C-FOREST: Parallel Shortest-Path Planning with Super Linear Speedup

Michael Otte and Nikolaus Correll

***Abstract*—C-FOREST is a parallelization framework for single-query sampling-based shortest-path planning algorithms. Multiple search-trees are grown in parallel (e.g., 1 per CPU). Each time a better path is found, it is exchanged between trees so that all trees can benefit from its data. Specifically, the path's nodes increase the other trees' configuration space visibility, while the length of the path is used to prune irrelevant nodes and to avoid sampling from irrelevant portions of the configuration space. Experiments with a robotic team, a manipulator arm, and the alpha benchmark demonstrate that C-FOREST achieves significant super linear speedup in practice for shortest-path planning problems (team and arm), but not for feasible path panning (alpha).**

***Index Terms*—Path Planning, Robots, Distributed Computing, Parallelization, Super Linear Speedup, Efficiency.**

I. Introduction

Path planning algorithms calculate a sequence of actions that cause a system to transition from an initial state to a goal state while avoiding obstacles, and thus facilitate many autonomous or semi-autonomous applications.

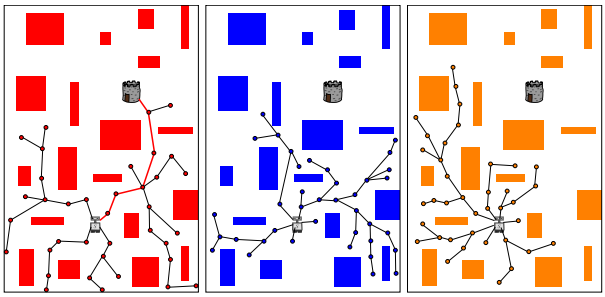
We present a parallelization algorithm for single-query[1] shortest-path planning[2] called *Coupled Forest Of Random Engrafting Search Trees* (C-FOREST). C-FOREST assumes a distributed architecture in which T CPUs communicate. Each CPU builds a *different* search tree between the *same* start and goal states (similar to OR-parallelization [10]). Although most growth is random and independent, message passing enables new exploration and pruning *of all trees* to be a function of the current best solution known to any tree in the forest (unlike OR-parallelization, in which each tree grows completely independently). Nodes from solution branches are also exchanged so they can be engrafted onto and improved by the other trees. The latter allows good solutions to be improved by all trees in the forest, and provides all trees increased visibility of the configuration space. Figure 1 depicts a simple 2D example with 3 trees (CPUs).

C-FOREST is a *parallelization framework* that is designed for single-query shortest-path planning algorithms; thus, it is not a path-planning algorithm *per se*. In other words, C-FOREST is more akin to OR-parallelization than to RRT*. Indeed, C-FOREST is designed to be used with any random tree algorithm operating in any configuration space, such that: (1) the search-tree has almost sure convergence to the optimal

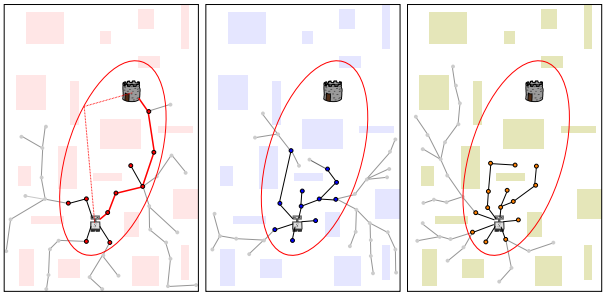
Michael Otte is with the Laboratory for Information and Decision Systems, Massachusetts Institute of Technology; However, the majority of the this work was done when he was at the University of Colorado at Boulder. Nikolaus Correll is with the Department of Computer Science, University of Colorado at Boulder. e-mail: ottemw@mit.edu.

[1]*Single-query* planners are used when the system is expected to encounter a new configuration space each time it plans [1] (in contrast *multi-query* planners expect to perform multiple searches through the same configuration space [2, 3, 4, 5, 6]).

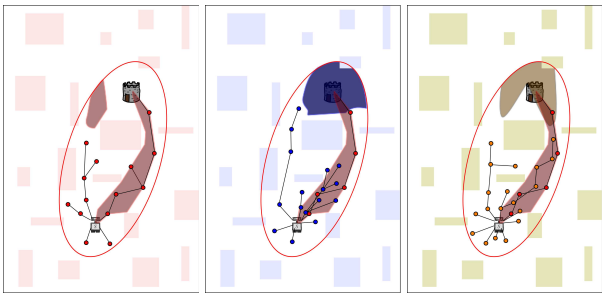
[2]*Shortest-path planning* searches for the *shortest* valid path with respect to a metric [7, 8] (In contrast, *Feasible-path planning* searches for *any* valid path between start and goal [9]).



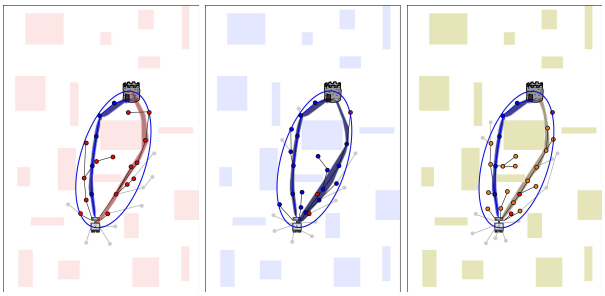
A robot (light gray) is planning a route to the tower (dark gray). Each CPU performs independent random planning until a better solution is found (bold red path at Left), and then the new solution is exchanged between CPUs.



The *length* of this solution defines a boundary (red ellipse). Future samples are drawn from inside the ellipse (because those outside cannot yield better solutions). This increases the probability of finding an even better solution. Existing nodes/branches are pruned (light gray), which decreases the time required to insert future nodes. Sharing the length of the solution (from left CPU to center and right CPUs) gives the advantages of knowing the left CPU's path to all CPUs.



Sharing the path *itself* increases the size of the sub-space from which new samples will yield better paths (dark shaded regions). Thus, sharing the path itself (from the left CPU to the center and right CPUs) further increases the probability of finding an even better solution on any/all CPUs.



As more solutions are found (e.g., center CPU), sharing data ensures that all CPUs can always prune, sample and improve based on the best solution known to any CPU.

Fig. 1. Example of C-FOREST in 2D Euclidean space. The planning on each CPU is represented by a different color (Red, Blue, Yellow).

solution (i.e., in the limit as time approaches infinity), and (2) the configuration space obeys the triangle inequality.

A. Super Linear Speedup vs. stopping criterion

Let w_1 and w_T be the wall time required to solve a particular problem with 1 or T CPUs, respectively. *Speedup* is traditionally defined $S = w_1/w_T$, and measures the relative time benefit of using T CPUs in parallel. Parallelization *efficiency* is defined as $\eta = S/T$ and is inversely proportional to the amount of electrical power required to solve a problem.

C-FOREST will normally be used with an 'any-time' stopping criterion—it will search for better solutions as long as possible, given constraints imposed by safety, time and/or energy expenses, etc. However, it can alternatively use a cost-based stopping criterion—running until the first solution better than a predefined target cost L_{target} is found.

By definition, S and η cannot be calculated when an any-time stopping criterion is used (for any algorithm)—since the any-time stopping criterion manually sets w_1 and w_T based on external factors. Therefore, all discussions on speedup and efficiency in the current paper refer to the L_{target} stopping criterion, which is also used in our experiments. The results obtained with the L_{target} stopping criterion transfer to the any-time stopping criterion because they show how much more (or less) planning time we expect would be required to get the same result using a different T. Note that the use of a particular stopping criterion does not change whether or not an algorithm (e.g., C-FOREST) has almost sure convergence to the optimal solution in the limit as time approaches *infinity* (i.e., when the algorithm never stops).

The experiments in Section IV show that C-FOREST can have super linear speedup ($S > T$, and $\eta > 1$) when the L_{target} stopping criterion is used—we even observe an average $\eta > 9$ in one scenario. This suggests that serially dividing computation between T trees on 1 CPU may also be beneficial (i.e., emulating a distributed architecture on a singe CPU). Therefore, we also describe and evaluate the latter idea, which we call Sequential C-FOREST.

II. Related Work

The path planning problem naturally lends itself to parallelization. Probabilistic Road-Map (PRM), a multi-query algorithm, has been shown to be 'embarrassingly' parallel on memory shared architectures [11]. Each CPU randomly samples and connects new points to the graph, and approximately linear speedup is achieved. A similar idea to [11] is presented in [12], except that RRT and RRT* are parallelized instead of PRM and $\eta = 1.13$ is observed. RRT and RRT* have also been implemented on a GPU such that obstacle detection is performed in parallel with sub-linear speedup [13]. [14] implements the kinodynamic *Planning by Interior-Exterior Cell Exploration* algorithm on a shared memory architecture with sub-linear speedup. In contrast to [11, 12, 13, 14], C-FOREST assumes a message passing architecture in which CPUs do not necessarily have access to shared memory, and we observe greater efficiency—up to $\eta > 9$.

OR-parallelization over a message passing architecture is used for feasible path-planning in [10, 15]. In [10] each CPU independently builds a random tree, and in [15] each CPU uses a "quasi-best-first search algorithm with backtracking." In either case, no data is exchanged between CPUs during the search. In contrast, C-FOREST is designed to solve the shortest-path planning problem and exchanges data during search. Comparing speedups is arguably unfair because [10] and [15] solve the feasible-path planning problem while C-FOREST solves the shortest-path planning problem. However, $\eta > 1$ is observed in both [10], where $\eta = 1.2$, and [15], where $\eta = 1.47$. In contrast, we observe greater speedup (up to $\eta > 9$). We believe that [15] is the first to suggest that it may be advantageous to "virtually" parallelize a super linear distributed algorithm on a single CPU, similar to Sequential C-FOREST.

The *Sampling-based roadmap of trees* (*SRT*) feasible-path planning algorithm has also been implemented on a message passing architecture [16]. SRT can be viewed as PRM with vertices representing trees instead of states. In the distributed version, master CPUs pick the root node of each tree and check for tree combinations. Slave CPUs each grow a single tree that is rooted at a *different* place in the configuration space. $\eta = 1.12$ is observed. In contrast, C-FOREST solves the shortest-path planning problem, trees are rooted at the *same* location and grown on homogeneous CPUs with one phase of operation (and $\eta > 9$, but different problems are being solved).

This paper extends upon our work on *Any-Com ISS* [8, 17]. In [8] the multi-robot shortest-path problem is solved in a distributed manner by a six robot team. [17] extends the idea for dynamic teams. Trees are assumed to be a unique type from [8] and the distributed architecture is a multi-robot team communicating over a wireless network. The current paper assumes a general message passing architecture, generalizes our earlier results to other algorithms (in particular RRT*) and other problem domains (manipulator arms, alpha benchmark).

Multiple trees have also been used for path planning with non-distributed architectures. *Reconfigurable Random Forrest* (*RRF*) [18] is a replanning algorithm where old trees, disconnected by obstacle movement, are saved and tested for connection vs. the current tree. Updated versions of this idea are explored by [19] and [20] and called *Lazy Reconfiguration Forest* and *Multipartite RRTs*, respectively. Major distinctions between these ideas and C-FOREST are that previous work: grows one tree at a time, assumes a serial architecture, and solves the feasible-path planning problem.

Any-Time RRT solves the shortest-path planning problem by building new trees while time remains, such that each new tree is guaranteed to be better than its predecessor [21]. A related idea for the feasible-path planning problem is to restart RRTs if a solution has not been found before a timeout occurs [22]. Both ideas assume a serial architecture and the next tree is not started until after the previous tree has been destroyed. In contrast, C-FOREST builds all trees simultaneously (even sequential C-FOREST uses time division so that all tees exist simultaneously). Since multiple trees exist at one time, cooperation between them contributes to planning progress.

[22] also contains a theoretical analysis applying restarting theory from [23] to the feasible path planning problem, and shows that one particularly extreme case of node pruning (e.g., pruning the entire tree) can be beneficial. The analysis is the

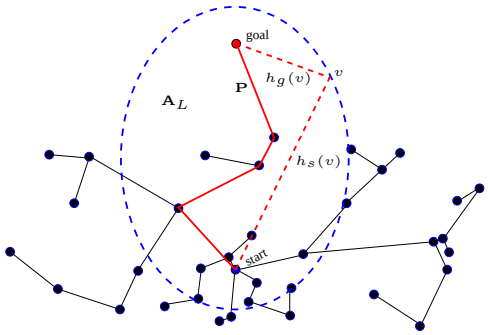


Fig. 2. Boundary beyond which new points cannot lead to better solutions (dashed-blue). The space within the region $\mathbf{A}_L$ and is defined by $h_s(v) + h_g(v) = L$ (dashed-red), where L is the length of the current best path $\mathbf{P}_{bst}$ (solid-red).

first attempt to quantify an advantage of using multiple trees in a path-planning algorithm. That said, the results of [22] do not immediately generalize to the shortest-path planning problem. This is because algorithms like RRT* rely on asymptotically dense node coverage (in the limit as time approaches infinity) to achieve almost sure convergence to the optimal solution (in the limit as time approaches infinity), and dense node coverage is impossible when restarting is used. In contrast, C-FOREST leaves intact the particular nodes required for almost sure convergence to the optimal solution (in the limit as time approaches infinity), while pruning only those nodes that cannot help convergence to the optimal solution.

III. C-FOREST

In the C-FOREST algorithm each CPU builds a *different* search tree between the *same* start and goal states. Tree growth happens in two ways. First, normal random tree growth relies on probabilistically independent node samples until a path is found. Second, each time a better path is found, it is sent to every other tree via message passing so that its nodes can be engrafted. The latter exchange is beneficial in three ways: (1) All trees expand into regions of the configuration space that are known to be beneficial by at least one tree. (2) All trees can prune themselves of globally outdated nodes (i.e., nodes that will not lead to a better solution). (3) All trees can focus their search by avoiding regions of the configuration space that cannot produce globally better solutions. (2) and (3) assume the existence of an admissible heuristic on path length, but can significantly reduce the time required to insert new nodes because node insertion time is dependent on the number of nodes already in a tree.

A. Definitions

A forest $\mathbf{T}$ is a collection of $T = |\mathbf{T}|$ trees. Each tree $\mathbf{t} \in \mathbf{T}$ is a graph consisting of nodes and edges. The prefix '$\mathbf{t}$.' is used to indicate association with a particular tree $\mathbf{t}$. Each node $v \in \mathbf{t}$ exists at a particular point in the configuration space that represents a unique state of the underlying system (e.g., the location and orientation of a robot in an environment). Let $\mathbf{S}$ and $\mathbf{G}$ represent the sets of all valid start and goal states, respectively (in practice it is common to have $|\mathbf{S}| = |\mathbf{G}| = 1$, but this need not be the case). Let $s \in \mathbf{S}$ and $g \in \mathbf{G}$. A path $\mathbf{P} \subseteq \mathbf{t}$ is composed of a sequence of nodes beginning at s and ending at g, such that it is both safe and possible for the system to move from the i-th state to the $(i+1)$-th state. The current best solution $\mathbf{P}_{bst}$ contains $\ell = |\mathbf{P}_{bst}|$ nodes labeled $\mathbf{P}_{bst,i}$ for $1 \leq i \leq \ell$. We assume that the configuration space is a metric space with distance metric $\|\cdot\|$ (and so obeys the triangle inequality). $L = \|\mathbf{P}_{bst}\|$ is the length of the best path.

Let $h(v_1, v_2)$ be an admissible[3] heuristic function that returns an estimate of the distance between v_1 and v_2. In the case of multiple goal states $h(v_1, \mathbf{G})$ is assumed to be admissible over the entire set of goals[4]. More accurate estimates returned by $h(v_1, v_2)$ will increase performance, but if no suitable heuristic can be found then it is possible to define $h(v_1, v_2) \equiv 0$. Let $h_s(v) = h(s, v)$ and $h_g(v) = h(v, \mathbf{G})$. Let $\mathbf{t}.d_s(v)$ be the actual distance from the start to v through tree $\mathbf{t}$. The tree that discovered $\mathbf{P}_{bst}$ is $\mathbf{t}_{bst}$ and $L = \mathbf{t}_{bst}.d_s(g)$ where $g \in \mathbf{t}_{bst}$. Assuming that at least one path to the goal has been found, any point v for which $h_s(v) + h_g(v) \geq L$ cannot possibly lead to a better $\mathbf{P}_{bst}$. Geometrically, $h_s(v) + h_g(v) = L$ describes the boundary of an open set in the search space (Figure 2). Let $\mathbf{A}_L$ be the Lebesgue measure of this open set. For the case of a single goal ($\mathbf{G} = \{g\}$) in Euclidean space $\mathbf{A}_L$ is bounded by an ellipsoid.

B. Algorithm

The C-FOREST algorithm is displayed in Figure 3-Left-Top. $\mathbf{P}_{bst}$ is initialized to the empty set and its length to ∞ (lines 1-2). Next, T trees are started, each on their own CPU (lines 3-4). Each tree $\mathbf{t}$ is an independent version of a random search tree (e.g., RRT*). The subroutine $\mathbf{StopCriterion}()$ returns $true$ once the stopping criterion has been met, otherwise it returns $false$. Once the allotted planning time has been exhausted, the best solution is returned (lines 5-7).

Most computation happens in $\mathbf{RandomTree}(\mathbf{t})$ (Figure 3-Middle-Top). $\mathbf{t}$ initializes $\mathbf{t}.\mathbf{P}_{bst}$ to the empty set and $\mathbf{t}.L$ to ∞ (lines 1-2). Search happens by picking a random point v from the configuration space using $\mathbf{RandomPoint}(L)$ and then inserting it into $\mathbf{t}$ with $\mathbf{t}.\mathbf{Insert}(v)$ (lines 4-5). $\mathbf{RandomPoint}(L)$ and $\mathbf{Insert}(v)$ are assumed to incorporate any specific logic required by the underlying random tree algorithm and/or configuration space. If adding v leads to a (globally) better path, then the new $\mathbf{P}_{bst}$ is sent to the other trees (lines 7-8). If a better solution is found by another tree, then it is added to the local tree using $\mathbf{AddPath}(\mathbf{P}_{bst})$ (lines 11-15). The local tree is pruned based on the global value of L using $\mathbf{t}.\mathbf{Prune}(L)$ (lines 9 and 14), and the sampling bounds are updated using $\mathbf{t}.\mathbf{SetSampleBounds}(L)$ (lines 10 and 15).

The subroutine $\mathbf{AddPath}(\mathbf{P}_{bst})$ is shown in Figure 3-Left-Center. Line 1 iterates over nodes in $\mathbf{P}_{bst}$ from start to goal (the start node does not need to be inserted, since it is guaranteed to exist in $\mathbf{t}$). The subroutine $\mathbf{t}.\mathbf{InTree}(\mathbf{P}_{bst,i})$ checks if $\mathbf{P}_{bst,i}$ is already in $\mathbf{t}$ to avoid duplicating points (line 2). If $\mathbf{P}_{bst,i}$ does not exist in $\mathbf{t}$ then it is inserted on line 3. $\mathbf{Insert}(v, \mathbf{P}_{bst,i-1})$ is a modified version of $\mathbf{Insert}(v)$ that explicitly includes

[3] $h(v_1, v_2)$ will never overestimate the distance between v_1 and v_2.
[4] $h(v_1, \mathbf{G}) \leq$ the distance from v_1 to any and all $g \in \mathbf{G}$.

CFOREST()
1: $L = \infty$
2: $\mathbf{P}_{bst} = \emptyset$
3: **for** $\forall \mathbf{t} \in \mathbf{T}$ **do**
4: **RandomTree**($\mathbf{t}$) on its own CPU
5: **while not StopCriterion**() **do**
6: sleep
7: **Return** $(L, \mathbf{P}_{bst})$

t.**AddPath**($\mathbf{P}_{bst}$)
1: **for** $i = 2$ to $|\mathbf{P}_{bst}|$ **do**
2: **if not** $\mathbf{t}$.**InTree**($\mathbf{P}_{bst,i}$) **then**
3: $\mathbf{t}$.**Insert**($\mathbf{P}_{bst,i}, \mathbf{P}_{bst,i-1}$)

$v =$ **RandomPoint**(L)
1: **repeat**
2: $v = (\mathbf{Rand}(0,1) * (c - b)) + b$
3: **until** $h_s(v) + h_g(v) < L$

RandomTree($\mathbf{t}$)
1: $\mathbf{t}$.**Insert**(s)
2: $\mathbf{t}.L = \infty$
3: $\mathbf{t}.\mathbf{P}_{bst} = \emptyset$
4: **while StopCriterion**() **do**
5: $v =$ **RandomPoint**(L)
6: $\mathbf{t}$.**Insert**(v)
7: **if** $\mathbf{t}.L < L$ **then**
8: $\mathbf{P}_{bst} = \mathbf{t}.\mathbf{P}_{bst}$
9: $L = \mathbf{t}.L$
10: $\mathbf{t}$.**Prune**(L)
11: $\mathbf{t}$.**SetSampleBounds**(L)
12: **else if** $L < \mathbf{t}.L$ **then**
13: $\mathbf{t}$.**AddPath**($\mathbf{P}_{bst}$)
14: $\mathbf{t}.L = L$
15: $\mathbf{t}$.**Prune**(L)
16: $\mathbf{t}$.**SetSampleBounds**(L)

$\mathbf{t}$.**Prune**(L)
1: **for** $\forall$ nodes $n \in \mathbf{t}$ **do**
2: **if** $h_s(n) + h_g(n) \geq L$ **then**
3: remove n and its descendants

$\mathbf{t}$.**Insert**(v)
1: $p_v =$ prospective parent of v according to the random tree algorithm being used
2: **if** $\mathbf{t}.d_s(p_v) + h(p_v, v) + h_g(v) < L$ **then**
3: insert v according to the random tree algorithm

$\mathbf{t}$.**Insert**(v, p_{bst})
1: $p_v =$ prospective parent of v according to the random tree algorithm being used
2: **if** $\mathbf{t}.d_s(p_v) + h(p_v, v) < \mathbf{t}.d_s(p_{bst}) + h(p_{bst}, v)$ **then**
3: **if** $\mathbf{t}.d_s(p_v) + h(p_v, v) + h_g(v) < L$ **then**
4: insert v according to the random tree algorithm
5: **else if** $\mathbf{t}.d_s(p_{bst}) + h(p_{bst}, v) + h_g(v) < L$ **then**
6: insert v according to the random tree algorithm with p_{bst} as its parent

SetSampleBounds(L)
1: $a = (L - |s - g|)/2$
2: $b = \max\{\min\{s, g\} - a, \mathbf{MinBounds}()\}$
3: $c = \min\{\max\{s, g\} + a, \mathbf{MaxBounds}()\}$

Fig. 3. Algorithm for C-FOREST (Left-Top) and selected subroutines. Note that any random tree algorithm can be used as a template for **RandomTree**($\mathbf{t}$), as long as it provides the necessary subroutines. **MinBounds**() and **MaxBounds**() return the minimum and maximum coordinates of the configuration space along each dimension.

$\mathbf{P}_{bst,i-1}$ in the possible neighbor set, but is otherwise identical. This ensures that $\mathbf{P}_{bst,i-1}$ can be the parent of $\mathbf{P}_{bst,i}$, but allows better nodes to be used if they exist.

We can ignore points not in $\mathbf{A}_L$ for random sampling (**RandomPoint**(L), line 6, Figure 3). We can also prune nodes not in $\mathbf{A}_L$ (as is done in **Prune**(L), line 2). Sampling directly from $\mathbf{A}_L$ 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 (**SetSampleBounds**(L), lines 1-3), and then disregard points outside $\mathbf{A}_L$ (**RandomPoint**(L), lines 1-3).

We have also found it useful to disregard any points for which $\mathbf{t}.d_s(v) + h_g(v) \geq L$ (**Insert**(v), line 2). That is, points that cannot lead to a better solution given their current distance-to-root through the tree plus the heuristic estimate of the distance to goal. This is a greedy strategy, since it does not account for the fact that future tree-remodeling may decrease $\mathbf{t}.d_s(v)$, and is similar to the priority heap weight used in the A* algorithm. We also prune the descendants of pruned nodes in a similar greedy approach (**Prune**(L), line 3).

C. Runtime Analysis

The runtime of C-FOREST is a combination of two things: (1) the inherited runtime, per node, of the particular underlying random tree algorithm (e.g., for RRT* $\mathcal{O}(c\log(n))$, where c is a predefined constant depending on dimensionality, and n is the number of nodes already in the tree), and (2) the time associated with sending and receiving messages. Sending a message requires time $\mathcal{O}(\ell)$ and receiving it requires time $\mathcal{O}(\ell + g(n) + \sum_{i=1}^{\ell} f(i))$, where $g(n)$ is the time required to prune nodes from the tree based on $\|\mathbf{P}_{bst}\|$, and $f(i)$ is the time required to insert the i-th node of $\mathbf{P}_{bst}$ into the tree and is also function of the underlying random tree algorithm (for RRT* $f(i) = c\log(n + i)$).

SequentialCFOREST()
1: $L = \infty$
2: $\mathbf{P}_{bst} = \emptyset$
3: **for** $\forall \mathbf{t} \in \mathbf{T}$ **do**
4: $\mathbf{t}.L = \infty$
5: $\mathbf{t}.\mathbf{P}_{bst} = \emptyset$
6: **while not StopCriterion**() **do**
7: **for** $\forall \mathbf{t} \in \mathbf{T}$ **do**
8: **RandTree**($\mathbf{t}$)
9: **Return** $(L, \mathbf{P}_{bst})$

RandomTree($\mathbf{t}$)
1: **if** $L < \mathbf{t}.L$ **then**
2: $\mathbf{t}$.**AddPath**($\mathbf{P}_{bst}$)
3: $\mathbf{t}.L = L$
4: $\mathbf{t}$.**prune**(L)
5: $\mathbf{t}$.**SetSampleBounds**(L)
6: **while TreeTimeLeft**() **and TimeLeft**() **do**
7: $v = \mathbf{t}$.**RandPoint**(L)
8: $\mathbf{t}$.**Insert**(v)
9: **if** $\mathbf{t}.L < L$ **then**
10: $\mathbf{P}_{bst} = \mathbf{t}.\mathbf{P}_{bst}$
11: $L = \mathbf{t}.L$
12: **Return**

Fig. 4. Sequential C-FOREST (left), and **RandomTree**($\mathbf{t}$) (right). Note that any random-tree algorithm can be used as a template for **RandomTree**($\mathbf{t}$), as long as it provides the necessary subroutines. Subroutines are described in Figure 3.

D. Sequential C-FOREST

It is possible to run any distributed algorithm on a serial architecture by using a *virtual* distributed architecture—where time division on a single CPU simulates having multiple CPUs [15]. The granularity of time division must be many orders of magnitude smaller than the total running time because each virtual CPU experiences a communication lag whenever it is not running. A full-blown virtual distributed architecture contributes additional overhead (e.g., messaging between CPUs) and is unnecessary for our task. Instead, we propose a sequential version of C-FOREST that is allotted $1/T$-th of the computation time on a single CPU for each tree $\mathbf{t} \in \mathbf{T}$. The algorithm is presented in Figure 4. 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. The subroutine **TreeTimeLeft**() returns $true$ if there is still time for tree $\mathbf{t}$ to plan during the current planning iteration. The rest of the subroutines are identical to those described in Figure 3 in Section III.

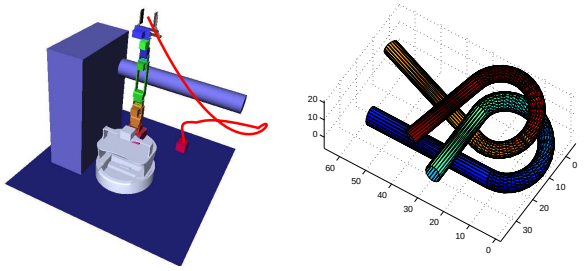


Fig. 5. 7 DOF arm with end-effector path (Left). The Alpha-1.5 feasible path-planning benchmark problem (Right).

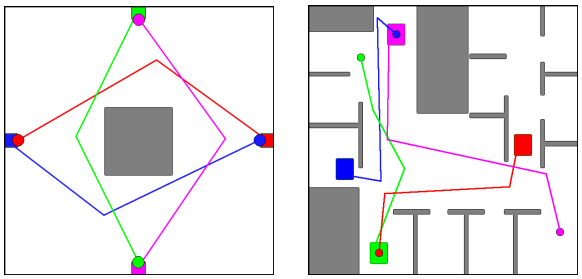


Fig. 6. Toy and Office environments used in experiment groups 2 and 3 with sample paths. Goal regions appear rectangular and robot starting locations are circular.

The algorithm moves to the next tree as soon as the previous tree has found a solution, even if time remains for the previous tree (**RandTree**(**t**), line 12). We have found this to help during the early phases of search because the next tree can focus on improving the current best solution. This can quickly reduce the search envelope at the beginning of the search, with positive effects that propagate forward with time.

Speedup *equals* efficiency for Sequential C-FOREST because only one CPU is used. Thus, while parallel C-FOREST can be useful when $\eta < 1$, sequential C-FOREST should *only* be used when $\eta > 1$.

IV. Experiments

We perform five sets of experiments involving four different planning problems. The first four experiments include: a seven degree-of-freedom manipulator arm, a multi-robot team in two different environments (office and toy, respectively), and the classic alpha-1.5 feasible-path planning benchmark problem [24]. The problems are illustrated in Figures 5 and 6. In the fifth experiment we evaluate the relative benefits of path sharing vs. path-length based pruning and sampling envelope reduction in the multi-robot team office problem. We use RRT* as the underlying random tree in all experiments.

Our parallel architecture contains 64 single CPU computers that communicate over a network, each CPU has a 1.2 GHz Xeon processor. Our serial architecture is a standard desktop computer with 1 CPU and a 1.596 GHz processor, all computers run the Ubuntu operating system. On the parallel architecture each C-FOREST or OR-parallelization tree is run on its own computer. On the serial architecture each tree plans

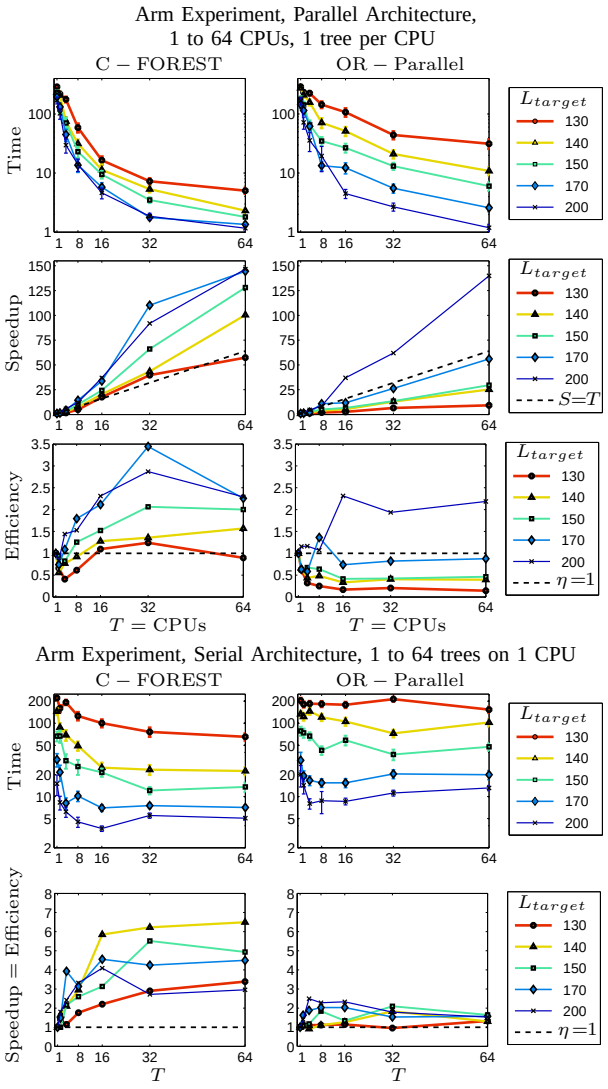


Fig. 7. 7 DOF arm. Color is target path length (L_{target}). Dashed lines show linear speedup ($\eta = 1$ and $S = T$) with respect to CPUs. Left and Right: C-FOREST and OR, respectively. Top three rows: time (mean, standard error), speedup, and efficiency for the parallel architecture using 1-64 CPUs, 1 tree per CPU (32 runs per data-point). Bottom two rows: time (mean, standard error), and speedup = efficiency for the serial architecture with 1-64 trees on a single CPU (35 runs per data-point), $S = E$ because 1 CPU is used.

for 1ms at a time. *Note that on the serial architecture efficiency equals speedup—because 1 CPU is used and efficiency is defined as speedup divided by the number CPUs. For brevity, we depict efficiency and speedup on the same plot for the serial algorithms.*

A. C-FOREST vs. OR algorithms

In the first four experiments we evaluate the performance of C-FOREST vs. OR-parallelization using a distributed architecture, *and* the performance of C-FOREST vs. OR-serialization using a sequential architecture. OR-parallelization grows T trees in parallel that *do not communicate* during search. OR-serialization is a 1-CPU version of OR-parallelization where

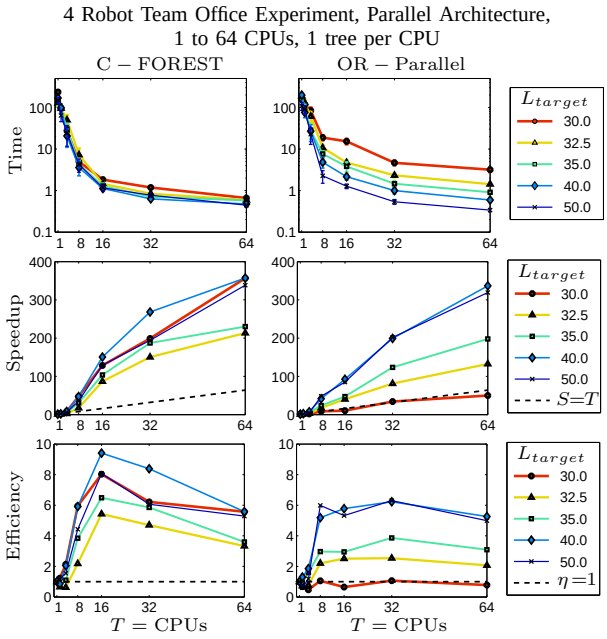


Fig. 8. 4 robot team in an office environment. Color is target path length (L_{target}). Dashed lines show linear speedup ($\eta = 1$ and $S = T$) with respect to CPUs. Left and Right: C-FOREST and OR, respectively. Top three rows: time (mean, standard error), speedup, and efficiency for the parallel architecture using 1-64 CPUs, 1 tree per CPU (32 runs per data-point). Bottom two rows: time (mean, standard error), and speedup = efficiency for the serial architecture with 1-64 trees on a single CPU (50 runs per data-point), $S = E$ because 1 CPU is used.

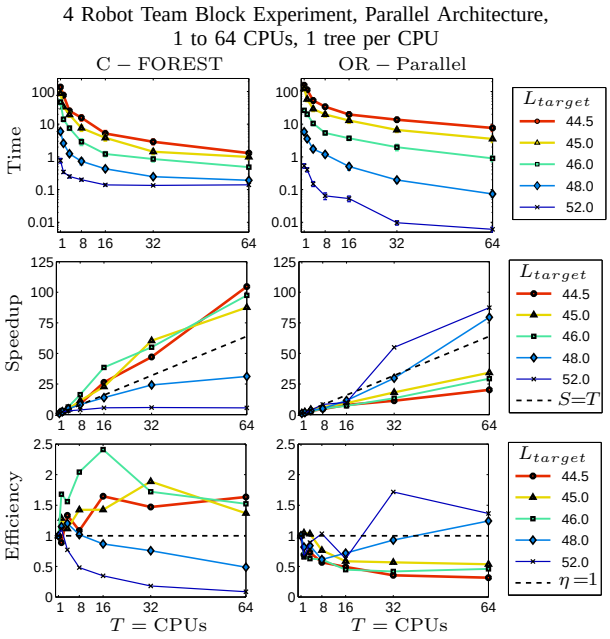
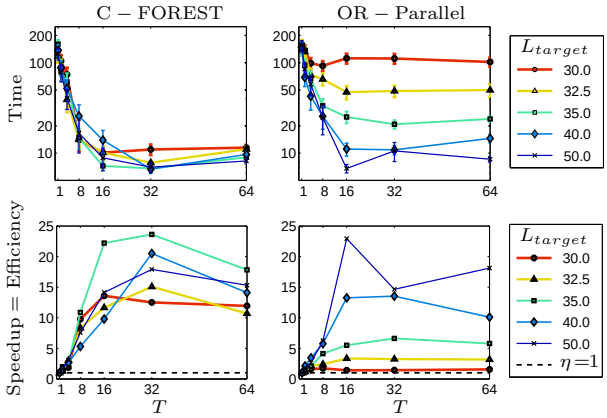
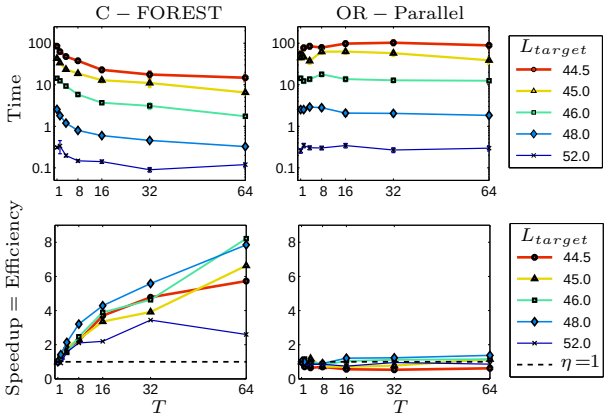


Fig. 9. 4 robot team in block environment. Color is target path length (L_{target}). Dashed lines show linear speedup ($\eta = 1$ and $S = T$) with respect to CPUs. Left and Right: C-FOREST and OR, respectively. Top three rows: time (mean, standard error), speedup, and efficiency for the parallel architecture using 1-64 CPUs, 1 tree per CPU (32 runs per data-point). Bottom two rows: time (mean, standard error), and speedup = efficiency for the serial architecture with 1-64 trees on a single CPU (50 runs per data-point), $S = E$ because 1 CPU is used.





computation is serially divided between non-communicating trees. Comparison of the C-FOREST algorithms to the OR algorithms is useful because it demonstrates the added advantage of sharing data *during planning* vs. the purely statistical benefits of drawing multiple random samples from the set of all paths, respectively.

For each combination of planning problem, architecture, and {C-FOREST, OR-parallelization} we perform repeated trials for $T = \{1, 2, 4, 8, 16, 32, 64\}$ and vs. different L_{target} (e.g., path length quality targets). The values of L_{target} are chosen to provide points along a spectrum of varying degrees of difficulty. The bounds of this spectrum are defined by the performance of a single tree, such that problems with relatively easy path-lengths are solved as soon as the first valid path is found, and relatively hard path-lengths are solved in three to five minutes, on average, depending on experiment.

The number of repeated trials represented by each data point varies by experiment. 32 repeated trials are performed per data point for all tests involving C-FOREST and OR-parallelization. For test involving sequential C-FOREST and OR-serialization, 35, 50, 50, and 20 trials are performed per data point in the arm, team office, team toy, and alpha-1.5 problems, respectively. This results in a total of $18,354$ individual trials.

Figures 7, 8, 9, and 10 display experimental results for the seven degree-of-freedom manipulator arm, multi-robot team in the office environment, multi-robot team in a simple toy

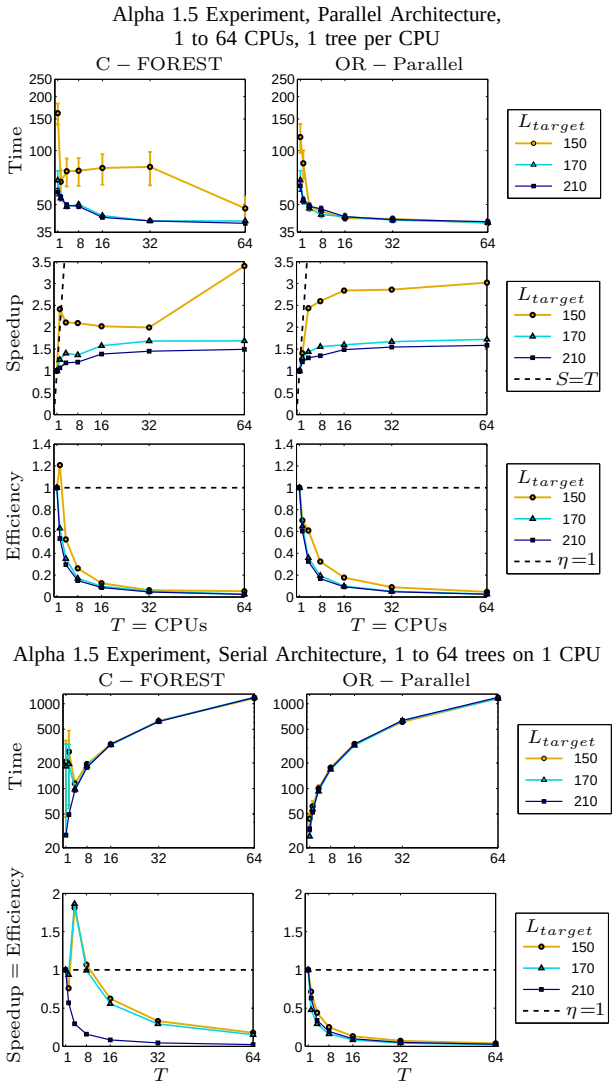


Fig. 10. Alpha-1.5 experiment. Color is target path length (L_{target}). Dashed lines show linear speedup ($\eta = 1$ and $S = T$) with respect to CPUs. Left and Right: C-FOREST and OR, respectively. Top three rows: time (mean, standard error), speedup, and efficiency for the parallel architecture using 1-64 CPUs, 1 tree per CPU (32 runs per data-point). Bottom two rows: time (mean, standard error), and speedup = efficiency for the serial architecture with 1-64 trees on a single CPU (20 runs per data-point), $S = E$ because 1 CPU is used.

environment, and the alpha-1.5 problem, respectively. The organization of each figure is identical as follows: Left and right columns show results for C-FOREST algorithms and OR algorithms, respectively. The top three rows show total time, efficiency, and speedup for C-FOREST vs. OR-parallelization. The bottom two show total time and speedup (efficiency) for sequential C-FOREST vs. OR-serialization.

When $T = 1$, the resulting algorithm is equivalent to the underlying random-tree algorithm (e.g., RRT*).

The Arm and Office Team experiments are representative of standard robotic path planning problems. In the first, a 7-DOF manipulator arm (Figure 5) must plan around an obstacle. Distance is defined as the total distance traveled by the end-effector of the arm. This is useful when we want to minimize the distance traveled by whatever the arm is holding.

In the second experiment the centralized multi-robot planning framework[5] is used for a team of four robots moving in an office environment. Distance is defined as euclidean distance through the combined configuration space of all robots.

The third experiment is similar to the second, except that the 4-robot team must exchange places around a block shaped obstacle (Figure 6-left). It is designed to evaluate C-FOREST vs. OR in an easy environment and test our analytical result that C-FOREST is less likely to have super-linear speedup in such a case.

The fourth experiment is the Alpha-1.5 benchmark problem [24]. It consists of two metal "wires" that are twisted around each other. The problem is solved when the wires are separated. The Alpha family of problems is a classic feasible-path planning scenario. Although C-FOREST is designed as a shortest-path planning algorithm, this experiment tests how well it fares in the feasible-path planning domain—where simply finding a viable solution is quite challenging. We use the Alpha-1.5 benchmark (vs. Alpha-1.0) because harder versions of the problem could not be solved by either C-FOREST or the OR algorithm within an amount of time that was conducive to running repeated trials (in fact, we failed to find a solution on Alpha-1.0 after running for more than an hour). We calculate distance using the method presented in [25]—i.e., as a weighted sum of (A) the magnitude of translation, and (B) a function of the inner-product between quaternions (Algorithm 5 in, [25]). We choose these weights such that that a 180 degree pure rotation has the same distance as the longest dimension of the (bent) wire.

B. C-FOREST solution sharing vs. pruning

C-FOREST differs from OR-parallelization in two ways: first, it shares paths between trees during planning; and second, it prunes old nodes and samples new ones based on the lengths of those shared paths. In order to assess the relative effects of these two differences, we rerun the office environment experiment for $L_{target} = 30$ meters with modified versions of the algorithm that: (1) do not prune or decrease the sampling envelope based on L, or (2) do not share and engraft the current best solution (length L is still shared and used for pruning and sampling envelope reduction). Figure 11 depicts the planning times and efficiencies for the parallel and sequential versions of C-FOREST (top and bottom sub-figures, respectively).

V. Discussion of Results

A. C-FOREST vs. OR for shortest-path planning

On shortest-path planning problems (i.e., the first three experiments) parallel C-FOREST outperforms OR-parallelization and sequential C-FOREST outperforms OR-serialization, in general. The few exceptions to this trend occur for target-lengths that allow relatively little time to improve upon an

[5]In the centralized multi-robot planning framework all robot are viewed as individual pieces of a single larger robot. The configuration space is the Cartesian product of all individual robots' configuration spaces. If the team consists of four holonomic 2D robots, then the configuration space has a total of eight degrees-of-freedom.

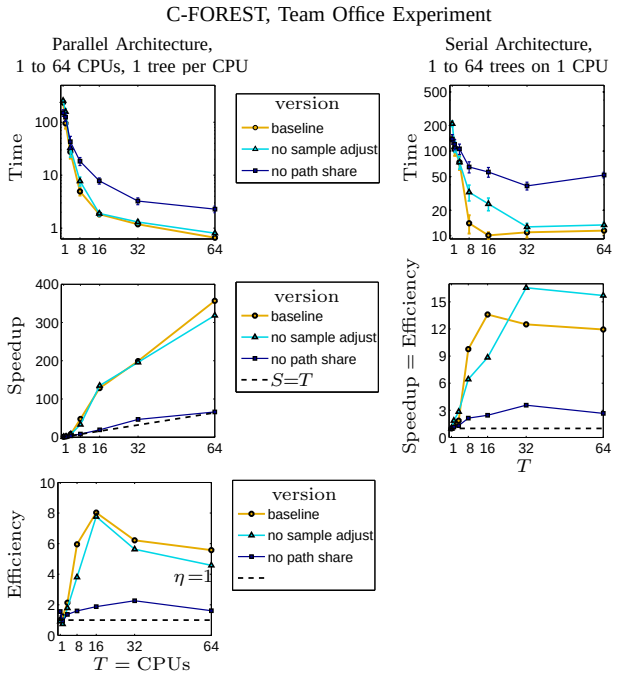


Fig. 11. C-FOREST variants on the office problem for $L_{target} = 30\text{m}$. "Baseline" (yellow) is C-FOREST or sequential C-FOREST (left or right, respectively), "no sample adjust" (teal) is a variant that does not decrease the sampling envelope, "no path share" (blue) is a variant that does not engraft solutions (solution lengths are still shared). Left: time (mean, standard error), speedup, and efficiency for the parallel architecture using 1-64 CPUs, 1 tree per CPU (32 runs per data-point). Right: time and speedup = efficiency for the serial architecture with 1-64 trees on a single CPU (35 runs per data-point). $S = E$ for the serial architecture because 1 CPU is used.

initial solution. C-FOREST algorithms have increasingly better speedup vs. the OR algorithms as L_{target} decreases—i.e., when the time used to improve a sub-optimal solution represents a larger percentage of total planning time. This reflects the fact that C-FOREST trees only work together after the initial solution is found (before the initial solution is found, there is no difference between C-FOREST and OR-parallelization).

Although the OR algorithms do exhibit super-linear speedup in many cases, the speedup of the C-FOREST algorithms is greater—by up to an order of magnitude.

Intuitively, exchanging paths is beneficial because bottlenecks and other obstructions can make exploring the space difficult, and finding a close-to-optimal solution unlikely. If the optimal path passes through multiple bottlenecks and/or snakes around multiple obstacles *but* other sub-optimal routes to the goal also exist; then only one $\mathbf{t} \in \mathbf{T}$ must get lucky per bottleneck/obstacle, instead of requiring a particular $\mathbf{t}$ to get lucky per all bottlenecks/obstacles—the latter is less likely.

Coupled sampling is beneficial because it allows all trees to focus effort on exploring only regions of the configuration that can possibly lead to better solutions. Coupled pruning is beneficial because it reduces the amount of work required to insert new nodes into each tree.

B. Shortest-path planning vs. feasible-path planning

C-FOREST has super linear speedup on all of the planning problems that come form the shortest-path planning domain (sometimes above 350), many efficiencies greater than 2, and some greater than 9; and Sequential C-FOREST algorithm has speedups as high as 20. In contrast, the speedups of C-FOREST and Sequential C-FOREST (and also the OR algorithms) are nearly always sub-linear on the Alpha-1.5 benchmark, which is a classic feasible path planning problem. This suggests that speedup is more likely to occur on problems for which an initial solution is relatively easy to find, compared to one with length L_{target}—for instance, when most of the planning time is spent improving a sub-optimal solution vs. simply finding an initial solution. It also suggests that C-FOREST is no better than OR-parallelization for *feasible* path-planning.

C. Exchanging paths vs. path lengths

Although sending an entire path requires more time and space than sending only its length, our results suggest that doing so is worth the price. This is indicated by experiment five (see Figure 11), where forests that do not share nodes exhibit relatively low efficiencies (slightly above 1). That said, reducing the sampling envelope does have a noticeable effect in all experiments.

D. T and Efficiency

Forests ($T > 1$) find the target solution more quickly, on average, than a stand alone tree ($T = 1$) in all experiments. However, in our experiments the best efficiencies for C-FOREST are observed when $T < 64$ (i.e., *not* with the most trees). Thus, with respect to efficiency, there may be an inherent limit to the cost-savings C-FOREST can provide. The observed power savings of up to 89% (for $\eta \approx 9.4$) is quite decent. Sequential C-FOREST attained an even better power savings of 95% (for $\eta \approx 23.6$). In fact, C-FOREST with $T > 1$ yields a better efficiency than $T = 1$ in nearly all shortest-path planning experiments.

An explicit method of selecting T to achieve the maximum efficiency is beyond the scope of the current paper. However, our results suggest that performance is stable vs. T (e.g., using an algorithm with 30% more or less trees has a relatively small effect on solution quality vs. time), and that it is better to error on the side of using too many trees than not enough. We believe that calibrating T on a similar problem will give a decent approximation to its optimal value, since there appears to be ample room for error.

E. Causes for C-Forest's super linear performance

C-FOREST exhibits super-linear speedup because it allows multiple trees to actively cooperate during the search for better and better solutions. In particular, sharing the nodes of the current best path appears to be the most helpful aspect of the algorithm[6] (as suggested by experiment five). The latter is advantageous because it biases all trees to be populated with nodes from "useful" locations in the search space, and allows each tree's visibility envelope to expand into nearby regions.

[6]Coupled sampling and pruning operations, and the statistical advantages of building multiple random trees in parallel are also helpful.

VI. Conclusions

C-FOREST allows multiple trees to actively cooperate during the search for better and better solutions. Experimental results suggest C-FOREST will perform better when: (1) an initial solution can be found in a fraction of the total planning time, (2) finding a relatively good solution is difficult. Further, most of C-FOREST's performance gains appear to come from sharing the nodes of the current best-path.

Both C-FOREST and OR-parallelization achieve super-linear speedup on the shortest-path planning problem; however, the speedup observed with C-FOREST is up to an order of magnitude greater. To the best of our knowledge, the best average efficiency observed with C-FOREST ($\eta > 9.4$) is significantly greater than any previous result observed in distributed single-query planning. Speedup values $S > 300$ are also observed; as well as speedup (and efficiency) of $S = \eta > 23.6$ for Sequential C-FOREST.

References

[1] I. A. Sucan and L. E. Kavraki, "On the implementation of single-query sampling-based motion planners," in *IEEE International Conference on Robotics and Automation*, 2010, pp. 2005–2011.

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

[3] Y. Koga and J.-C. Latombe, "On multi-arm manipulation planning," in *Proc. IEEE International Conference on Robotics and Automation*, vol. 2, 1994, pp. 945–952.

[4] G. Sanchez and J.-C. Latombe, "On delaying collision checking in prm planning: Application to multi-robot coordination," *The international Journal of Robotics Research*, vol. 21, pp. 5–26, 2002.

[5] ——, "Using a prm planner to compare centralized and decoupled planning for multi robot systems," in *Proc. IEEE International Conference on Robotics and Automation*, vol. 2, 2002, pp. 2112–2119.

[6] C. M. Clark, S. M. Rock, and J.-C. Latombe, "Dynamic networks for motion planning in multi-robot space systems," in *International Symposium on Artificial Intelligence, Robotics and Automation in Space*, 2003, pp. 3621–3631.

[7] S. Karaman and E. Frazzoli, "Incremental sampling-based algorithms for optimal motion planning," in *Proceedings of Robotics: Science and Systems*, 2010.

[8] M. Otte and N. Correll, "Any-com multi-robot path-planning: Maximizing collaboration for variable bandwidth," in *Proc. International Symposium on Distributed Autonomous Robotics Systems*, 2010.

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

[10] S. Caselli and M. Reggiani, "Randomized motion planning on parallel and distributed architectures," in *Euromicro Workshop on Parallel and Distributed Processing*, 1999, pp. 297–304.

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

[12] J. Ichnowski and R. Alterovitz, "Parallel sampling-based motion planning with superlinear speedup," in *In Proc. IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2012.

[13] Bialkowski, Karaman, and Frazzoli, "Massively parallelizing the RRT and the RRT*," in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2011.

[14] I. A. Sucan and L. E. Kavraki, "Kinodynamic motion planning by interior-exterior cell exploration," in *Algorithmic Foundation of Robotics VIII (Proceedings of Workshop on the Algorithmic Foundations of Robotics), STAR*, 2009, pp. 449–464.

[15] D. J. Challou, M. Gini, and V. Kumar, "Parallel search algorithms for robot motion planning," in *IEEE International Conference on Robotics and Automation*, 1993, pp. 46–51.

[16] E. Plaku, K. E. Bekris, B. Y. Chen, A. M. Ladd, and L. E. Kavraki, "Sampling-based roadmap of trees for parallel motion planning," *IEEE Transactions on Robotics*, vol. 21, pp. 587–608, 2005.

[17] M. Otte and N. Correll, "Any-com multi-robot path-planning with dynamic teams: Multi-robot coordination under communication constraints," in *Proc. International Symposium on Experimental Robotics*, 2010.

[18] T.-Y. Li and Y.-C. Shie, "An incremental learning approach to motion planning with roadmap management," in *IEEE International Conference on Robotics and Automation*, 2002, pp. 3411–3416.

[19] R. Gayle, K. R. Klingler, and P. G. Xavier, "Lazy reconfiguration forest (LRF) - an approach for motion planning with multiple tasks in dynamic environments." 2007, pp. 1316–1323.

[20] M. Zucker, J. J. Kuffner, and M. S. Branicky, "Multipartite RRTs for rapid replanning in dynamic environments," in *International Conference on Robotics and Automation*, 2007, pp. 1603–1609.

[21] D. Ferguson and A. Stentz, "Anytime RRTs," in *Proc. IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2006, pp. 5369–5375.

[22] N. A. Wedge and M. S. Branicky, "On heavy-tailed runtimes and restarts in rapidly-exploring random trees," in *AAAI Conference on Artificial Intelligence*, 2008.

[23] M. Luby, A. Sinclair, and D. Zuckerman, "Speedup of las vegas algorithms," in *In Proc. of the 2nd Israel Symposium on the Theory of Computing and Systems*, 1993, pp. 128–133.

[24] B. Yamrom, "Alpha puzzle," Provided by GE Corporate Research & Development Center, via Parasol Lab at Texas A & M University, http://parasol.tamu.edu/dsmft/benchmarks/mp/.

[25] J. J. Kuffner, "Effective sampling and distance metrics for 3d rigid body path planning," in *Proc. IEEE Conference on Robotics and Automation*, 2004.