On Parallel RRTs for Multi-robot Systems

Stefano Carpin, Enrico Pagello
Department of Electronics and Informatics
The University of Padova - ITALY

Abstract. Rapidly-exploring Random Trees are planning algorithms recently introduced for a broad class of path planning problems. In this paper we provide three different ways to better the performance of such algorithms. Numerical results obtained implementing them over a parallel system outline an optimal speed up.

1 Introduction

In the last few years a number of new application fields requiring the solution of problems involving not only state space constraints but also differential constraints have been introduced. The so called kinodynamic motion planning problem ([6]) has to be solved when designing digital actors, humanoid robots, virtual prototyping systems, or to study molecular structures ([14]). Since even the simpler generalized mover's problem is PSPACE-hard ([17]), approximated and random techniques have been introduced ([10]). Probabilistic path planners achieve probabilistic completeness, that is, provided that a solution exists, by allotting more time to the planner we can improve the chance it will eventually find a solution.They are then being used to solve kinodynamic problems, characterized by high dimensional configuration spaces. In this scenario, Rapidly-exploring Random Trees (RRT) ([15, 16]) exhibit good results and have been applied for solving real world problems about systems involving many degrees of freedom ([11]). Nevertheless searching a path in a high dimensional configuration space is still a time consuming task. In this paper we illustrate how it is possible to implement a parallel version of RRT based motion planners which yields optimal speed up.

2 The RRT algorithm

The problem statement is the following: given a metric space X, a starting point $x_{init} \in X$ and a goal region $X_{goal} \subset X$ or goal state $x_{goal} \in X$, find a continuous path from x_{init} to x_{goal} which does not intersect the region $X_{obs} \subset X$. In a classical motion planning problem the metric space X is the configuration space of the robot. In a kinodynamic motion planning problem X is constrained to be a subset of the tangent bundle of the configuration space ($X \subset \mathcal{T}(C)$). We briefly review RRT-connect ([12]), one of the basic versions of RRT. As illustrated in algorithms 1.a and 1.b, two trees are incrementally built, one starting from the initial configuration and the other starting from the goal configuration. Each tree node is associated with a configuration satisfying the dynamic constraints of the system. The search ends when the two trees join each other or when the allotted time expires (this to prevent infinite search over an unsolvable problem instance). During the $EXTEND$ step a new node tree is added by integrating the differential constraints of the system (NEW_CONFIG) and

```
1: RRT_CONNECT_PLANNER(x_init, x_goal)
2: INPUT starting and goal points x_init and x_goal
3: τ_a.init(x_init)
4: τ_b.init(x_goal)
5: for k = 1 to K do
6:    q_rand ← RANDOM_CONFIG
7:    if NOT EXTEND(τ_a, q_rand) = Trapped then
8:       if EXTEND(τ_b, q_new) = Reached then
9:          RETURN PATH(τ_a, τ_b)
10:      end if
11:   end if
12:   SWAP(τ_a, τ_b)
13: end for
14: RETURN Failure
```

(a) Bidirectional RRT

```
1: EXTEND(τ, q)
2: INPUT a tree τ and a random configuration q
3: RETURN Trapped or Reached or Advanced
4: q_near ← NEAREST_NEIGHBOR(q, τ)
5: if NEW_CONFIG(q, q_near, q_new) then
6:    τ.add_vertex(q_new)
7:    τ.add_edge(q_near, q_new)
8:    if q_new = q then
9:       RETURN Reached
10:   else
11:      RETURN Advanced
12:   end if
13: end if
14: RETURN Trapped
```

(b) Extension of the RRT

Figure 1: Basic RRT

checking its belonging to X_{free}. The new configuration is created starting from a randomly generated configuration and the closest configuration in the tree being grown.

The strength of this approach is that with this expansion technique, the tree is biased to grow towards unexplored regions of the free configuration space (X_{free}). Another advantage is that the path being built satisfies the dynamical constraints of the systems, thus avoiding the classical two steps approach involving path search and subsequent smoothing. An analysis of the asymptotic behavior of the RRT algorithm illustrates that the distribution of the samples converges to the random sampling process used to get the samples. It can also be shown that the RRT-Connect algorithm is probabilistic complete (see [12] for details). It is clear that the size of the output produced, i.e. the number of nodes in the solution tree, is a random variable itself.

3 Parallel formulations of the RRT algorithm

One of the possible ways to improve the performance of the RRT algorithm is to develop a parallel implementation. Indeed *parallel motion planning algorithms* have been already studied and it has been shown that this can be a viable opportunity to speed up paths computation ([8]). In the framework of *randomized* algorithms, two major ways can be undertaken. The first approach is based on the *OR paradigm* approach, where a set of processors is engaged in the solution of the same instance of the search problem. Each processor executes the same algorithm and the first one which finds a solution sends a message to the others to stop their computation ([4, 5]). In this way it is not necessary to reformulate the algorithm, but it suffices to just add a few communication steps. In the second approach it is necessary to modify the algorithm so that every processor contributes to the same solution of the problem. In this case we will adopt an approach similar to the one proposed in [1].

3.1 OR parallel RRT implementation

In the so called *OR parallel* paradigm a set of processors solves the same computational problem using the same randomized algorithm[1]. The solution found by the set of processors is the solution found by the first processor which found a solution for the problem. It is also possible to let all the processors terminate the computation and then take the best one as the solution produced by the set according to some quality index (see [2] for an example), but we will not consider this variant here. In the basic OR parallel computation every processor then carry out all the computation on its own and communication is performed just when the first processor finds a solution. Then a *termination message* is broadcasted to all the other processors to let them terminate (for this reason this method is also called *barrier parallel* because the first which finds a solution stops all the others). The goal of this method is to minimize the time needed to compute a solution. The theoretical explanation ([5]) stems from the observation that the time required to find a solution to the motion planning problem using the RRT algorithm is a random variable. Let us suppose that m processors execute the RRT algorithm to solve the same problem instance and let T^i be the time spent by the i-th processor ($1 \leq i \leq m$). All the T^is are independent and identically distributed. Let $P_i(t)$ be the probability that the time spent by processor i to solve the problem will not exceed t, i.e.

$$P_i(t) = \Pr[T^i < t] \tag{1}$$

Thus the probability $P^m_{none}(t)$ that none of the m processors will find a solution in time t is

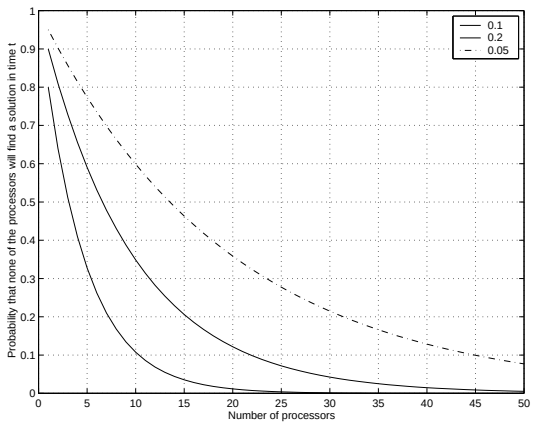
$$P^m_{none}(t) = (1 - P_1(t))^m. \tag{2}$$

Then, by increasing the number of processors it is possible to decrease the probability that solving an instance of the problem will take more than a fixed amount of time t.
Figure 2.a illustrates the trend of $P^m_{none}(t)$ for different values of $P_i(t)$ as a function of the number of processors engaged in the OR computation. It is clear that even for poor (low) values of $P_i(t)$, good (low) values of $P^m_{none}(t)$ can be reached as the number of processors increases.
Algorithm 2.b illustrates how to extend the bidirectional RRT algorithm to get a parallel search to solve a motion planning problem. It is assumed that when the BROADCAST instruction (line 10) is executed by one processor, subsequent evaluations of the test in line 6 by other processors will return false, so that their execution stops. This implements the *barrier*.

3.2 Embarrassingly parallel RRT implementation

The analysis of algorithms 1.a and 1.b outlines that if we allow a set of processors to concurrently (and cooperatively) work to build the RRT, most of the operations can be performed in parallel. Indeed, coordination is required just for the steps 6 and 7 of algorithm 1.b, to deny concurrent inconsistent modification of the tree being built. Communication is required only to stop processors' computation when the solution is found by one of them. Then the parallel implementation belongs to the class of *embarrassingly parallel* algorithms. On the basis of the former observations, we illustrate the parallel algorithm to solve an instance of the motion

[1] the use of a deterministic algorithm is inappropriate since all the processors will then execute the same instructions and then produce the same solution



(a) Trend of $P^m_{none}(t)$ as a function of the number of processors for different values of $P_i(t)$

1: **OR_RRT_BIDIRECTIONAL**(x_{init},x_{goal})
2: INPUT starting and goal configurations x_{init} and x_{goal}
3: $\tau_a.init(x_{init})$
4: $\tau_b.init(x_{goal})$
5: $done \leftarrow$ FALSE
6: **while** $(NOT\ done)\ AND$
$(termination\ message\ not\ yet\ received)$ **do**
7: $q_{rand} \leftarrow$ RANDOM_CONFIG
8: **if** NOT EXTEND(τ_a, q_{rand}) = *Trapped* **then**
9: **if** EXTEND(τ_b, q_{new}) = *Reached* **then**
10: BROADCAST *termination message*
11: RETURN PATH(τ_a, τ_b)
12: **end if**
13: **end if**
14: SWAP(τ_a, τ_b)
15: **end while**

(b) OR Parallel Bidirectional RRT

Figure 2: Or parallel RRT

planning problem.

As a first step it is necessary to choose the computational model which will execute the parallel algorithm, as this will strongly influence its design and coding. Among the great number of models introduced in literature we adopt the *shared memory* model. In this model a number of processors can execute their own local program and they can communicate by exchanging data through a shared memory ([9]). We assume a *concurrent read exclusive write* (CREW) memory access policy. This choice stems from the coherence of this model with the symmetrical multiprocessor (SMP) machines available in our department, so that we can easily implement the algorithm to verify its effectiveness.

It is assumed that the tree being built resides in the shared memory, so that each processor can access and update its nodes. Algorithms 3.a and 3.b show the embarrassingly parallel implementation of a motion planning algorithm based on RRTs. The shared memory contains three data structures shared among all the processors, namely τ_a, τ_b and $done$. τ_a and τ_a are initialized as a couple of trees with a single node which holds the starting and goal configurations respectively, while $done$, initially set to $false$, is used to stop the computation on all the processors when the trees τ_a and τ_b meet each other. To prevent concurrency related problems when updating shared data, binary semaphores are used to guarantee exclusive write access. We suppose that a semaphore is associated with every shared data and that the LOCK and UNLOCK operations atomically acquire or release the semaphores. It is assumed that each processor owns a model of X_{free} and starts executing PARALLEL_RRT_BIDIRECTIONAL. The correctness and convergence of the embarassingly parallel version of the RRT based motion planner trivially descends from the corresponding properties already proved for the sequential version. This because parallelism is introduced to just speed up nodes generation, but the properties of the nodes being generated and added to the tree do not change.

```
1: PARALLEL_RRT_BIDIRECTIONAL(τ_a, τ_b, done)
2: INPUT τ_a, τ_b, RRTs rooted on starting and goal
   configurations and a boolean shared variable done
3: while NOT done do
4:    q_rand ← RANDOM_CONFIG
5:    if NOT PARALLEL_EXTEND(τ_a, q_rand)
      = Trapped then
6:       if PARALLEL_EXTEND(τ_b, q_new)
         = Reached then
7:          LOCK(done)
8:             done ← TRUE
9:          UNLOCK(done)
10:      end if
11:   end if
12:   SWAP(τ_a, τ_b)
13: end while
```

(a) Parallel RRT based motion planning

```
1: PARALLEL_EXTEND(τ, q_rand, q_goal)
2: INPUT a tree τ, random and goal configurations q_rand
   and q_goal
3: RETURN Trapped, Reached or Advanced
4: q_near ← NEAREST_NEIGHBOR(q_rand, τ)
5: if NEW_CONFIG(q_rand, q_near, q_new) then
6:    LOCK(τ)
7:       τ.add_vertex(q_new)
8:       τ.add_edge(q_near, q_new)
9:    UNLOCK(τ)
10:   if q_new = q_goal then
11:      RETURN Reached
12:   else
13:      RETURN Advanced
14:   end if
15: else
16:   RETURN Trapped
17: end if
```

(b) Parallel EXTENSION of the RRT

Figure 3: Basic RRT

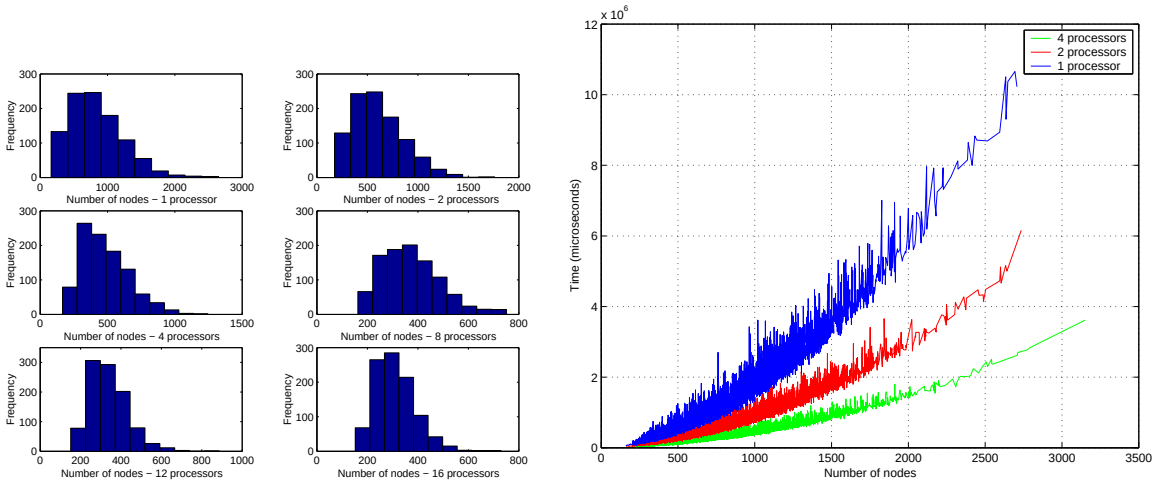
3.3 Combining the OR parallel and the Embarrassingly Parallel algorithms

In the previous two subsections two different parallel implementations of the RRT algorithm were illustrated. The so called *OR Parallel* implementation aims to decrease the time spent to build the solution tree by allowing more computational units to solve the same problem. The speed up then stems from finding a solution tree which includes a small number of nodes. As illustrated, the chance of finding such a tree increases with the number of searching processes. On the other hand, the *embarrassingly parallel* algorithm decreases the time spent to find a solution allowing more processors to cooperatively work to build the same tree.
It is then possible to follow two different ways to obtain the goal of fast solution generation. Each of the two approaches has its own advantages. It is of course possible to combine the two approaches to get the best of the two. Given a parallel computer with a shared memory architecture, we can divide processors into groups and let every group solve the given problem using the embarrassingly parallel algorithm. The first group of processors which finds a solution sends a termination message to other groups as required by the OR parallel approach. We call *embarrassingly OR parallel* this hybrid technique. Its implementation trivially descends the two base algorithms.

4 Numerical results

4.1 Parallel Computer Architecture

All the numerical results illustrated in this section have been obtained by running the parallel algorithms on an IBM SP/R6000 parallel computer based on *Winter Hawk* nodes ([7]). The computer is composed of 4 nodes connected by a high speed switch which warrants 500 Mbits of bandwidth. Every node includes 4 processors arranged in an SMP like architecture. 2 Gbytes of memory is shared in every node. Processors on different nodes can communicate and exchange data using the Message Passage Interface (MPI). The software has been de-



(a) (b)

Figure 4: OR parallel and Embarrassingly parallel performance

veloped in ANSI C++ and has been compiled using the highly optimized IBM compiler.This architecture is well suited for implementing the three different parallel strategies previously illustrated. For the OR parallel algorithm, each of the 16 processors performs its computation independently and the termination message is sent or received using the MPI primitives. For the embarrassingly parallel implementation we used the 4 four processors of a single node. This because processors in different nodes do not share memory. Shared address space access among the (up to four) processes is obtained using the Pthread API. Finally, the combined implementation of the OR parallel and embarrassingly parallel algorithms is trivial. Every node of 4 processors executes the embarrassingly parallel algorithm and the first node which solves the problem sends a termination message to other nodes.

4.2 The Motion Planning Problems Studied

In this section we compare the performance of the three parallel implementations of the RRT based motion planner. We set up a framework to solve a multi-robot motion planning problem involving circular holonomic robots moving adminst polygonal obstacles in a shared planar environment (see also [3] for an indepth discussion of this framework). Since every robot has two degrees of freedom (i.e. the coordinates of the center of its bounding cylinder), planning the motion of a system composed by N robots implies planning a motion in a subset of R^{2N}. It should be noticed that even if we are not studing a kinodynamic problem, this does not affect the speed up we get.

Figure 4.a illustrates the results of the OR parallel version of the RRT run over sets of processors of different sizes. Accordingly to the theoretical explanation previously illustrated, it is evident that increasing the number of processors, the expected size of the solution RRT decreases. Numerical data refer to 1000 trials of a two robots motion planning problem (i.e. planning is performed in a subset of R^4).

Figure 4.b compares the performace of the embarrassingly parallel version of the RRT algorithm run over one, two and four processors. The chart plots the time spent to solve the

problem versus the size of the RRT built to solve it. For fairness of comparison, we compare the results obtained in 5000 trials run over the same motion planning problem.
A way to have a numerical measure of the performance gain is to evaluate the *speed up* of the parallel algorithm ([13]). We compare the trend of the time spent versus the size of the tree for one, two and four processors. Such comparison however is not straightforward since the functions are not smooth. To eliminate the influence of spikes, when comparing the performances we interpolated each function with quadratic polynomials fitting the data in a least-square sense and then compared the three polynomials. The choice of quadratic polynomials stems from the implementation.

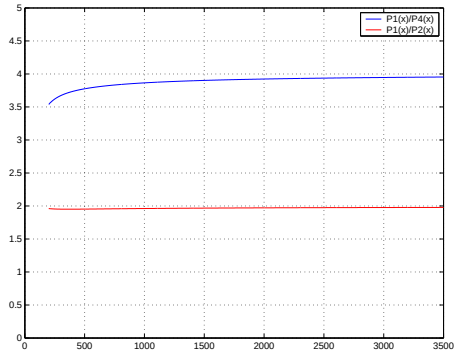
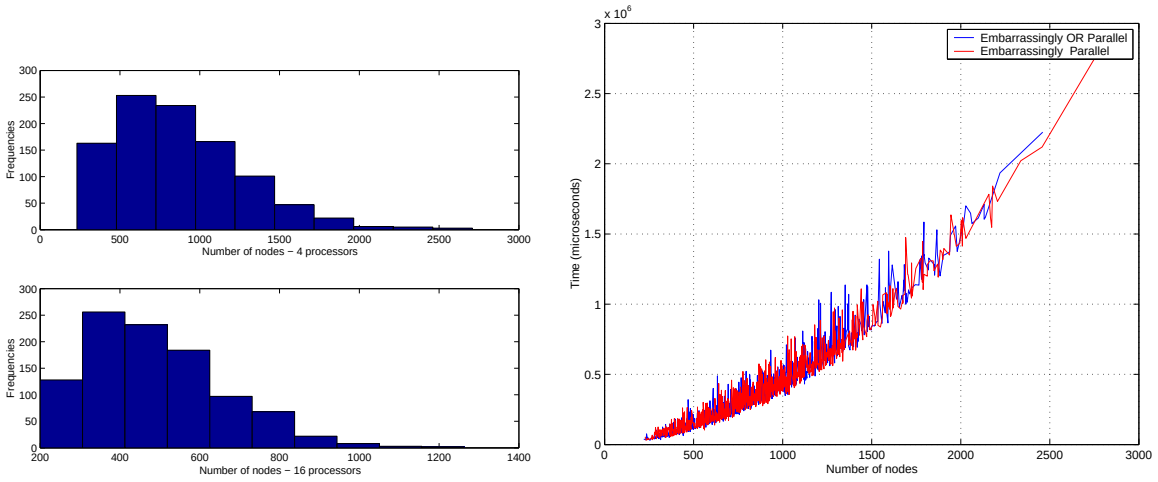


Figure 5: Speed up obtained with the embarrassingly parallel algorithm run over two and four processors

Since we do not adopt particular techniques for nearest neighbor searching, the complexity of the algorithm is quadratic in the number of nodes in the solution tree. Let $P_1(n)$, $P_2(n)$ and $P_4(n)$ the polynomials interpolating the functions obtained with one, two and four processors respectively. Figure 5 plots the ratio between such functions to illustrate the speed-up obtained. It is evident that the speed up gained is near optimal (we remind that the speed up can not exceed the number of processors used, i.e. two and four, respectively).
Finally we run the embarrassingly OR parallel algorithm over the set of 16 processors divided in four blocks of four processors. As expected, this is the best we can get. The number of nodes generated to find a solution is smaller when compared with the simple embarrassingly parallel algorithm run over four processors and on the other hand the trend of the function of time versus the size of the tree is the same we got with the embarrassingly parallel algorithm. This is clearly illustrated in figures 6.a and 6.b, respectively.



(a) (b)

Figure 6: Performance of the embarassingly OR parallel algorithm

5 Conclusions

In this paper we illustrated how it is possible to improve the performance of RRT based motion planners using a parallel computer. Two basic techniques can be applied, namely classical parallel and cooperative embarassingly parallel computation. Finally the two techniques can be combined to get the best of the two apporaches. All the techniques have been implemented on an parallel system, and numerical results give evidence of the speed up obtained.

References

[1] N.M. Amato and L.K. Dale. Probabilistic roadmaps are embarrassingly parallel. In *Proc. of ICRA*, pages 688–694, 1999.

[2] S. Carpin and E. Pagello. A distributed algorithm for multi-robot motion planning. In *Proceedings of EUROBOT*, pages 207–214, 2001.

[3] S. Carpin and E. Pagello. Exploiting multi-robot geometry for efficient randomized motion planning. In *Intelligent Autonomous Systems 7*, pages 54–62, 2002.

[4] S. Caselli and M. Reggiani. Erpp: An experience-based randomized path planner. In *Proc. of ICRA*, pages 1002–1008, 2000.

[5] D. Challou, D. Boley, M. Gini, V. Kumar, and C. Olson. Parallel search algorithms for robot motion planning. In K. Gupta and A.P. del Pobil, editors, *Practical Motion Planning*, pages 115–132. John Wiley & Sons, 1998.

[6] B. Donald, P. Xavier, J. Canny, and J. Reif. Kinodynamic motion planning. *Journal of the ACM*, 40(5):1048–1066, November 1993.

[7] M.R. Barrios et al. *Inside the RS6000/SP*. IBM International Support Organization, 1998.

[8] D. Henrich. A review of parallel processing approaches to motion planning. In *Proc. of ICRA*, pages 3289–3294, 1996.

[9] J. JáJá. *An Introduction to Parallel Algorithms*. Addison Wesley, 1992.

[10] L.E. Kavraki, P. Švestka, J.C. Latombe, and M.H. Overmars. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *IEEE Transactions on Robotics and Automation*, 12(4):566–580, 1998.

[11] J. Kufner, K. Nishiwaki, S. Kagami, M. Inaba, and H. Inoue. Motion planning for humanoid robots under obstacle and dynamic balance constraints. In *Proc. of ICRA*, pages 692–698, 2001.

[12] J.J. Kufner and S.M. LaValle. Rrt-connect: An efficient approach to single-query path planning. In *Proc. of ICRA*, pages 995–1001, 2001.

[13] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing*. The Benjamin/Cummings Publishing Company, 1994.

[14] J.C. Latombe. Motion planning: A journey of robots, molecules, digital actors, and other artifacts. *The International Journal of Robotics Research - Special Issue on Robotics at the Millennium*, 18(11):1119–1128, 1999.

[15] S.M. LaValle and J.J. Kufner. Randomized kinodynamic planning. *International Journal of Robotics Research*, 20(5):378–400, 2001.

[16] S.M. LaValle and J.J. Kufner. Rapidly-exploring random trees: Progress and prospects. In D. Rus B. Donald, K. Lynch, editor, *Algorithmic and Computational Robotics: New Directions*, pages 45–59. A.K. Peters, 2001.

[17] J.H. Reif. Complexity of the mover's problem and generalization. In *Proceedings of the 20th IEEE Symposium on Foundations of Computer Science*, pages 421–427, 1979.