MT-RRT: a general purpose multithreading library for path planning

Andrea Casalino[1], Andrea Maria Zanchettin[1] and Paolo Rocco[1]

***Abstract*— Rapidly Random exploring Trees are popular algorithms in the field of motion planning. A feasible path connecting two different poses is found by incrementally building a tree data structure. They are powerful and flexible, but also computationally intense, requiring thousands of iterations before their termination. The aim of this article is to show the capabilities of MT-RRT, a general purpose library which exploits four different multithreading strategies to speed up the planning process of Rapidly Random exploring Trees. The performance achieved by MT-RRT are verified on various benchmarks, providing general results valid for a great class of planning problems. The strategies proposed are able to significantly reduce the computation time absorbed by the planning process.**

I. Introduction

Path planning is one of the most classical problems in robotics. Essentially, the problem consists in finding a feasible trajectory leading a manipulator, or more in general a dynamical system, from a starting state to an ending desired one. Some planners obtain the solution step-by-step, as the result of a closed-loop control scheme. Such algorithms are typically designed for on-line implementations. Examples are the approaches based on repulsive field [1], [2]: a motion for the robot is obtained by applying some virtual forces, generated by the obstacles populating the scene. Such methods are typically affected by local minima problems, leading the robot to some equilibrium point far from the desired ending configuration. MPC approaches like [3] or [4] are other examples of closed-loop approaches. Generally, these methods are not computationally intense but fail to find a solution for those problems having a complex shaped admitted region.

On the opposite, other classes of planners are designed to be deployed off-line, computing in a single step an entire trajectory solving the planning problem. Rapidly Random exploring Trees (RRT) [5] are the most representative example. They explore the admitted space in an incremental way, building a search tree. They are more computationally demanding, but they are able to find feasible paths also for those problems characterized by very cluttered environments. With respect to the original formulation proposed in [5], many variants have been proposed to improve the searching process. For example, [6] proposed a reducing metric sensitivity to promote the expansion of the tree, starting from those nodes resulting far from the obstacles, in order to obtain in a faster way the solution. [7] proposed a version specifically designed to find feasible trajectories passing through narrow passages. RRT were also proved to be deployable as kinodynamic planners, designing optimal LQR controller driving a generic dynamical system to a desired final state, see [8] and [9].

On the other hand, all the planners proposed, even for medium-complex problems, require thousands of iterations to obtain a solution. Therefore, multithreading strategies can be deployed to speed up any kind of planning problem. In particular, the aim of this article is to show the capabilities of MT-RRT: a general purpose path planning library, which exploits four different possible multithreading strategies for parallelizing RRT algorithms.

Some parallel computation strategies for this kind of planning algorithms were already proposed, even though the related literature is not rich. A review of this topic, addressing not only RRTs, can be found in [10]. [11] uses GPU to parallelize only the collision check phase, even though this activity might not be the most time consuming one. [12] made use of a distributed memory approach, proposing some multi-processing strategies. Processes are coordinated via message-passing, resulting in augmented overheads in comparison to the multithreading strategies proposed here.

To the best of the authors' knowledge, it is difficult to find works reporting general results, valid for any kind of planning problems. In this work, the performance achievable by MT-RRT will be tested in many different planning scenarios, showing the scalability with respect to the number of employed threads. Such scenarios are representative of a great class of planning problems, providing general results about which kind of strategy is more indicated to speed up a particular class of planning problems.

The rest of this article is structured as follows: Section II will briefly review the basic mechanism of the RRT algorithms; while Section III will describe four different strategies for parallelizing them, whose performance are reported in Section IV. Section V will provide a discussion of the achieved performance.

II. Background on RRT algorithms

RRTs are able to find a trajectory connecting two states: a starting one x_o and an ending one x_f building a search tree $T(x_o)$. Each node $x_i \in T$ (apart from the root) is connected to its unique father x_{fi} by a trajectory $\tau_{fi \to i}$. In the following of this article, we will use also this notation: $Fath(x_i) = x_{fi}$. The set $\mathcal{X} \subseteq \mathbb{R}^d$ will contain all the possible states x, while $\underline{\mathcal{X}} \subseteq \mathcal{X}$ is a subset describing the admissible region induced by a series of constraints to be considered. The basic version of an RRT algorithm is described by Algorithm 1.

[1] The authors are with Politecnico di Milano, Dipartimento di Elettronica, Informazione e Bioingegneria, Piazza L. Da Vinci 32, 20133, Milano, Italy (e-mail: name.surname@polimi.it).

Algorithm 1 Canonical RRT

procedure RRT SEARCH(x_o, x_f)
  $T(x_o) = \{x_o\}$;
  for k=1 : I **do**
    sample a $x_R \in \mathcal{X}$;
    x_s = Expand(T, x_R);
    if $x_s \in \underline{\mathcal{X}}$ AND $\|x_s - x_f\| \leq \epsilon$ **then**
      return Path to root(x_s)$\cup x_f$;
  end for
end procedure

procedure EXPAND(T, x_R)
  $x_{Nearest}$ =Nearest Neighbour(T, x_R) ;
  x_s = Steer($T, x_{Nearest}, x_R$);
  if $x_s \in \underline{\mathcal{X}}$ **then**
    $Fath(x_s) = x_{Nearest}$;
    $T = T \cup x_s$;
  return x_s;
end procedure

procedure NEAREST NEIGHBOUR(T, x_R)
  return $\underset{x_i \in T}{\operatorname{argmin}}(C(\tau_{i \to R}))$;
end procedure

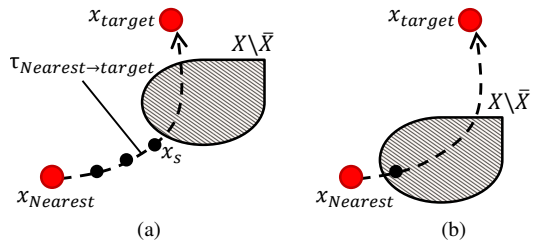

Fig. 1: The dashed curves in both pictures are the optimal trajectories connecting the pair of states $x_{Nearest}$,x_{target}, while the filled areas are regions of X not allowed by constraints. The steering procedure integrates with a fixed step the optimal trajectory, with the aim of finding the furthest admissible state from $x_{Nearest}$ lying along the trajectory. For the example on the right, the steering is not possible: the first state obtained is returned as x_s although is not valid.

The Steer function in the $Expand$ procedure is problem-dependent. Basically, It has the aim to extend a certain state x_i already inserted in the searching tree, toward another one x_R. For this purpose, an optimal trajectory $\tau_{i \to R}$, agnostic of the constraints, going from x_i to x_R is taken into account. A series of equispaced states $x_s^{1,\cdots,K}$ lying along τ, are verified to be in $\underline{X}$, in order to find the furthest one from x_i (denoted as x_s) admitted by constraints. When none of the states $x_s^{1,\cdots,K}$ result to be in $\underline{X}$, the first intermediate state computed is returned, even though it is considered not valid, refer also to Figure 1.

The Nearest Neighbour procedure relies on the definition of a cost function $C(\tau)$. Therefore, the closeness of states does not take into account the shape of $\underline{\mathcal{X}}$. The algorithm terminates when a steered configuration x_s sufficiently close to x_f is found.

The previously described algorithm can be modified to consider a bidirectional strategy, expanding simultaneously two different trees [13] (see the picture in the middle of Figure 2). Indeed, at every iteration one of the two trees is extended toward a random state. Then, the other tree is extended toward the steered state previously obtained. At the next iteration, the roles of the trees are inverted. The algorithm stops, when the two trees meet each other. The detailed pseudocode is reported by Algorithm 2.

Algorithm 2 Bidirectional RRT

procedure RRT BIDIRECTIONAL SEARCH(x_o, x_f)
  $T_{master} = T(x_o) = \{x_o\}$;
  $T_{slave} = T(x_f) = \{x_f\}$;
  for k=1 : $\frac{I}{2}$ **do**
    sample a $x_R \in \mathcal{X}$;
    x_s = Expand(T_{master}, x_R) ;
    if $x_s \in \underline{X}$ **then**
      x_{s2} = Expand(T_{slave}, x_s) ;
      if $x_{s2} \in \underline{X}$ AND $\|x_s - x_{s2}\| \leq \epsilon$ **then**
        $Path_1$ =Path to root(x_s);
        $Path_2$ =Revert(Path to root(x_{s2}));
        return $Path_1 \cup Path_2$;
    Swap(T_{master}, T_{slave});
  end for
end procedure

This solution offers several advantages. For instance, the computational times absorbed by the Nearest Neighbour search is reduced since this operation is done separately for the two trees and each tree contains at an average half of the states computed (see Section II-A).

Notice that there are many (possibly infinite) $\tau_{o \to f} \subset \underline{\mathcal{X}}$. Among the aforementioned set, we might be interested in finding the trajectory minimizing $C(\tau_{o \to f})$. The basic version of the RRT algorithm is proved to find with a probability equal to 1, a suboptimal solution [14]. The optimality issue is addressed by a variant of the RRT, called RRT* [14], whose pseudocode is contained in Algorithm 3 (d is the cardinality of $\mathcal{X}$). Essentially, the RRT* version, after inserting in the search tree a newer steered state, tries to undertake local improvements to the connectivity of the tree, in order to minimize the cost-to-go of the states in the $Near$ set. This is proved to converge to the optimal solution. There are no precise stopping criteria for the RRT*.
Figure 2 reports some examples of trees obtained by following the three exposed strategies on the use-case detailed in Section IV-A.

A. Computational time analysis

Let t_τ denotes the mean time absorbed by the computation of an optimal trajectory τ connecting two states and t_V

Algorithm 3 RRT*

procedure RRT* SEARCH(x_o, x_f)
  $T(x_o) = \{x_o\}$;
  $Sol = \emptyset$;
  for k=1 : I **do**
    sample a $x_R \in \mathcal{X}$;
    x_s =Expand star(T, x_R);
    if $x_s \in \underline{\mathcal{X}}$ AND $\|x_s - x_f\| \leq \epsilon$ **then**
      $Sol = Sol \cup x_s$;
  end for
  $x_{best} = \underset{x_S \in Sol}{\text{argmin}}$(Cost to root($x_S$));
  return Path to root(x_{best}) $\cup\, x_f$;
end procedure

procedure EXPAND STAR(T, x_R)
  x_s=Expand(T, x_R);
  $N = size(T)$;
  $Near = \{x_i \in T | C(\tau_{i\to s}) <= \gamma(\frac{log(N)}{N})^{\frac{1}{d}}\}$;
  Rewind($Near, x_s$);
  return x_s;
end procedure

procedure REWIRD($Near, x_s$)
  $x_{bestfath} = x_{Nearest}$, $C_{min} = C(\tau_{Nearest\to s})$;
  for $x_n \in Near$ **do**
    if $\tau_{n\to s} \subset \underline{\mathcal{X}}$ AND $C(\tau_{n\to s}) < C_{min}$ **then**
      $C_{min} = C(\tau_{n\to s})$; $x_{bestfath} = x_n$;
  end for
  $Fath(x_s) = x_{bestfath}$;
  C_s =Cost to root(x_s);
  for $x_n \in Near$ **do**
    if $\tau_{s\to n} \subset \underline{\mathcal{X}}$ **then**
      $C_n = C(\tau_{s\to n}) + C_s$
      if $C_n <$ Cost to root(x_n) **then**
        $Fath(x_n) = x_s$;
  end for
end procedure

procedure COST TO ROOT(x_i)
  return $\sum_{x_p \in Pathtoroot(x_i)} C(\tau_{Fath(p)\to p})$;
end procedure

the mean time required to check whether a certain state x is contained within $\underline{X}$. The proportion between the two aforementioned quantities can vary a lot, considering the specific planning problem addressed. The total computation time T absorbed by the invocation of a basic RRT algorithm can be estimated as follows:

$$\begin{aligned} T &= T_\tau + T_V \\ T_\tau &= \mathcal{O}\left(\sum_{i=1}^{I}((i+1)t_\tau)\right) = O\left(\frac{I(I+1)}{2}t_\tau\right) \\ T_V &= \mathcal{O}\left(It_V\right) \end{aligned} \quad (1)$$

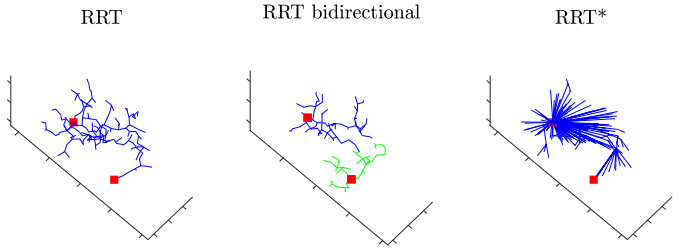

Fig. 2: Examples of trees obtained when solving the planning problem detailed in Section IV-A. In all the pictures the states x_o and x_f are depicted as big red squares.

where I is the number of iterations needed to complete the search. Indeed, at every iteration i, in the worst case, the tree contains i elements for which the nearest neighbour search absorbs i evaluations of τ; while every expansion of the tree requires at least one feasibility check.
When considering the bidirectional version, assuming the same value for I, we have roughly half of the nodes in every tree and therefore the time spent for the Nearest Neighbour search changes, leading to:

$$T_\tau = \mathcal{O}\left(\frac{I(I+1)}{4}t_\tau\right) \quad (2)$$

Finally, when considering the RRT* formulation, the Near set determination and the evaluation of the possible reconnection (also called rewirds) drastically improves the computational time T, leading to:

$$\begin{aligned} T &= T_\tau + T_V + T_{Rew} \\ T_\tau &= \mathcal{O}\left(I(I+1)t_\tau\right) \\ T_{Rew} &= \mathcal{O}\left(\gamma t_V \sum_{i=1}^{I}\left(\frac{log(i)}{i}^{\frac{1}{d}}\right)\right) \end{aligned} \quad (3)$$

III. MT-RRT STRATEGIES

The aim of this Section is to show how the RRT algorithms are parallelized by making use of MT-RRT. As already exposed, the proposed library exploits a multithreading environment, whose peculiarity is the presence of a common memory shared by threads. This characteristic requires to design accurately the critical regions, i.e. those parts of the program where threads are allowed to modify (one at a time) the values of the shared variables. In the following, the four possible strategies adopted by MT-RRT will be exposed.

A. Parallelization of the querying activities

By the analysis of equations (1), (2) and (3), it is clear that the querying operations (i.e. the ones requiring to scan the entire list of elements of a tree) are computationally demanding. Therefore, the first possible strategy for a parallelization could simply consist in executing collectively (among threads) the Nearest Neighbour search and the determination of the Near set. Clearly, it is inefficient to re-open and close a new parallel region every time a new collective search is required, but it is more convenient to launch a pool of

threads at the beginning of the algorithm and then use a shared barrier to notify to threads the need to execute a newer collective search [1]. The maintenance operations on the search tree are executed always by the main thread. The mean computation time expected when considering a perfect parallelization and adopting P threads can be estimated as follows:

$$\begin{aligned} & T(P) &=& \frac{T_\tau}{P} + T_V \\ \textit{for RRT*} \quad & T(P) &=& \frac{T_\tau}{P} + T_V + T_{Rew} \end{aligned} \tag{4}$$

B. Parallel exploration on a shared tree

Another way to obtain a parallelization is to actually do simultaneously, every single step of the algorithms detailed in Section II. Therefore, we can imagine having threads sharing a common tree, executing in parallel every step of the expansion process. Some critical regions must be designed to allow the threads executing the maintenance of the shared tree (inserting new nodes or executing new rewirds) one at a time. Ideally, the computation time of this solution can be estimated as follows:

$$\begin{aligned} & T(P) &=& \frac{T_\tau + T_V}{P} \\ \textit{for RRT*} \quad & T(P) &=& \frac{T_\tau + T_V + T_{Rew}}{P} \end{aligned} \tag{5}$$

However, the presence of the critical regions can severely compromise the efficiency of the parallelization for certain planning problem, enlarging the total overhead times. Indeed, the computational times obtained by adopting this strategy could be far beyond the time described by equation (5) (see Section IV).

C. Parallel expansions of copied trees

To limit as much as possible the overheads induced by the presence of critical regions, we can consider a version similar to the one proposed in the previous Section, in which every thread has a private copy of the search tree. After a new node is added by a thread to its own tree, $P-1$ copies are computed and dispatched to the other threads. At every iteration, all the threads take into account the list of nodes received from the others and insert some of them into their private trees. This mechanism is able to avoid the simultaneous modification of a tree by two different threads, avoiding the use of critical regions. The pseudocode of this solution is reported in Algorithm 4.

When considering the bidirectional strategy, the mechanism is analogous but introducing for every thread a private copy of both the searching trees. Instead, for the RRT* formulation, every thread communicates to the others not just the new states found, but also the rewirds executed. Moreover, a common list of solutions found is maintained during the expansions. At the end of all the explorations, the path in that list minimizing the cost $C(\tau)$ is returned. In this way, when following this approach, there is no need to

[1]Which is implemented as a parallel for, with a subsequent reduction operation.

Algorithm 4

```
1:  procedure TREE COPIES SEARCH(x_o, x_f, P)
2:      Path = ∅;
3:      for kp=1 : P do
4:          T_kp = {x_o};
5:          J^kp = ∅;
6:          C_kp = 0;
7:          launch thread( Thread search(kp));
8:      end for
9:      return Path;
10: end procedure
11:
12: procedure THREAD SEARCH(th_id)
13:     for k=1 : I/P do
14:         if Path ≠ ∅ then return ;
15:         S = size(J^{th_id});
16:         for kj=1 : (S − C_{th_id}) do
17:             C_{th_id} = C_{th_id} + 1;
18:             T_{th_id} = T_{th_id} ∪ J^{th_id}_{kj};
19:         end for
20:         sample a  x_R ∈ X;
21:         x_s = Expand(T_{th_id}, x_R) ;
22:         if x_s ∈ X  AND ‖x_s − x_f‖ ≤ Threshold then
23:             critical region begin
24:             Path = Path to root(x_s) ∪ x_f;
25:             critical region end
26:             for kp=1 : P do
27:                 if kp != th_id then
28:                     j_kp = x_s;          ▷ copy this node
29:                     J_kp = J_kp ∪ j_kp;
30:             end for
31:             return ;
32:     end for
33: end procedure
```

synchronize threads for executing rewirds. Assuming to t_M equal the time required to copy a node inserted in the tree, the expected computation time of this strategy turns out to be:

$$\begin{aligned} & T(P) &=& \frac{T_\tau + T_V}{P} + T_M \\ \textit{for RRT*} \quad & T(P) &=& \frac{T_\tau + T_V + T_{Rew}}{P} + T_M \\ & T_M &=& \mathcal{O}\left(\frac{I(P-1)}{P} t_M\right) \end{aligned} \tag{6}$$

D. Blinded multiagents explorations

The strategy proposed in this Section aims at exploiting a significant number of threads, with both a reduced synchronizing need and allocation memory requirements. To this purpose, we have developed a variant of the RRT for which every exploring thread has not the entire knowledge of the tree, but it is conscious of a small portion of it. Therefore, we can deploy many threads to simultaneously explore the region $\underline{X}$ (ignoring the results found by the other agents) for

a certain amount of iterations N_b. After completing this sub-exploration, all data incoming from the agents are collected and stored in a centralized data base while the agents wait to begin a new explorative batch, completely forgetting the nodes found at the previous iteration. The described behaviour resembles one of many exploring ants, which reports the exploring data to a unique anthill. The pseudocode in Algorithm 5 details the proposed approach. The total amount of iterations I is split into C cycles of N_b explorations for every agent(thread). Clearly $I \simeq N_b(P-1)C$.

Algorithm 5 Multi Agent RRT

procedure BACKGROUND(x_o, x_f, P)
  $T_M = \{x_o\}$;
  for kp=1 : P **do**
    $T_{kp} = \emptyset$;
    launch thread(Agent work(kp));
  end for
  $k = 0$;
  while $k < I$ **do**
    for kp=1 : P **do**
      sample a $x_R \in T_M | \mathbb{P}(x_i) \propto \frac{1}{1+C(\tau_{i\to f})}$;
      $T_{kp} = \{x_R\}$;
    end for
    threads barrier;
    threads barrier;
    $T_M = \{T \cup T_1, \cdots, T_P\}$;
    if $\exists x_i \in T_M s.t. \|x_i - x_f\| \leq \epsilon$ **then**
      Kill agents;
      return Path to root(x_i)$\cup x_f$;
    $k = k + N_b \cdot P$;
  end while
  Kill agents;
end procedure

procedure AGENT WORK(th_{id})
  while TRUE **do**
    threads barrier;
    if $T_{th_{id}} = \emptyset$ **then**
      break;
    RRT search(); ▷ on $T_{th_{id}}$, with $I = N_b$;
    threads barrier;
  end while
end procedure

procedure KILL AGENTS
  for kp=1 : P **do**
    $T_{kp} = \emptyset$;
  end for
  threads barrier;
end procedure

Notice that there is no need to physically copy the states computed by the agents when inserting them into T_M, since threads share a common memory: a reference to the newer states is simply added to T_M. When considering this approach a bidirectional search is not implementable, while the RRT* can be extended as reported in Algorithm 6.

Algorithm 6 Multi Agent RRT*

procedure MULTI AGENT SEARCH *(x_o, x_f, P)
  $T_M = x_o$;
  for kp=1 : P **do**
    $T_{kp} = \emptyset$;
    launch thread(Agent work * (kp));
  end for
  $Sol = \emptyset$;
  $k = 0$;
  while $k < I$ **do**
    for kp=1 : P **do**
      sample a $x_R \in T_M | \mathbb{P}(x_i) \propto \frac{1}{1+C(\tau_{i\to f})}$;
      $T_{kp} = \{x_{kp}\}$;
      $Near^{kp} = \{x_i \in T_M | C(\tau_{i\to s}) <= \gamma(\frac{log(N)}{N})^{\frac{1}{d}}\}$;
      $T_{kp} = \{T_{kp} \cup$ Path to root(x^{kp}) $\forall x^{kp} \in Near^{kp}\}$
    end for
    threads barrier;
    threads barrier;
    $T_M = \{T \cup T_1, \cdots, T_P\}$;
    for $x_m \in \{x | x \in T_p \bigwedge \|x - x_f\| \leq \epsilon \ \forall p = 1, \cdots, P\}$ **do**
      $Sol = Sol \cup x_m$;
    end for
    $k = k + N_b \cdot P$;
  end while
  Kill agents;
  $x_{best} = \underset{x_S \in Sol}{\text{argmin}}$(Cost to root($x_S$));
  return Path to root(x_{best}) $\cup\, x_f$;
end procedure

procedure AGENT WORK *(th_{id})
  while TRUE **do**
    threads barrier;
    if $T_{kp} = \emptyset$ **then**
      break;
    RRT* search(); ▷ on T_{kp}, with $I = N_b$;
    threads barrier;
  end while
end procedure

When considering this formulation, we are modifying the canonical behaviour of an RRT algorithm, since agents explore every time some new roots, ignoring all the previously computed nodes. However, we empirically found that the global behaviour of the path search is not deteriorated. In particular, the optimality properties of the RRT* seems to be preserved, see Figure 3d.

The mean computation time of this kind of strategy can

be estimated as follows:

$$\begin{aligned} & T(P) &=& \frac{T_\tau + T_V}{P} \\ \textit{for RRT*} \quad & T(P) &=& \frac{T_\tau + T_V + T_{Rew}}{P} \\ & T_\tau &=& \mathcal{O}\left(C(N_b+1)N_b t_\tau\right) \\ & T_{Rew} &=& \mathcal{O}\left(C\gamma t_V \sum_{i=1}^{N_b}\left(\frac{log(i)^{\frac{1}{d}}}{i}\right)\right) \end{aligned} \quad (7)$$

The mean time spent for the querying operations is considerably lower, since such operations are performed by agents considering only their own local reduced size trees.

IV. Performance Analysis

This Section will present the performance obtained when applying the strategies proposed in Section III. The following will describe four different representative planning benchmarks, which will be employed later to show the computational times obtained applying the strategies proposed, when varying the number of threads.

A. Problem 1 : path planning

Here we consider a classical path planning problem. The robot adopted is a 3 d.o.f. robot with simplified shaped links. Every state $x \in X$ is simply the pose of the robot, i.e. the 3 values describing the angular joint displacements. The admissible region $\underline{X}$ is represented by the free configurational space, i.e. the set of poses x for which the robot is not in collision w.r.t. all the fixed obstacles populating the scene. The workspace of the robot is represented in Figure 3. An optimal trajectory $\tau_{1\to 2}$ is the segment in the joint space whose extremals are $x_{1,2}$, and the length of such segment is considered as $C(\tau_{1\to 2})$. The states x_s^k are obtained by following the segment connecting $x_{Nearest}$ to x_{target} with a fixed length step l (apriori decided):

$$\begin{aligned} L &= l \cdot \frac{x_{target} - x_{Nearest}}{\|x_{target} - x_{Nearest}\|} \\ x_s^k &= x_{Nearest} + k \cdot L \end{aligned} \quad (8)$$

B. Problem 2 : path planning with a 7 d.o.f robot

Another path planning problem is considered here, similar to the previous one and with the same workspace. However, here the robot considered, reported in Figure 4, has 7 d.o.f. Notice that time T_V is more than doubled with respect to Problem 1; while time T_τ is almost the same.

C. Problem 3 : trajectory planning

In this case, we consider the same scenario of Problem 1, but considering a trajectory planning problem. The state x is composed as follows:

$$x = [q^T \quad \dot{q}^T]^T \quad (9)$$

where q is the pose of the robot (angular displacements), while $\dot{q}$ are the joint velocities. The second order minimum time trajectory connecting two states $x_{1,2}$ is considered as $\tau_{1\to 2}$. Such a trajectory can be computed by making use of the Reflexxes library [15], which takes into account the kinematic limitations of a manipulator, but not the obstacles populating the scene. The cost $C(\tau_{1,2})$ is assumed as the time for reaching x_2 when following $\tau_{1,2}$. The states $x_s^{1,\cdots,K}$ considered by the steering function are separated by a (small) fixed time Δt: $x_s^k = \tau(k\Delta t)$. Notice that for this case time T_V is equal to the same when considering Problem 1, while time T_τ is significantly higher.

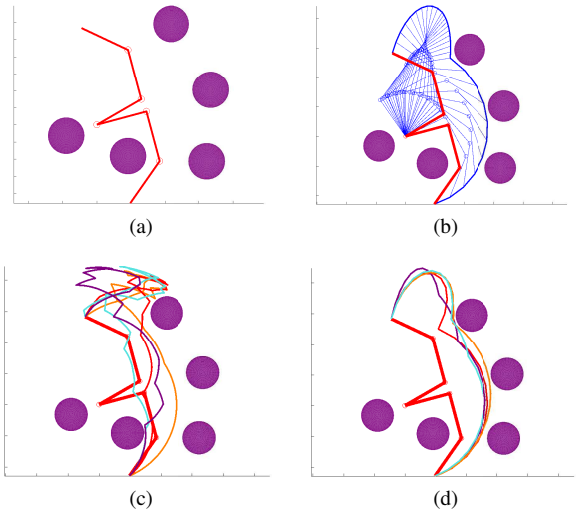

Fig. 3: The two pictures on the top depict the planning problem considered for Problem 1. The purple spheres are the obstacles populating the workspace of the robot. On the left the poses x_o and x_f, while on the right the optimal path obtained by applying RRT* with some intermediate poses and the trajectory of the E.E. The pictures at the bottom are examples of paths found when applying a standard (serial) RRT on the left; the multiagents version of RRT* described in Section III-D on the right.

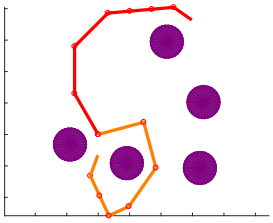

Fig. 4: The planning problem considered for Problem 2. The starting and ending poses of the robot are depicted respectively in red and in orange.

D. Problem 4 : kinodynamic planning

This benchmark considers an optimal control problem. Consider the following non linear dynamical system whose equation of motion is as follows:

$$\dot{x} = f(x) + u \quad (10)$$

$$f(x) = T^{-1} \cdot diag(-1 - e^{-\alpha_1 x_1^2}, \cdots, -1 - e^{-\alpha_6 x_6^2}) \cdot T \quad (11)$$

where all coefficients $\alpha_{1,2,3,4,5,6}$ are > 0, while T is an orthogonal matrix. The state-space considered for planning is clearly the state-space of the above dynamical system. A trajectory $\tau_{1\to 2}$ is obtained integrating eq. (10) forward in time, under the effect of the optimal linear regulator obtained considering a linearization of the system:

$$\begin{aligned} \dot{x} &= (f(x_1) - Ax_1) + Ax + u \quad \texttt{where} \quad A = \frac{\partial f}{\partial x}|_{x_1} \\ u &= \hat{u} + A(x_1 - x_2) - f(x_1) \\ \hat{u} &= K(x_2 - x) \end{aligned} \tag{12}$$

The gain matrix K is computed such that:

$$K = \underset{K}{\operatorname{argmin}} \left(\int_0^\infty \left((x - x_2)^T Q (x - x_2) + \hat{u}^T R \hat{u} \right) dt \right) \tag{13}$$

The steering function simply follows τ with fixed sample time. The admissible region is structured as follows:

$$\begin{aligned} x \in \underline{X} \Rightarrow -2 \quad &\leqslant x_i \leqslant \quad 2 \quad \bigwedge \\ -5 \quad &\leqslant u_i(x) \leqslant \quad 5 \quad \forall i = 1, \cdots, 6 \end{aligned} \tag{14}$$

E. Results

Figures 5, 6, 7 and 8 report the performance obtained when applying the strategies contained in MT-RRT, i.e. those described in Section III, when varying the number of employed threads. ξ is the efficiency of the parallelization (normalized speed up, see [16]), which is defined as follows (adopting the same notation of Section III):

$$\xi = \frac{T_{serial}}{T(P) \cdot P} \tag{15}$$

where T_{serial} is the computation time absorbed by the standard serial versions presented in Section II. In particular T_{serial}, was assumed as the mean value recorded for a batch of simulations. When having a good parallelization, ξ should be close to 1 or higher. For all the simulations, the maximal number of iterations was set equal to 2500, and the termination criteria for all the versions adopting an RRT or a bidirectional RRT strategies was disabled: even in a case a solution was found, the search was not interrupted. These results were obtained using a quad core laptop embedded with an Intel(R) core i7-6500u.

V. Conclusions

The strategy presented in Section III-A results to be effective for Problem 3, i.e. the scenario for which $T_\tau \gg T_V$, while for all the other problems its performance result to be poor in comparison to the other approaches. The reason is the presence of severe overhead times: after every collective search of the Nearest Neighbour or the determination of a Near set, threads must wait for all the others to end that search. Moreover, this happens at every iteration.

The strategies reported in Sections III-B and III-C achieve comparable performance when considering the classical RRT and its bidirectional version, while for RRT* the approach presented in Section III-C outperforms the one of Section III-B. The reason is the presence of the critical regions for applying rewirds, which leads to time waste.

The multi agent approach of Section III-D is significantly faster w.r.t all the other methods for all the RRT versions considered, because threads perform parallel extensions on reduced-size trees. However, when considering the RRT* algorithm, the approach in III-D is an approximation of the canonical version, that might lead to solutions that are near-optimal. Therefore, for those cases for which the optimality is crucial, the approach of Section III-C can be efficiently deployed. The entire MT-RRT library is available at [17], which contains also detailed directions to derive a customized planning problem to be solved with the usage of MT-RRT.

References

[1] O. Khatib, "Real-time obstacle avoidance for manipulators and mobile robots," in *Autonomous robot vehicles*. Springer, 1986, pp. 396–404.

[2] E. Rimon and D. E. Koditschek, "Exact robot navigation using artificial potential functions," *IEEE Transactions on robotics and automation*, vol. 8, no. 5, pp. 501–518, 1992.

[3] M. Nolte, M. Rose, T. Stolte, and M. Maurer, "Model predictive control based trajectory generation for autonomous vehicles-an architectural approach," 2017.

[4] G. B. Avanzini, A. M. Zanchettin, and P. Rocco, "Constrained model predictive control for mobile robotic manipulators," *Robotica*, vol. 36, no. 1, pp. 19–38, 2018.

[5] J. J. Kuffner and S. M. LaValle, "Rrt-connect: An efficient approach to single-query path planning," in *Robotics and Automation, 2000. Proceedings. ICRA'00. IEEE International Conference on*, vol. 2. IEEE, 2000, pp. 995–1001.

[6] P. Cheng and S. M. LaValle, "Reducing metric sensitivity in randomized trajectory design," in *Intelligent Robots and Systems, 2001. Proceedings. 2001 IEEE/RSJ International Conference on*, vol. 1. IEEE, 2001, pp. 43–48.

[7] L. Zhang and D. Manocha, "An efficient retraction-based rrt planner," in *Robotics and Automation, 2008. ICRA 2008. IEEE International Conference on*. IEEE, 2008, pp. 3743–3750.

[8] D. J. Webb and J. v. d. Berg, "Kinodynamic rrt*: Optimal motion planning for systems with linear differential constraints," 2012.

[9] A. Tahirovic and F. Janjoš, "A class of sdre-rrt based kinodynamic motion planners," in *American Control Conference (ACC)*, vol. 2018, 2018.

[10] D. Henrich, "A review of parallel processing approaches to motion planning," in *Robotics and Automation, 1996. Proceedings., 1996 IEEE International Conference on*, vol. 4. IEEE, 1996, pp. 3289–3294.

[11] J. Bialkowski, S. Karaman, and E. Frazzoli, "Massively parallelizing the rrt and the rrt," in *Intelligent Robots and Systems (IROS), 2011 IEEE/RSJ International Conference on*. IEEE, 2011, pp. 3513–3518.

[12] D. Devaurs, T. Siméon, and J. Cortés, "Parallelizing rrt on distributed-memory architectures," in *Robotics and automation (ICRA), 2011 IEEE International Conference on*. IEEE, 2011, pp. 2261–2266.

[13] J. J. Kuffner and S. M. LaValle, "Rrt-connect: An efficient approach to single-query path planning," in *Robotics and Automation, 2000. Proceedings. ICRA'00. IEEE International Conference on*, vol. 2. IEEE, 2000, pp. 995–1001.

[14] S. Karaman and E. Frazzoli, "Sampling-based algorithms for optimal motion planning," *The international journal of robotics research*, vol. 30, no. 7, pp. 846–894, 2011.

[15] T. Kröger, "Opening the door to new sensor-based robot applications:the reflexxes motion libraries," in *Robotics and Automation (ICRA), 2011 IEEE International Conference on*. IEEE, 2011, pp. 1–4.

[16] B. Wilkinson and M. Allen, *Parallel programming*. Prentice hall Upper Saddle River, NJ, 1999, vol. 999.

[17] https://github.com/andreacasalino/MT_RRT.

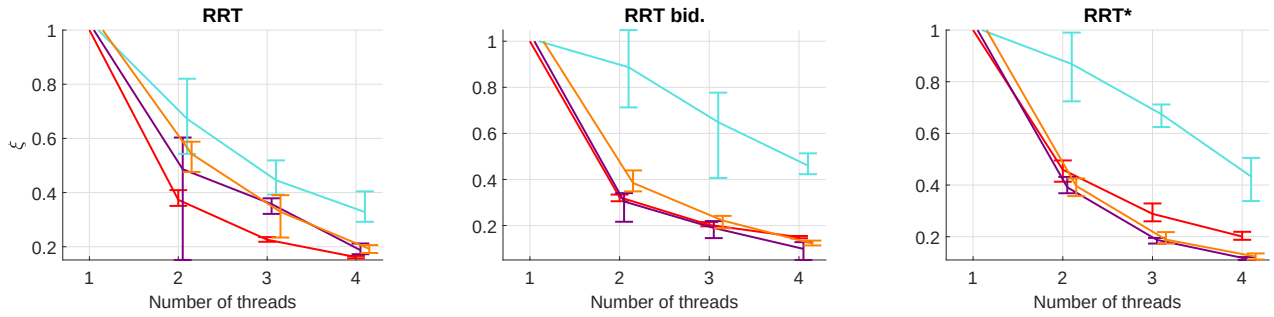

Fig. 5: Performance obtained adopting the strategy described in Section III-A.

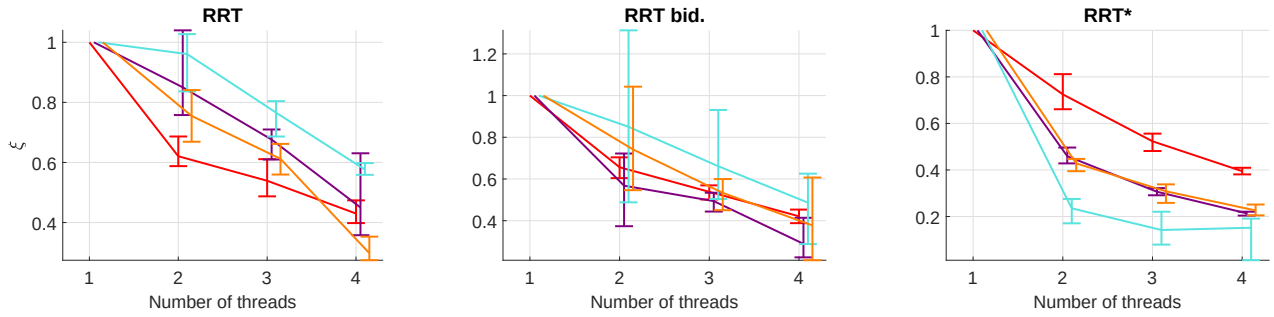

Fig. 6: Performance obtained adopting the strategy described in Section III-B.

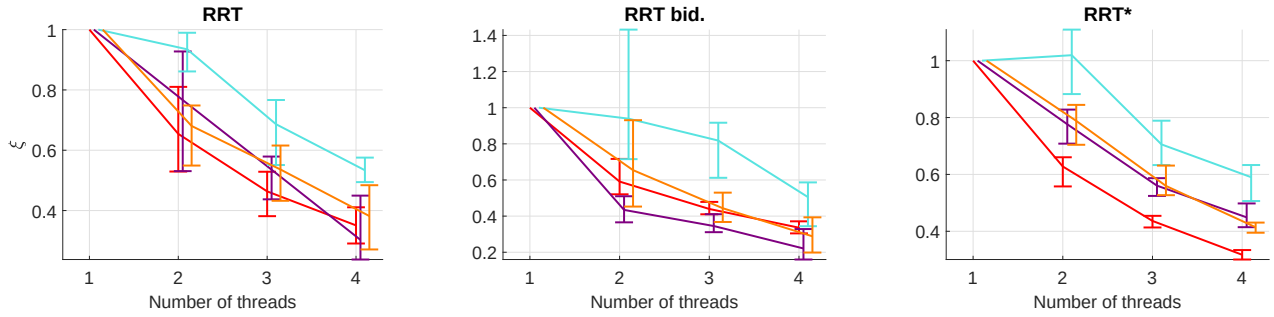

Fig. 7: Performance obtained adopting the strategy described in Section III-C.

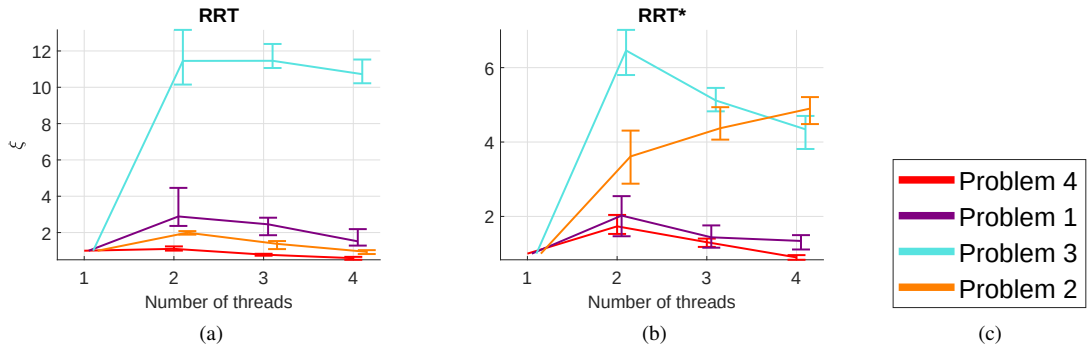

Fig. 8: Performance obtained adopting the strategy described in Section III-D.