected to the existing graph. Points that cannot be connected to the graph are ignored. The forest algorithms presented in this paper build directly on single-query random trees.

One of the earliest and most widely used single-query planners is the *rapidly exploring random tree* or *RRT* (LaValle and Kuffner [14, 15]). Re-planning versions also exist (Ferguson et al. [8], Ferguson and Stentz [7]). While RRT provides nice probabilistic coverage guarantees, the resulting paths tend to wander—for instance, Karaman and Frazzoli [11] prove that RRT will almost surly converge to a sub-optimal solution. A number of attempts have been made to eliminate wandering by finding increasingly optimal solutions as time permits. *Any-Time RRT* works by building new trees while time remains, such that each subsequent tree is guaranteed to be better than its predecessor (Ferguson and Stentz [6, 7]). *RRT** carefully chooses a set of candidate nodes such that the resulting algorithm almost surly converges to the optimal solution (Karaman and Frazzoli [11]). The *Any-Com ISS* random tree described by Otte and Correll [16] attaches new nodes in a way that minimizes cost given the current tree, and ongoing remodeling guarantees convergence to the optimal solution. In Section V we evaluate forests that use both RRT* and the Any-Com ISS random tree.

Given that recent breakthroughs have provided algorithms with optimal convergence, it is clear that future advances in path planning will focus on increasing the rate of convergence. We believe that distributed computation is a natural way to achieve this type of acceleration, and that forests are an elegant way to achieve distribution. On the other hand, given the potential for super linear speedup, we also believe that forests of random trees can benefit search in non-distributed settings.

B. Related work

Probabilistic road-maps, a popular multi-query planner, have been shown to be 'embarrassingly parallel' (Amato and Dale [1]). Here parallelization is achieved by having each process randomly sample new points to connect to the graph. Results show approximately linear speedup vs. the number of processors. While it seems reasonable that single-query planners could be parallelized in the same way, this form of distributed algorithm requires each process to have direct access to a single tree stored in memory. In contrast, each tree in our forest algorithm maintains its own memory footprint—knowledge transfer is achieved via exchanging small messages between processes. The latter framework allows the forest to be distributed between processes that do not necessarily have access to a shared global memory structure (e.g., between multiple machines connected by a network).

The Any-Time RRT algorithm builds multiple trees sequentially (Ferguson and Stentz [6]). However, the next tree is not started until the previous tree has been completed and destroyed. In contrast, our forest idea builds all trees simultaneously. Since multiple trees exist at one time, cooperation between them contributes to search progress.

Any-Com ISS (Otte and Correll [16]) is the most closely related work to our own. In that work, the authors present a multi-robot search algorithm that is solved in a distributed manner by a six robot team. The specific algorithm is similar to the distributed version of forests of random trees that we present here, except that the underlying trees are assumed to be a unique type described in that work. Otte and Correll [16] note that their experimental results show super-linear speedup, however no theoretical explanation is given. In contrast, a main contribution of our paper is the theoretical proof that parallelizing search in a coupled forest of random trees can lead to super linear speedup for many different types of random trees. Another difference is that we show how to harness the super linear speedup to improve non-distributed path planning. One final difference is that we perform experiments using forests with many more than six trees, and multiple underlying random tree algorithms.

It is also worth mentioning that, in the field of machine learning, forests of decision trees have proven to be much more powerful than a single decision tree for the problems of classification and regression (Breiman [2]). While the tasks of regression and classification are quite different from the path planning problem we are concerned with, this body of work is an interesting analogue.

II. Parallel Forest Algorithm

The parallel version of the forest algorithm is displayed in Figure 1-top. The best solution $\mathbf{P}_{bst}$ is initialized to the empty set and its length is initialized to ∞ (lines 1-2). Next, T trees are started, each on their own process or computational unit (lines 3). The subroutine $\mathbf{TimeLeft}()$ returns true while planning time remains, otherwise it returns false. Once the allotted planning time has been exhausted, the forest returns the best solution found by any tree (lines 5-7).

The bulk of the algorithm takes place within each individual tree (Figure 1-bottom). Each tree begins by initializing its best known solution to the empty set and its length is to ∞ (lines 1-2). Search happens by picking a random point v from the configuration space with $\mathbf{RandomPoint}(L)$ and then inserting it into the tree using $\mathbf{Insert}(v)$ (lines 4-5). Random sampling happens in a special way that depends on L, the length of the best solution know to all trees; however, we postpone a full discussion on this topic until later. $\mathbf{Insert}(v)$ is assumed incorporate any specific logic required by the underlying random tree. If the new node leads to a (globally) better path, then the new solution is distributed to the other trees (lines 7-8). This can be accomplished in shared memory or via messaging over a network. If a better solution has been found by another tree, then it is added to the local tree using $\mathbf{AddPath}(\mathbf{P}_{bst})$ (lines 10-12), taking care to avoid point duplication. Finally, the local tree is pruned based on the global value of L using $\mathbf{prune}(L)$ (lines 9 and 12).

ParallelForest()

1: $L = \infty$
2: $\mathbf{P}_{bst} = \emptyset$
3: **for** $t = 1$ to T **do**
4: **RandomTree**(t) on its own process
5: **while** **TimeLeft**() **do**
6: sleep
7: **Return** $(L, \mathbf{P}_{bst})$

RandomTree(t)

1: $t.L = \infty$
2: $t.\mathbf{P}_{bst} = \emptyset$
3: **while** **TimeLeft**() **do**
4: $v = t.$**RandomPoint**(L)
5: $t.$**Insert**(v)
6: **if** $t.L < L$ **then**
7: $\mathbf{P}_{bst} = t.\mathbf{P}_{bst}$
8: $L = t.L$
9: $t.$**prune**(L)
10: **if** $L < t.L$ **then**
11: $t.$**AddPath**$(\mathbf{P}_{bst})$
12: $t.L = L$
13: $t.$**prune**(L)

Fig. 1. Algorithm for parallel forests of random-trees (top), and forest tree (bottom). Note that any random tree algorithm can be used, as long as it provides the necessary subroutines.

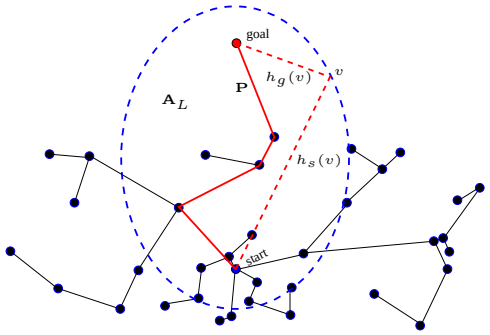

Fig. 2. Boundary of ellipsoid beyond which new points cannot possibly lead to better solutions (dashed-blue). The space within the ellipsoid is denoted $\mathbf{A}_L$ and its boundary is defined by $h_s(v) + h_g(v) = L$, where L is the length of the current best path $\mathbf{P}$ (red).

Let s and g represent the start and the goal states, respectively. Let $h(v_1, v_2)$ be a heuristic function that returns the distance between two states v_1 and v_2 assuming no obstacles are present. Let $h_s(v) = h(s, v)$ and $h_g(v) = h(v, g)$. Let $d_s(v)$ represent that actual distance from the start to v through the current tree. Assume that at least one path to the goal has been found, and the algorithm is currently working on finding better solutions as time remains. The current shortest path between the start and the goal is denoted $\mathbf{P}$, and the distance (or cost) along this path is $L = d_s(g)$.

Any point v for which $h_s(v) + h_g(v) \geq L$ cannot possibly lead to a better $\mathbf{P}$. Geometrically, $h_s(v)+h_g(v) = L$ describes an ellipsoid in the search space (see figure 2). Let the space

$v =$ **RandomPoint**(L)

1: $a = (L - |\mathrm{s} - \mathrm{g}|)/2$
2: $b = \max\{\min\{\mathrm{s}, \mathrm{g}\} - a,$ **MinBounds**$()\}$
3: $c = \min\{\max\{\mathrm{s}, \mathrm{g}\} + a,$ **MaxBounds**$()\}$
4: **repeat**
5: $s = ($**Rand**$(0, 1) * (c - b)) + b$
6: **until** $h_s(v) + h_g(v) < L$

Insert(v)

1: calculate p_v, the prospective parent of v, according to the particular tree algorithm that is being used
2: **if** $d_s(p_v) + h(p_v, v) + h_g(v) < L$ **then**
3: continue to insert v according to the tree algorithm

prune(L)

1: **for** n = each node in the tree **do**
2: **if** $h_s(n) + h_g(n) \geq L$ **then**
3: remove n and all of its descendants

Fig. 3. Subroutines used in the forest of random trees algorithms. **MinBounds**() and **MaxBounds**() return the minimum and maximum coordinates of the configuration space along each dimension.

within the ellipsoid be denoted A_L. We can ignore all points not in A_L for random sampling (**RandomPoint**(L), line 6, Figure 3). Similarly, we can prune any nodes not in A_L (**prune**(L), line 2). Sampling directly from the ellipsoid can be difficult in practice. Instead, we perform initial sampling from the hypercube described by $h_s(v)+h_g(v) = L$ per each dimension (**RandomPoint**(L), lines 1-3), and then disregard points outside the ellipsoid (lines 4-6).

We have also found it useful to disregard any points for which $d_s(v) + h_g(v) \geq L$ (**Insert**(v), line 2). Note this is a greedy strategy, since it does not account for the fact that future tree-remodeling may decrease $d_s(v)$ (also note, this is similar to the priority heap weight used in the A* algorithm). In another greedy approach, we also prune the descendants of pruned nodes (**prune**(L), line 3).

Coupling the sampling and pruning mechanisms of all trees enables the entire forest to grow based on the best solution found so far. Sharing the current best solution gives all trees a chance to improve it.

A few variations on the algorithm are also possible. Often, the insertion and pruning operations can be combined such that irrelevant nodes are removed as they are discovered during the insertion operation. If pruning is especially time consuming, a similar effect can be achieved by deleting trees with a small probability and then regrowing them to contain only the current best solution. Obviously, deletion must be done judicially, since it removes potentially valuable information from the tree. In practice, we have found small values (e.g., probability of deletion 0.01 per 0.01 seconds) to be useful.

III. Analysis

There are two subspaces within $\mathbf{A}_L$ that are relevant. $\mathbf{A}_{\mathbf{P}_{now}}$ is the union of all points not in the tree that, if added, would

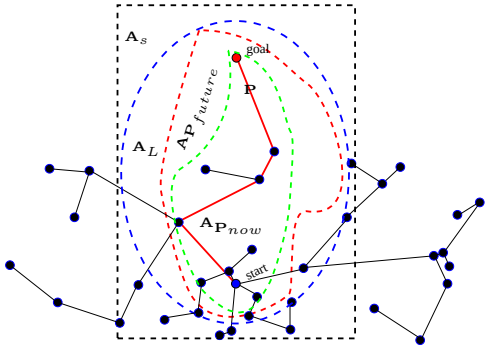

Fig. 4. The relationship between the subspaces $\mathbf{A}_{\mathbf{P}_{now}}$ (dashed-green), $\mathbf{A}_{\mathbf{P}_{future}}$ (dashed-red), $\mathbf{A}_L$ (dashed-blue), and $\mathbf{A}_s$ (dashed-black).

immediately result in a better solution. $\mathbf{A}_{\mathbf{P}_{future}}$ is the union of all points not in the tree that, if eventually added, would eventually enable a better path. $\mathbf{A}_{\mathbf{P}_{now}} \subseteq \mathbf{A}_{\mathbf{P}_{future}}$ and $\mathbf{A}_{\mathbf{P}_{future}} \subseteq \mathbf{A}_L$. Verification of $v \in \mathbf{A}_{\mathbf{P}_{now}}$ is easy, assuming we are alerted when better solutions are found.

$\mathbf{A}_{\mathbf{P}_{future}}$ can be broken into two disjoint subspaces: $\mathbf{A}_{\mathbf{P}_{future,see}}$ contains points that are also within the visibility region of the current tree, and $\mathbf{A}_{\mathbf{P}_{future,blind}}$ contains points that cannot see the tree due to collisions. If sampled, points from $\mathbf{A}_{\mathbf{P}_{future,see}}$ will be added to the tree, while those in $\mathbf{A}_{\mathbf{P}_{future,blind}}$ will not. Note $\mathbf{A}_{\mathbf{P}_{now}} \subseteq \mathbf{A}_{\mathbf{P}_{future,see}}$.

We assume $\mathbf{A}_{\mathbf{P}_{future,see}}$ cannot be calculated in practical time—if it could, then better planning algorithms exist (e.g., gradient decent). Let $\mathbf{A}_s$ denote the space we actually use for sampling. In practice $\mathbf{A}_s \supseteq \mathbf{A}_L$. Figure 4 depicts the relationship between $\mathbf{A}_{\mathbf{P}_{now}}$, $\mathbf{A}_{\mathbf{P}_{future}}$, $\mathbf{A}_L$, and $\mathbf{A}_s$.

Let the volume of a subspace be denoted $\|\cdot\|$. Assuming points are sampled uniformly at random from $\mathbf{A}_s$, and a subspace $\mathbf{A} \subseteq \mathbf{A}_s$, it is possible to express the probability of choosing a point in $\mathbf{A}$ as $\mathrm{P}(v \in \mathbf{A}) = \|\mathbf{A}\|/\|\mathbf{A}_s\|$. The probability a point sampled uniformly at random leads to a better solution is $\mathrm{P}(v \in \mathbf{A}_{\mathbf{P}_{now}}) = \|\mathbf{A}_{\mathbf{P}_{now}}\|/\|\mathbf{A}_s\|$.

If the sizes of the subspaces were not tied to the structure of the tree, then the expected number of samples needed to find a better path would be $\mathrm{E}_{\mathbf{P}} = \|\mathbf{A}_s\|/\|\mathbf{A}_{\mathbf{P}_{now}}\|$. However, since adding new points changes the sizes of $\mathbf{A}_{\mathbf{P}_{now}}$ and $\mathbf{A}_{\mathbf{P}_{future,see}}$, we cannot do this. However, we know that if $\mathbf{A}_{\mathbf{P}_{future,see}}$ changes due to a node insertion that does not result in a better path being found (i.e., the node $v \in \mathbf{A}_{\mathbf{P}_{future,see}}, \notin \mathbf{A}_{\mathbf{P}_{now}}$), then $\mathbf{A}_{\mathbf{P}_{future,see}}$ will not get smaller (and $\mathbf{A}_{\mathbf{P}_{future,blind}}$ will not get bigger). Similarly, $\mathbf{A}_{\mathbf{P}_{now}}$ will not get smaller until a better path is found. Thus, the probability of picking a node in $\mathbf{A}_{\mathbf{P}_{now}}$ will not decrease until a better path is found, and an upper bound on the expected number of insertions required to find a better solution is given by $\mathrm{E}_{\mathbf{P}} < \|\mathbf{A}_s\|/\|\mathbf{A}_{\mathbf{P}_{now}}\|$.

Let $p = \mathrm{P}(v \in \mathbf{A}_{\mathbf{P}_{now}})$. We assume that a better path can actually be found. Thus, $0 < p \leq 1$. Let $f(n)$ represent the insertion time required to add a new node to a tree, assuming n nodes are already in the tree. $f(n)$ is a stand-in for something like $\log(n)$ or n. In all non-trivial random trees $f(n) = \Omega(c)$, for a constant c, since attaching new nodes in a useful way requires performing a search of the tree. Assuming that every sampled point is added to the tree, the expected time to find a better path is:

$$\mathrm{E} = \sum_{j=1}^{\infty} \left[(1-p)^{j-1} p \left(\sum_{i=0}^{j-1} f(n+i) \right) \right] \tag{1}$$

Where $(1-p)^{j-1}p$ is the probability the j-th node insertion leads to a better path, and $\sum_{i=0}^{j-1} f(n+i)$ is the cumulative time required to insert j nodes into the tree. Assuming $\lim_{m\to\infty} (1-p)^m f(n+m-1) = 0$ (that is, $f(n) = o(c^n)$, note the little 'o'), Equation 1 can be reduced to:

$$\mathrm{E} = \sum_{i=0}^{\infty} (1-p)^i f(n+i) \tag{2}$$

Assume that we build a forest of T random trees, such that all trees simultaneously search for a path between the same start and goal locations in the configuration space. Let p_t denote the probability an insertion into tree t leads to a better solution. If all trees simultaneously add a new point, then the probability at least one tree finds a better path is:

$$\mathrm{P}_T = 1 - \prod_{t=1}^{T} (1-p_t) \tag{3}$$

If all trees require the same amount of time to insert a new node (i.e., they are all the same size), then the expected number of iterations until a better path is found is $1/\mathrm{P}_T$. In practice, different trees may have different n and different values for $f(n)$. We can calculate an upper bound on the expected time to a new better path, if we use the maximum $f(n)$ over t per each iteration. Assuming the insertion function is non-decreasing, $f(n+1) \geq f(n)$ for all $n > 0$, then it follows that $f(\max_t(n_t) + i) = \max_t f(n_t + i)$. Let $n_{max} = \max_t(n_t)$. The upper bound on the expected time until the discovery of a better forest path is given by:

$$\mathrm{E}_T \leq \sum_{j=1}^{\infty} \left[(1-\mathrm{P}_T)^{j-1} \mathrm{P}_T \left(\sum_{i=0}^{j-1} f(n_{max}+i) \right) \right] \tag{4}$$

Assuming $\lim_{m\to\infty} (1-\mathrm{P}_T)^m f(n+m-1) = 0$ (again, that is $f(n) = o(c^n)$ with a little 'o'), we obtain:

$$\mathrm{E}_T = \sum_{i=0}^{\infty} (1-\mathrm{P}_T)^i f(n+i) \tag{5}$$

Comparing Equations 2 and 5 shows how the parallelized multi-tree version of the forest is expected to perform vs. a single search tree, respectively. The expected time required to find a better path is less for the forest than for a single tree, as long as $1 - \prod_{t=1}^{T}(1-p_t) \geq p$, since $0 < p_t \leq 1$ and $0 < p \leq 1$, and $T > 1$.

If we assume that $p_t = p$ for all t (that is, the probability a new node leads to a better path is the same for all forest trees and the single tree), then we see that the forest with $T > 1$ is always expected to find a better path before a single tree. This

is intuitive, since more samples from a random distribution increases the probability of finding what we are looking for. Note $\lim_{T\to\infty} 1 - \prod_{t=1}^{T}(1-p_t) = 1$, and as a result, the expected time to a new solution decreases to 0 as the number of trees in the forest approaches infinity.

We would like to investigate what conditions, if any, allow super linear speedup. Namely, when is the following true:

$$\mathrm{E}_T < \mathrm{E}/T. \tag{6}$$

In general, super linear speedup is rare, showing when it can happen is a delightful treat. This is also of particular interest because it tells us when we would theoretically do better to use a forest instead of a single tree when using a single processor (e.g., splitting computation time between each tree).

In particular, we want to see when Equation 6 holds for popular search-tree algorithms (e.g., when $f(n)$ is linear, logarithmic, etc.). The basic idea is to assume that each tree is equally likely to find a better solution $p = p_t$, and then solve for a relation between n_{max} and n.

Given that the expected time to a solution is less for the forest (regardless of super linear speedup or not), the number of nodes in a forest tree is likely to be less than that in a stand-alone tree ($n_{max} \leq n$), especially after a few solutions have already been found (e.g., due to pruning and reduced sample space). Smaller tree size translates into a smaller insertion time per new node, which helps reduce the expected time to a better path. Hence, we would like to know how large forest trees can get (n_{max}), relative to a stand-alone tree (n), while still facilitating super linear speedup.

The assumption $p = p_t$ is fair if we assume that all trees have the same L, and that the size of the subspaces depicted in Figure 2 are dependent on L given a particular environment (recall that all trees in the forest have the same L), and/or that the search space is sufficiently populated such that $\mathbf{A}_{\mathbf{P}_{future,see}} >> \mathbf{A}_{\mathbf{P}_{future,blind}}$ (so helpful points are unlikely to be ignored due to unnecessary obstacle collisions). Also, if we assume that the tree is sufficiently populated such that adding new points does not significantly change the size of $\mathbf{A}_{\mathbf{P}_{now}}$, then we can assume that p is a static value independent of j. For computational ease, we let $q = 1 - p$, where $0 \leq q < 1$. Substituting into Equation 6 gives:

$$\sum_{i=0}^{\infty} q^{Ti} f\left(n_{max}+i\right) < \frac{1}{T}\sum_{i=0}^{\infty} q^{i} f\left(n+i\right) \tag{7}$$

We start with the linear case $f(n) = c_1 n$, where c_1 is a constant greater than 0. Note that c_1 cancels from either side.

$$\sum_{i=0}^{\infty} q^{Ti} \left(n_{max}+i\right) < \frac{1}{T}\sum_{i=0}^{\infty} q^{i} \left(n+i\right) \tag{8}$$

Solving for the limit of the sum as the number of terms approaches infinity, we are able to obtain the following closed-form solution:

$$n_{max} < \frac{(1-q^T)(pn+q)}{Tp^2} - \frac{q^T}{1-q^T} \tag{9}$$

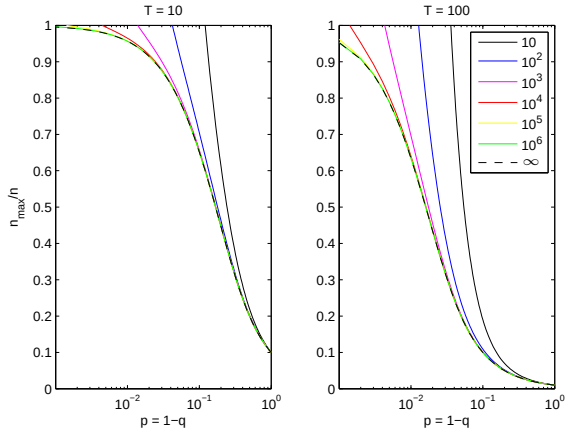

Fig. 5. Linear insertion function $f(n)$ super linear speedup range. The vertical axis displays the maximum ratio n_{max}/n that allows super linear speedup vs. T, given the values of n, T, and p. Different colors indicate different values of n.

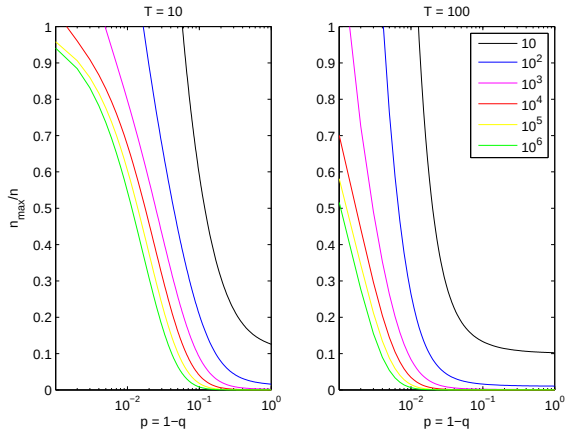

Fig. 6. Logarithmic insertion function $f(n)$ super linear speedup range. The vertical axis displays the maximum ratio n_{max}/n that allows super-linear efficiency in T, given the values of n, T, and p. Different colors indicate different values of n.

Figure 5 displays the maximum ratio of n_{max}/n for which the inequality holds, given various values of n, T, and $p = 1 - q$. The area beneath a curve represents the region in which the expected speedup is super linear. For example, given $T = 10$ trees and $p = 10^{-1}$, if a single tree would have $n = 10^3$ nodes given a particular L, then super linear speedup is expected when forest trees have less than about $0.6n$ nodes. Recall that forest trees tend to require fewer nodes per L because they are expected to find solutions faster—regardless of super linear speedup or not. As n approaches infinity, the ratio n_{max}/n approaches $(1 - q^T)/(Tp)$. This is represented by the black dashed line in Figure 5.

The next case we examine is when $f(n)$ is logarithmic $f(n) = c_2 \log_2(n)$, where c_2 is a constant greater than 0. Note that c_2 cancels from either side.

$$\sum_{i=0}^{\infty} q^{Ti} \log_2\left(n_{max}+i\right) < \frac{1}{T}\sum_{i=0}^{\infty} q^{i} \log_2\left(n+i\right) \tag{10}$$

We are unable to find a closed-form solution, however it is still possible to evaluate the inequality numerically using partial summations to obtain an estimate that is arbitrarily accurate. Note that the size of the terms rapidly decreases due to the exponentiation q^i vs. the slow growth of $\log_2(n+i)$. Figure 6 displays the the maximum ratio of n_{max}/n for which the logarithmic inequality holds.

The final case we examine is the insertion function used by RRT*. In RRT* all nodes within a particular d-ball are evaluated for possible connection to a new node, where d is the dimensionality of the configuration space. The radius of the d-ball is calculated as $\min\left\{c_3((\log n)/n)^{1/d}, c_4\right\}$, where c_3 and c_4 are constants defined in terms of d (see Karaman and Frazzoli [11] for more details). This means that the volume of the d-ball is $\mathbf{A}_{ball} = \min\{c_5((\log_2 n)/n), c_6\}$, where c_5 and c_6 are constants dependent on d. If we assume that nodes are evenly distributed in $\mathbf{A}_s$, then the expected number of nodes evaluated per insertion is $nr\|\mathbf{A}_{ball}\|/\|\mathbf{A}_s\|$, where r is the ratio between the number of nodes in $\mathbf{A}_s$ and the number of nodes in the tree. We can assume r is a constant if most nodes are in $\mathbf{A}_s$, and $r = 1$ if the tree is pruned as described in 1. This gives an expected node insertion time of:

$$f(n) = \min\{c_7 \log_2(n), c_8 n\} \tag{11}$$

Where c_7 and c_8 are constants. While it is possible to substitute this result into Equation 8 and then solve numerically as we have done before, we note that logarithmic case wins out whenever $\log_2(n) < c_9 n$ and $\log_2(n_{max}) < c_9 n_{max}$, where $c_9 = c_8/c_7$. Therefore, results for the logarithmic $f(n)$ can be used for RRT*, assuming trees are sufficiently big.

Although we have only shown that the forest can be better for a single iteration, it is easy to see that this can extend to the entire run of the algorithm by induction. Each new value of L significantly affects p via A, and a particular path length reduction reduces $\|\mathbf{A}_s\|$ by an amount exponential in d. Since $\|\mathbf{A}_s\|$ correlates directly to random sampling and pruning, we expect forests of random trees to become more and more useful with increasing d.

IV. Sequential Forest Algorithm

Given the potential for super linear speedup vs. T, we propose a sequential version of the forest algorithm that uses $1/T$-th of computation time on a single processor for each of T trees. The algorithm is presented in Figure 7. The subroutine $\mathbf{TreeTimeLeft}()$ returns true if there is still time for this tree to plan during the current planning iteration. The amount of time allotted to each tree per iteration is small (e.g., on the order of 0.01 second), so that many loops through the forest occur over the course of the search. Note that we move on to the next tree as soon as the previous tree has found a solution, even if time still remains for the previous tree ($\mathbf{RandomTree}(t)$, line 11). We have found this to help during early phases of search, since it enables the next tree to focus a disproportional amount of its effort on improving the current best solution. This effect is diminished once the entire forest is established. However, quickly reducing the search space at the beginning of the search has positive effects that propagate through the rest of the runtime.

$\mathbf{Forest}()$

```
1:  L = ∞
2:  P_bst = ∅
3:  for t = 1 to T do
4:      t.L = ∞
5:      t.P_bst = ∅
6:  while TimeLeft() do
7:      for t = 1 to T do
8:          RandomTree(t)
9:  Return (L, P_bst)
```

$\mathbf{RandomTree}(t)$

```
1:  if L < t.L then
2:      t.AddPath(P_bst)
3:      t.L = L
4:      t.prune(L)
5:  while TreeTimeLeft() and TimeLeft() do
6:      s = t.RandomPoint(L)
7:      t.Insert(s)
8:      if t.L < L then
9:          P_bst = t.P_bst
10:         L = t.L
11:         Return
```

Fig. 7. Sequential forests of random-trees algorithm (top), and individual tree (bottom). Note that any random-tree algorithm can be used, as long as it provides the necessary subroutines. Subroutines are described in Figure 1 in Section II.

V. Experiments

We perform five experiments to evaluate the performance of forests of random trees vs. stand-alone trees. Two different type of trees are used: Any-Com ISS trees [16], which have a linear node insertion function, and RRT* trees [11], which have a logarithmic insertion function in most cases.

The first two experiments use the parallel forest algorithm with Any-Com ISS trees and RRT* trees, respectively. In Experiment 1, robots are given the centralized multi-robot planning task of navigation in an office environment. In Experiment 2, a single robot is given the task of navigation through a maze-like environment. Both experiments are run on a simulated[1] cluster of T computers for $T = [1...64]$. Results are displayed in Figure 8-top and -bottom, respectively.

Plots show mean and standard deviation of solution quality over 20 runs vs. number of trees in the forest (note the logarithmic scale used on the horizontal axis). Different colors

[1]We hope to rerun this experiment on an actual cluster in future work. The simulation works by running each of the T simulated computational units for a small amount of time in a sequence, rewinding the global clock for each unit. Messages sent from each computational unit are not made available until after all units have finished calculation per the current iteration, and message passing is included in the time calculation. Since a real cluster would not limit communication in this way, we expect a real cluster to marginally improve results.

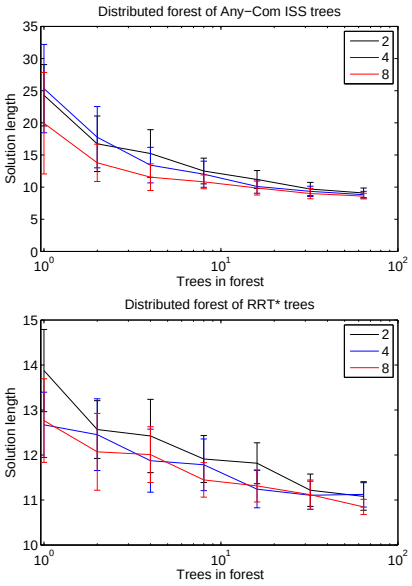

Fig. 8. Experiments 1 and 2. Parallel forests of Any-Com ISS trees, 5 robots in an office environment (top). Parallel forests of RRT* Trees, 1 robot planning in a maze environment (bottom). Results show solution quality vs. forest size. Color denotes planning time. Data-points show mean and standard deviation over 20 runs. Note the log scale of the horizontal axis.

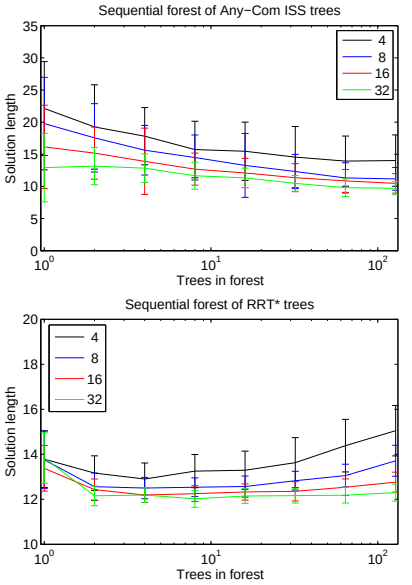

Fig. 9. Experiments 3 and 4. Sequential forests of Any-Com ISS trees, 5 robots in an office environment (top). Sequential forests of RRT* Trees, 1 robot planning in a maze environment (bottom). Results show solution quality vs. forest size. Color denotes planning time. Data-points show mean and standard deviation over 50 runs. Note the log scale of the horizontal axis.

represent different planning times. Assuming T_1 and T_2 represent two different forest sizes, and P_1 and P_2 represent two different planning times, super linear speedup can be observed by comparing data-points for which $T_1 P_1 = T_2 P_2$.

Experiments 3 and 4 are similar to 1 and 2, respectively, except that the sequential version of the forest is used. Forest size $T = [1...128]$. Results are displayed in Figure 9-top and

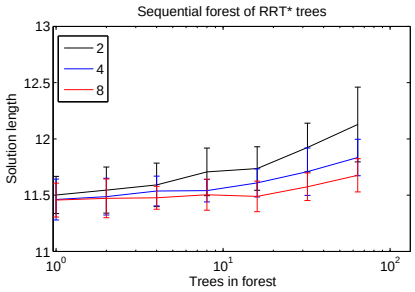

Fig. 10. Experiment 5, Sequential forests of RRT* Trees, 1 robot planning in a simple 2D environment with random obstacles. Results show solution quality vs. forest size. Color denotes planning time. Data-points show mean and standard deviation over 20 runs. Note the log scale on the horizontal axis.

-bottom, respectively. In this case, super linear speedup is observable as a decrease in solution length vs. T.

In Experiment 5 we hope to provide an example of an environment in which the single-processor forest does not have super linear speedup for any T. According to the analysis in section III, this should happen in environments that are relatively easy to plan through (i.e., p is relatively large). The RRT* algorithm is used, since logarithmic insertion functions tend to shrink the super-linear range of p vs. T. Planning is performed in a simple 2D environment with random obstacles, and results are displayed in Figure 10.

VI. Discussion of Results

Overall, we find that planning with forests of random trees works exceptionally well. The parallel forest algorithm gives significantly better results vs. a single tree—more trees correlate to better solutions, regardless of the type of tree being used. We observe super-linear speedup in the Any-Com ISS tree, in general, and super linear speedup for small numbers of trees with RRT*. Recall that Figures 5 and 6 in Section III show that the range of p conducive to super linear speedup stabilizes vs. T for linear insertion functions but decreases with T for logarithmic insertion functions. Thus, the experimental results agree well with the theoretical analysis, since the Any-Com ISS tree has a linear order insertion function and function and RRT*'s is essentially logarithmic.

Results are similar for the sequential forest algorithm. This algorithm is only useful when speedup is super linear vs. T. As with the parallel forest, the number of trees that produced super-linear efficiency tended to be small for RRT* (logarithmic insertion function), especially for short planning times. The Any-Com ISS tree (linear insertion function) was observed to perform better with a large numbers of trees. It is important to note that super linear speedup is observed for some T in forests using either type of tree.

Forests of random trees tend to work well in challenging environments, where the probability a new node produces a better solution is relatively low. In very easy environments a single tree may outperform a forest in the non-distributed version of the algorithm, especially when the insertion function is logarithmic and planning time is short (e.g., Experiment

five). However, when parallelization is available (and solution quality per clock-time is the most important evaluation metric), it is always better to use more trees.

VII. Conclusions

We present a new approach to single-query path planning that uses forests of random trees. Forests are coupled such that they sample new random nodes, and prune existing ones, based on the best solution known to any tree in the forest. The current best solution is also shared between trees so that all trees have a chance to improve it.

A significant contribution of this work is the theoretical analysis that shows forests can achieve super linear speedup vs. their number of trees (that is, efficiency can be > 1). Super linear speedup is very rare. When it exists, it is possible to design faster non-distributed algorithms by dividing processing time between each sub-problem (i.e. instead of devoting a unique cpu to each sub-problem). Therefore, we also propose a non-distributed sequential forest of random trees algorithm that operates according to this concept.

We perform experiments using two different state-of-the art underlying random trees. The experiments show that parallel forests always out-perform a stand-alone tree, often with super linear speedup. Likewise, sequential forests outperform a stand-alone tree in many circumstances (implying super linear speedup). In general, forests of random trees work especially well for difficult problems—when the probability of finding a better path is small.

Acknowledgments

The Authors would like to thank Dustin Reishus and Jason Marden for their conversations about this work.

References

[1] N M Amato and L K Dale. Probabilistic roadmap methods are embarrassingly parallel. In *Proc. IEEE International Conference on Robotics and Automation*, pages 688–694, 1999.

[2] Leo Breiman. Random forests. *Machine Learning*, 45: 5–32, 2001.

[3] Christopher M Clark, Stephen M Rock, and Jean-Claude Latombe. Dynamic networks for motion planning in multi-robot space systems. In *Proc. 7th International Symposium on Artificial Intelligence, Robotics and Automation in Space: i-SAIRAS*, pages 3621–3631, 2003.

[4] E W Dijkstra. A note on two problems in connection with graphs. In *Numererical Mathematics*, volume 1, pages 269–271, 1959.

[5] Dave Ferguson and Anthony Stentz. Using interpolation to improve path planning: The field D* algorithm. *Journal of Field Robotics*, 23:79–101, 2006.

[6] Dave Ferguson and Anthony Stentz. Anytime rrts. In *Proc. IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5369–5375, 2006.

[7] Dave Ferguson and Anthony Stentz. Anytime, dynamic planning in high-dimensional search spaces. In *Proc. IEEE International Conference on Robotics and Automation*, pages 1310–1315, 2007.

[8] Dave Ferguson, N Kalra, and A Stentz. Replanning with rrts. In *IEEE International Conference on Robotics and Automation*, 2006.

[9] P Hart, N Nilsson, and B Raphael. A formal basis for the heuristic determination of minimum cost paths. In *Proc. IEEE Transactions On System Science and Cybernetics (SSC-4)*, pages 100–107, 1968.

[10] J E Hopcroft, J T Schwartz, and M Sharir. On the complexity of motion planning for multiple independent objects; pspace-hardness of the warehouseman's problem. *The International Journal of Robotics Research*, 3: 76–88, 1984.

[11] S. Karaman and E. Frazzoli. Incremental sampling-based algorithms for optimal motion planning. In *Proceedings of Robotics: Science and Systems*, Zaragoza, Spain, June 2010.

[12] Sven Koenig and Maxim Likhachev. Improved fast replanning for robot navigation in unknown terrain. In *Proc. IEEE International Conference on Robotics and Automation (ICRA'02)*, 2002.

[13] Yoshihito Koga and Jean-Claude Latombe. On multi-arm manipulation planning. In *Proc. IEEE International Conference on Robotics and Automation*, volume 2, pages 945–952, 1994.

[14] S M LaValle and J J Kuffner. Randomized kinodynamic planning. In *IEEE International Conference on Robotics and Automation*, pages 473–479, 1999.

[15] S M LaValle and J J Kuffner. Rapidly-exploring random trees: Progress and prospects. In *Algorithmic and Computational Robotics: New Directions*, pages 293–308, 2001.

[16] Michael Otte and Nikolaus Correll. Any-com multi-robot path-planning: Maximizing collaboration for variable bandwidth. In *Proc. 10th International Symposium on Distributed Autonomous Robotics Systems*, 2010.

[17] M Overmars and P Svestka. A probabilistic learning approach to motion planning. In *Algorithmic Foundations of Robotics (WAFR)*, pages 19–37, 1995.

[18] J H Reif. Complexity of the movers problem and generalizations. In *Proceedings of the IEEE Symposium on Foundations of Computer Science*, 1979.

[19] Gildardo Sanchez and Jean-Claude Latombe. On delaying collision checking in prm planning: Application to multi-robot coordination. *The international Journal of Robotics Research*, 21:5–26, 2002.

[20] Gildardo Sanchez and Jean-Claude Latombe. Using a prm planner to compare centralized and decoupled planning for multi robot systems. In *Proc. IEEE International Conference on Robotics and Automation*, 2002.

[21] Anthony Stentz. Optimal and efficient path planning for partially-known environments. In *Proc. IEEE International Conference on Robotics and Automation (ICRA'94)*, 1994.